A PROVABLY SECURE OPERATING SYSTEM: THE SYSTEM, ITS APPLICATIONS, AND PROOFS

Computer Science Laboratory Report CSL-116
Second Edition

May 7, 1980

By: Peter G. Neumann,
Robert S. Boyer,
Richard J. Feiertag,*
Karl N. Levitt, and
Lawrence Robinson**

Computer Science Laboratory
Computer Science and Technology Division

*Now at Sytek Corporation, Sunnyvale, California
**Now at Ford Aerospace and Communications Corp., Palo Alto, California

Approved:

Jack Goldberg, Director
Computer Science Laboratory

David H. Brandin, Executive Director
Computer Science and Technology Division

A PROVABLY SECURE OPERATING SYSTEM:

THE SYSTEM, ITS APPLICATIONS, AND PROOFS --

SECOND EDITION, 7 May 1980

ABSTRACT

This report provides a detailed description of the design of a secure operating system and some of its applications, along with informal proofs of some of the properties related to security. Discussed here are:

* a formal methodology for the design and implementation of computer operating systems and applications subsystems, and for the formal proof of properties of such systems, with respect to both the design and the implementation;

* the design of a secure capability-based operating system according to this methodology to meet advanced security requirements, together with relevant implementation considerations;

* the design of several application subsystems for this operating system, including support for multilevel security classifications, for confined subsystems, for a secure relational data management system, and for monitoring of security;

* the statement and informal proof of properties of the design for the operating system and for certain application subsystems, and the consideration of proofs of implementation correctness;

* consideration of the system design with respect to distributed systems and networks;

* an evaluation of the significance of this work, and considerations for future development of secure systems and subsystems.

TABLE OF CONTENTS

PREFACE TO THE SECOND EDITION

This report is being issued in a second edition for several reasons. First, several design changes have occurred since the first edition of February 1977. Second, substantially more is known about the verification of designs and of implementations -- although work still remains to be done on modeling the PSOS protection mechanism formally. Third, the on-line tools supporting the methodology have progressed further, and can have a big impact on any future PSOS developments. Fourth, the state of the hardware art has progressed substantially, to the point where an implementation is now quite feasible, based on partly on the availability of new machines with sufficiently flexible microcode. Fifth, there is additional related work that is worth referencing. Sixth, a detailed prototype implementation study is just beginning. Finally, the demand for copies of the first edition has exceeded the supply. Thus it seems appropriate to make a revised version available at this time.

For the convenience of the careful reader of the 1977 report, an incremental summary of the changes introduced in this second edition is included here. For historical reasons, we have chosen to leave intact the original incremental preface which related the 1977 report to its precursor of 1975. A reader not familiar with the earlier versions may well choose to skip both incremental prefaces, at least on the first reading of this report. We have also added a new chapter, to provide a summary of recent related work.

PREFACE TO THE FIRST EDITION [11 February 1977]

This report provides a self-contained description of work on the design of a secure operating system and on proofs of significant properties of this system with respect to security. This work has been done by members of the Computer Science Laboratory of SRI [with the work beginning on 3 April 1973].

The [first edition of this] report [in 1977] supersedes an earlier report (Neumann et al. [75]) dated 13 June 1975. For the convenience of readers of that report, an incremental preface follows that indicates the differences between the two reports. For readers unfamiliar with the earlier report, the incremental preface may be omitted without loss of understandability.

INCREMENTAL PREFACE FOR THE SECOND EDITION [May 1980]
(FOR READERS OF THE 1977 REPORT AND FOR HISTORICAL PERSPECTIVE)

This second edition includes a few design changes that have been made in the three years since the previous report. It also includes references to new material that represents significant advances in the development and use of the hierarchical development methodology and related proof technology.

The only substantive change in the design involves the introduction of "windows", which are a means of efficiently obtaining protectable subsegments that can be addressed directly by capabilities. Windows are particularly useful for small-sized objects such as stack frames.

A minor change in the design affects primarily the internal structure of the lowest levels of the specification hierarchy. What was previously a level containing several modules has been reorganized into several levels. The changes are reflected in the tables at the end of the technical summary (Chapter 0).

Part V has been extended. It includes a brief summary of recent uses of HDM for systems other than PSOS, as well as recent progress in formal verification. Major secure systems developments currently using HDM include KSOS-11 (Ford) and KSOS-6 (Honeywell). Further, SRI has designed and specified a family of secure, real-time systems, known as TACEXEC (for the Army). In addition, SRI is using HDM to develop SIFT, an ultrareliable fault-tolerant system, which includes both design proofs of fault tolerance and code proofs. Perhaps most notably, the KSOS-11 kernel specifications have now been thoroughly subjected to tools determining whether they are multilevel secure. Various verification systems have been built and are being applied to some of these systems.

The references have been upgraded to include recent material. Particularly noteworthy are new material on HDM (e.g., the 3-volume Handbook), a 1979 NCC paper on the foundations of PSOS (added here as Appendix G), a 1978 NCC paper on the impact of HDM on security (included here as Appendix H), a report on proving multilevel security mechanically, Boyer and Moore's new book on theorem proving, their axiomatization of HDM, and various successful proofs. Also included are relevant references to KSOS, TACEXEC, and SIFT.

Rigorous proofs that the specifications satisfy the desired security assertions are outlined. These proofs are largely syntactic in nature, and can be extensively automated by the use of the specification checking tools and related tools.

Proofs of implementation correctness are outlined to support the claim that it is feasible to prove formally that implementation programs are consistent with the specifications. This is done first in a uniprogramming mode, and then in a multiprogramming mode. A fundamental theorem has been stated and proved, relating these two modes under certain well-defined conditions. Thus, if the proofs hold in a uniprogramming sense, and if the required conditions are met, the proofs hold in the multiprogramming environment of the operating system. The proof effort thus looks reasonable for proofs of the entire operating system implementation. Illustrative proofs of implementation correctness are now included.

Initial authorization and login protocols are discussed. The conformance of the initial authorization to the desired security of the system is also discussed.

Detailed designs are provided for various application areas. The secure document manager has been extended, and proofs are given to show that the specifications satisfy the desired multilevel security properties. A design is given for the confined subsystem manager. Part of a design for a data management system is also given.

Monitoring is discussed in greater detail than before. Specific examples are given to show how monitoring would be carried out, and what its power would be. It is also demonstrated that monitoring does not compromise system security.

INCREMENTAL PREFACE FROM THE FIRST EDITION (INCLUDED FOR HISTORICAL PERSPECTIVE)

For readers of the original PSOS report, dated 13 June 1975, this incremental preface provides a brief indication of the differences between the two reports. The entire report is newly written, and considerable new material has been added.

The methodology has been extended and sharpened. The specification language, now called SPECIAL (SPECIfication and Assertion Language), has been modified to simplify readability and proof. It is defined in terms of a formally stated syntax. An on-line environment has been developed to support the checking of the consistency of the hierarchical design, the specifications and the mappings between levels. These tools aid in the design and the proofs. Contemplated tools to aid in the implementations and their proofs are also outlined.

The operating system has been expanded to include input-output. The ordering of levels has been modified slightly, with linkage moved above the user process level. Revocable capabilities have been eliminated, at least insofar as they must be supported by the lowest levels of the system (including the hardware). They may still exist at a higher level, and are in fact found in the visible i-o level in a somewhat different form. A new kind of capability has been added, called a store-limited capability. Such capabilities may be limited to the process in which they are created (or just to its stack), cannot be transferred further, and disappear altogether on process termination (or stack pop). These capabilities are useful in implementing some of the applications indicated below.

All of the specifications have been subjected to the checking of the on-line environment noted above. Thus they are essentially free of syntactic errors.

The hardware required to support this design has been studied in detail, including support for the store-limited capabilities. (The elimination of revocable capabilities simplifies both the hardware and the software.) Specific hardware design considerations are included.

System initialization, backup, and recovery are treated in greater detail than before.

LIST OF TABLES

LIST OF FIGURES

INDEX OF TERMS

TECHNICAL SUMMARY OF THIS REPORT

INTRODUCTION

This report describes work on the design of a general-purpose computer operating system intended to meet advanced security requirements. The design is formally specified such that properties about the security of the system can be formally proved. The design is also amenable to efficient implementation and to efficient use. Discussed here are:

* a formal methodology (HDM) for the hierarchical design and implementation of computer operating systems and applications subsystems, and for the formal proof of properties of such systems, with respect to both the design and the implementation;

* the design of a secure capability-based operating system (PSOS) designed according to this methodology to meet advanced security requirements, together with relevant implementation considerations;

* the design of several application subsystems for this operating system, including support for multilevel security, for confined subsystems, for a secure relational data management system, and for monitoring of security;

* statement and informal proof of properties of the design for the operating system and for certain application subsystems, and the consideration of proofs of implementation correctness;

* consideration of the system design with respect to distributed systems and networks;

* an evaluation of the significance of this work, and considerations for future development of secure systems and subsystems.

Future work remains to be done in several areas to demonstrate the feasibility of efficiently implementing the design of the operating system and its applications. Such areas include formal proofs of design properties, further support for a suitable implementation language, the implementation of the existing specifications, and formal proofs of implementation correctness.

THE METHODOLOGY

The Hierarchical Development Methodology (HDM), initially developed and first used for the design of the operating system, is intended to be generally useful throughout all stages of system development, including design, implementation, debugging, integration, and verification of the system. The methodology integrates these stages by providing both a unified language for specifications and assertions (SPECIAL) and a hierarchical structure

for the design that is reflected in the implementation and in the proofs. The methodology provides two dimensions of structure: a staged decomposition of the system development effort, and a hierarchical decomposition of the system design.

The first structural dimension partitions the design and implementation into five stages, progressing in roughly sequential order (with appropriate iteration) from a loosely defined system concept, to a completely specified design, to a completely implemented system. This staging encourages decisions to be made as they are needed, in a reasonable order, and at a time appropriate to the development.

The second structural dimension entails a hierarchical decomposition of the design into different modules at different levels. The functions at any one level depend for their implementation exclusively on functions of lower levels or of the same level. The specification of each level is self-contained; i.e., it is independent of the specifications of other levels. It is also relatively independent of the implementation. The hierarchical decomposition contributes notably to the understandability and manageability of the system development. It also contributes to system reliability and recovery, initialization, and monitoring, in that each of these tasks is distributed in a hierarchical manner reflecting the design structure. Although hierarchical structures have been used previously, this work is the first attempt to formalize such structure on a large scale and to take advantage of it in related proof efforts.

Associated with each stage are properties that can be stated and proved, first about the design and then about the implementation. The proofs at successive stages provide increasing confidence in the suitability of the ultimate system. The approach is also suitable for handling incremental changes to the design, the implementation, and the proofs. Changes at some stage affect only that and later stages; proofs need be reconsidered only if they apply to those stages. Thus a change in implementation that does not alter the specifications does not affect the validity of the design proofs.

THE SYSTEM DESIGN

The methodology mentioned above has been applied to the design of PSOS. "PSOS" might be considered an acronym for a "Potentially Secure Operating System", in that PSOS has been carefully designed in such a way that it might someday have both its design and its implementation subjected to rigorous proof; considerable effort has gone into trying to enhance that possibility. PSOS is generally called a "Provably Secure Operating System", although that name is really quite presumptuous and speculative until such time as the proofs have actually been carried out. However, whenever that does happen, "PSOS" can easily be made to stand for "Proved Secure Operating System". (It might be noted that we sometimes get asked about our "Probably Secure Operating System".)

The PSOS concept is based on various data and procedure abstractions available to users of PSOS that include the software and the hardware. These include processes and virtual memory objects (segments), as well as directories that provide catalogues for symbolically named objects. They also include extended-type objects of types that may be user-created. All objects of a particular type are created, maintained, and deleted by a collection of programs known as the type manager for that type. An extended-type manager coordinates the creation of new types.

In the operating system, all objects are accessed by means of capabilities. A capability is a protected piece of data that refers to a particular object. Each capability is protected in the sense that it can be created only by the system, and cannot be forged or modified. Each capability contains protection information (access rights) that indicates how the corresponding object may be used with that capability. In addition, capabilities may be constrained so that their movement is restricted. Such capabilities are said to be store-limited. The levels of abstraction provided by various hierarchical levels within the operating system are summarized in Table 0.1, while a generic decomposition of these levels is given in Table 0.2. The lower levels would normally be implemented in hardware (or microcode). For historical purposes, the earlier decomposition of 1977 is given in Table 0.3.

APPLICATIONS OF THE OPERATING SYSTEM

An important potential application for the operating system is in support of particular security policies. One such policy is considered here as an illustrative example, in which multilevel security requirements are typified by TOP SECRET, SECRET, CONFIDENTIAL, and UNCLASSIFIED levels of classification, along with rules as to when reading and writing is permitted. The extension of the design to support such an application of the operating system is given. This illustration is not meant to justify the particular policy, which is in fact incomplete. However, it demonstrates the power of PSOS.

A subsystem to support confined execution is also given. This subsystem provides an environment in which it can be guaranteed that certain types of information leakage can be avoided.

The design framework and part of the design of a relational data management system are also considered. Here the emphasis is not on the design of a data-base management system, but rather on the demonstration that the methodology is applicable to such subsystems.

Another application area considered is the monitoring of both system performance and system security. The way in which this can be done without introducing new security violations is indicated.

The final application area considered is that of distributed systems. First the notion of implementing PSOS as a distributed system is considered. Then the notion of designing a network of

operating systems such as PSOS is considered. Various design alternatives are considered.

SYSTEM SECURITY

The desired security of the system and of its application subsystems must be formally stated in terms of the methodology, in order that security be formally provable. Such statements and their proofs are facilitated by the use of capabilities in the operating system, as is the security of the system itself.

A first step in the direction of design proofs is indicated here. The two main properties that are desired of PSOS relate to authorized reading and authorized writing of information. In simple terms, these properties are summarized as follows. Each is formally stated and proved in the report.

DETECTION PRINCIPLE: There shall be no unauthorized reading of information.

ALTERATION PRINCIPLE: There shall be no unauthorized writing of information.

For the multilevel security classification environment, corresponding properties have been formulated by Bell and LaPadula (the simple security condition and the so-called "*-property"), and these are used here. A more recent formulation that has been extremely useful in obtaining automatic proofs of multilevel security is given in Feiertag [80].

Proofs of these properties, appropriately formulated in terms of the methodology, require the demonstration of the consistency of the precise formulations with the specifications. These proofs are presented informally. More rigorous proofs have been carried out for other systems, and are indicated.

Additional properties relating to system security concern denial of service and leakage of information through channels other than reading and writing (e.g., inferences based on system performance). These issues are discussed, but are by no means resolved.

The security of any implementation ultimately depends on the correctness of the implementation, i.e., its consistency with the specifications. Significant advances are described here that will aid in such proofs.

CONCLUSIONS

The work described here demonstrates the feasibility of designing a general-purpose operating system to meet advanced security requirements, and proving significant properties about such a design. It also indicates that the design can be efficiently implemented. However, much work remains in order to achieve such an implementation and to carry out the proofs.

Table 0.1
PSOS ABSTRACTION HIERARCHY

Level	PSOS Abstraction or Function
16	user request interpreter *
15	user environments and name spaces *
14	user input-output *
13	procedure records *
12	user processes * and visible input-output *
11	creation and deletion of user objects *
10	directories (*)[c11]
9	extended types (*)[c11]
8	segments and windows (*)[c11]
7	paging [8]
6	system processes and input-output [12]
5	primitive input/output [6]
4	arithmetic and other basic operations *
3	clocks [6]
2	interrupts [6]
1	registers (*) and addressable memory [7]
0	capabilities *

* = module functions visible at user interface.
(*) = module partially visible at user interface.
[i] = module hidden by level i.
[c11] = creation/deletion hidden by level 11.

Table 0.2
PSOS GENERIC HIERARCHY

Generic Level	Generic Abstraction in PSOS	PSOS Level
F	user abstractions	14-16
E	community abstractions	10-13
D	abstract object manager	9
C	virtual resources	6-8
B	physical resources	1-5
A	capabilities	0

Table 0.3
SUMMARY OF THE EARLIER PSOS DESIGN (1977)

Level	Abstraction or Function
13	user request interpreter
12	user environments and user name spaces
11	user input-output
10	procedure records
9	user processes (scheduling, interprocess communication, synchronization), visible i-o
8	user objects
7	directories
6	extended types and extended-type objects
5	segmentation
4	paging
3	system processes and system input-output
2	arithmetic operations
1	addressable memory and primitive input-output
0	capabilities, interrupts, registers, clock

PART I
THE APPROACH

CHAPTER I-1
OVERVIEW OF THE REPORT

1. INTRODUCTION

The Computer Science Laboratory at SRI has been engaged from April 1973 to February 1977, with some recent effort as well, in the design of a general-purpose computer operating system called PSOS, for Provably Secure Operating System. It was desired from the outset that properties of PSOS could be formally proved about both its formally stated design and its implementation. The approach described here relies heavily on the use of a formal methodology that encompasses design, implementation, and proof. The methodology developed for this purpose is called HDM, for Hierarchical Development Methodology. HDM uses a hierarchical decomposition of the design, with formally stated specifications for each system function and formal assertions of each desired system property. The desired design properties to be proved relate to the security of the operating system and certain of its applications.

HDM is also being applied to the design of several other systems, including two versions of KSOS, a UNIX(@TM Bell Laboratories)-compatible system based on a security kernel, and an ultrareliable computing system for commercial aircraft (see Wensley et al. [76,78]). In the former case, the specifications are being proved consistent with a formal model of multilevel security similar to that described by Bell and LaPadula [74] and by Millen [75]. In the latter case, properties are being proved about the system fault-tolerance (e.g., stability of recovery and reconfiguration) with respect to the design. The methodology has also been used earlier by SRI for the design of a family of real-time systems (with security as one option) (Feiertag et al. [79a]) and by Honeywell and SRI for the design of a proposed kernel retrofit for Multics.

The design of PSOS is described here. This design is essentially complete for the operating system, and is intended to be illustrative for several application subsystems designed for use with PSOS. The multilevel security requirements are supported by one of these applications, for which proofs of the desired properties have been carried out. Also, security properties have been proved informally for the specifications of the functions of the visible interface to PSOS itself. These properties demonstrate that there can be no "unauthorized" modification or acquisition of information by use of the operating system functions (for a particular useful definition of "authorization").

A language for writing specifications and assertions has been developed in accordance with the methodology. This language is called SPECIAL (SPECIfication and Assertion Language), and is described in Appendix A. (See also Robinson et al. [76], Roubine et al. [76]). In addition, on-line tools have been developed to support the use of this language. These tools are intended to simplify the

overall development and proof effort. They contribute to the design by providing an on-line editable form for specifications, with automated checks of syntactic consistency. These tools also contribute to the proofs of security of the design.

To support the proof of the multilevel security properties for the multilevel security environment for PSOS (and for the Multics kernel work), the Bell and LaPadula model has been restated in terms of the concepts of the SRI methodology. This restatement is also presented, providing the basis for proof of the correspondence between the model and the specifications. A more recent formulation is found in Feiertag [80].

Tools have also been and are continuing to be developed for stating and proving semantic properties of programs. These tools are designed to be compatible with the tools mentioned above. As more of these verification tools become available, semiautomatic proofs of implementation correctness will become more feasible. Some preliminary experience is cited in Chapter V-1, along with further details on some of the other work mentioned above.

This remainder of this chapter provides an overview of the work reported in this report. It is written as a reasonably self-contained document, and may be read to any desired depth on the first reading of the report. Subsequently, this chapter may be used as a detailed summary description of the work. In the remainder of this chapter, the methodology is summarized, first with respect to design and implementation, and then with respect to verification. The desired security properties are then summarized. In addition, the desired properties of the Bell and LaPadula model are given. Properties of SPECIAL related to security are summarized, and the desired security properties are explicitly restated using the concepts of SPECIAL. After the design of the operating system is outlined, the correspondence proofs between the desired properties and the specifications are discussed. Tools to support the verification effort are also discussed, as are several considerations involved in implementing the system. Chapter I-2, which follows, presents the methodology in greater detail. SPECIAL is described in Appendix A.

Part II discusses the design of the operating system (specified in Appendix B) and some of its implementation issues. Part III considers the proofs of security for the design (Chapter III-1) and also outlines the effort required to carry out proofs of implementation consistency (Chapter III-2). Specific implementations and proofs of implementation correctness are given in Appendix C. Part IV describes the design of several application subsystems suitable for implementation using the operating system, and gives proofs for security properties for some of these applications. In particular, Chapter IV-1 discusses the design of the secure object manager supporting multilevel security (with specifications in Appendix D). Chapter IV-2 discusses the confined subsystem manager (specified in Appendix E). Chapter IV-3 considers the design of a relational data-base system (specified in part in Appendix F). Chapters IV-4 examines the monitoring of security. Chapter IV-5

considers the distributed system and networking implications of PSOS. Chapter V-1 provides more background on related work using HDM, particulaly those efforts concerned with security. Finally Chapter V-2 presents conclusions resulting from this work.

2. THE METHODOLOGY FOR DESIGN AND IMPLEMENTATION

The methodology is summarized here. More complete documentation is found in the three-volume HDM Handbook (Robinson [79], Silverberg et al. [79], and Levitt et al. [79]). Earlier documents are also relevant (Robinson et al. [75] and Robinson and Levitt [75]). The methodology as described in this report separates the development of a computer system or subsystem into stages corresponding to

(S0) the choice of the visible interface,
(S1) the hierarchical design,
(S2) the specification of each function at each node of the hierarchy,
(S3) the definition of mappings among the data representations at connecting nodes, and
(S4) the writing of implementation programs for the functions at each node.

These stages of design and implementation are described below.

(S0) INTERFACE DEFINITION

In the initial stage (S0), the desired interface is defined that is visible to the callers of the system or subsystem in question. This interface is then decomposed into a set of MODULES (i.e., a set of facilities), each of which manages OBJECTS of a particular type. An object is a system resource such as a segment, a directory, or a process. Each module consists of a collection of FUNCTIONS (corresponding to operations and data-structure accesses). Each function has an argument list and can be invoked either by a program or directly by a user. Each function is either an O-function (Operation) that changes the state of the module to which it belongs, a V-function (Value-returning) that characterizes the state of the module, or an OV-function that both changes the state and returns a value.

(S1) HIERARCHICAL DECOMPOSITION OF THE SYSTEM

The modules of the visible interface, together with other modules whose functions are hidden by the interface but are part of the eventual implementation, are arranged into a hierarchy of collections of modules. For descriptive simplicity, it is assumed here that there is only one visible interface, and so it can also be assumed that the hierarchy is a linear ordering of these module collections, each of which can then be referred to as a LEVEL. (For all cases considered here, this simplified description is applicable.) The implementation of each level depends only on the behavior of the next lower level. However, a module may be included in more than one level of the design -- e.g., the module supporting the "user" hardware instructions, which would be part of most levels in a

typical operating system. The structure of the decomposition is thus explicitly declared at this stage.

(S2) MODULE SPECIFICATION

For each module, a FORMAL SPECIFICATION is developed. In this methodology, specifications are used that are similar to those suggested by Parnas [72a,b,c,d]. However, Parnas' original approach has been extended substantially here, in that the specification language (SPECIAL) and the hierarchical structure have been formalized and supported by an on-line environment. V-functions of a module are either PRIMITIVE (necessary for characterizing the state of the module) or DERIVED (computed from the values of primitive V-functions). Some V-functions are VISIBLE at the interface to a module (i.e., can be called by programs), while others are HIDDEN. The specification of each O- or OV-function precisely describes the effect of that operation as a state change. The state change is defined by a set of EFFECTS, each of which relates values of primitive V-functions before the call on the specified function to values of primitive V-functions after the return from that call. The specification of each V-function gives either the INITIAL VALUE of the function (if it is primitive) or its DERIVATION from primitive values (if it is derived). The specification for each visible function also gives EXCEPTION CONDITIONS for which a call on that function is to be rejected. Specifications are written independently of decisions that concern only the implementation of the module. For a well-conceived module, the specification is much easier to understand than its implementing program (see (S4)).

(S3) MAPPING FUNCTIONS

For each module above the lowest level of the design, a MAPPING FUNCTION is written that characterizes the state of the module in terms of the states of lower-level modules. A mapping function is written as a set of expansion rules, in each of which a higher-level V-function value is expanded as an expression containing lower-level V-function values. These expansion rules, called MAPPING FUNCTION EXPRESSIONS, are also written in SPECIAL. In this way, assumptions are explicitly stated as to how the data structures of a module are represented in terms of lower-level modules. For example, a mapping function would relate the data structure of a user process to that of a system process, or a segment to a sequence of pages.

(S4) IMPLEMENTATION

Programs are then written to implement the visible functions of each level (except for the lowest level) in terms of those at the next lower level. Such programs are called ABSTRACT PROGRAMS since they call functions that implement abstract operations specified at lower levels of the system. The abstract programs may be directly compiled into executable code.

Stages 0, 1, 2, 3, and 4 have formalized the following five conventional steps in software development, respectively: interface definition and decomposition; system modularization; specification;

data representation; and coding. The results of Stages 0, 1 and 2 are considered to constitute the DESIGN; those of Stages 3 and 4 constitute the IMPLEMENTATION. In the methodology referred to here, Stages 2, 3, and 4 are carried out for each level in the hierarchy.

3. THE METHODOLOGY FOR VERIFICATION

The stages of development provide the basis for the verification effort. Associated with each of these stages of design and implementation is the statement or verification of correctness properties appropriate for that stage, for each level in the hierarchical design.

(S0') INTERFACE PROPERTIES

At Stage 0, the desired properties of the system can be explicitly stated. The desired set of such properties for the operating system is given by two principles regarding the prevention of unauthorized reading and writing. The set of properties for the environment supporting multilevel security consists of the *-property and the simple security condition of Bell and LaPadula [74], along with their duals for integrity. These properties are discussed subsequently.

(S1') HIERARCHY PROPERTIES

At Stage 1, the consistency of the hierarchical structure and of the naming of functions can be demonstrated.

(S2') SPECIFICATION PROPERTIES

At Stage 2, the desired properties can be proved about the design (i.e., about the specifications of the visible interface), independent of subsequent implementations. For the purposes of this report, SYNTACTIC properties are those that are algorithmically checkable, while SEMANTIC properties need a formal proof procedure -- in general, undecidable -- to establish them. Proofs are based on

* syntactic properties of SPECIAL;

* syntactic properties not in SPECIAL but required of all specifications in the system under consideration;

* semantic properties of the specifications.

The first class of syntactic properties is called INTRINSIC, while the second is called EXTRINSIC. In addition, each module specification can be shown to be self-consistent (i.e., satisfiable) at this stage.

(S3') MAPPING PROPERTIES

At Stage 3, it can be proved that the mapping functions are consistent with the specifications and the hierarchical decomposition. An inconsistent mapping function is one in which two distinct states at a higher level can both correspond to a single

state at a lower level. Consistency of the mapping functions with the outputs of the previous stages is demonstrated similarly to those for self-consistency of the specifications.

(S4') IMPLEMENTATION PROPERTIES

At Stage 4, the implementation can be proved consistent with the specifications and mapping functions resulting from the previous stages. Proofs of implementation consistency are done level by level. A given program intended to implement a visible V-, O-, or OV-function is proved correct with respect to its specifications, the specifications of the modules that implement the program, and the mapping function expressions relating these modules.

Once a verified system is obtained, it will tend to evolve with time. Changes in specifications and in implementations require corresponding reverification. However, reverification is required only where changes in specifications and implementations have affected the validity of the earlier verification. The staged application of this methodology and the formally defined modular decomposition can considerably simplify the reverification effort.

On the basis of its applications to date, the staged development appears to give successively greater confidence in the resulting systems, from one stage to the next, first in terms of the suitability of the design and then in terms of the correctness of its implementation. Subtle design bugs have been discovered relatively easily in attempting proofs. Significant savings in development costs can result from detection of inherent insecurity in the design or implementation as early as possible.

4. SECURITY OF THE OPERATING SYSTEM

The desired security of the operating system is described by two basic principles that must be satisfied by all user-visible functions of the operating system.

> DETECTION PRINCIPLE: There shall be no unauthorized acquisition of information.
>
> ALTERATION PRINCIPLE: There shall be no unauthorized alteration of information.

The precise notion of authorization relates to the use of capabilities in the operating system. In essence, authorization to access a particular piece of information requires the possession of a suitable capability for that information. The creation and passage of capabilities is controlled by the operating system.

Two additional principles are related to security in a general-purpose operating system.

> THE GUARANTEED SERVICE PRINCIPLE: There shall be no unauthorized denial of service.

THE CONFINEMENT PRINCIPLE: There shall be no leakage of information.

The guaranteed service principle states that users should never be denied access to a resource to which they are entitled, for example, as the result of an unfair scheduling algorithm. Intuitively, a denial of service may be a security violation in the sense that it can compromise a user's ability to perform an authorized task. Whereas the alteration and detection principles are satisfied by the operating system design, the guaranteed service principle is only partially satisfied.

The confinement principle concerns information that may be inferred from the behavior of the system, rather than read directly. Such a leakage of information may occur from the system to a user, or from one user to another. Although certain forms of leakage can be characterized and controlled, this is not possible in general. Nevertheless, a confined environment is provided for users of PSOS, in which a well-defined sense of confinement can be ensured.

5. THE BASIC MULTILEVEL SECURITY MODEL

The security model of Bell and LaPadula [74] is considered next. For security, each object (i.e., operating system resource such as a segment or process) being written into or read from has a classification level and a category set, collectively referred to as the OBJECT SECURITY LEVEL. Also, each user has a clearance level and a category set, collectively referred to as the USER SECURITY LEVEL. Clearance and classification levels are linearly ordered (e.g., TOP SECRET, SECRET); category sets are partially ordered. One security level is AT LEAST that of another if and only if its classification or clearance level is at least that of the other and its category set contains the category set of the other. Similarly, for integrity, each object or user has its own integrity level, and partial ordering is defined as for security levels. The ordered pair consisting of the security level and the integrity level is called the ACCESS LEVEL. (To avoid confusion, each such level is always identified by an adjective. The term "level" used by itself refers to a collection of modules of the hierarchical design.)

The Bell and LaPadula model is expressed approximately as follows.

SECURITY CONDITIONS:

The *-property for security: Writing is permitted only into an object with AT LEAST the user's security level. That is, there is no writing downward in security level.

The simple security condition: Reading is permitted only from an object with AT MOST the user's security level. That is, there is no reading upward in security level.

Note that writing up is not considered to be insecure, but is nevertheless often undesirable. For example, overwriting existing

information at a higher security level could be very damaging. Thus writing up is also forbidden in many cases.

The desired integrity conditions are formally the duals of these two security conditions, as follows.

INTEGRITY CONDITIONS:

The *-property for integrity: Writing is permitted only into an object with AT MOST the user's integrity level. That is, there is no writing upward in integrity level.

The simple integrity condition: Reading is permitted only from an object with AT LEAST the user's integrity level. That is, there is no reading downward in integrity level.

Appropriate use of the security and integrity levels enables the system security officer to put strict limits on undesirable reading and writing.

To prove that these four properties hold with respect to formal specifications, it is desirable to restate them in terms of the specification and assertion language, SPECIAL, whose properties are discussed next. Following that, the restatement of the desired security properties in the form to be proved is given.

6. SPECIFICATION LANGUAGE PROPERTIES RELATED TO SECURITY

SPECIAL is a formal, nonprocedural specification language. It permits each function of a module to be specified independent of its implementation. Thus, properties of the design may be stated (in SPECIAL) in an implementation-independent manner.

In SPECIAL, the effects of an O- or OV-function of a module of some design level are defined in terms of the new values of the primitive V-functions of that level as related to the old values of those V-functions, to the arguments to the specified function, and to the parameters of the modules of the given level. (A PARAMETER of a module is a symbolic constant that is fixed for each particular instance of that module, as for example the maximum size of a segment.) Similarly, the value of a derived V-function or an exception condition is defined in terms of the values of the primitive V-functions of the level, the arguments of the specified derived V-function, and parameters of the level. The initial values of primitive V-functions are defined in terms of the module parameters and the arguments of the function. The following definitions are useful.

A primitive V-function value is CITED by the specified function if and only if it appears as an old value in either an effect, an exception, or a derivation.

A primitive V-function value is MODIFIED by the specified function if and only if it appears as a new value in an effect.

SPECIAL requires as an intrinsic syntactic property that all V-function values cited or modified in any module specifications must be values of V-functions of the design level to which the given module belongs. In addition, the PSOS specifications require that all modified values must be values of V-functions of the same module. This is an extrinsic syntactic property. Finally, semantic properties are required of each cited or modified V-function value to ensure that all of the arguments and cited values are legitimate, and to assure that all modified values are dependent on only legitimate values according to the following definitions. These definitions are applicable to each specified function.

> A READ REFERENCE is a cited V-function value, or a parameter of the level of the specified function, or an argument to the call on that function.
>
> A WRITE REFERENCE is a modified V-function value, or the value returned by a visible V-function or an OV-function, or the value of an exception condition.
>
> A read reference is LEGITIMATE if and only if it is the value of a V-function all of whose arguments are arguments of the specified function or parameters of the module or legitimate read references.
>
> A write reference is DEPENDENT on a read reference in a specification if and only if two different legitimate values exist for the read reference that would cause the write reference to assume correspondingly different values.

It should be noted that exception conditions are included in the definition of a write reference because the presence or absence of an exception condition can itself result in information transfer. The occurrence of an exception condition can be viewed as a status value that is returned for each function call.

7. THE DESIGN OF THE PROVABLY SECURE OPERATING SYSTEM

The basis for security in PSOS is the use of capabilities to protect all user-visible objects. A CAPABILITY is a protected token for an object and is interpreted by the operating system and the hardware as an address for the object. Each capability also contains authorization information defining how the object may be used, subject to the rules for objects of the particular type. A capability is itself protected in that it is nonalterable and nonforgeable.

The design of the operating system is decomposed hierarchically into various levels. The design is presented in detail in Part II, with specifications given in Appendix B. The levels of the design are summarized below, in order of decreasing abstraction (increasing hardware dependence).

```
user environments *
user input-output *
```

procedure records *
user processes *
user objects *
directories *
extended types *
virtual memory (segments and windows) *
paging
system processes and input-output
basic operations (e.g., arithmetic)
real memory
capabilities * and interrrupts

Each of these levels provides an abstraction useful in the implementation of the operating system. Some of these abstractions (denoted by "*") are visible at the interface to the operating system (except for the creation and deletion of objects). Note that interfaces at a higher level than the above can hide some of these abstractions from view -- e.g., capabilities, new types, or new processes, as desired.

8. A DESIGN TO SUPPORT THE MULTILEVEL ACCESS PROPERTIES

Security levels and integrity levels are associated with every object visible at the interface about which the desired properties are to be proved. In the case of the subsystem supporting secure objects for PSOS, the visible interface consists of five modules, as follows.

secure processes
secure directories
secure extended types
secure segments
secure capabilities

These five levels mimic five modules of PSOS noted above (namely processes, directories, extended types, segments, and capabilities) that do not have security and integrity levels associated with them. The design is considered in Chapter IV-1. In implementation, each of these modules uses only lower-level modules, with secure processes being the highest and secure capabilities the lowest. (Note that if every system application were to require the multilevel security properties, these five modules could completely replace the corresponding five operating system levels.)

9. FORMULATION OF THE SECURITY PROPERTIES IN TERMS OF THE METHODOLOGY AND THE SPECIFICATIONS

The various security properties stated above must be formally stated in terms of the concepts of the specification language to be formally proved. Although inadequate detail is available to the reader at this point in order to fully comprehend the formality, the following restatements of the above properties are given as an illustration of the power of the methodology and the basic simplicity of the statements to be proved. For the basic operating system, the

alteration and detection principles may be stated as follows, based on the notion of the accessibility of capabilities.

> ITERATIVE ACCESSIBILITY: A capability is ITERATIVELY ACCESSIBLE if and only if it is the value of a read reference, with arguments that are themselves iteratively accessible.
>
> DETECTION PRINCIPLE: For the specification of any visible function, the capability value of any read reference must be iteratively accessible.
>
> ALTERATION PRINCIPLE: For the specification of any visible function, capability values assumed by write references of the specified function must be iteratively accessible.

These properties are satisfied by the specifications for the operating system given in this report. These properties relate the concept of authorized use with the mechanisms supporting capabilities in the operating system. In essence, no access to a particular piece of information is permitted without an appropriate capability for the information; furthermore, the acquisition of capabilities is controlled. These and additional properties of PSOS are discussed in Chapter III-1.

For multilevel security, in the specifications of each function of the visible interface, each possible primitive V-function value (i.e., for each given set of argument values) has an access level associated with it. This level is fixed. Similarly, each process able to call that interface has an access level. The arguments of all visible functions are at the same level as the caller (i,e., the process invoking the function) because they are supplied by the caller. The desired security conditions may then be expressed in terms of SPECIAL as follows.

> SECURITY CONDITIONS: In the specification of a visible function, the security level of each write reference must be
>
> (a) AT LEAST the security level of the caller, and
>
> (b) AT LEAST the security level of each read reference upon which the write reference is dependent.

Part (a) is precisely the *-property for security, while Part (b) is the desired restatement of the simple security condition. The duals for integrity are achieved by interchanging SECURITY and INTEGRITY and by interchanging AT LEAST and AT MOST, as follows.

> INTEGRITY CONDITIONS: In the specification of a visible function, the integrity level of each write reference must be
>
> (a) AT MOST the integrity level of the caller, and
>
> (b) AT MOST the integrity level of each read reference upon which the write reference is dependent.

These conditions are satisfiable by the specifications for the multilevel security interface for PSOS. Not shown is that the stated conditions imply conformance with the Bell and LaPadula model as stated here.

The correspondence between these properties and the specifications for a set of visible functions for some system guarantees that if the system was initially in a secure state, it will subsequently be in a secure state after the execution of a call on any of the functions of the visible interface. It remains, however, to show that the initial state is secure. Although checks on the consistency of the initial conditions can be made, it is noted that the security of the initial system (i.e., the assignment of levels to users and shared objects) is basically a policy matter.

The above multilevel access properties must then hold for the specifications of every O-, OV-, and V-function visible at the interface. That is, no calls of a visible O-, OV-, or V-function are defined except those satisfying the above security and integrity conditions. Instead of returning an exception on a prohibited call, every possible call follows the security rules. The proof technique is illustrated below, following a summary of a typical design to support the access properties.

A recent formulation that is more suitable for formal proof is given in Feiertag [80].

10. CORRESPONDENCE PROOFS BETWEEN SPECIFICATIONS AND THE SECURITY PROPERTIES

The proofs of the alteration and detection principles for PSOS as stated in Section I-1.9 require the demonstration that the specification of each user-visible function satisfies those properties. Part of this is ensured if the specifications satisfy the syntactic requirements of SPECIAL -- that is, if they are well-formed specifications. The remainder is ensured because of the close relationship between the capabilities of PSOS and the designators of SPECIAL. A designator is precisely a nonforgeable token for an object and is primitive in SPECIAL. In particular, the PSOS design requires that every piece of information in the system have one or more capabilities associated with it. Every function that references a piece of information must be presented the appropriate capabilities as arguments. Since there are no operations in the operating system that alter a capability or derive from it a capability for a different existing object, the alteration and detection principles can be readily enforced.

The proof procedures that demonstrate the multilevel security properties for the PSOS subsystem described above all entail very simple deductions. The proofs can be accomplished by deriving simple rules from the security properties. When applied to the specifications, these rules yield logical formulas whose validity implies that the design is secure. Similar proofs apply to proving the correspondence between the specifications for the proposed Multics security kernel and the desired security properties.

Another security-related subsystem for PSOS provides an environment in which processes may be constrained to operate in a confined manner (e.g., see Lipner [75]). This is the confined subsystems manager, discussed in Chapter IV-2. For this subsystem, it is possible to state and prove that such a confined process cannot transfer information out of the environment in which it must execute, at least not by "storage channels". (A storage channel is any communication that results from the use of V-function values, as opposed to a "time channel", which uses inferences about the times required for various programs.) However, at present these proofs have not been carried out. In addition, the statement and proof of security properties could be accomplished for the data management system discussed in Chapter IV-3 and the monitoring subsystem of Chapter IV-4, as well as for networks of operating systems (see Chapter IV-5).

In the same way, the approach can be applied to arbitrary applications. A further example of the potential applicability of this approach is provided by the access control lists of Multics, which are not a part of the proposed revised security kernel, but which could also be specified as a higher-level interface. Similarly, for PSOS access control lists may be formally specified, and their desired security properties modeled. The proof approach described here is then applicable to this additional software. In this way, an implemented policy can become a mechanism.

11. TOOLS TO SUPPORT THE DESIGN AND THE CORRESPONDENCE PROOFS

We have developed an on-line environment to support the first four stages of the methodology, i.e., the interface definition, the hierarchical decomposition, the specifications, and the mapping functions. This environment is also useful in performing the syntactic checks needed in the proofs of correspondence between the desired properties and the specifications. The design of this environment is open-ended and is expected to be extended to support implementations and proofs of implementations.

The environment currently runs on TENEX and is being implemented on Multics. It is directly applicable to both PSOS and the Multics security kernel, as well as other related efforts. The environment currently exists in three parts, as follows.

(P1) The HIERARCHY MANAGER permits the establishment of a hierarchy of collections of modules, and which is responsible for maintaining the design structure.

(P2) The SPECIFICATION ANALYZER determines if each module specification is syntactically correct. This part includes type checking.

(P3) The MAPPING FUNCTION ANALYZER determines if the mapping function expressions are syntactically correct and syntactically consistent with the specifications of the modules involved.

In addition to these existing tools, a fourth tool is desirable to

prove those cases involving semantic dependencies in the correspondence proofs.

(P4) The MODEL CONSISTENCY CHECKER performs extrinsic syntactic checks, and also generates logical formulas whose validity is equivalent to the satisfaction of the semantic conditions for consistency with the model. Such a tool exists for proving mechanically that every function in a set of specifications satisfies a multilevel security model. A similar tool is contemplated for the PSOS protection properties and for various security policies that particular PSOS type managers may be called upon to enforce.

The generation of the logical formulas is straightforward, and experience shows that proof of many of the formulas is trivial while others require theorem proving. Mechanical generation and proof are vastly preferable to this effort being done manually, in order to eliminate human error from the proof process.

12. IMPLEMENTATION CONSIDERATIONS

The design of PSOS has been examined from the point of view of achieving an efficient realization. For a prototype development, various existing hardware could support PSOS in a modest system configuration, particularly with th help of new microcode. For a commercial development, new hardware is preferable, although it may be possible to modify hardware currently in development. All of the hardware required for efficient implementation of PSOS can be found in various existing machines, but no single machine currently has all of the desired features. A small associative memory would have a dramatic effect comparable to that in Multics, for example. The system is scalable, in that it could exist on one processor or on a multiprocessor configuration, or as a distributed system. Furthermore, the processors could vary in capacity from minis to maxis.

In general, the choice of programming language is not critical for implementing specifications. Appropriate syntactic constraints on the language can contribute to the qualtity of the resulting software. However, if proofs of program correctness are desired, the choice of a programming language in which to write the implementation programs strongly influences the provability of these programs. The language must be well structured and should provide considerable intrinsic security -- e.g., via strong type checking and restrictive scope rules. It must relate well to the methodology. It should also simplify the task of program verification. It should include modern features such as those found in EUCLID, ADA, MODULA, ALPHARD, SIMULA, CLU, and GYPSY (such as module encapsulation, protection and data abstraction). However it should not be as unconstrained as either of the first two of those languages in their full power.

The problem of implementing a module that is shared by concurrent processes is important generally, as well as with respect to security. The SRI methodology includes a model of concurrent computation that makes it possible to state and prove that a shared

implementation is correct. In addition, synchronization conditions have been derived that enable a set of correct stand-alone programs to be automatically modified so that together they constitute a correct concurrent implementation. Alternatively, it is possible to check automatically whether the given synchronization conditions are met. Thus, programs can be verified in isolation. If the required synchronization conditions are satisfied, correctness in the real operating environment is immediately ensured, given correct hardware operation. (Unfortunately the view of concurrency that SPECIAL currently supports is overly restrictive. This provides an important topic for future research.) Issues related to implementation and proofs of implementation are discussed in Part II and in Chapter III-2, respectively.

13. TOOLS TO SUPPORT IMPLEMENTATION

In addition to the tools outlined above to support the design and the correspondence proofs, the following tools are also under development at SRI to support implementation and program verification.

(P5) The PROGRAM HANDLER determines for each program its syntactic correctness and its syntactic consistency with the specifications and mapping functions. Program handlers currently exist for a subset of Modula (for use with KSOS) and for a subset of Pascal (for SIFT). Both of these also provide a bridge to the verification system in that they perform translation necessary for interfacing with the verifier.

(P6) The DEVELOPMENT DATA-BASE MANAGER maintains a data base of the specifications, programs, and proofs in (P1), (P2), (P3), (P4), and (P5), and keeps track of which modules are specified, mapped, implemented, and verified.

Other tools that will support semiautomatic program verification are also being developed, and can be used as appropriate. These include various verification condition generators (including a meta-VCG suitable for use with several different programming languages), logical simplifiers, various program transformers (e.g., optimizers that preserve program equivalence), and other tools that would enhance program verification. A more general view of the goals of such an overall environment, including compatible tools for partial execution, partial simulation, debugging, testing and performance evaluation, is outlined in Neumann [74].

14. OUTLINE OF THIS REPORT

In the next chapter, the SRI methodology for the design, implementation, and proof of large programming systems is described. The specification language, SPECIAL, is described in Appendix A. Part II presents the design in detail, with specifications in Appendix B representing the results of the design part of stage 2. Chapter III-1 gives the properties of the specifications related to security, and outlines their proofs. This represents the verification part of stage 2. Chapter III-2 discusses implementation

and proofs of implementation. Illustrative mapping functions (stage 3) and abstract programs (stage 4) are given in Appendix C, along with illustrative proofs. Part IV then gives examples of the use of the methodology (and of PSOS) for various applications. Part V presents the conclusions of the work, which are now summarized briefly.

15. CONCLUSIONS

The operating system design described here has been developed according to a formal methodology intended to support design, implementation, and proof. Informal proofs of security properties of the specifications have been carried out, partly manually and partly with the help of on-line tools. Formal correspondence proofs are conceptually relatively simple, aided substantially by the syntax of the specification language and its specification analyzer, by the abstraction afforded by the specifications, and by the simplicity of the model. Experience with the automatic multilevel security design proofs in KSOS demonstrates the value of such proofs.

Considerable progress has been made toward proving the correctness of implementations. Verification systems now exist at SRI for meaningful subsets of Modula and Fortran, and one for Pascal is under development for SIFT. Much work remains to be done.

The methodology is also being applied in other projects at SRI. For example, it is being applied to the design proofs of KSOS, and to illustrative code proofs. It is also being applied to the development and proof of other secure systems and of systems in which security is not the major concern (as in Wensley et al. [76], in which fault-tolerance properties are to be proved).

A realistic assessment of the PSOS design presented here requires a prototype implementation of the system, possibly accompanied by proofs of the correctness of the implementation. Although preliminary arguments indicate that PSOS can be implemented efficiently, these arguments require demonstration. Similarly, the security properties depend on correct implementation. A prototype implementation study is just beginning.

A realistic assessment of the methodology requires such a development, including implementation and proof of implementation. The KSOS developments have shown the value of the methodology for design and the value of formal specifications as a medium for communication (e.g., between the two KSOS efforts) and for design proofs. A major conclusion of the PSOS work is that further development seems warranted. Another conclusion is that much research remains to be done, particularly in the areas of implementation and proofs of implementation.

CHAPTER I-2

THE METHODOLOGY FOR DESIGN, IMPLEMENTATION, AND PROOF

This section provides greater detail on the methodology outlined in Chapter I-1. Additional detail concerning the use of the methodology is found in the discussion of the design (Part II), the proofs (Part III) and the applications (Part IV). Further background on the methodology is given in Robinson and Levitt [75].

In recent years there has been increased recognition of the benefits of using programming methods that introduce several refinements (i.e., levels of abstraction) between the statement of the programming task and the runnable code. The stepwise refinement method (Dijkstra [72]) allows a programmer to elaborate his concept of a program in a sequence of refinements and to prove -- formally or informally -- the correctness of each refinement. From the sequence of verified refinements, it is then possible to deduce the correctness of the entire program.

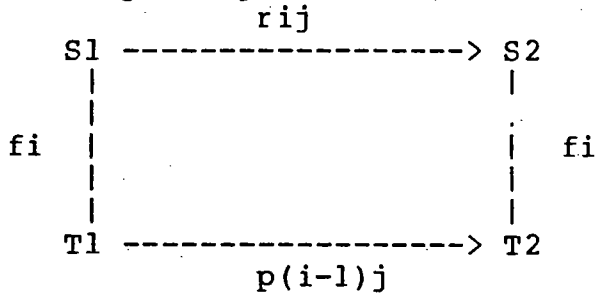
The methodology is based on a formalization of the refinement concept. Here a sequence of abstract machines (M0,M1,...,Mn) is defined, where M0 is the most primitive machine (possibly the hardware or a programming language). Each abstract machine consists of a state and a set of transformations for effecting state changes. If M0 is realized by hardware, the state is the register set and physical memory, and the transformations are the machine instructions. If M0 is represented by a programming language, then the state is the set of program variables, and the transformations are the statements in which assignment to variables takes place.

An abstract program is a program (written in an abstract programming language) that can be run on an abstract machine. Every abstract machine has an abstract machine interpreter that can execute an abstract program on that abstract machine. Thus the semantics of the abstract programming language are defined by the actions of the interpreter. The issues concerning abstract programmming languages and their interpreters are omitted from this paper. The formal definition or specification of the behavior of an abstract machine contains the only assumptions that an abstract program (or its writer) may make about an abstract machine. However, each non-primitive abstract machine in the hierarchy (Mi, $0<i\leq n$) is implemented in terms of its descendant (Mi-1). In the terminology of Parnas [74], abstract machines are the elements of the hierarchy, and the "implements" relation is the defining relation of the hierarchy. The "implements" relation has the following properties:

(1) The state of Mi is derivable from the state of Mi-1 by a mapping fi.

(2) Each transformation rij of Mi is carried out by the execution of an abstract program p(i-1)j running on Mi-1.

The mapping and the abstract program must be consistent with the specifications of Mi and Mi-1, in order for the implementation to satisfy the specification. Thus the following diagram must commute, where S1 and S2 are states of Mi, T1 and T2 are states of Mi-1, and rij and p(i-1)j are as above:



This first view of a hierarchy is restrictive in the sense that it is a linear ordering, rather than a tree or a directed acyclic graph. Thus each abstract machine occupies an entire level. In this view, a level is implemented only in terms of the level immediately below it (see Figure I-2.1a). Since each non-primitive level i (i>0) of such a system adds features to those available at level i-1, level i is different from -- but contains many of the features of -- level i-1. On the other hand, there may be some features that are available at level i-1, but not at level i. When a feature of level i-1 (i>0) is visible above level i, we say that the feature is transmitted by level i (or that level i is transparent with respect to the feature). When a feature of level i-1 (i>0) is not visible above level i, we say that the feature is hidden by level i (or that level i hides the feature). However, for purposes of description, it is important to look at several other views of this same hierarchy, views in which an abstract machine may not completely define a level.

In this second view, an abstract machine may be divided into one or more component abstract machines called modules. There are many possible modularizations of an abstract machine. However, we shall be concerned here with a particular modularization schema, having the following properties:

(1) At level i (i>0), machine Mi (of Figure I-2.1a) consists of module Mii (the defined module at level i), and at most i other modules Mij ($0 \leq j < i$), where Mij is equivalent to Mjj. Thus Mii represents the set of features that level i adds to the system, and Mij represents the set of features that are transmitted to level i from machines at lower levels (see Figure I-2.1b).

(2) A module may be absent from the definition of a level's abstract machine (see M31 of Figure I-2.1b). This absence represents the features that level i hides from the levels above it. Thus if a module Mij is absent, the module Mkj must also be absent, for all k, k>i. This rule is needed in order to be consistent with the first view of a hierarchy, that a level's features must be either defined by that level or transmitted from lower levels.

It is also possible to regulate transparency on the basis of individual functions of a module, rather than an entire module. If any of a module's features is visible at a level, then statements can be made about the module's behavior at that level. Thus the entire module can be treated as visible at that level (even if some of its functions are invisible there). The highest-level module, according to this view, defines the user interface of the system.

It is desired to provide modularization within a level, to avoid redundant module definitions, and (at the same time) to retain the rigor of the first two views of the hierarchy. Thus, a third view is adopted, in which only a level's defined module (in the second view) is explicitly described. The new diagram of the same hierarchy appears in Figure I-2.1c. Here module Mi' is equivalent to Mii of Figure I-2.1b, for all i. The specifications of the example will be presented in this way, but the system behaves in the same way as the equivalent hierarchies of Figures 1a and 1b.

Each module is formally defined using a notation derived from Parnas [72a, 72b]. This notation allows the compact description of a module having a large (possibly infinite) number of states. A module's state and transformations are available to its environment by means of functions that can be called by abstract programs. Following the Parnas notation, the state information is available via the outputs of V-functions (value-returning). A module's state is changed by calling a member of the set of transformations or O-functions (operation). In certain cases the state information is available as the result of a transformation. In this case the function is said to be an OV-function.

Each V-function's specification contains a definition of its initial value, and a delineation of the conditions (i.e. exception conditions) under which it may not be successfully called. The specification of an O-function or an OV-function contains exception conditions, and a description of the effects of a call to the function. If an exception condition is satisfied, no value is returned in the case of a V- or OV-function, and no effects are executed in the case of an O- or OV-function. In both cases control is returned to the calling program, with special notification that an exception condition

has been satisfied.

Initial values, exception conditions, and effects are expressed in terms of assertions. For initial values, the assertions are written in terms of module constants. For exception conditions, the assertions are written in terms of constants and the module's V-function values (describing the module's state before the call). For effects, the assertions are written in terms of constants and two sets of V-function values for the module: those before the call (in single quotes) and those after the call (unquoted). The assertions for the effects must be true after the call is completed. Thus, the effects describe a relation between the two states that must be satisfied upon exit from the O-function.

The progression from the initial design formulation to the specification and implementation of the system can proceed in five stages. Except for the first stage, some amount of formal proof can take place at each stage. The stages are intended to represent the desired protocol for constructing a hierarchically structured software system, but they may overlap somewhat -- with possible backtracking when design changes must be made. However, the most important fact is that, when completed, the design and implementation will be visible in distinct stages as follows.

Stage 0--The initial stage involves the definition of the desired interface visible to callers of the system or subsystem being developed. This interface is then decomposed into a set of modules, each of which manages objects of a particular type. (An object is a system resource such as a segment, a directory, a process, or a data base.) Each module consists of a collection of functions (corresponding to operations and data-structure accesses). Each function has an argument list and can be invoked either by a program, or directly by a user. Each function is either an O-function, an OV-function, or a V-function.

Stage 1--Here all of the above modules, together with others whose functions are hidden by the interface but are to be part of the eventual implementation are placed in a hierarchical ordering. For each module, we list the O- and V-functions that it will contain, and decide at which higher levels (if any) each of these functions will be visible.

Stage 2--Each module is formally specified as described above. Based on the specifications for a module, it is possible to attempt to prove that the specifications are self-consistent and that certain global assertions are true. A proof of self-consistency involves demonstrating that, for each O-function, no set of assertions in the effects is self-contradictory over the domain of definition for the V-functions. In general, an inconsistent module specification prevents a proof of the implementation of the module from being successful, and thus a proof of inconsistency need not be

explicitly carried out. However, for diagnostic purposes, it is useful to perform consistency checking at this stage. Global assertions (see Price [72]) are expressions written in terms of the V-functions of the module. In order to show that they apply to the module, it must be shown that they are true for the initial state of the module and also after any sequence of O-function calls. A similar method for proving properties of an abstraction is suggested by Hoare [72], and Spitzen and Wegbreit [75]. Global assertions describe general properties of a module, and thus may be used as lemmas to simplify the proof of a program that calls functions of the module. Some general system properties (e.g., those pertaining to security) can be represented as global assertions. General system properties are usually associated with the user interface.

Stage 3--At this stage, decisions are made regarding the representations (or mappings) of the state of level i (characterized by its V-functions) in terms of the state of level i-1 (taking the view of Figure I-2.1a). It is also possible to state and prove properties of the representations. This approach is similar to that of Hoare [72]. The generalization to the hierarchies of Figures 1b and 1c is straightforward.

The state of a module is defined by a particular assignment of values to the module's V-functions. In terms of the state space S of Mi and the state space T of Mi-1, a mapping function of level i is defined as a surjective (i.e. "onto") function from T' to S, where T' is a subset of T. It is first shown that this concept of a mapping function conforms to the desired properties of the data representation. A method is then given for writing relations among V-function values of Mi and those of Mi-1 in a manner that ensures that such relations constitute a mapping function.

With regard to the state mappings from T' to S, the following are observed:

(a) Numerous states in Mi-1 can map to a single state in Mi, as depicted in Figure I-2.2a, due to the possibility of delaying the decision on the precise representation of the state of Mi.

(b) Not all states of Mi-1 have images in Mi. As illustrated in Figure 2b, a direct transition (S1,S2) in Mi might correspond to a transition (T1,T2) in Mi-1 that traverses several intermediate states that do not map to states in Mi. Moreover, there might be several paths between T1 and T2 (as shown), corresponding to different implementation algorithms.

(c) The most general case is that in which each of the states in a direct transition pair of Mi corresponds to several states in Mi-1. In Figure I-2.2c, S1 corresponds to T1 and T2, and S2 corresponds to T3 and T4. A correct implementation of the direct transition

(S1,S2) in Mi could be any of the following transitions in Mi-1: (T1,T3), (T1,T4), (T2,T3), or (T2,T4).

A mapping function for level i is written as a set of expressions containing the V-function values of machines Mi and Mi-1. Each mapping function expression contains a V-function value of Mi (with formal parameters) set equal to an expression containing V-function values of Mi-1, which gives the value of the higher-level V-function in terms of values relevant to the lower level. A V-function value of Mi maps down to an expression containing V-function values of Mi-1, even though mapping functions are defined as an upward mapping of states. This is because we can apply the mapping function expressions to a higher-level assertion in order to derive a lower-level assertion (see the discussion of mapped specifications below).

Once mapping function expressions have been defined for each of the V-functions of Mi, it remains to be determined if they characterize the properties of a mapping function, as enumerated in the state-space description of abstract machines. The mapping function expressions are consistent if such is the case. If mapping function expressions are inconsistent, it is impossible to find an implementation satisfying the specifications of the modules Mi and Mi-1. The consistency of mapping function expressions between Mi and Mi-1 is proved with respect to the specifications of Mi by creating mapped specifications for Mi, i.e., substituting each V-function reference in the specification of Mi by its instantiated mapping function expression. The mapped specifications can be proved consistent in the same manner used to show the self-consistency of a module specification in Stage 2.

As discussed below, mapping function expressions are used to transform module specifications of Mi into assertions expressed in terms of only V-functions of Mi-1.

The effort in the first three stages results in a design for a system. In our notion of a design, many of the important system properties can be stated and proved before any code is written. The specifications of Mi and Mi-1, and the mapping function of level i are sufficient to generate the correctness criteria (i.e., input and output assertions) for an implementation of Mi in terms of Mi-1.

Stage 4--Each of the functions of Mi, i>0, is implemented as an abstract program using the functions of Mi-1 and the control constructs of some formally-defined programming language. These programs complete the binding of the decisions that were left incomplete by the mapping functions of Stage 3. Each of these abstract programs must be proven to be a "successful" implementation, with respect to the specifications of Mi and to the mapping functions between Mi and Mi-1. This is accomplished by deriving input and output assertions for the implementing

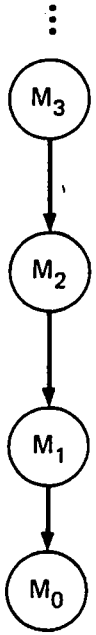
programs, which (if satisfied) imply such a "successful" implementation. The input and output assertions for the implementing programs are trivially derivable from the mapped specifications of Mi. Then the correctness of the implementing programs is proved with respect to these assertions using an extension of Floyd's method (Floyd [67]) described in Robinson and Levitt [75]. The extension axiomatizes the generation of verification conditions for programs calling O-functions.

Each implementing program has an entry point with no pre-conditions (i.e., an input assertion of TRUE) and several exits. The program can exist without a pre-condition, because it contains its own machinery for detecting exceptions and for reporting them back to the caller (via the multiple exits). If the function has n exception conditions, there are n+1 exits: one for each of the n exception conditions (with the mapped exception conditions serving as corresponding output assertions), and one for the normal exit. In the case of a V-function V (Figure I-2.3a), the output assertion for the normal exit states that the program returns a value equal to the instantiated mapping function expression for V. In the case of an O-function O (Figure I-2.3b), the output assertion for the normal exit consists of the effects of the mapped specification for O.

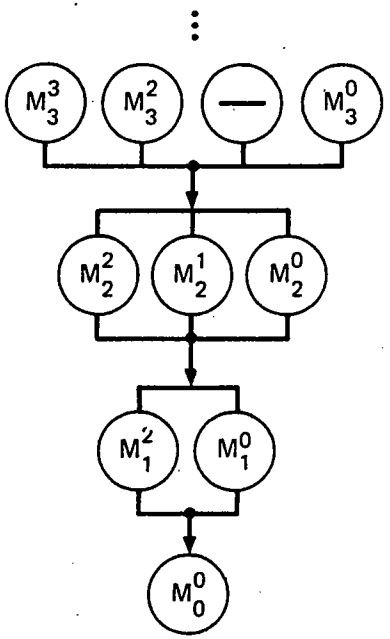
Each of the primitive functions, both of M0 and of the abstract programming language used in Stage 4, is implemented in terms of programs in an available well-defined implementation language, e.g., the instruction set of the hardware or a high-level programming language. The communication mechanism between the levels must also be decided. It could be any one of the following: macro expansion at compile time, procedure calls, or interprocess communication. The formal semantics of the communication mechanism must be stated and proved correct. The end product is a software system with the behavior of the highest-level abstract machine Mn (according to Figure I-2.1a). The users of such a system need not be concerned with the lower levels, because their existence and behavior is completely hidden by Mn.

The methodology localizes design issues to the proper context, and it separates the issues of the behavior of an abstraction from those of data representation and implementation. This localization and separation is attractive because it reduces the complexity to be dealt with at a given level and stage. The methodology seems to lead to systems whose designs are understandable and whose properties are intuitively evident, even in the absence of proofs. The methodology reduces the proof of a large program to the proofs of numerous small programs, and simplifies the input and output assertions that are applied to each program. Due to the data abstraction provided by the hierarchy, the assertions tend to be expressed in terms of functions relevant to a particular level. Currently the main impediments to proving large programs involve the difficulty of

framing assertions for the program and the difficulty of carrying out the deductions for large unstructured programs. This methodology holds attractive prospects for the proof of large software systems. The remainder of this report shows how the methodology has been applied to the design of a secure operating system whose security properties can be proved.



(a) ONE ABSTRACT MACHINE PER LEVEL

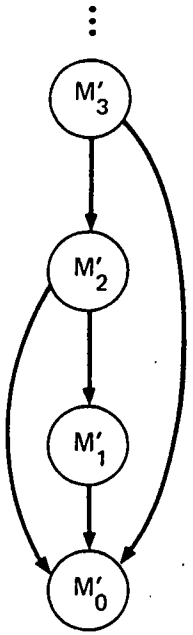


(b) MODULARIZATION OF LEVELS

$$\forall_i \; (i \geq 0 \Rightarrow M_i = \bigcup_{j=0,\ldots i} M_i^j)$$

$$\forall_{ij} \; (j > 0 \wedge j < i \Rightarrow M_i^j = M_j^j \; v \; M_i^j = \text{UNDEFINED})$$

$$\forall_{ijk} \; (j > 0 \wedge j < i \wedge k > i \Rightarrow (M_i^j = \text{UNDEFINED} \Rightarrow M_k^j = \text{UNDEFINED}))$$



(c) CONSIDERING A LEVEL IN TERMS OF ITS DEFINED MODULE ONLY

$$\forall_i \; (i \geq 0 \Rightarrow M'_i = M_i^i)$$

SA-4063-4

FIGURE I-2.1 THREE EQUIVALENT VIEWS OF THE TYPE OF HIERARCHY DEFINED BY THE METHODOLOGY

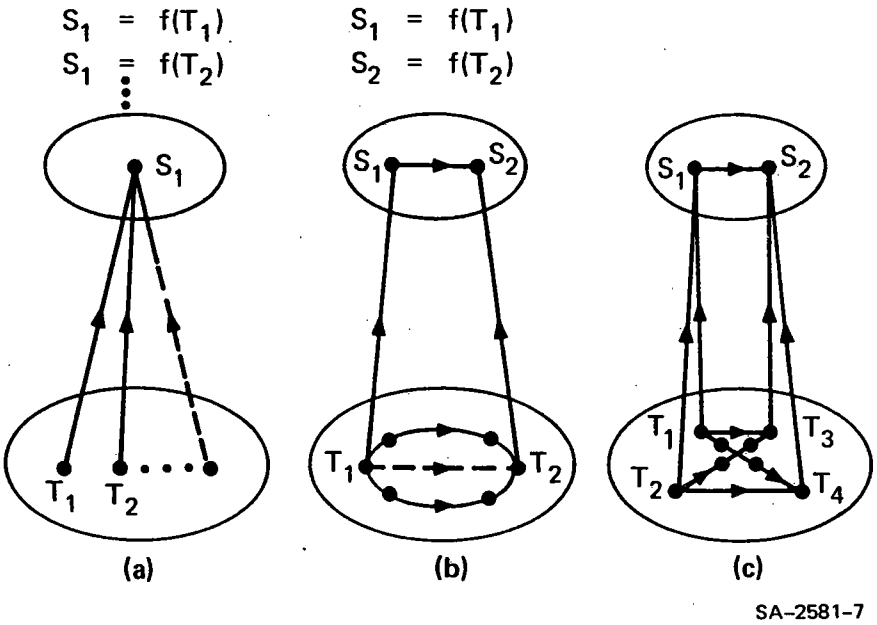


FIGURE I-2.2 MAPPING FUNCTION f RELATING THE STATES OF TWO ABSTRACT MACHINES

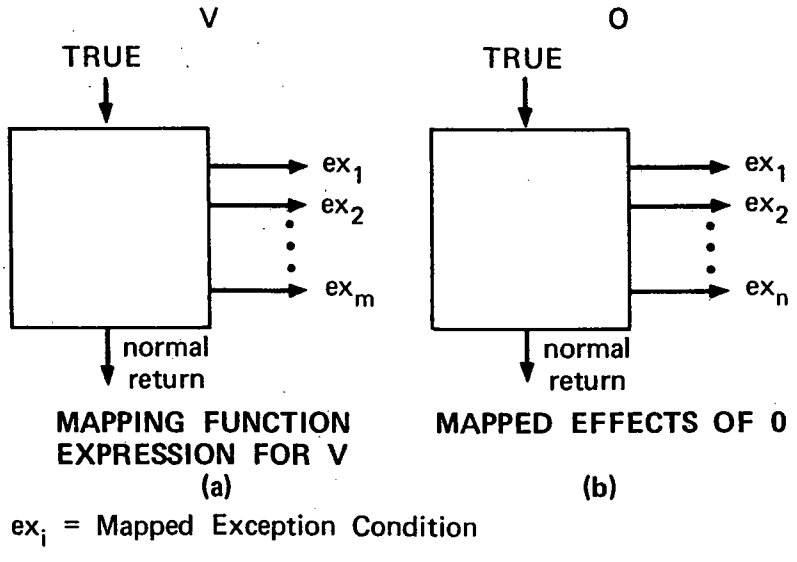


FIGURE I-2.3 FLOWCHART DIAGRAMS FOR PROGRAMS IMPLEMENTING A V-FUNCTION AND AN O-FUNCTION SHOWING INPUT AND OUTPUT ASSERTIONS

PART II
CHAPTER II
DESCRIPTION OF THE OPERATING SYSTEM

This chapter presents a description of the operating system. A complete and rigorous description of the operating system is given in the formal specifications of Appendix B. No attempt is made in this chapter to duplicate the contents of the specifications. Rather, the following discussion considers the more significant parts of the design. Readers interested in details of the design should consult the specifications. In addition to the design itself, this section discusses the motivations for the design and some of the implementation considerations of the design.

The operating system is intended to be general-purpose and therefore contains several facilities common to many general-purpose systems such as virtual memory, multiprocessing, a file system, and virtual I/O devices. These facilities provide a basic and sufficiently general set of resources for efficiently implementing a wide variety of applications. These facilities have been designed to be easy to use and understand so as to simplify the implementation of the applications.

Since proving the security of the operating system is a major objective of this work, the means of achieving security is crucially important to the system design. Capabilities are an excellent choice for the protection mechanism because they are simple enough to be introduced as the most basic part of the system design. All protection in the system derives from the basic capability mechanism. Therefore, once a few security properties are proven about capabilities, it is a straightforward matter to prove that security properties hold for the entire system.

There are many different types of security policies that one might wish to enforce upon a system or subsystem. Since the operating system is intended to be general-purpose, it should be able to support a wide variety of different security policies. The capability mechanism is both general and efficient enough to allow construction of a wide variety of protection mechanisms which can enforce many different security policies.

THE OPERATING SYSTEM INTERFACE

The operating system can be viewed by a user as a collection of virtual resources. Examples of such resources are memory, processing, and I/O. Each virtual resource is divided into abstract units called objects. For example, virtual storage is divided into objects called segments. The user-visible interface to the operating system will be described in terms of the user-visible abstract objects and the operations that can be performed on them.

CAPABILITIES

For purposes of security, the most basic type of object in the system is the capability. Capabilities are the means by which all other objects in the system are identified and are the basis of protection and security in the system.

In order to perform any operation upon an abstract object of the operating system, a user must invoke an operating system function appropriate to that object. All user-visible functions that operate upon an abstract object require a capability for that object as an argument. Every capability has two parts. One part is a unique identifier. The unique identifier can be thought of as an integer that serves to identify the object being operated upon. Therefore, each distinct object must be identifiable by one unique identifier, necessarily distinct from all others. The other part of the capability is a set of access rights to the object. The access rights define those operations which are permitted on the object. The access rights are defined with respect to the type of object to be operated upon. Therefore, each type of object can have different types of access rights. There is a maximum number of access rights that may be defined for any type of object.

By invoking a system function, any program can obtain the access rights associated with a given capability. It is necessary for programs to be able to determine if the unique identifier parts of two different capabilities are the same in order to determine if they represent the same object. A slave capability for a given capability is defined to be the given capability with no access rights. The function GET_SLAVE returns the slave capability of a given capability. The important property of a slave capability is that there is only one slave capability for each unique identifier, therefore, if two capabilities have the same unique identifier (i.e., they represent the same abstract object) they will have the same slave capability. Determining if two capabilities represent the same object is accomplished by comparing the slaves of the two capabilities.

There are five visible functions that operate upon capabilities. These functions comprise the CAPABILITIES module. There is a function just described, GET_SLAVE, that returns the slave of a given capability, and a function GET_ACCESS that returns the access rights of a given capability. The function CREATE_CAPABILITY creates a new capability with a unique identifier

which has never appeared in a previously created capability and which has all possible access rights. The function CREATE_RESTRICTED_CAP creates a new capability with a previously unused unique identifier as with CREATE_CAPABILITY; however, only a given subset of the possible access rights are included in the new capability. The function RESTRICT_ACCESS returns a new capability with the same unique identifier as a given capability and with a specified subset of the access rights of the given capability. These are the only operations permitted on capabilities. Note that none of these operations modifies an existing capability, and that whenever a new capability is created with the same unique identifier as an existing capability by invoking the function RESTRICT_ACCESS, the new capability contains a subset of the access rights of the existing capability. These two properties are necessary for maintaining the security of the system.

SEGMENTS

Segments are the primary means of storing data on the operating system. Each segment consists of an array of storage cells. Each cell is indexed by an integer from zero to one less than the number of cells. Each cell can contain either a capability or an integer. Any cell can be addressed by giving a capability for a segment and the index of the desired cell in the segment. The READ_WINDOW function of the SEGMENT module returns the contents of the cell at the given address. The WRITE_WINDOW operation changes the contents of the cell at the given address to be the value given. Functions exists to create new segments and delete existing segments.

WINDOWS

It is often useful to be able to permit access to only a portion of the data in a segment. A window permits access to any contiguous block of data of a segment. The function CREATE_WINDOW returns a capability that can be used to access some contiguous portion of a given segment. The CREATE_WINDOW function specifies the beginning and length of the portion of the segment to be accessible from the window. A window capability can be used as an argument to the READ_WINDOW and WRITE_WINDOW functions in a manner similar to a segment capability. Functions exist to delete a window and to change the block of data to which a window refers.

PROCEDURES

A procedure is simply a sequence of instructions which can be used by the operating system to control the execution of a process. The instructions of a procedure are stored in a segment. All segments can contain procedures. A segment capability can be used as an argument to the CALL function (described later) to cause the system to actually begin to interpret the instructions of the procedure. A procedure may have several entry points, i.e., cells at which execution of the procedure may begin. The entry points must all be at the beginning of the procedure segment, i.e. the entry points must occupy the first 'n' cells of the segment where 'n' is the number of entry points. The CALL function will not

permit entering a procedure at an instruction that is not an entry point.

I/O DEVICES

Any general-purpose operating system must support several types of I/O devices. However, the choice of precisely which I/O devices to support depends upon the specific needs of the users of the system. For whatever I/O devices are selected, it is assumed that the operating system will provide a well engineered set of functions for operating those devices.

Whatever I/O devices are supported by the system, it is likely that needs will arise to support other I/O devices. In addition to functions that support specific I/O devices, it is necessary to have functions which allow utilization of arbitrary I/O devices. It is these functions that are described below.

Each device has four communication paths to the user program. Each path contains one fixed-size buffer. Two of the paths, "command" and "status", are for transmitting control information between the device and the user program and the other two, "input" and "output", are for transmitting data. The function SEND_COMMAND is invoked to send control information to a device. This function places the control information in the "command" buffer. The device invokes the DEVICE_COMMAND function to retrieve the information from the command buffer. In the opposite direction, the device invokes the function CHANGE_STATUS to place control information in the "status" buffer. This control information can be read by the user program by invoking the RECEIVE_STATUS function. Similar functions exist for passing data between the device and the user.

It is assumed that the device will constantly monitor the "command" buffer for control information, since executing these commands is its only purpose. However, it is unreasonable to expect the user program to constantly monitor the "status" buffer, because such monitoring is a waste of time for the user process. For this reason, the CHANGE_STATUS function will optionally notify a waiting process of a change in device status by invoking the WAKEUP function (see section on process coordination).

Note that none of the above functions incorporates knowledge of specific devices. The communication paths have been designed to be sufficiently general to permit communication with almost any type of I/O device. Knowledge about any particular device must be incorporated in the programs that invoke the communication functions.

DIRECTORIES

Directories provide a means of associating names with capabilities and, thereby, the objects that the capabilities represent. A directory is a collection of named entries. Every name within a directory must be unique. However, the same name may appear in different directories. Each entry contains one

capability. The function GET_CAP of the DIRECTORIES module returns the capability contained in the entry with the given name in the given directory. Every entry is locked. This means that GET_CAP will not return the capability unless the appropriate key is supplied. The key is simply another capability. The lock mechanism provides a way to control access to each entry in a directory individually. The functions ADD_ENTRY and REMOVE_ENTRY create and delete entries. The functions CREATE_DIRECTORY and DELETE_DIRECTORY create and delete directories.

Certain designated entries in directories are treated specially. These entries are termed distinguished. The function REMOVE_ENTRY cannot be applied to a distinguished entry and the function DELETE_DIRECTORY cannot be applied to a directory containing a distinguished entry. A distinguished entry is added to some designated directory whenever a new object is created by the system. The entry contains the capability for that object. The object may be a segment, window, directory, user-defined object, or process. The distinguished entry is removed only when the object is deleted. Therefore, there is one and only one distinguished entry for each existing user-visible object. The purpose of distinguished entries is to guarantee that there exists at least one capability for each existing user-visible object. This guarantees that there is always a means of accessing every existing user-visible object.

PROCESSES

Processes are the most complex objects of the system. This is largely because processes are made up of many other objects which are themselves complex. However, the key to understanding the operation of the system is understanding how a process operates. The following subsections describe distinct aspects of a process.

i) Registers

Each process has associated with it a set of registers. The registers are of two types: general registers and address registers. The general registers can contain any type of data, i.e. capabilities or integers. There is a fixed number of general registers, and there is an operation to load each register and to read the contents of each register. The address registers contain addresses of cells (words) in segments or windows. Each address register, therefore, contains a capability for a segment or window and an offset of a word in that segment or window. There are operations to load each address register and to read the contents of each address register.

Some of the address registers are given special interpretations by the system: the program counter, the stack register, and the arguments register. The program counter contains the address of the word of storage containing the instruction currently being executed by the process. The program counter is automatically incremented after execution of each instruction is completed or it is modified by a CALL or RETURN (described below). The stack and arguments register are used by the CALL and RETURN functions as part of the

discipline of invoking procedures. The interpretation and use of these registers is described below.

There are two special purpose capability registers: the process register and the instruction class register. The process register contains the capability for the process. From the point of view of each process, the value of the process register is constant and can only be read. The instruction class register enables the execution of instructions, i.e., an instruction can not be executed unless an appropriate capability is contained in the instruction class register. Typically, a single capability placed in the instruction class register enables many instructions. For example, a single capability enables all user executable instructions. Functions are provided to read and modify the instructions class register.

ii) Execution of instructions

The purpose of each process is to perform computations. These computations are defined by a programs that are contained in procedures. The essential component of each procedure is a sequence of instructions. These instructions dictate the sequence of functions invoked by the process. Each process interprets instructions as follows:

1. Read the address in the program counter.
2. Read the contents of the word specified by this address.
3. Interpret the contents as an instruction, and invoke the functions necessary to accomplish the intended effects of the instruction.
4. Modify the program counter to contain the address of the next instruction if the program counter has not already been modified in the course of executing the instruction.
5. Go back to step 1.

Instructions may have only effects that are compositions of the functions described in the specifications. For example, a load register instruction might compose the READ_WINDOW function of the WINDOWS module and the LOAD_GENERAL_REGISTER function of the REGISTERS module. The choice of what functions to compose to make up instructions should be largely dependent on the intended applications of the system. The basic design of the system is not dependent on the instruction set chosen.

iii) Procedure activation

A procedure activation is a very important element in the operation of a process. At any given time, all the objects accessible to a process must derive from the procedure activation, i.e. a process cannot reference an object unless a capability for that object can be derived from a capability contained in the process's procedure activation. A procedure activation consists of three parts:

1. a procedure,
2. an activation record, and
3. a procedure record.

A procedure has been defined above, and is primarily a sequence of instructions and possibly some data. The data and instructions may contain capabilities, and these capabilities may be used to access other objects. An activation record is an array of data words that is accessible only to the process while it is executing in the procedure activation. In other words, the activation record contains data that is local to a given activation of a procedure. The process registers are considered to be part of the current activation record. An activation record is automatically created when a procedure is invoked via the CALL function (see below). The activation record initially contains the arguments of the procedure invocation. The RETURN function deletes the activation record for the activation.

A procedure record is also an array of data words. However, the scope of a procedure record is different from an activation record. A procedure record is accessible from all activations of a given procedure within a process. A procedure record can be read or written by invoking the functions READ_PR or WRITE_PR of the PROCEDURE_RECORDS module. The size of the procedure record cannot be changed. The procedure record is deleted when the process is deleted.

The three basic objects of a procedure activation provide the various different types of storage necessary to an activation. The procedure segment itself provides permanent storage. The procedure record provides storage for the duration of the process and the activation record provides storage during the activation.

A procedure activation is initiated by invoking the CALL function of the USER_INVOKE module and the activation is terminated by invoking the RETURN function of the same module. The CALL function requires arguments: the address of the procedure being invoked, the address of the instruction to return to when the new activation is completed, the address of the arguments for the new activation, and the address of the current top of the stack.

There are actually three types of procedure invocation: the system procedure call, the protected call, and the unprotected call. The system call is used whenever a system procedure is invoked and looks to the user exactly like a protected call. The system call is described fully below. The protected call creates a new segment to be used as the activation record of the new activation and places a pointer to this new segment in the stack register. It also places the address of the arguments in the arguments register. The return address is remembered by the system so that it may be utilized during the return from the new activation. The address of the called procedure is placed in the program counter causing control to transfer to the called procedure.

The unprotected call does not create a new segment to be used as the activation record of the new activation. Instead it simply loads the stack register with the given address of the top of the stack. In other words the activation record for the new activation will be placed wherever the calling procedure specifies the top of

the stack to be. This implies that the new activation is not protected from the calling activation. In the unprotected call the arguments are passed in the same manner as the protected call, i.e., by placing the address of the arguments in the arguments register. The address returned by the unprotected call function is simply the return address supplied by the caller. This information is used by the return operation to return to the calling activation.

The RETURN function is invoked by a procedure activation in order to terminate the activation. The RETURN function requires one argument: the return information returned by the CALL or UNPROTECTED_CALL function. If the call was protected then the return must find the return address from the system data base. If the call was unprotected then the argument to the RETURN function is the return address. Control is then transferred to a fixed location. It is assumed that the instructions at that location will then perform the necessary operations to cause the actual return to the calling activation using the return address as the source of return information.

A few general comments are in order to fully explain the invocation of procedures. The protected CALL function is provided to permit isolation between the calling and called activations. The only direct communication path between the calling and called activations in a protected call is through the arguments. The calling activation must be careful to pass as the arguments address only sufficient capabilities for the arguments it wants the called activation to have access to. The calling activation will likely create a window to use as the arguments address. The unprotected call is used when both the calling activation and called activation are willing to trust one another. Since, in an unprotected call, the calling activation provides the activation record for the called activation record and the called activation has access to the return information of the calling activation, they must both trust each other. Because an unprotected call implies mutual trust, either the calling or the called activation can demand a protected call. The calling activation demands a protected call by invoking the protected call operation instead of the unprotected call operation. The called activation demands a protected call by failing to provide "read" access in the capability contained in its address.

The return address supplied as an argument to a call must actually be the address of sufficient information to restore the state of the calling activation sufficiently so that it may proceed. At least two pieces of information are necessary to accomplish this: the address of the activation record for the activation and the address of the instruction at which execution is to be resumed in the calling procedure. Other information may also reside here such as the values of some of the registers which are to be restored.

It is important to note that when a new activation begins execution via a protected call, the only objects it has access to are those within the procedure segment, activation record (including the registers), or procedure record of the new activation or objects accessible via calling other procedures or functions for which

capabilities exist in the procedure segment, activation record, or procedure record of the new activation. The only ways for the two activations to share objects is through arguments or by incorporating shared capabilities into the procedure segments themselves. This makes it easy to construct procedures that are well protected from other procedures in the system, and makes possible the building of secure subsystems.

iv) Process Coordination

Different processes can share information by sharing data in commonly accessible segments. There is a need for processes to coordinate their activities in order that they do not interfere with one another. The system provides two very basic, but powerful, coordination facilities. The first of these is a very simple "test and set" type of instruction provided by the SEGMENTS module. The function DECREMENT_AND_TEST decrements by one the integer value stored in the given word of the given segment and returns true if the resulting value is a negative integer. The function INCREMENT_AND_TEST increments by one the value of the specified word and returns true if the original value of the word was a negative integer. These instructions can be used to assure mutual exclusion of processes in a critical section of program and are as powerful as the P and V operations of Dikjstra.

The second coordination facility is provided by functions of the COORDINATOR module. These functions permit one process to queue itself and wait for a wakeup from another process and permit a process to cause another process to cease waiting and continue execution. The test and set functions of the SEGMENTS module and the wait and wakeup functions of the COORDINATOR module can be used to construct more complex process coordination operations.

v) Clocks and timers

The system maintains a clock. The clock is simply a integer value that is incremented at uniform intervals of time. The value of the clock is maintained by invoking the function UPDATE_CLOCK of the USER_TIMERS module. It is assumed that some system controlled process (probably dedicated to this purpose) is responsible for invoking the UPDATE_CLOCK function at regular intervals. The value of the clock is obtained by invoking the function READ_CLOCK.

Each process may (by invoking the function SET_CLOCK_TIMER) declare a clock time at which it is to be awakened. When the value of the clock becomes greater than the specified timer value, the wakeup is performed.

Each process also has a virtual clock of its own. This clock records the time the process spends executing instructions. The virtual clock time does not include the time the process spends suspended or waiting. The precise definition of this time is implementation dependent. The function READ_PROCESS_TIMER returns the value of this clock. Each process may also declare a process clock time at which it is to be awakened. This declaration is made by invoking the function SET_PROCESS_TIMER.

vi) Creation and deletion of processes

As stated above, the objects accessible to a process at a given time are determined by the procedure activation that is executing in the process at the given time. The objects accessible to a process for its entire lifetime are determined by the procedure activation that executes when the process is created. The task of creating a process is logically similar to creating a procedure activation. The function used to create a process (CREATE_UPROC of the USER_PROCESS module) appears similar to the CALL function.

The CREATE_UPROC function requires four arguments: a directory capability and entry name for the directory entry into which the capability for the new process is to be kept, the address of the procedure to be invoked in the new process, and the address of the arguments to this procedure. The CREATE_UPROC function creates an activation record for the new activation. It also creates a set of registers for the process and sets the program counter to the specified procedure address. A new directory is also created for the new process. This directory, called the process directory, is to be used by the process for retaining capabilities for temporary objects whose lifetime is less than or equal to the lifetime of the process. A capability for this directory is returned by the function GET_PROCESS_DIR. This directory, and all the objects in the directory will be deleted when the process is deleted. The process is suspended from executing. When the function START_UPROC is invoked for the process, the process will begin execution in the newly created procedure activation.

A process is deleted by first invoking the function STOP_UPROC. This function stops the execution of the process to be deleted. Invoking the function DELETE_UPROC then destroys the process. This involves deleting all activation and procedure records of the process, and deleting the process directory and all objects in the process directory.

USER CREATED OBJECTS

Users can, of course, write procedures that create and maintain objects having properties that the user desires. The user must simply create procedures with an entry for each function that is permitted upon the new type of object. The procedures must use existing objects to represent the new type of object. Capabilities can be used to identify and control access to the new type of object. The system provides functions to assist in creating and maintaining user defined objects. Each type of object is identified by a distinct unique identifier. The function CREATE_TYPE of the EXTENDED_TYPES module creates a new type and returns the slave capability for the type. The function CREATE_OBJECT returns a capability for an object of a given type. The extended type module maintains a vector of capabilities for objects (primitive or extended) which comprise the representation of each extended object it has created. This representation may be modified only if the extended type manager is presented a capability for the object whose representation is to be modified and a capability for the object

type with appropriate access rights. There are four functions that can be used to modify a representation. CREATE_IMPL_OBJ creates a new object of the specified type and inserts a capability for that new object into the representation vector at a specified location. INSERT_IMPL_OBJ simply inserts a given capability into the representation vector at a specified location. DELETE_IMPL_OBJ deletes a specified capability from the representation vector and deletes the object represented by the deleted capability if the object was created by CREATE_IMPL_OBJ. DELETE_IMPL_CAP deletes a specified capability in the representation vector if the capability was inserted by INSERT_IMPL_OBJ. Since objects created by CREATE_IMPL_OBJ do not have directory entries, in order to assure that a capability exists for each existing user-visible object, an extended object can be deleted only if all the objects created for its representation by CREATED_IMPL_OBJ have been deleted.

THE BASIC SYSTEM DESIGN

The operating system is decomposed into sixteen levels. The levels are listed in Table 2-2. The short description of each level below indicates the major functions implemented at that level that are not implemented at some lower-numbered level. Each level is discussed in turn, beginning with the lowest-numbered level and proceeding sequentially upward through the levels. The functionality of some of the levels has already been described as part of the system interface above. These descriptions are not repeated. Specifications for levels 0 through 13 are given in Appendix B. The descriptions below concentrate on these levels. All the security-related mechanisms of the system are contained within these levels. Levels 14 through 16 relate to tailoring the user interface to the needs of the users and will be largely dependent upon the applications desired for the system. These higher levels are described, therefore, in general terms only.

The specification of each level consists of a collection of modules. The modules comprising each level are shown in Table 2-1. The description of each level is in terms of modules. In the case where a module appears in the specification of more than one level, the description is given as part of the lowest level in which the module appears.

LEVEL 0

CAPABILITIES

Capabilities and the functions related to them are described above.

LEVEL 1

REGISTERS

Registers are described briefly above. Each processor has its own set of registers. The general registers and address registers may be used by procedure activations to store appropriate values temporarily. The special-purpose registers are used by various other modules of the system for specific purposes. One of these special-purpose registers, the process register PROC_CAP, contains a capability for the processor of which the process register is a part. The registers are distinct from other forms of memory because they are constructed to be extremely efficient in performing a particular job.

PRIMARY MEMORY

Primary memory consists of arrays of cells. Each cell can contain any type of information of a predetermined fixed size. The arrays themselves are fixed size. Each array is called a memory

block. The cells of a block are accessed by providing a cell address. A cell address consists of a capability for the block containing the cell and the index of the cell in the array. The function BLOCK_READ of the MEMORY module returns the current value of a cell, and the function BLOCK_WRITE changes the value of a cell. The functions BLOCK_DECREMENT_AND_TEST and BLOCK_INCREMENT_AND_TEST respectively decrement and increment the value of a given cell if that cell contains an integer. These functions also indicate whether the value of the cell is negative. These functions can be used to synchronize processors or processes.

The size of each block is determined during initialization. Each block provides memory for the implementation of some low software level of the design. This enables the lower levels to be isolated from one another. The higher levels of the design use the virtual concept of segmentation to achieve this isolation.

LEVEL 2

INTERRUPTS

The hardware base of the operating system has been designed to accommodate several processing units operating in parallel. There may be several main processing units, called processors, that execute user and system procedures. There may also be I/O processors, communications processors, a capability generating processor, a clock updating processor, etc. These various processors can coordinate their activities by means of interrupts.

Each processor has a fixed number of interrupts. A processor may send an interrupt to another processor or to itself by invoking SET_INTERRUPT. A processor receives an interrupt by invoking RECEIVE_INTERRUPT. It is assumed that all processors invoke RECEIVE_INTERRUPT frequently in order that they will quickly respond to any pending interrupts. RECEIVE_INTERRUPT modifies the program counter to a predefined value for the interrupt that has occurred and saves the previous value in the program counter. The new value of the program counter should be the address of the first instruction of a procedure that responds to the interrupt. Interrupts for each processor can be individually masked or all interrupts for a given processor can be inhibited for that processor. If two interrupts are simultaneously pending on the same processor, the one with higher priority will be processed first. The address of the procedure that responds to a particular interrupt can be set by invoking SET_INT_HANDLER.

The interrupt mechanism is by necessity quite primitive. It is not well suited to the coordination needs of higher levels of the operating system. It is also difficult to verify programs written using interrupts. For these reasons the interrupt mechanism is used only by levels 0 through 6. At level 6 a new coordination mechanism is introduced that is better suited to the needs of higher levels, and the interrupt mechanism is made invisible to higher levels.

LEVEL 3

CLOCK

Clock related functions have been described above. However, since processes do not exist at this level there are no virtual time clocks and since the function WAKEUP_PROCESS does not exist at this level notification that some time has elapsed is made by sending an interrupt.

LEVEL 4

ARITHMETIC AND LOGIC FUNCTIONS

The ARITHMETIC module provides the functions necessary to perform arithmetic, logical, and boolean computations. The functions are ADD, SUBTRACT, MINUS, MULTIPLY, DIVIDE, LOGICAL_AND, LOGICAL_OR, LOGICAL_NOT, LESS_THAN, GREATER_THAN, and EQUAL. Other similar functions might be added for convenience. Note that all the functions in this module are derived and there are no 0-functions.

LEVEL 5

PRIMITIVE I/O

The primitive form of input and output is very similar to the user-visible form of input and output. The same four communication paths, "command", "status", "input", and "output", are used to pass control information and data between the system and the device. Since the coordination function WAKEUP_PROCESS does not exist at this level, the CHANGE_STATUS function of the PRIMITIVE_I/O module uses an interrupt to notify a procedure that a change of status has occurred.

In order to take the burden of performing trivial I/O operations away from the main processors of the system, some systems introduce special-purpose processors for the purpose of performing the simpler I/O functions. These special processors are generally called channels. A specification for an I/O facility using such channels is given in the CHANNEL_IO module of Appendix B. The introduction of channels changes the usage of the data communication paths, "input" and "output". Rather than actually supplying or receiving a word of information in the WRITE_DEVICE or READ_DEVICE functions as in the PRIMITIVE_I/O module, the addresses of cells in main memory are supplied to these functions. The channel will either read these cells and send the data contained therein to the device, or place data from the device into these cells. A single invocation of READ_DEVICE or WRITE_DEVICE can involve the transmission of many words of data. The main advantages of channels in systems built using contemporary technology is that the main processors do not have to coordinate their timing with the timing of the device and the resources of the main processor are not wasted on very simple tasks. The channel processor handles the timing

problems. If the CHANNEL_IO module is used to replace the PRIMITIVE_IO module at level 1, then a channel like interface should be used for input and output at the higher levels as well, e.g., for the SYSTEM_IO and VISIBLE_IO modules.

LEVEL 6

SYSTEM PROCESSES

System processes are essentially the same as the processes described as part of the user interface (hereafter called user processes). There are four major differences. First, there are a fixed number of system processes. Therefore, system processes are not created or deleted. All the system processes exist after initialization. Second, in system processes, procedure activations do not have procedure records. Procedure activations are composed of a procedure storage unit (a procedure storage unit can be a procedure segment, procedure page, or procedure memory block depending upon the level at which the procedure exists) and an activation record. Third, system processes cannot be stopped or started. Fourth, system processes do not have a process directory. With the exception of these four differences, system processes operate as do user processes.

System processes serve two purposes. They may perform certain system tasks such as removing pages from primary memory, or may be used to implement user processes. In general, a small fixed number of system processes will be assigned to implement the large number of user processes. This can be accomplished by time-multiplexing the system processes among the running user processes. The function INSTANTIATE_SPROCESS of the SYSTEM_PROCESS module exists for this latter purpose. If a user process is currently bound to a given system process, the binding can be changed by invoking INSTANTIATE_SPROCESS. A different process can be bound to a given system process by giving the state of the new process. INSTANTIATE_SPROCESS will change the state of the given system process to this state and return the old state of the system process. This old state defines the state of the previously bound process and can be saved until some later time when it can be bound to a system process once again. The state of a process is simply the contents of all the registers of that process.

REGISTERS

The registers of level 6 are exactly the same as the registers of level 1 with the exception that at level 6 there is one set of registers for each system process rather than one set per processor. (This is why a new instantiation of the REGISTERS module is shown at level 6 in Table 2-1.) The process register contains a capability for the system process with which the register is associated. The program counter contains the address of the cell containing the instruction being executed; however, the address may refer to a memory block, a page, or a segment depending upon the level at which the executing procedure resides.

CLOCK AND TIMERS

The clock and timers of level 6 have the same functions as those visible at the user interface (described above as part of the operating system interface). At level 6, the virtual timers apply to system processes rather than user processes.

PROCESS COORDINATION

The coordination functions of the COORDINATOR module are exactly those visible at the user interface (described above as part of the operating system interface). The process capabilities refer to system processes rather than user processes.

PROCEDURE INVOCATION

As stated above, a procedure activation at this level, and all higher levels through level 12, consists of an activation record and a procedure storage unit. The invocation of a procedure activation is accomplished by means of the SYSTEM_CALL. The system call is quite similar to the unprotected user call, however, there are a few differences. First, because a system procedure may reside in primary memory (rather than in a segment), the address of the procedure cannot be distributed to users of the procedure. For system procedures, special system procedure capabilities are used for procedure invocation. The system maintains the mapping between the procedure capabilities and the procedure addresses. Second, calls to system procedures are always made from a procedure implementing a higher or the same level of the system or from user procedures. Calls are never made from a procedure at some level to a procedure at a higher level. Therefore, it is necessary to protect the called activation from a calling activation at a higher level, but it is never necessary to protect the calling activation from the called activation. A portion of memory or a segment is set aside for the activation records of each level within each system process. All the activation records within a level within a system process will follow a stack discipline within the storage set aside for that level in that process. Procedure invocations from an activation at some level to an activation at a lower level cause a stack switch so that the activation record of the activation at the lower level is not accessible to activations at a higher level. However, the arguments and return information passed from the higher level activation are accessible to the lower level activation. This is consistent with the general protection policy of the system.

Note that the user call and system call have been designed to be uniform in structure so that a program need not know whether or not it is or is invoking a system procedure when it is being compiled. This simplifies the maintenance and testing of programs as well as simplifying the compiler.

SYSTEM I/O

The I/O functions of the SYSTEM_I/O module are identical to the functions of the PRIMITIVE_I/O module of level 5, with one

exception. Instead of using an interrupt to signal a change of status, the CHANGE_STATUS function of the SYSTEM_I/O module uses the WAKEUP_PROCESS function introduced at this level. This method of signalling is better suited to the procedures of higher levels.

LEVEL 7

PAGES

A page is a convenient abstraction useful in the particular implementation of segments chosen for this system. Conceptually, a page is a fixed size block of contiguous storage cells in which each cell can be addressed by giving a capability for the page and the offset of the cell within the page. The functions READ_PAGE and WRITE_PAGE are invoked to read and modify the contents of a cell in a page. The functions CREATE_PAGE and DELETE_PAGE create a new page and delete an existing page respectively. In the implementation the page may be a very complex object because the cells of a page may be represented as cells of a block of memory or cells of one or more secondary storage devices, and the representation may change with time. The complexity of the implementation does not appear in the abstract view of the specification.

LEVEL 8

SEGMENTS

The functions of the SEGMENTS module have already been described in the discussion of segments in the user interface.

WINDOWS

The functions of the WINDOWS module have already been described in the discussion of windows in the user interface.

LEVEL 9

EXTENDED TYPE

The functions of the EXTENDED_TYPE module have already been described above under user-created objects of the user interface. The functions of the EXTENDED_TYPE module can be used by higher levels of the system as well as by users to create new object types. The directory is a good example of where the system uses the functions of the EXTENDED_TYPE module to create a new system type.

LEVEL 10

DIRECTORIES

The functions of the DIRECTORIES module have already been

described as part of the user interface. The functions ADD_DISTINGUISHED_ENTRY and REMOVE_DISTINGUISHED_ENTRY that add and delete distinguished entries are not visible at the user interface. Distinguished entries can be added and removed by the user by creating or deleting some object using functions of level 11. The use of locks and keys is described later.

LEVEL 11

USER OBJECTS

The purpose of the USER_OBJECTS module is to enforce the constraint that for every user created object there is exactly one distinguished directory entry. The USER_OBJECTS module contains functions to create and delete segments, directories, and extended-type objects. Whenever a user procedure creates an object by invoking one of the create functions of this module, a distinguished entry is created for that object. Whenever a user procedure deletes an object by invoking a delete function of this module, the distinguished entry for that object is deleted. Since a user procedure has no other means to create or delete objects (the create and delete functions of the SEGMENTS, WINDOWS EXTENDED_TYPE, and DIRECTORIES module are not accessible to user procedures) the desired relationship is ensured.

LEVEL 12

USER PROCESSES

All of the functions introduced at this level have been described as part of the user interface. The primary differences between this level and level 6 are:

1. All references to processes at level 12 are to user processes rather than system processes.
2. Processes can be created and deleted.
3. Processes can be stopped and started.
4. Each process has a process directory in which it can create entries for objects for the lifetime of the process.

REGISTERS

The registers of level 12 are identical to the registers of levels 1 and 6 with the exceptions that each user process has a set of registers and the process register contains a capability for the user process.

COORDINATION

The COORDINATOR module of level 12 is the same as the COORDINATOR module of level 6. Because the level 12 COORDINATOR can be implemented with segmented virtual memory and the level 6 COORDINATOR cannot, the implementation of the two COORDINATOR

modules may be different. They are, therefore, shown in Table 2-1 as separate instantiations.

USER TIMERS

The USER_TIMERS module of level 12 is the same as the TIMERS module of level 6 except that each user process has a set of timers.

PROCEDURE INVOCATION

The call and return operations of level 12 have already been described in the description of the user interface.

VISIBLE I/O

The user visible I/O of this level is identical to the system I/O of level 6, with one exception. At level 6 each set of communication paths has a single capability to identify it. This capability is defined for the life of the system. If a capability for an I/O device is to be given to a user procedure, it is necessary that there is some means of invalidating this capability so that a capability for the same device can later be given to some other user procedure with the assurance that only that procedure has access to the device. For this reason the ASSIGN_DEVICE function of the VISIBLE_IO module creates a temporary capability for a device given a permanent capability for the device. Either a permanent or a temporary capability can be used in any of the other functions of the VISIBLE_IO module. The function DEASSIGN_DEVICE invalidates the temporary capability for the specified device.

LEVEL 13

PROCEDURE RECORDS

The use of procedure records has been described as part of the user interface. The only visible functions of the PROCEDURE_RECORDS module are READ_PR and WRITE_PR which read and modify the cells of the given procedure record. The function CREATE_PR creates a new procedure record and associates with a given procedure. The capability for a procedure record for some procedure can be obtained by invoking the function GET_PR_ADDRESS, giving the capability for the procedure as an argument.

LEVEL 14 - USER I/O

The user-visible I/O interface of level 12 is very primitive and very general. It allows the connection to and use by the system of a very large class of I/O devices. However, it requires that the user procedure invoking the functions of level 12 know all the details of operating every different type of device. Most applications will require somewhat less generality and somewhat more convenience. An I/O interface is more desirable for most users if

it provides a few generic functions applicable to many different types of devices and hides most of the particular details of operating a particular device such as buffering, timing, data format, and detailed control. The purpose of level 14 is to provide such an interface. An example of such an interface is given by Feiertag and Organick [71]. The interface is not specified here because it is not immediately relevant to the security of the system.

LEVEL 15 - USER ENVIRONMENTS AND USER NAME SPACE

Any usable operating system must contain some functions that perform high-level tasks. The implementation of these functions translates the high level requests into the more basic functions of the operating system. Examples of such high-level tasks are compilers, debuggers, mail, linking, text editing, sorting, billing, report generation, etc. It is also convenient to be able to use names to denote certain objects rather than capabilities; a user cannot type a capability into a terminal. The purpose of level 15 is to provide such functions. These functions are not specified here because the totality of all such functions is too numerous and the ones that are useful to any given instance of the operating system depend upon the applications of that instance. Some examples of such functions are given later in this section under the heading "TYPICAL USER ENVIRONMENT". The difference between a user environment and an application is not at all clear. Applications such as the Secure Object Manager described in Section IV-1 can be considered environments.

LEVEL 16 - USER REQUEST INTERPRETER

If any process of the operating system is to perform tasks on behalf of the user, there must exist some language comprehensible to both a process and a user. Level 16 contains functions which accept requests in some such language and interpret them, i.e., invoke the functions necessary to implement the requests. A single operating system may have many request languages and, therefore, may require several request interpreters.

PROTECTION MECHANISMS

A simple capability mechanism, with capabilities containing only unique identifiers, is sufficiently general to implement any desired form of protection. However, some useful types of protection are difficult to construct using this simple capability mechanism alone. For this reason, the operating system has four different protection mechanisms other than a simple capability mechanism. The four mechanisms are:

1. capabilities with access rights,
2. capability store limitations,
3. protection of hierarchical levels, and
4. directory entry locks and keys.

Each of these mechanisms is described, in turn, below.

CAPABILITIES WITH ACCESS RIGHTS

Each capability has associated with it a fixed number of boolean values called access rights. The access rights serve to determine which of the operations on the object designated by the capability can be invoked using this capability. For example, among the access rights associated with a segment capability are the rights for reading and writing the segment. If the read access right of a segment capability has the value TRUE, then the read operation, when invoked with this capability as an argument, will complete successfully (assuming all other preconditions of the read function are met). If the read access right has the value FALSE, then invoking the read function with this capability will result in an exception. The analogous actions will occur for the write operation. There does not necessarily have to be a one-to-one correspondence between access rights and functions; one access right can control access to more than one function and one function could require more than one access right.

Access rights are represented as a vector of boolean values. The rights must be interpreted by the functions which operate upon the type of object that the capability designates. The access rights, therefore, only have meaning with respect to the type of object that the capability containing those rights designates. The number of different access rights that a single object can have is limited by the number of boolean values that can be associated with a single capability. This number is fixed for a given implementation of the system. A list of the types of objects in the system and the access rights associated with those types of objects is given in Table 2-3.

The function CREATE_CAPABILITY of the CAPABILITIES module returns a capability with a new unique identifier and with all its associated access rights set to TRUE. The function RESTRICT_ACCESS returns a capability with the same unique identifier as a given capability, but with fewer access rights than the given capability (i.e., some of the access rights that were true in the given

capability may be false in the returned capability). The function CREATE_RESTRICTED_CAP returns a capability with a new unique identifier and with a given set of access rights. These are the only functions that create capabilities. There is no function that can modify a capability.

There is one important property that must be maintained by all functions with respect to access rights. Assume that the presence of an access right in a capability is designated by the value TRUE in the appropriate boolean value and the absence of that right is designated by the value FALSE. If there are two capabilities, C1 and C2, for the same object (i.e., the capabilities have the same unique identifier), and that C2 has a subset of the access rights of C1 (i.e., all the boolean values that are FALSE in C1 are FALSE in C2 and some of the values that are TRUE in C1 may be FALSE in C2) then any function that completes without exception using C2 as an argument will also complete without exception using C1 as that same argument. In other words, C1 is always at least as powerful as C2. Recalling that RESTRICT_ACCESS is the only function that can create a new capability for an object that already has a capability and that the newly created capability must have the same or fewer access rights than the given capability, it is evident that a process cannot generate a capability for an object that is more powerful than those already accessible to the process. This result is important to the security of the system and to the proof of security of subsystems.

CAPABILITY STORE LIMITATIONS

One of the difficulties of any general capability-based system is controlling the propagation of capabilities. Any process can make copies of any capability it has access to, and can store these copies in any segment to which it has access. Such unrestricted propagation of capabilities is not always desirable. A feature has been included in the system to permit selected restriction on the propagation of capabilities. The restriction used is called the capability store limitation.

A capability store operation is an O- or OV-function in which the value of some primitive V-function is modified to return a capability. Capabilities can propagate only by means of capability store operations. Examples of capability store operations are the WRITE_SEGMENT function of the SEGMENTS module, the ADD_ENTRY function of the DIRECTORIES module, and the LOAD_GENERAL_REGISTER function of the REGISTERS module.

All possible instances of V-functions in user visible modules whose values can be capabilities are partitioned into objects, i.e., each capability returning V-function is assigned to some object. For example each possible instance of the H_READ hidden V-function of the SEGMENTS module is part of some segment object. Each object has a set of capability store permits associated with it. Each permit is a boolean value. If all the permits are TRUE, then any capability may be assigned to any V-function of the object (subject, of course, to other constraints). Similarly, each capability has a

set of store permissions. The store permissions are boolean values and have all the properties of access rights. In fact, capability store permissions are part of the access rights of each capability. If all the store permissions of a capability are TRUE, then that capability may become the value of any V-function of any object (again subject to other constraints). Each capability has a predetermined number of store permissions and each object has the same number of store permits. Each store permission has a corresponding store permit. The general rule regarding store permissions of capabilities and store permits of objects is that a V-function may take on a capability as a value only if, for each store permission of the capability, that store permission is TRUE or the corresponding store permit of the object of which the V-function is a part is TRUE or both are true. If both the store permission and the corresponding store permit are FALSE, then the capability store operation will generate an exception. For the simple case of one store permission for each capability and one store permit for each object, the following table summarizes the result:

		Capability Store Permission	
		TRUE	FALSE
Object Store Permit	TRUE	OK	OK
	FALSE	OK	EXCEPTION

The system supports one type of capability store limitation (the user can construct subsystems to support other types of store limitation if desired). The one type is termed the "process" store limitation. The purpose of the process store limitation is to guarantee that when certain selected capabilities are made accessible to a process, the process cannot make these capabilities or copies of these capabilities accessible to any other process. Such a limitation is useful if one user wishes to give access to some object to another user but wants to be assured that access will go no further. The process store limitation is also useful in the implementation of the Secure Object Manager in that the Secure Object Manager can give some process direct access to a segment and be assured that the direct access cannot be transmitted to some other process. The remainder of this section shows how the above purpose can be achieved using the capability store permissions and object store permits described above.

The store permits of an object are determined when the object is created and are fixed for the life of the object. When an object is created, the store permissions of the capabilities created to access the new object must be the complement of the store permits of the new object. For example, if a new segment is created with the process store permit being TRUE, then the two capabilities created to reference that object (the two capabilities are the one returned by CREATE_SEGMENT and the one put in the directory entry for the

segment) must have a process store permission of FALSE. This guarantees that a capability for an object with a particular set of store permits can only be stored in an object with at least those store permits.

The system is initialized with one directory, the root directory. The root directory's store permits are all FALSE. Therefore, no object with store permits of TRUE can be created in the root directory. This is because the capability for any object with store permits of TRUE would have some store permissions FALSE and therefore could not be stored in the root directory which has only FALSE store permits. In fact, no object with TRUE store permits can be created in any directory descended from the root directory for the same reasons. Since all user-created objects must be entered in some directory, how can a user object with store permits be created? Objects with store permits can be created only in a process directory. All process directories have the process store permit set to TRUE. The other objects in the system that have the process store permit set to TRUE are the process temporary segments and windows created by functions in the USER_PROCESS module and the registers of the REGISTERS module. All of these objects are private to some process, and are not shared and can not be shared with another process. Therefore, a capability stored in one of these objects is not accessible to more than one process.

In order to create a capability accessible to only one process, a capability must be created with its process store permission being FALSE. Such a capability can be created by invoking RESTRICT_ACCESS. This capability can be stored only in the registers, the process directory, objects with the process store permit that are descendant from the process directory, and the temporary segments and windows, and, therefore, is accessible only to that process. Any copy made of this capability must also have its process store permission as FALSE and, by the same reasoning, is accessible only to the same process.

PROTECTION OF HIERARCHICAL LEVELS

The operating system has several mechanisms for protecting its own implementation. It is necessary to protect the implementation from unauthorized modification by user programs (e.g., not allowing a user program to modify the unique identifier generating algorithm) and to protect parts of the system from unauthorized modification by users and other parts of the system (e.g., not allowing a program in level 10 to delete an arbitrary page of level 7). This section describes how the implementation is protected.

The functions that are implemented in hardware are protected physically. A hardware algorithm cannot be changed without physically changing the hardware itself. Functions that are implemented in software are implemented as procedures. The instructions for the procedures will reside on some form of storage, i.e. memory blocks, pages, or segments. This storage will be partitioned by level so that a particular memory block, page, or segment will contain procedures for only a single level. The

procedures will be protected because the capabilities for the memory blocks, pages, or segments that contain the procedures for a given level will be accessible to that level only. Therefore, no higher level procedure or user procedure will be able to read or modify the algorithms that implement the functions of a given level.

It is also necessary to control the accessibility of functions at different levels. For example, many functions are not accessible at the user interface. For software implemented functions, accessibility is controlled by distribution of procedure capabilities of the procedures that implement the functions. If a function is to be accessible to a procedure of a higher level or to a user procedure, then the capability for the procedure that implements that function is made accessible to the higher-level procedure. If a function is not accessible to a certain level, then no procedures in that level should have access to the capability for the procedure implementing the function.

The accessibility of hardware implemented functions also needs to be controlled. The most straightforward technique for controlling accessibility of hardware implemented functions is to require that each instruction present an authorizing capability as one of its operands. This technique requires that each instruction have an additional operand and that each procedure contain the capability for each of the instructions that it contains. These requirements are both cumbersome and inefficient. The technique adopted in the operating system is to have overlapping classes of instructions. Each class contains the instructions necessary to perform some useful set of operations. Each class has a distinct unique identifier. In order to execute an instruction, a capability with the unique identifier of a class containing the instruction must be presented. Rather than make this capability an operand of the instruction, the capability must be placed in a special register called the instruction class register. The algorithm for each hardware instruction must include a check to determine that the instruction is a member of the class associated with the unique identifier of the capability in the instruction class register. The contents of the instruction class register need be changed only when the class of instructions used by the executing procedure changes. All user accessible instructions are contained in a single instruction class.

DIRECTORY ENTRY LOCKS AND KEYS

The directory mechanism provides an additional protection facility to allow control of access to capabilities in individual directory entries. With each directory entry is associated a set of locks. Locks are simply slave capabilities. In order to access the capability in a directory entry, the calling procedure must present a proper key for that entry. A proper key is simply a capability whose slave is one of the locks associated with the entry. In order to get a capability from a directory entry it is necessary to present two capabilities: the capability for the directory with the "load" access right and a proper key for the entry. This permits the protection of individual entries or groups of entries.

TYPICAL USER ENVIRONMENT

The secure operating system presents to the programmer a very diverse collection of facilities. It is the purpose of this section to demonstrate how these diverse facilities can be used to create a good working environment for the user of the system. The demonstration will be by means of examples. The examples will be of modules that would constitute part of a general purpose programming environment. The purpose of the examples is to demonstrate how some of the system functions can be used, how a user environment can be constructed, and how a typical user can accomplish useful work on the system. The examples are not to be taken as the suggested or preferred way to accomplish a particular task, and the reader should not consider the examples to be part of the system. The examples provide only one possible alternative for use of the system. One of the great strengths of the secure operating system is that it provides a good base for many possible alternatives and ways of accomplishing the alternatives.

Three facilities are described:

1. Linker - The linker is a procedure that returns a capability for an object when given the symbolic path name of that object. Since all user objects must have a symbolic path name, the linker provides a means of getting to all user objects subject to access constraints.
2. User - The concept of a "user" is identified. Each user is associated with a particular initial process state. A user coordinator process is created that listens to selected I/O devices for requests to create a process for a user named in the request. The process is created with the initial state associated with the user. The new process is given the capability for the I/O device upon which the request was received. This facility permits users to log into the system and have a process created to execute their requests.
3. Interuser messages - Functions are defined to permit data to be sent to a message queue object associated with a particular user. Each user has an associated symbolic name. The messages may be read by a process created on behalf of the target user.

The specifications for these facilities is given in Appendix B.

LINKER

As stated earlier, all objects accessible to a procedure activation must be derived from either the arguments to that activation or capabilities in the procedure segment of that activation. It is often convenient to be able to reference objects by symbolic name rather than by capabilities. The linker provides one means of obtaining a capability for an object when given a symbolic name for the object. All user objects must have symbolic names because every user object must have a named distinguished entry in some directory. Since every directory is an object, it

must also have a named distinguished entry. The only exception to this rule is the single directory that exists when the system is initialized. This directory is called the root directory. All objects must have entries that are descendents of the root directory. An example of a directory structure is diagrammed in Figure 2-1. A capability for any object can be obtained by invoking the function GET_CAP of the DIRECTORIES module with a capability for a directory containing an entry for the object and the name of the entry. The capability for this first level directory can be obtained by invoking GET_CAP with a capability for a directory containing an entry for the first level directory and the name of this entry. The capability for the second level directory can be obtained from a third level directory and this process can be repeated until the directory containing the entry is the root directory. At any given point in time, a capability for a particular object can be obtained with a capability for the root directory and the list of entry names of all the directory entries that must be interrogated to finally get the desired capability. This list of names is called the path name of the object. Given a capability for the root, a path name of an object, and the capability for the procedure implementing GET_CAP, a capability for the object can be obtained by repeatedly invoking GET_CAP to obtain capabilities for the directories in the path and finally the capability for the object itself (1).

For example, a path name for the object numbered 3 in the example of Figure 2-1 is ALPHA, QED, GOOD. This path name uniquely identifies the object at this point in time. Since entries may be deleted and recreated, it is possible that at some future time the path name might indicate a different object. There may also be many path names for the same object. In the example the path names ALPHA, QED, GOOD; GAMMA, ALPHA, LOWER; and ALPHA, QED, UGLY, ALPHA, LOWER all refer to object 2. Path names are, therefore, not as unique in identifying objects as unique identifiers, however, it is this nonuniqueness that makes using names advantageous in some situations.

The LINKER is the procedure that obtains a capability for the object indicated by a given path name. The LINKER is used in conjunction with the compilers and assemblers of the system to permit symbolic references to objects in programs. Several conventions must be obeyed in the procedure segment generated by a compiler or assembler. Each symbolic reference is assigned a word in the template procedure record in the procedure segment. This word contains an integer indicating where the path name for the object can be found in the procedure segment. This integer also indicates where the keys necessary to unlock the entries in the path can be found in the procedure segment. A procedure activation of the procedure references the desired object by reading the contents of the word in the procedure record allocated to the object. The activation then uses the contents of this word as if it were a

(1) It is also necessary to have the keys to all the entries in the path and that the capabilities for the directories in the path have the "load" access right.

capability for the object. On the first symbolic reference to the object in a process, this will result in an exception because the word will contain an illegal address, not a capability. Upon encountering this exception, the procedure activation will invoke the RESOLVE_REFERENCE function of the LINKER with the integer and the capability for the procedure segment. The LINKER uses the capability for the procedure segment and the integer to find the path name and keys for the object. Since the procedure segment for the LINKER contains a capability for the root directory, the LINKER can obtain the capability for the desired object. The LINKER returns this capability to the calling activation. The activation then writes the capability into the associated word in its procedure record and reinvokes the function on which it took the exception. On all subsequent uses of this same reference in the same process, the word of the procedure segment will contain the desired capability and the LINKER need not be invoked. The binding of the symbolic path name to an object occurs on the first reference in each process. This linking technique is similar to that used in Multics.

It should be noted that the linker specification contains only V-functions, there are no O-functions or OV-functions. This means that the linker has only one state. The linker is simply a procedure with no memory. Any user can write a linker procedure. The system could have several different linkers. A capability for some linker procedure must be included in each procedure segment for procedures that will make symbolic references. A given procedure may use more than one linker or it may not use any linker. The latter case would occur if the procedure did not reference any other objects or if the capabilities for all objects that it references are embedded in the procedure segment. Having different linkers available is necessary if different types of symbolic referencing are desired. For example, some procedures might wish to use path names beginning at a directory other than the root directory. A slightly modified form of the linker described above could implement such symbolic names. Since capabilities for procedures that implement the system functions will be kept in some directory descendant from the root directory, system functions can be referenced symbolically using the linker described above as well as other linkers. If the only capability that is embedded in a particular procedure segment is one for a linker, then the linker can be used to control what objects activations of that procedure can access.

USERS

It is very useful to be able to associate a process within the operating system with a person or group of persons using the system. The concept of a "user" must be identified within the system, and some means of associating processes with users must be established. The functions of the user module accomplish this. Each user has a name. The only constraint on the name being that it be unique for each different user. With each user is associated initial process conditions. (Recall that the initial conditions of a process are specified by the capability for the initial procedure of the process

and the arguments to that initial procedure. Each process that is created on behalf of this user will have these initial conditions. These initial conditions should include capabilities for the objects that all processes of this user should have. The function CREATE_USER_PROCESS of the USER module creates a process with the initial conditions associated with the given user. The functions CREATE_USER and DELETE_USER create and delete users. A process can determine the user for whom it was created by invoking GET_USER_NAME. The function GET_DEVICE_CAP can be invoked by a process to obtain the capability for the device with which it is to establish communication with the user himself if such communication is desired (see below for example).

The following scenario illustrates how the above functions can be used to allow users to log into the system, i.e., create processes on their behalf, and use these processes to perform computations. There is a special process called the user coordinator. This process maintains a list of the names of all existing users together with the capabilities for those users. The user coordinator also has capabilities for a selected set of I/O devices that are all terminals or terminal like. The user coordinator sends commands to all of these devices to attempt to read from the terminal and to notify it, via wakeup, when something has been read. The user coordinator then waits for something to be read by waiting for a wakeup. When the wakeup is performed, the user coordinator attempts to read each device until it finds the one that contains data. Once it has read all the data up to a specified terminating character, it looks to see if the data matches a user name. If not, it ignores the data and starts from the beginning. If the data is a user name then the user coordinator creates a process for the user by invoking CREATE_USER_PROCESS. A temporary capability for the device is created by invoking the ASSIGN_DEVICE function of the VISIBLE_IO module and this temporary capability is included as an argument in the invocation of CREATE_USER_PROCESS. The new process will begin executing when START_PROC of the USER_PROCESS module is invoked. The initial procedure activation of the newly created process can obtain the capability for the device by invoking GET_DEVICE_CAP. The activation can then communicate with the user at the terminal. The initial procedure can verify that the user at the terminal is the proper one by requesting a password or series of passwords, by knowing that the device is accessible only to this particular user, or by any other means deemed appropriate. Once the identity of the user has been established, the initial procedure can begin accepting and interpreting requests of the user (an example is provided in a later section). There should be a procedure that performs the function of interpreting standard user requests. The initial procedure can invoke this request interpreter. By inputting requests to the request interpreter, the user can have the process perform desired computations. When the user completes his desired tasks, he can input a "log out" request to the request interpreter. This will cause the request interpreter to destroy the process.

INTERUSER MESSAGES

In order for processes to communicate with one another, they must share some object. If they share a segment, they can communicate by using the segment to pass data between them. If the two processes were created by the same process, the creating process can place capabilities for a shared object in both their initial states. However, it is useful to have a mechanism by which processes can communicate without deliberate prearrangements on their part. This is the purpose of the MAIL module.

The mail facility allows a user to send data to another user. Since each user must have a process to perform computations on his behalf, the mail facility can also be used to communicate between processes. The function SEND_MESSAGE sends the data, called the message, to a designated user. The target user is designated by name rather than by capability. This makes it possible for a user at a terminal to designate the target user. This would not be possible if capabilities were used because capabilities cannot be entered at a terminal. Names, however, do not provide protection as do capabilities. A separate protection mechanism has been provided. Each user, by invoking ACCEPT_MESSAGE, must indicate the other users from whom he is willing to receive messages. The function SEND_MESSAGE will cause an exception if the target user has not invoked ACCEPT_MESSAGE with the name of the sending user. REJECT_MESSAGE is invoked to reject subsequent messages from the given user. RECEIVE_MESSAGE is invoked to retrieve messages that have been sent to a user. Messages are retrieved one at a time on a first in/first out basis. Each received message contains the name of the user who sent the message. This allows the receiving user to accurately determine who sent the message.

The three facilities described above are all quite simple to specify and implement. In a particular instance of the secure operating system, these facilities could be modified and augmented to fulfill specific needs of users. These examples have been presented to illustrate how the operating system can be used. In an actual implementation of the operating system, many more facilities and more elaborate facilities will probably be helpful in aiding users in accomplishing their tasks.

EXAMPLE OF SYSTEM USE

In order to illustrate how the system might be used in a general purpose way by a person communicating via some terminal, a sample session follows. It is assumed that the person is known to the system. The login procedure was discussed in the section entitled USERS. The person enters his user name to the user coordinator process which is initially listening to the I/O device. The user coordinator process then creates a process for the user. The initial procedure for the new process is a user request interpreter and overseer. The user request interpreter may first wish to validate that the user is whom he claims to be. This might be done by requesting a password be typed or that an encoded card be inserted into the terminal, by more sophisticated techniques such as handwriting, fingerprint, or speech recognition, or by a simple

technique such as stationing a guard next to the terminal who assures that each person enters his own name. The request interpreter must obtain a capability for the user's I/O device by calling the procedure GET_DEVICE_CAP. In order to call GET_DEVICE_CAP the request interpreter must get a capability for it by calling the linker. It is assumed that the request interpreter has a capability for the linker in its procedure segment. The linker must contain, in its procedure segment, capabilities for all the procedures that it will invoke. Once the request interpreter has a capability for the procedure for GET_DEVICE_CAP, it can retain this capability for future use in its procedure record. Similarly the capability for the user's I/O device can be retained in the procedure record. In the remainder of this description it is assumed that on first call to any procedure from some other procedure within this process the linker is invoked to obtain a capability for the called procedure. This capability is then retained in the procedure record of the calling procedure for future calls.

The request interpreter might now output a message requesting a password to the user's I/O device by calling SEND_COMMAND and WRITE_DEVICE and can similarly input the response. After validating the password by comparing it to the correct value known to the request interpreter (for example, the correct value might be passed to the request interpreter as an argument by the user coordinator process), the request interpreter can acknowledge receipt of the password and accept a request from the user. Assume that the user wishes to input, compile, and execute a program. To do this the user enters the name of some editor. The editor may be known to the request interpreter in which case it may have a capability for it or the request interpreter may link to it. If the editor is not explicitly known to the request interpreter, a capability for it may be found by looking in the system library. The system library is some directory containing capabilities for procedures of general interest to users. The request interpreter has a capability for the directory containing the system library and can obtain a capability for the editor by calling GET_CAP. The editor, once called by the request interpreter, can create a temporary segment to temporarily hold inputted text. Once editing is complete, the editor can transfer the text to some permanent segment specified by the user and delete the temporary segment.

The user can then enter the name of a compiler procedure. When the compiler procedure is invoked by the request interpreter it will generate a procedure segment containing the object code for the indicated program. The procedure segment must also contain the linking information and a capability for the linker (if the new procedure is to link to anything).

The new procedure is now invoked by the user by entering the path name of the directory entry containing the procedure capability. The request interpreter extracts the procedure capability from the directory and calls the procedure (it is assumed that procedures invoked in this way by the request interpreter are called at offset zero). The new procedure is then executed. The

procedure can link to any other procedure because it has a capability for the linker, it can communicate with the user because it can obtain the capability for the user's I/O device by calling GET_DEVICE_CAP, and it can create temporary objects by obtaining a capability for the process directory. If the user desires that some program he is compiling run in a restricted environment without access to all the facilities of the system, the compiler does not place a capability for the linker in the procedure segment of the compiled procedure. The compiler may, instead, embed in the procedure segment a capability for a special linker that will link to only certain restricted procedures or the compiler may "prelink" the newly compiled procedure directly to the other procedures the new procedure may call by embedding the capabilities for those other procedures into the new procedure segment.

When the user has finished his task, he terminates the session by requesting a logout. Logout can be accomplished by having the request interpreter send a message, via the mail facility, to the user coordinator process. The user coordinator process can then stop and destroy the user's process. This sample session illustrates only one of many possible ways the PSOS might be used to carry out tasks for users.

SYSTEM INITIALIZATION

In order for an operating system to be correct it must be properly initialized. A system is properly initialized if its state is consistent with the initial state prescribed by the specifications of the system. In order to define a correct system initial state, some terms must be defined. A level is in a correct initial state if the values of all the V-functions of the level are consistent with the initial assertions of the specifications of the V-functions. A level is in a consistent state if the state can be achieved by starting with the level in a correct initial state and invoking O-functions and OV-functions in some order. A correct initial state is a consistent state.

From these definitions it is possible to define a correct initial state for a system. A system is in a correct initial state if the topmost level of the system is in a correct initial state and all other levels are in consistent states. Note that by this definition the lower levels need not be in an initial state for the system to be in an initial state. It is expected that the initial state for a given level will be constructed by invoking O-functions and OV-functions of lower levels that have already been initialized. Therefore, the lower levels will no longer be in their initial state once the topmost level has been initialized. A module that is included as part of the specification of many levels might, therefore, assert a different initial state at each level. For the sake of brevity, each module in the specifications of Appendix B asserts only the initial state of the lowest level of which it is a part. A complete specification should include separate assertions of initial conditions for each level at which a module resides.

All operating systems must have some procedure or procedures for taking the system from the non-functional or powered-down state to the system initial state. These procedures may be manual or automatic in software or hardware or some combination of these. From the point of view of provability, the only important property of these initialization procedures is that they place the system in a correct initial state. There are no preconditions on the initialization procedures, only post conditions. Although the methodology does not in any way restrict how the initialization procedures can go about achieving the correct initial state, there is one technique that is straightforward and in keeping with the general philosophy of the methodology. This technique is to perform initialization by levels, beginning by initializing the lowest level, and successively initializing the next higher level until all levels are initialized. Each level is initialized to a correct initial state for that level. Then the initialization of the next higher level can be performed using only the functions of the level just initialized. All that need be proved is that each level is brought to a correct initial state. Since, once a level is initialized, only functions of that level are used by the next higher level, it is clear, by definition, that the level will always be in a consistent state. Once the topmost level is brought to a correct initial state then system initialization is complete. This

technique is in keeping with the general philosophy of the methodology because the specifications, implementation, and proof of initialization can be done one level at a time. The proof that system initialization is correct follows directly from the proofs that initialization of each level is correct.

For those levels that are implemented as hardware, initialization consists of setting all the registers of that level to their proper initial value. The algorithms for the hardware are permanently instantiated into the hardware circuitry. For those levels implemented at least in part as software, it is necessary to first instantiate the algorithms that implement the functions of that level by loading programs into memory from some I/O device and then initializing data bases to their proper initial values.

SHUTDOWN AND REINITIALIZATION

It is sometimes necessary to temporarily return an operating system to its non-functioning or powered-down state so that the equipment may be maintained as required or used for some other purpose. When the system is returned to service, it should be in essentially the same state as when it left service. This temporary interruption of service is accomplished in two phases. The first phase of bringing the system from an operating state into a non-functioning state is called shutdown. The second phase of bringing the system from a non-functioning state back to its previous operating state is called reinitialization.

The shutdown procedures must bring the system to a non-functioning state in such a way that all the information necessary to restore the system to its previous state is preserved in storage. The storage media chosen for preserving this information should be logically separable from the rest of the operating equipment of the system, so that it is not possible that this information could be accessed or modified in any way during the non-functioning period. This is necessary to preserve the security of the system and to guarantee proper restoration of the system state at reinitialization. The reinitialization procedures must use the information preserved during shutdown to reconstruct the state of the operating system. The state of most operating systems is generally very complex requiring the preservation of large amounts of information. However, it is usually the case that much of this information is already on storage media suitable for preservation and does not have to be moved during either shutdown or initialization. For example, most of the state information of the operating system described above resides in segments. Segments are made up of pages and most pages in the system reside on secondary storage devices such as disks. The disks are likely to be a suitable medium for preserving state information, and so the bulk of the state information for this operating system need not be moved.

In keeping with the hierarchical structure of the methodology, both shutdown and initialization can be accomplished in a

hierarchical fashion. Each level is responsible for shutting itself down and reinitializing itself. Shutdown is begun by invoking an O-function called "shutdown" at the topmost level of the operating system. This function does the necessary preserving of state for that level and, as its last action, invokes the shutdown O-function of the next lower level. This process repeats through all levels down to the lowest level. When the lowest level has completed preserving its state, its last act is to put the system in a non-functioning state. Reinitialization proceeds much like initialization. The lowest level is reinitialized and then each succeeding higher level is reinitialized using the functions of the newly reinitialized next lower level. Rather than bring each level to a correct initial state, each level is brought to the state it was in when the shutdown O-function of that level was invoked. In order to restore this state, the reinitialization procedure uses the state information preserved by the shutdown procedure of the level. When reinitialization of a level is complete, the level returns to the next higher level as if it were returning from the shutdown O-function.

To the next higher level, this temporary loss of service looks like a simple O-function invocation. Since the reinitialization procedure for each level returns that level to the same state it was in when the shutdown O-function of that level was invoked, the shutdown O-function has no effects (a rather strange O-function). The specifications of all shutdown O-functions have no effects, although they may have arguments and they may have exceptions. The proof of the implementation of shutdown and reinitialization involves proving that for each level, reinitialization restores the state that existed when shutdown was invoked. In other words, the combination of the shutdown and reinitialization procedures has no visible effects. Shutdown and reinitialization are completely specified by the shutdown O-function of each level. Since the shutdown O-function is part of the specifications, its design is proved secure as part of the proof of security of the specifications.

The implementation of shutdown and reinitialization can be made easier by restricting shutdown to being invoked only in certain system states. By carefully choosing the allowable states at which shutdown can occur, the amount of state information which has to be preserved can be minimized. For example, it might be reasonable to permit shutdown only when a single user process exists in the operating system described above. In this case it is only necessary to preserve the state of one process. Before attempting shutdown, the system must destroy all the user processes but one. Clearly, the restrictions on shutdown should be carefully chosen. For example, it would be unreasonable to have the restriction that no user segments exist at shutdown. This restriction defeats one of the main purposes of the system, i.e., long term retention of stored information.

FAULT DETECTION AND RECOVERY

One objective of this study is to design a system that is provably secure. All proofs, however, are based on assumptions. In the case of PSOS, one of the assumptions is that the primitive functions of the specifications, i.e., those functions which are not implemented in terms of lower level functions, can be correctly implemented. Unfortunately, due to fundamental physical properties of hardware, the actual realizations of the primitive functions can, at best, be only close approximations to the function specifications. If the system is to be secure, it must compensate for such incorrect behavior in the realization of the primitive functions. This is the purpose of recovery.

It is usually impossible to enumerate all those deviations from the specifications which could possibly occur in a particular realization. Fortunately, in any reasonable realization most possible deviations have such a low probability of occurrence that they are unlikely to occur within the lifetime of any implementation. For a given realization it should be possible to enumerate all potential deviations from the specifications which are likely to occur within the lifetime of the system.

In order for the system to be able to deal with a deviation, it must be able to detect the deviation. One means by which the system may detect a deviation is by observing that some level is in an inconsistent state. For example, on an attempt to read the contents of a word of a segment, the implementation of the segment level would invoke the page read function on a particular page. If invoking the page read function caused an exception to occur because the page did not exist, then the segment level has detected an inconsistent state (assuming a particular implementation) because only the segment level can create and delete pages and should never cause that particular exception. The occurrence of an inconsistent state implies that a deviation has occurred in some primitive function, for, given the system will have been proved correct assuming correct realization of primitive functions, it is only within these functions that the deviation could have occurred.

It is, however, quite possible that a deviation occurring in a primitive function could cause the system to incorrectly change from one consistent state to another consistent state. For example, a deviation in the memory level could cause the contents of a word of memory to change. If this memory word is part of a page which is part of a user segment, then the system will still be in a consistent state at all levels. Since the system is still in a consistent state, the deviation cannot be detected from the system definition alone, the system requires additional information to detect such a deviation. This additional information must be a redundant form of the state information already present. The amount of redundancy may vary from minimal, i.e., a parity bit on a word of data, to complete duplication of the information, to many duplicates of the information. There are many possibilities between these extremes. Different pieces of information may necessitate different

amounts of redundancy. The amount of redundancy desired should be determined by weighing the value of the information and the probability that the information will be lost due to a deviation, against the cost of maintaining the redundant information. Once a deviation has been detected, an attempt must be made to correct the cause of the deviation and to restore the system to the state it would be in had the deviation not occurred. Unfortunately, the detection techniques discussed above do not necessarily aid in determining what the deviation was or the component that caused the deviation. Determining what the deviation was and the component at fault is diagnosis. Restoring the system state to what it would have been had the deviation not occurred is recovery.

Neither diagnosis nor recovery can be 100% reliable. Both the diagnosis and recovery algorithms can, at best, provide a partial solution to the problem they are attempting to solve. However, if the cases which are covered by the partial solutions are the ones which are likely to occur within the lifetime of the system, a very high degree of reliability can be obtained. Because both the diagnosis and recovery algorithms are highly dependent upon the particular implementation chosen for a given system design, a meaningful set of algorithms for these purposes is not given in this document. However, some general principles and specific examples are given below.

Diagnosis of deviations could be performed at many levels of the system. Some diagnosis can be incorporated into the hardware directly. For this purpose some functions may be added to the hardware levels of the system design in order that the hardware diagnosis can be invoked by system software, and results of diagnosis can be returned to the software. Some diagnosis algorithms may be incorporated in the lowest levels implemented by software. This is necessary because certain types of deviations can render most of the software of the system incorrect. For example, if the add function ceased to work properly, much of the software of the system would operate incorrectly. However, diagnostic programs that are carefully written and not dependent upon other software or the add instruction can diagnosis a deviation such as this. Some diagnosis programs may be incorporated at higher levels of the system. For example, some deviations may not be diagnosable from a single symptom. However, the deviation may be recognizable by observing a pattern of symptoms. A high-level program could record all symptoms which have not already been diagnosed and look for patterns that may identify the deviation. Any diagnosis algorithm implemented on some level of the system which has a visible effect at some higher level must be included in the system specifications as part of the design. Since the diagnostic functions are intended to be included in the specifications, they will be proved secure as part of the design security proof and there is no possibility of the diagnostic functions creating a security violation.

In order for recovery algorithms to be able to restore the proper state of the system after a deviation, state information must be stored redundantly. The redundant information used for detection of deviations and the redundant information used for recovery may or

may not be the same. As with detection, the extent to which a given piece of information is maintained redundantly is dependent upon the probability that the information may be destroyed, the value of the information, and the cost of maintaining the redundant information. Techniques for recovery will be illustrated by examples, followed by general principles for designing recovery into systems using the methodology.

One type of deviation that occurs in many types of memories is that one bit of a memory word will be modified with no apparent cause. This type of deviation from the specifications can be detected by adding to each word of memory a parity bit, a single bit of redundant information. Using the parity bit, the system can detect a spurious modification in a single bit of each word of memory. Many techniques have been developed for recovering from such errors within the memory itself. By adding some additional bits of redundant information to each word of memory, the original contents of the word can be reconstructed with a high degree of reliability when a hardware failure occurs. In general, the greater the number of redundant bits, the greater the reliability of the recovery. The advantage of such techniques to the methodology is that the correction takes place at the same level as the failure and the failure is totally invisible outside the level. No modifications to the specifications are necessary.

If, however, the correction is not implemented at the same level as the failure, the failure will be visible outside the level and a deviation from the specifications can occur. It is, therefore, necessary to incorporate the detection of the deviation into the design of the level at which the deviation occurs so that higher levels can detect the deviation and act upon it. The most common way detected deviations will be reflected in the specifications is through exceptions. In the case of the memory parity error, the function block_read should have an additional exception:

```
parity_error(u, i);
```

Because deviations are caused by events not naturally describable in terms of the specification language, their description in terms of the language must be artificial. To describe the parity error exception requires a new V-function and a new O-function in the memory module. The new functions are defined as follows:

```
VFUN parity_match(slave_capability u; INTEGER i)
     -> BOOLEAN b;
    $( true if parity of word i of block u is correct )
    HIDDEN;
    INITIALLY b = TRUE;

OFUN cause_parity_error(capability c; capability b;
                        INTEGER i);
    $( set parity bit for word i of block  b )
    EXCEPTIONS
        no_parity_capability(c);
        no_block(get_slave(b));
```

```
        EFFECTS
            'parity_match(get_uid(b), i) = FALSE;
```

The exception parity_error can now be defined as:

```
    parity_match(u, i) = FALSE;
```

This definition for parity errors assumes that there is some special processor which invokes the function cause_parity_error. The first argument of this function is a special capability for causing parity errors possessed only by this special processor. The exception no_parity_capability assures that only the parity processor can cause parity mismatches. Of course, the special processor and the function cause_parity_error are not explicitly implemented in the system, they are simply a convenient way to describe a phenomena that is not otherwise describable in the specifications.

Consider now two methods(1) of recovering from a memory parity error. These are not the only methods, but are just two possible solutions. The first method involves those memory words which are used for paging, i.e., memory words that are constituents of pages. Since a single parity bit per word provides only a detection mechanism and is not sufficiently redundant to aid in recovery of data, some additional redundancy is necessary. Assume that the page level implementation maintains two copies of each page residing in primary memory. Each invocation of the function page_write causes both copies to be updated. When the function page_read is invoked, an attempt is made to read one copy of the word of memory. If this attempt is unsuccessful due to a parity error exception, then the other copy of the word of the page is read. It is assumed that the probability of parity errors occurring in both words is sufficiently low as to be very unlikely to occur within the lifetime of the system. (If one wished to account for the possibility of having a parity error on both copies one could reflect this event as a parity error exception of the page module).

The main disadvantage to the above method is its cost. The method consumes twice the required memory and necessitates twice the number of memory references on each write as the nonrecoverable situation. A less costly solution is more desirable. Consider now a method implemented at the segment level. Two copies of each segment are maintained by the segment level, however, rather than updating both copies at each write, one copy is updated less frequently. For example, one copy may be updated only at specific time intervals if a write operation on the segment has occurred in the interval. This copy can be stored on a less expensive storage medium than the original, e.g., on magnetic tape.

This second method is much less costly than the first method. Even though the second method requires at least two copies of each

(1) Note that these methods are presented solely for the purpose of illustrating some of the difficulties encountered in recovery techniques. The discussion is superficial and does not constitute a recommendation for or against the use of the two methods.

word of memory, one of these copies can be on an inexpensive storage medium. Also the second method does not require two memory writes for each page on each segment write. The main advantage of the first method is that parity errors and parity error recovery are totally invisible above the paging level (assuming that the possibility that parity errors can occur on both copies of the same word is negligible). The specifications for the paging level are unchanged by the detection of and recovery from parity errors. This is unfortunately not the case with the second method. Because copies are not made on each segment write operation, full duplication is not maintained and it is possible that a read of a word of a segment that encounters a parity error upon referencing the most recent copy of that word, would not be retrieving the value supplied in the last invocation of write on that word. The value so retrieved might be a previous value which has since been modified. This previous value may be inconsistent with the current state of the segment. Even if the entire contents of the segment is retrieved, there is no guarantee that the saved copy is consistent because the copy may have been saved in the middle of an operation. In this case the read function is actually causing a change of state to the system and is therefore an OV-function rather than a V-function. Clearly, in this case, some change to the specification of the segment level is necessary if the specification is to be consistent with the implementation. This may be accomplished simply by redefining the read function, or, in order to preserve the intuitive meaning of read, it may be more desirable to reflect the parity error as an exception in the read function and introduce a new O-function which recovers the state of the word from the most recent copy. This new O-function would require write access in the capability for the segment and would therefore preserve the intuitive meanings for read and write access. The important property is that whatever aspects of detection or recovery are observable at any level must be included in the specifications of that level. Since the specifications are to be proven secure, it is assured that the design of detection and recovery is secure. In order to minimize the impact of recovery upon proof it is advantageous to make recovery invisible as in the first method above. This may, however, lead to system designs which cannot be implemented in a cost effective manner. Some balance between cost of implementation and cost of proof must be attained.

Recovery methods do not necessarily have to be applied uniformly to all objects of a given level. For example, some segments may be considered more critical than others, and it might be most effective to apply the first recovery method above to the pages of these critical segments. Also, some segments might be critical for only a short period, and it might be advantageous to apply the first method to the pages of these segments for only the short period.

This nonuniform approach to recovery is particularly useful in PSOS where the primary goal is security. Not all algorithms and data in the system are essential to the security of the system, and it is, therefore, reasonable to apply superior detection and recovery techniques to those parts of the system that are critical

to security. For example, the correctness of a capability is, in general, more critical to security than the correctness of other forms of data and might be represented in a more highly redundant form to assure its correctness. Similarly, the algorithms that implement the capability module might be implemented redundantly, for example, as triply redundant hardware with voting.

The above paragraphs provide a few examples of how recovery might be accomplished under certain circumstances. These examples are not intended to represent optimal strategies in any sense, for it is not possible to fully evaluate a recovery strategy without detailed knowledge of the system implementation. The examples are meant to illustrate some of the tradeoffs in recovery techniques and to demonstrate how the recovery techniques are incorporated into a system design using the methodology.

Some general principles of detection of and recovery from deviations can now be stated:

1. Hardware failures can, in general, occur at any level of the system. Detection of and recovery from such failures can take place at the level of the failure or at any level higher than the failure or some combination of these. A failure need not be dealt with at any particular level, but may be dealt with at the level or levels that the designer deems appropriate. In the examples above, a failure at the memory level is recovered from at the memory, page, and segment level.

2. Each level of the system may provide functions which permit implementations of higher levels to check the consistency of the given level or to cause the given level to recover to a consistent or proper state. These types of functions may be necessary because some level may detect an inconsistency in its state which may be due to an inconsistency in the state of some lower levels that have not been detected by the lower levels.

3. To promote the simplest design, it is desirable that deviations be made invisible at the lowest possible level, i.e., detection and recovery are implemented at this low level. However, this goal may be incompatible with cost effective implementation.

4. The most important principle is that all effects observable at any level be included in the specifications. The design can then be proven secure. This principle is, of course, applicable to all areas of system design, not only recovery.

Certain tests for consistency are too costly to be performed dynamically, i.e., to be performed routinely as the system is running. Consider, as an example, the case of a capability that is about to be deleted. The user object manager guarantees that there will remain at least one capability for the object whose capability is to be deleted. However, checking that this is actually true is a

very costly operation requiring a search of all directories in the system for this other capability. Examples of other expensive consistency checks include assuring that a given page is part of only one segment and that CREATE_CAPABILITY returns a capability with a unique identifier that does not appear in any existing capability. Such consistency checks may be tried on an occasional basis to try to detect otherwise undetected deviations. In addition these consistency checks can be used to aid in restoring the system to a consistent state once it has been determined that there is an inconsistency. These checks might also be tried in the case that an inconsistency is suspected. For example, a user might discover that a segment to which only he should have access contains data that he did not place there. Running these consistency checks might disclose that the segment in question contains a page that is also in another segment. It may be impossible to determine what the proper consistent state should be. In a secure system, if the proper consistent state is unknown, a conservative rectification of the inconsistency is desirable. If unauthorized release of information is paramount, an extreme solution might be to destroy the system. A more moderate solution would be to destroy the page in question. If the system is properly implemented, the occurrence of such situations will hopefully be extremely rare, however if proper operation of the system is critical, it is necessary to develop policies to handle these situations and to design the system to aid in detecting and rectifying such inconsistencies. For example, in PSOS prohibiting capabilities from being written into user segments could make the detection and correction of certain inconsistencies related to capabilities much easier. This prohibition is a significant restriction in the functionality of the system, but it may be worthwhile in certain applications where security is crucial.

None of the techniques of detection and recovery described above is new or unique to the design and proof methodology. However, the methodology does enhance the usefulness of these approaches by making the conditions that define a consistent state of the system explicit and precise. Also the methodology, i.e., the combination of specifications, mapping functions, and programs, makes it straightforward to determine the effect upon the system interface or any internal interface of a particular deviation. This is essential to determining which deviations can have undesirable effects, what the cost of a deviation is, and the best means for recovering from the deviation.

IMPLEMENTATION CONSIDERATIONS

Although this report does not include a specific implementation for the system described above, it is important to consider selected issues relating to the implementation of the system. Some of the decisions made in the design of the operating system are derived from knowledge of how the system could be implemented. Also the choice of implementation can seriously effect the efficiency and verifiability of the implementation. The following sections relate to specific topics related to software and the hardware/software interface. Much of the implementation of PSOS is conventional, i.e., there are existing systems that reasonably implement many of the features of PSOS. There are several aspects of the implementation that are fairly unique to PSOS. Two such aspects, table management and procedure invocation, are now discussed.

TABLE MANAGEMENT

Much of the work of any operating system regards the management of the resources of the system. Such management of resources is accomplished through the maintenance of data bases or tables which describe the state of the resources being managed. In those cases where resources are utilized frequently and the utilization of the resources requires reference to the tables associated with the resource, effective table management becomes an important factor in the overall efficiency of the system.

PSOS introduces a new wrinkle on the table management problem by its use of capabilities to reference objects. In other systems, references to frequently utilized resources is made via integers (e.g., segment numbers, channel numbers, file numbers, device numbers). Such integers are typically indices into tables used to manage the type of resource being referenced. Locating the table entry corresponding to a given resource is accomplished by a simple arithmetic calculation involving the known beginning of the table and the integer used to reference the resource. In PSOS, all references to resources (objects) are made via capabilities. The "integer" identifying the resource being referenced is actually the unique identifier in the capability. Unfortunately, if the unique identifier is used as a simple index to reference an entry in a table, all tables would have to be very large and the meaningful entries in the table would be very sparse. This is because the number of unique identifiers is very large and the unique identifier for each virtual resource is unique for all time and not reused. It is necessary to have a very efficient means of mapping a given unique identifier into the table entry corresponding to the resource (or perhaps into the resource itself).

Consideration of this problem is divided into three cases:

1. perpetually existing resources,

2. resources created during initialization or reinitialization of the system, and

3. resources created while the system is in normal operation.

Perpetually existing resources mainly consist of the physical resources of the system such as primary memory, processors, I/O devices, and the clock. These devices can be assigned permanent unique identifiers. Because these unique identifiers associated with perpetual resources can be determined before the system is constructed and because these identifiers are unchanging, a judicious choice of identifiers can make the process of mapping them into the actual resources very efficient. For example, the unique identifiers for I/O devices could consist of two parts: one part being a code that indicates that this identifier corresponds to an I/O device and the other part being the physical address of the device. An operation referencing an I/O device via such a unique identifier simply checks the first part to make sure the identifier does correspond to an I/O device and then uses the second part to make the actual reference. All the unique identifiers for the physical resources can be assigned in this manner. Of course, the algorithm that generates new unique identifiers for newly created capabilities must never generate any of these predetermined unique identifiers.

The best examples of resources create during initialization or reinitialization of the system are the system procedures. Operating system functions are invoked by using a capability for a system procedure as an argument to the call operation. The system maintains a table of all system procedures and, in order to perform the invocation of a system procedure, the call operation must locate the entry in this table for the system procedure corresponding to the unique identifier in the given capability. All system procedures are created during system initialization. If the unique identifier corresponding to a system procedure could somehow be directly related to the location of the entry for that system procedure, then the call operation could be simplified and made more efficient. One way to accomplish this would be to generate the unique identifiers for all the system procedures in the sequence their entries appear in the table, e.g., the system procedure appearing in the fifth table entry would have the fifth unique identifier in the sequence of unique identifiers generated for system procedures. If the call operation knows the first identifier in the sequence, and from this can easily determine the relative position of all succeeding identifiers, then the location of the table entry can easily be determined. Of course, this technique assumes that the sequence of unique identifiers generated by the unique identifier generating algorithm can be easily determined. This is true of the simple integer successor algorithm, but is not true for all algorithms.

Resources created while the system is running present more difficult problems. The most obvious way a representing and efficiently accessing a large and sparse table such as a table of unique identifiers is via hash tables. It is expected that all system tables indexed via unique identifiers will be hash tables. Although there are a wide variety of hashing technique and hashing algorithms that may be applicable, it is expected that one or two

techniques and algorithms will be chosen in order that they may be implemented efficiently in hardware or firmware. Consider, for example, the table that is used to implement the get_pr_address function of the procedure_records module. This table is indexed by a procedure (segment) unique identifier. Efficiency might dictate that this lookup be done by single machine instruction, requiring that the hashing algorithm be implemented in hardware.

Consider now the table that is used to implement the read_device, write_device, send_command, and receive_status functions of the visible_io module. This table is indexed by a virtual I/O device unique identifier and each entry contains the capability associated with the physical I/O device. If very efficient I/O is desired to support certain applications, then simply implementing the hashing algorithm in hardware or firmware may not be efficient enough. There is still a necessity for at least two primary memory references for each table reference: one to compare the referencing identifier with the entries identifier to ascertain that the correct entry has actually been found, and one to retrieve the capability for the I/O device. These memory references can be eliminated in the majority of references and the hashing eliminated by the use of a set of buffer registers that can be addressed associatively. Because such associative memories are expensive and cannot operate as quickly for a large number of entries, the number of entries is kept small and only a few entries of the table are stored in the associative memory. These entries are typically the most recently referenced entries. A mechanism must be constructed for adding entries to and removing entries from the associative memory in a manner that keeps the most recently used entries in the associative memory. It has been demonstrated that using such associative buffer memories can eliminate most of the references to primary memory necessary for table lookups.

The table necessary to implement the functions read_segment and write_segment of the segments module introduces a further complexity. Clearly, these functions must be implemented in a highly efficient manner for every machine instruction invokes one of these functions at least once (the read_segment necessary to read the instruction itself). The problem arises because the table that is used to translate segment unique identifiers into page capabilities is likely to be very large because it has one entry for every segment in the system. This table will likely be stored in secondary memory necessitating long delays for each reference. The use of an associative memory as described above would help significantly. However, when a reference is made to a segment whose entry is not in the associative memory, a long delay would still be incurred. In order to obtain sufficient overall efficiency, the number of references to secondary storage must be a very small fraction of the overall references. It is not cost effective to build an associative memory large enough to reduce the number of secondary storage references to a sufficiently small fraction of the overall number of references. This problem can be solved by using a two level buffer strategy. The associative memory is still used, however a second buffer is kept in primary memory. The number of entries kept in primary memory can be much larger than the number of

entries in the associative memory, but will still be much smaller than the total number of entries in the table. An attempt is made to resolve each reference to the table first using only the entries in the associative memory, if this fails then the primary memory entries are used, and only if this fails is secondary storage referenced. If the most recently used entries are kept in the associative memory and primary memory, then references to secondary storage for table entries should be very infrequent.

All the methods for table management described above require very close cooperation between hardware, firmware, and software. In the case of the table necessary to support the implementation of segments, the associative memory and the algorithms that access the associative memory are likely to be implemented in hardware, the algorithms for adding entries to and removing entries from the associative memory are likely to be implemented in firmware, and the algorithms for adding entries to and removing entries from primary memory are likely to be implemented in software. This implies a great deal of coordination between the hardware, firmware, and software, with much information accessible to all modes of implementation. Such information as the location and size of many system tables will have to be accessible to hardware firmware and software.

PROCEDURE INVOCATION

The mechanism for procedure invocation is another area where close cooperation is necessary between hardware, firmware, and software. This is true both because the invocation mechanism requires hardware assistance, the hardware algorithms are dependent on software maintained tables, and many of the functions supported by the system are implemented as software procedures and these procedures must properly utilize the invocation mechanism in order to assure proper operation of the system and maintain the security of the system.

Each activation of a procedure has associated with it an activation record. The activation records contains storage for the values of variables local to the procedure activation. The activation record should also contain control information necessary to support the control structures of the system and the programming language.

An activation record is created whenever an activation is created (i.e., when a procedure is invoked via a call operation). The address of the activation record is placed in the stack register by the call operation. The call operation also returns an address that is passed as an argument to the return operation when the activation is to be terminated. This address is used by the return operation to return to the invoking activation. In a particular implementation this return address will likely be placed in a designated register by the call operation.

It is essential that procedures obey certain conventions at procedure invocation and return in order to maintain their

protection. If procedure P1 wishes to invoke procedure P2 in a manner such that procedure P2 cannot interfere with P1 then procedure P1 must do the following before executing the call operation on P2:

1. P1 must create a segment or window containing only the arguments for P2. The address for this window will be passed to P2.

2. P1 must clear all the registers containing information that P1 does not want P2 to have access to. P1 may wish to save the contents of the cleared registers before clearing them. The hardware might provide an instruction that saves the contents of and clears specified registers to make this step more efficient. (Note that the instruction class register is always effectively cleared, i.e., set to minimum privilege, on a call, but P1 may wish to save its value for later restoration.)

3. P1 must store the address of the instruction to be executed upon return at some location established by convention. This location will be at a fixed offset relative to the return address passed to the call operation. The saved contents of registers must also be placed at some predetermined offset from this return address in order that the contents of the registers can be restored.

P1 must execute a protected call so that a new segment is created for the activation record of P2 and so that the return address is protected from access by P2.

If procedure P2 wishes to assure that it is invoked by any procedure in a manner such that the invoking procedure cannot interfere with the operation of P2 then the following must be done:

1. All capabilities for the procedure (segment) P2 must not have the "read" or "write" access right and must have the "call" access right. This guarantees that P2 can be invoked only in a protected manner (will have a new segment created for its activation record) and that P2 itself cannot be modified.

2. Before executing the return operation, P2 must clear the registers containing values it does not want returned to its invoking activation.

Using these conventions, the caller can be protected from the callee, the callee can be protected from the caller, and the caller and callee can be mutually protected. Since there is some overhead involved in a protected call, an unprotected_call operation has been provided for those invocations where neither the caller nor callee requires protection.

The call to a system procedure operates somewhat differently. This is because within the system, the caller never needs protection from the callee. Also, some system procedures do not reside in

segments. Unlike capabilities for user procedures, capabilities for system procedures are not segment capabilities. If a call operation with a capability for a system procedure is attempted, the call operation recognizes the capability as being for a system procedure and determines the address of the system procedure by reference to the system procedure table. The system procedure table also specifies the location for the activation record for the procedure and the contents of the instruction class register for the procedure. The latter is necessary because the instruction class determines whether addresses are interpreted as segment addresses or memory block addresses. If a system procedure has a capability for another system procedure within the same system level, then that capability may contain the "read" access right. This allows the two procedures to share the same storage space for their activation records. If a system procedure has a capability for another system procedure at a lower system level, then that capability will have the "call" access right, but not the "read" access right. This guarantees that the two procedures will not share the same storage space for their activation records.

An attempt has been made in the design of the call, unprotected_call, system_call, return, and system_return operations to make usage and conventions as uniform as possible. This makes writing a compiler for a programming language much simpler because the compiler does not have to know what kind of procedure it is calling or whether or not the call is protected.

It is expected that the call and return operations will be implemented in firmware. This is necessary for both conceptual (otherwise how does one invoke call) and efficiency reasons. Modern programming language make extensive use of modularization and procedures and an inefficient procedure invocation mechanism leads to poor system performance. The initialization of the system procedure table will likely be done by software during system initialization and the location and size of this table will have to be made accessible to the firmware by the initialization software.

IMPLEMENTATION CONSIDERATIONS

Although this report does not include a specific implementation for the system described above, it is important to consider selected issues relating to the implementation of the system. Some of the decisions made in the design of the operating system are derived from knowledge of how the system could be implemented. Also the choice of implementation can seriously effect the efficiency and verifiability of the implementation. First, issues relating to hardware will be discussed with respect to efficiency of the system. Second, issues relating to verifiability of the software will be discussed.

HARDWARE

The functions implemented by hardware are the visible functions of levels 0 through 5 of the specifications, plus a few additional features. Hardware (as opposed to software) is chosen as the implementation medium for levels 0 through 5 primarily for two reasons. First, levels 0 through 5 are the primitive levels of the system, i.e. there are no mapping functions which map all the states of these levels to other lower levels. The existence of mapping functions for all other levels assures that the other levels can be implemented in terms of the functions of the primitive levels. The nonprimitive levels can, therefore, be implemented in software if the functions of the primitive levels are implemented as hardware. Second, the functions of levels 0 through 5 are, in general, the most frequently invoked functions of the system and, therefore, should be the fastest functions. Implementation of these functions in hardware is the most convenient way of attaining fast operation. In addition to the functions of levels 0 through 5 a few other operations are implemented in hardware. These other operations are also invoked frequently during the operation of the system and are implemented in hardware to attain speed of execution.

The hardware described below is well within the state of the art. All the hardware features exist, in some form, in commercially available computers, although no single existing computer contains them all. Construction of this hardware simply involves combining and integrating existing techniques.

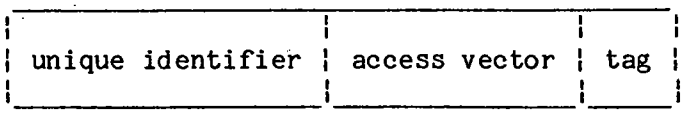
It is no coincidence that the primitive levels of the secure operating system contain, in general, the most frequently invoked functions and that these functions are straightforwardly implementable in contemporary hardware. One of the objectives of the design of any operating system by the methodology should be to assure this association of primitive levels, frequently invoked functions, and straightforward hardware implementation. This is the case of the secure operating system where the choice of the functions of the primitive levels was made with contemporary hardware in mind.

The design of hardware to implement most of the functions of levels 0 through 5 should be straightforward for any experienced

hardware designer as most of the functions are very conventional (e.g., those relating to interrupts, registers, and the clock). Some functions are not as conventional (e.g., those relating to capabilities) and require some discussion. Only issues relating to these less conventional functions will be discussed.

Capabilities

Since capabilities are the basis of all protection in the secure operating system, it is essential that only the operations of the capability module can access capabilities. To insure this it is necessary that capabilities be distinguishable from other data in the system. The most straightforward way of distinguishing capabilities is to mark each piece of data with a single bit of information indicating whether or not the data is a capability. This bit of information is called a tag. In addition to the tag, the capability must contain a unique identifier (the value retrieved by GET_SLAVE) and an access vector (the value returned by GET_ACCESS). This can be represented schematically as follows:



This diagram indicates that a capability has three fields. The tag field is one bit in length and always contains the same value. The length of the unique identifier field must be large enough to take on a sufficient number of different values so that the system will never run out of unique identifiers during its lifetime. We can put a reasonable upper bound upon this number by assuming that a system might generate one unique identifier every microsecond and that the lifetime of the system is about 60 years. These assumptions are probably worse than could be expected in any real system. In this case the number of different unique identifiers required would be about 2^{51}. This would require a field length of 51 bits. Any real system could probably get away with a field shorter than 51 bits. The length of the access vector field depends upon the number of different access rights required for any given object. Although there is no limit to the number of access rights an arbitrary object might need, a reasonable number would be between six and twelve. There is no system maintained object that requires more than six access rights. Using the maximum possible length for each field, the total length of the capability would be $1 + 51 + 12 = 64$ bits. Some number of bits for error detection and correction should also be added. From the point of view of capabilities, some value slightly greater than or equal to the capability size should be the optimal size for a memory word. However, a machine word less than this size could also be accomodated by making capabilities multiple word data items. The important properties that must be maintained by the hardware are:

1. only the three hardware implemented operations GET_SLAVE,

GET_ACCESS, and RESTRICT_ACCESS can read a data item whose tag indicates that it is a capability, and

2. only the operations CREATE_CAPABILITY, CREATE_RESTRICTED_CAP, and RESTRICT_ACCESS can generate a data item whose tag indicates that it is a capability.

The algorithms for the capability operations GET_SLAVE, GET_ACCESS, and RESTRICT_ACCESS follow directly from the specifications and the format of the capability itself. These capability operations must be sure that their argument is a capability (by examining the tag) before returning any value. The only capability operations whose implementations are not straightforward are CREATE_CAPABILITY and CREATE_RESTRICTED_CAP. These operations must generate a different unique identifier each time they are invoked. A simple algorithm such as adding one to the value of the last previously used unique identifier is acceptable. However, any algorithm which generates a different unique identifier each time is also acceptable.

As stated in the introduction to this section, it is not necessary that the hardware that implements capability operations be separate from the hardware for other levels of the system. Although the specifications call for a single instance of the capability functions, it is quite clear that the operations GET_SLAVE, GET_ACCESS, and RESTRICT_ACCESS can be distributed into the processors. Each processor can have an instance of these operations. The distribution of the CREATE_CAPABILITY operation is not quite as easy because each created capability must be unique for the entire system rather than for just one processor. The system can either have a single instance of the CREATE_CAPABILITY operation or, if there are multiple instances of CREATE_CAPABILITY, these instances will have to cooperate. The desirability of each of these two alternatives for CREATE_CAPABILITY will depend upon the precise machine architecture chosen.

The three hardware implemented object types (interrupts, memory blocks, and I/O devices) are addressed via capabilities. This would seem to imply that the hardware must maintain some mapping from the unique identifiers within the capabilities to the physical addresses of the objects. Logically, this implication is correct, however, in practice the mapping can be trivial. Since the hardware implemented objects are known when the system is initialized, some unique identifiers for these objects can be reserved for use by the hardware. The unique identifiers reserved should be those bit patterns which consist of the physical addresses of the objects plus some bits to distinguish the three different types of hardware objects from each other and from all other unique identifiers that the system may generate. When the hardware is invoked to reference one of the hardware objects, it simply extracts the unique identifier from the capability (once the access rights have been checked) and checks to see if the type bits indicate a hardware object of the correct type. If so, the bits which contain the physical address of the hardware object are used to reference the object itself. This scheme makes insignificant the cost of using capabilities to reference hardware implemented objects.

There remains the problem of efficiently implementing software references to objects via capabilities. It must be possible for a program implementing some abstract object to convert the slave capability for that object into the representation of that object. For programs implemented above level 9 (extended types), the extended type manager can be used for this purpose. For programs implemented below level 9 or for programs not wishing to use the extended type manager some other facility is necessary. The simplest mechanism that can be implemented efficiently for this purpose is to implement a hashing function in the hardware. The hashing function can be defined as follows:

```
VFUN hash(slave_capability u, INTEGER hs) -> INTEGER hi;
    INITIALLY hi >= 0 AND hi < hs;
```

This is a constant function that always returns the same integer for a given slave capability within the range of integers between 0 and hs. This function can be used to efficiently locate an entry for the object corresponding to a given capability in a hash table of size hs. The algorithm of the HASH function must be carefully chosen to permit efficient implementation and to minimize conflicts in the hash table. The HASH function should logically be included in the specification of the CAPABILITIES module.

Processors

The processor level provides the basic computational power of the system. The processor level, therefore, includes functions for arithmetic such as addition and subtraction, boolean algebra such as 'and' and 'not', and relations such as equal, greater than, and less than. The processor level also provides the primitive mechanism for executing sequences of operations; there will, therefore, be functions for controlling sequencing, such as transfers and conditional transfers. Virtually all computers have processors with functions such as these and any of these processors is generally suitable for use in the secure operating system. The precise choice of computational functions should depend more on the language chosen to implement system and application programs than on the operating system design. For this reason most of the computational functions of level 4 are not given in the specifications. The specification of these functions is generally trivial. A few have been specified to illustrate how they might be done. The specification is by no means complete. Also, most of the specifications of level 4 could be modified to fit a different processor architecture without effecting the overall system design.

Hardware assistance for higher levels

There are several instances, in the higher levels of the operating system, of functions that will be invoked so frequently that it is essential that their implementation be exceedingly fast if the system is to operate at reasonable speed. For this reason, the most frequently executed paths of the algorithms that implement these functions must be implemented in hardware.

Hardware assistance is clearly necessary for operations that read and write pages and segments since these operations are invoked on every instruction. Hardware should be used to compute the location of and retrieve or modify data in primary memory. Techniques for efficiently accomplishing these tasks in hardware can be found in many existing machines and typically involve the use of buffer or associative memories. It may also be desirable to utilize similar techniques for referencing I/O devices. Recall that programs above level 12 are given temporary capabilities for I/O devices so that access to the device may be revoked at some later time. The implementation of the VISIBLE_IO module must, for each transaction with a device, convert the temporary capability for the device into the permanent capability for the device. If it is essential that operations on I/O devices be highly efficient, the functions that operate upon devices can be implemented in hardware using an associative memory to obtain the permanent capability for the device when given the temporary capability.

SOFTWARE

The most important software implementation consideration with respect to the methodology is the choice of programming language. The programming language must be suitable to the application (i.e., it must be a good system programming language) and it must be amenable to program verification techniques. In order to make proof as easy as possible, the programming language must relate well to the methodology and particularly to the specification language. In general, programs that are easily verifiable and constructed using the methodology should have clarity and be eminently readable. Such a language should contain features such as procedures and functions, restrictive scope rules, strong type checking, simple but general data and control structures, and means for generating and handling exceptions.

Functions have been incorporated into the design of the hardware to make it easy for a compiler to generate code for such a language. The hardware has the CALL and RETURN functions for procedure and function calls and a convention for the passing of arguments. The existence of activation records and procedure records can be used to enforce restrictive scoping of variables. Conventions are included that allow a procedure to indicate to its caller that an exception has occurred.

Level	PSOS Abstraction
16	user request interpreter
15	user environments and name spaces
14	user input-output
13	procedure records
12	user processes and visible input-output
11	creation and deletion of user objects
10	directories
9	extended types
8	segmentation and windows
7	paging
6	system processes and input-output
5	primitive input/output
4	arithmetic and other basic operations
3	clocks
2	interrupts
1	registers and addressable memory
0	capabilities

Table 2.1 - PSOS levels of abstraction

		PROCEDURE_RECORDS	
LEVEL 13	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_INVOKE	COORDINATOR	TIMERS
	SEGMENTS	WINDOWS	EXTENDED_TYPES
	DIRECTORIES	USER_OBJECTS	USER_PROCESS
	USER_INVOKE	VISIBLE_IO	
	USER_PROCESS	USER_INVOKE	VISIBLE_IO
	COORDINATOR	TIMERS	
LEVEL 12	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_INVOKE	SEGMENTS	WINDOWS
	EXTENDED_TYPES	DIRECTORIES	USER_OBJECTS
		USER_OBJECTS	
LEVEL 11	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_PROCESS	SYSTEM_INVOKE	COORDINATOR
	TIMERS	SYSTEM_IO	SEGMENTS
	WINDOWS	EXTENDED_TYPES	DIRECTORIES
		DIRECTORIES	
LEVEL 10	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_PROCESS	SYSTEM_INVOKE	COORDINATOR
	TIMERS	SYSTEM_IO	SEGMENTS
	WINDOWS	EXTENDED_TYPES	
		EXTENDED_TYPES	
LEVEL 9	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_PROCESS	SYSTEM_INVOKE	COORDINATOR
	TIMERS	SYSTEM_IO	SEGMENTS
	WINDOWS		
	SEGMENTS	WINDOWS	
LEVEL 8	CAPABILITIES	REGISTERS	ARITHMETIC
	SYSTEM_PROCESS	SYSTEM_INVOKE	COORDINATOR
	TIMERS	SYSTEM_IO	
		PAGES	
LEVEL 7	CAPABILITIES	REGISTERS	MEMORY
	ARITHMETIC	SYSTEM_PROCESS	SYSTEM_INVOKE
	COORDINATOR	TIMERS	SYSTEM_IO

Table 2.2 (continue on next page)

	SYSTEM_PROCESS TIMERS	SYSTEM_INVOKE SYSTEM_IO	COORDINATOR
LEVEL 6	CAPABILITIES ARITHMETIC	REGISTERS	MEMORY
		PRIMITIVE_IO	
LEVEL 5	CAPABILITIES INTERRUPTS	REGISTERS CLOCK	MEMORY ARITHMETIC
		ARITHMETIC	
LEVEL 4	CAPABILITIES INTERRUPTS	REGISTERS CLOCK	MEMORY
		CLOCK	
LEVEL 3	CAPABILITIES INTERRUPTS	REGISTERS	MEMORY
		INTERRUPTS	
LEVEL 2	CAPABILITIES	REGISTERS	MEMORY
	REGISTERS	MEMORY	
LEVEL 1	CAPABILITIES		
LEVEL 0		CAPABILITIES	

Table 2.2 - Modules of levels 0 through 13
(Modules above the dotted line of each level represent functions newly added to the system at that level. Modules below the dotted line represent functions actually implemented at lower levels. A module appearing above the dotted line in more than one level indicates different instantiations of the module, i.e., different implementations of the module with possibly different parameter values.)

Type of object	Access rights
clock	read, modify
interrupt	set_handler, mask, set_int
memory block	read, write
primitive I/O device	read, write, control, device
processor	read, modify
system process	read, modify
system I/O device	read, write, control, device
page	read, write, delete
segment	read, write, delete, call
extended type	create, manage, delete, interrogate, add_rep, delete_rep
directory	add_entries, remove, load, list, add_locks, delete
user process	read, modify
visible I/O device	read, write, delete, control, device
procedure records	read, write

Table 2.3 - Access rights for system objects

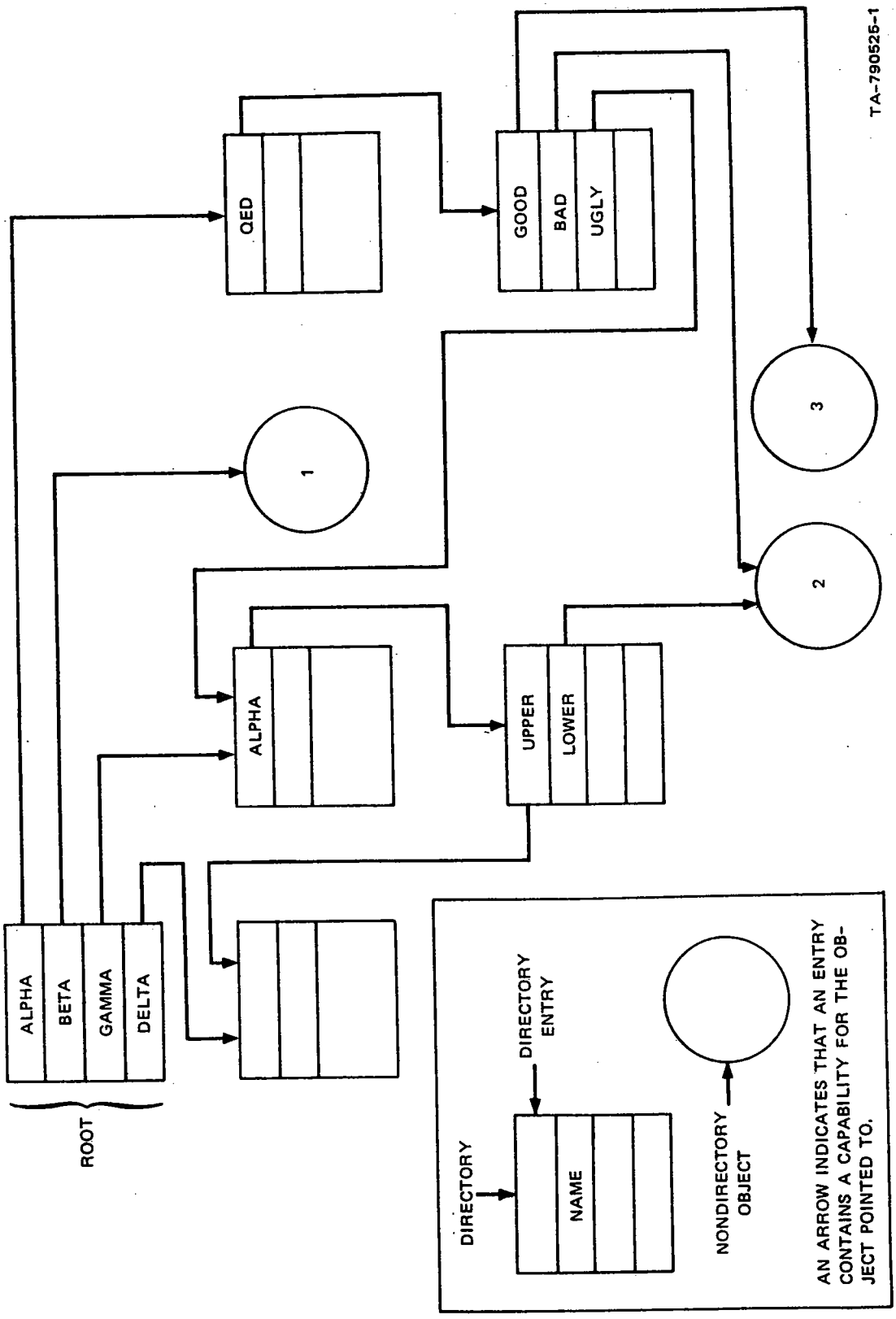


FIGURE 2.1 EXAMPLE DIRECTORY HIERARCHY

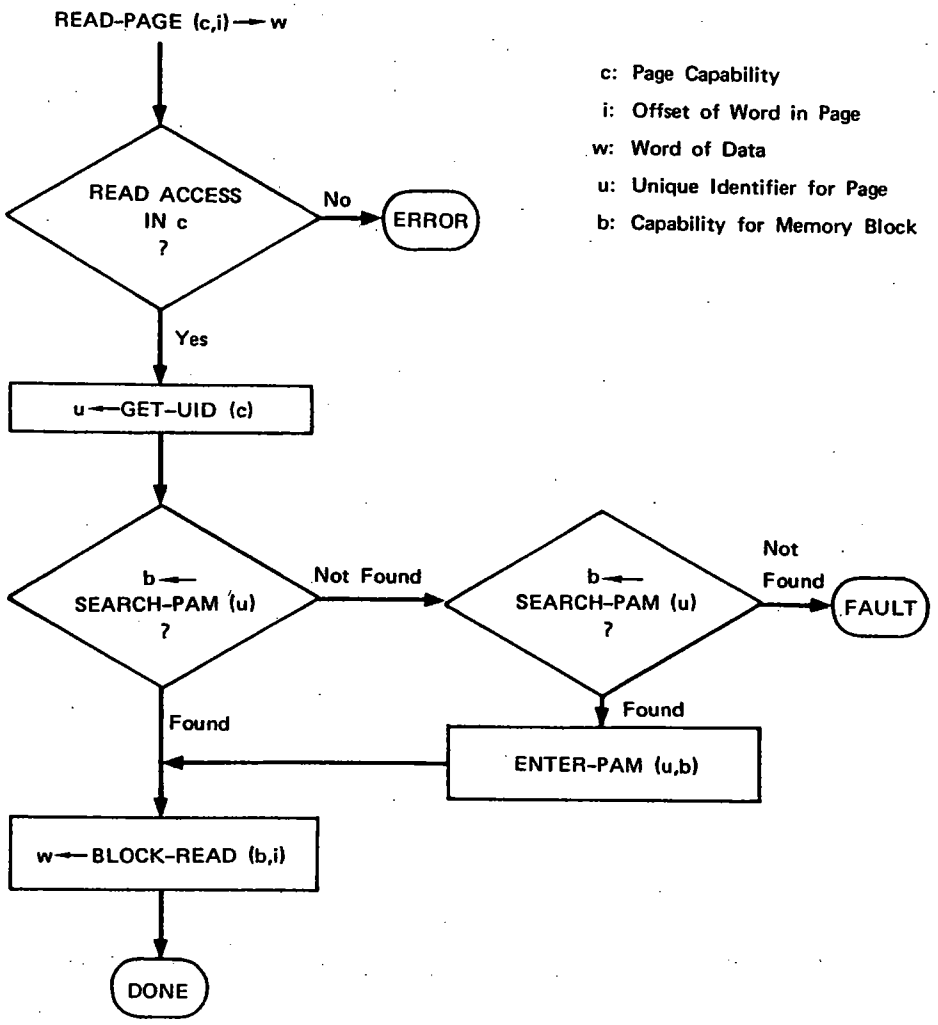


FIGURE 2.3 COMPLETE HARDWARE ALGORITHM FOR READING A PAGE

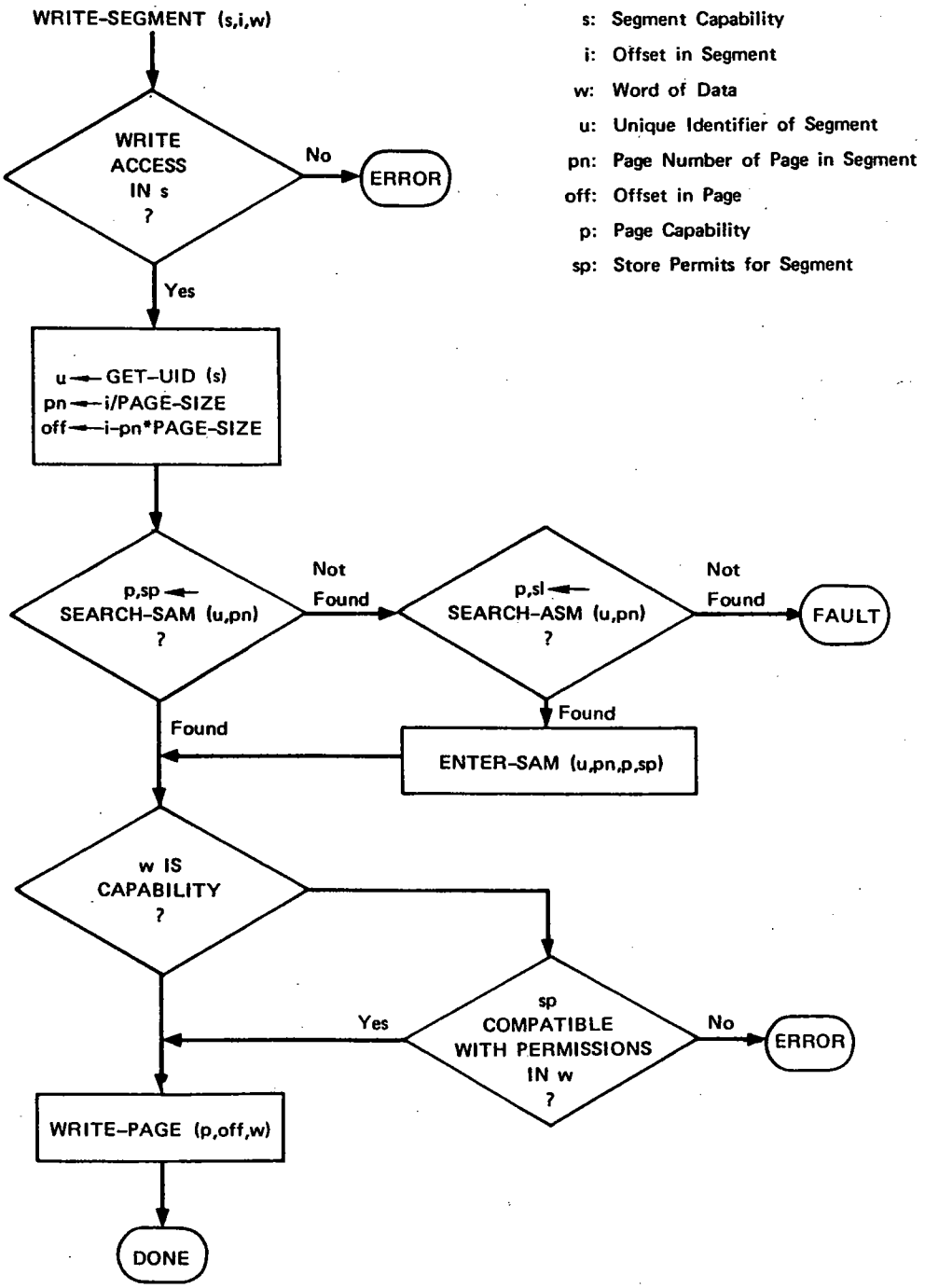


FIGURE 2.4 HARDWARE ALGORITHM FOR WRITING INTO A SEGMENT

PART III
PROOF CONSIDERATIONS

CHAPTER III-1
PROOFS OF SECURITY -- SPECIFICATION PROPERTIES

This section presents a statement of specific security-related properties about the operating system design, and outlines proofs of these properties. It is seen that the security of the basic design thus rests on easily-enforced simple properties of the specifications. The security of any application rests on its abstract type managers. [The authors feel strongly that this chapter is only a preliminary formulation. Further research is needed to develop models that better capture the user-view intuitive meaning of security -- particularly to model environment changes. Also, the present description is too dependent upon SPECIAL.]

The staged development process discussed in Part I makes a distinction among the following stages:

(S0) the definition of the visible interface to the system or subsystem to be developed;

(S1) the decomposition of the functions of that interface (along with internal system functions) into a hierarchy of levels, each consisting of a set of modules;

(S2) the design as represented by formal specifications for each module;

(S3) the abstract data representations as defined by mapping function expressions; and

(S4) the implementation of each function in terms of lower-level functions.

The proofs of operating system security follow the same distinction. First, proofs of design properties are considered, i.e., proofs of the correspondence between the desired properties and the formal specifications. Then proofs of the consistency of the design (stage 2) and the implementation (stages 3 and 4) are considered. This chapter is concerned with proofs of properties relative to the design. The next chapter considers the proofs of implementation consistency.

Two properties are basic to the security of system operation. These are the Alteration Principle and the Detection Principle, which state that there shall be neither unauthorized modification nor unauthorized acquisition of information. These principles are stated informally in Section 4 of Chapter I-1, and somewhat more rigorously in Section 9 of Chapter I-1. In this

chapter, these principles are given a more detailed axiomatization in terms of the specification language and the concept of capabilities in the operating system design. It is observed that these principles are satisfied by the specifications for the operating system (Appendix B). Furthermore, it is possible to check this consistency automatically, although the anticipated on-line tools for doing this checking have not yet been completed.

Additional properties are given that contribute to the understanding of the security of the design. These properties are decomposed into several categories, some relating to the methodology in general or to the specification language in particular, the rest relating to the operating system design.

TERMINOLOGY AND NOTATION

The terminology used here concerns O-, OV-, and V-functions, hidden and visible V-functions, and derived and primitive (nonderived) V-functions. For V-functions, "H" stands for "hidden", "V" for "visible", "P" for "primitive", and "D" for "derived".

The external interface to each level (e.g., that of the user-visible interface) consists of a collection of visible V-functions, O-functions, and OV-functions, each of which is permitted to be called from certain higher levels. Each level is composed of a collection of modules, specified independently. Internal to a module are hidden V-functions (HV), which define the state of the module. In specifications, hidden V-functions may be cited and modified only by functions at the level of the module to which they belong. They are totally invisible above that level. Hidden V-functions may however appear in assertions (e.g., mapping functions) about system behavior.

In general, protection is intrinsic to the functions of the external interface to each module, via the exception conditions and the use of capabilities as arguments. That is, each visible function requires as arguments appropriate capabilities for any objects to be accessed; these capabilities must contain proper access rights in order for the function to be executed. Since the hidden functions are not accessible, no protection is applied to them.

Certain V-functions of each module are primitive (PV), namely those whose values are used in the specification of effects for the O- and OV-functions of given module. Other V-functions of the module are derived (DV), with the value of each being derived as some function of the values of primitive V-functions of the module. Being able to write effects solely in terms of the most primitive V-functions of each module greatly simplifies the specifications.

All visible V-functions of the operating system are derived V-functions VDV, being derived from corresponding hidden functions HPV or HDV. As noted above, access rights are checked by the visible V-functions.

PROPERTIES OF THE APPROACH AND OF THE DESIGN

The properties defined here are of four categories, relating to

(C1) intrinsic syntactic properties of the specification language,

(C2) extrinsic syntactic properties (over and above the intrinsic syntactic properties) and semantic properties required of all PSOS specifications,

(C3) properties of the capability module of PSOS, and

(C4) the desired security properties of the PSOS visible interface, to be derived from the actual specifications for the modules forming that interface.

The properties of category C4 considered here relate specifically to the mechanism provided by the user-visible operating system interface. Additional policy relating to various applications is not discussed in this section, but is described in detail in Part IV. Included there are the secure object manager and the confined subsystem manager. For the operating system interface, the relevant properties of category C4 include the avoidance of unauthorized modification and unauthorized acquisition of information via the operating system. These properties will be useful in proving correctness of the implementations of policies using PSOS. For the secure document manager, the relevant properties include the "*-property" and "simple security condition" of the Bell and LaPadula model for handling classified objects. Note that these two properties of the Bell and LaPadula model are in fact specific instances of the avoidance of unauthorized modification and unauthorized acquisition, respectively.

In each of the following sections, the properties of each category are defined. The satisfaction of these properties then leads to a chain of reasoning resulting in a meaningful and well-defined sense in which the operating system design is said to be secure. Proofs of these properties are then outlined, and the implications on the security of the implementation of the system are considered.

1. Syntactic Properties of the Specification Language

The SPECIfication and Assertion Language (SPECIAL) is described and defined precisely in Appendix A. In general, a module specification at the highest level of simplification has

the following form.

```
MODULE modulename
    TYPES
    DECLARATIONS
    PARAMETERS
    DEFINITIONS
    EXTERNALREFS
    FUNCTIONS
END_MODULE
```

The form of a function specification depends on whether the function is an O-, OV-, or V-function. For example, an O-function specification at the highest level of simplification has the following form.

```
OFUN functionname ([arglist]);
     $(purpose as a comment)
     DEFINITIONS
     EXCEPTIONS
     EFFECTS
```

Here each effect in the specification of an O- or OV-function is a list of expressions, relating the desired new (quoted) values of primitive V-functions of the module to old (unquoted) values of those functions. The language forbids the quoted appearance of a nonprimitive V-function. As noted in the next section, each expression is typically (but not necessarily) of the form

'PV(x) = f(PV(x), PV1(y)),

where x and y are sets of arguments, where PV(x) and PV1(y) are primitive V-functions of the module to which the specified function belongs, where PV(x) denotes the value of PV(x) before the execution of the specified function, where 'PV(x) denotes the desired value of PV(x) after the execution of the specified function, and where f(PV(x), PV1(y)) is a function of PV(x) and PV1(y), of parameters of the module, and of external references to V-functions of modules at the same level. Effects may also include the construct "effects_of", which provides a means of writing an implicit effect equivalent to the effects of O- and OV-functions of other modules of the same level.

The specification language imposes scope rules and type checking on all expressions of the language. As seen below, these constraints contribute significantly to the enforcement of the desired security properties. The properties discussed in this section are common to all specifications, and relate to the syntax of the specification language used for the operating system development.

For the purposes of this section, a primitive V-function value in the specification of a module is CITED if and only if it appears as an old value in an effect, an exception, or a

derivation. The value is MODIFIED by the specified function if and only if it appears as a new value in an effect. The cited variables have values that are established before the time a function is invoked. The modified variables have values that are established (explicitly) as a result of the execution of O- or OV-functions of the module. SPECIAL requires that all V-function values cited or modified in any module specification must be V-functions of the level to which the given module belongs. Note that a module may appear as part of various (consecutive) levels.

Values are characterized as read references or write references, as follows.

READ REFERENCES for a given specified function of some module are precisely

* the arguments to the given function,

* the parameters of the module (which are fixed for the module as a whole),

* the primitive objects of the specification language (designators [e.g., capabilities and unique identifiers in the operating system], integers, reals, booleans, and characters; sets, vectors, and structures thereof),

* cited values (unquoted occurrences) of the primitive V-functions of the module, all of whose arguments are read references, and

* external references to cited values (unquoted occurrences) of V-functions of modules belonging to the same level, all of whose arguments are read references of the specified function.

The WRITE REFERENCES of a specified O- or OV-function of a module are those that may take on a modified value as a result of that O- or OV-function. Write references of a function may be either

* modified values (quoted occurrences) of the primitive V-functions of the module or of another module of the same level (via the effects_of construct for an external reference), or

* modified values returned by the function (if it is an OV-function of the module),

* the values of exception conditions of the specified function.

In the specification of any function, the arguments to the V-functions whose values are either cited or modified must be read references of the specified function (or in the case of an OV-function specification, the argument may be the return value).

Note that quoted values of nonprimitive V-functions may not occur. Quoted values of primitive V-functions of other modules at the same level do not occur, although this is an example of an extrinsic syntactic property, required not by SPECIAL but by the design of PSOS.

The value of a derived V-function of a module specification is determined indirectly by changes in corresponding primitive V-functions, never directly by mention of a quoted derived V-function value in the effect of an O- or OV-function. Specifications for derived V-functions make this correspondence explicit. A derived V-function may be hidden (HDV) or visible (VDV). The value of a hidden derived V-function may depend only on the read references of the function; the value of a visible derived V-function may depend only on the read references of the function and on the values of the hidden derived V-functions of the module. Thus VDV are derived from HDV and HPV, while HDV are derived from HPV. This partial order avoids cyclic derivations. In specifications, derived variables appear only in unquoted form.

As an illustration of these definitions, consider the directory module, as specified in Appendix B. The relevant read references of the function "add_entry(d,n,c)", for example, are the arguments, d, n, c; the parameters of the directory module, root_slave, entry_name_length, and the access codes; the defined variable u = get_slave(d); and the old values of the primitive V-functions of the directory module, h_valid_dir(u), h_get_cap(u,n), and h_dir_permits(u). The write references of the function are the new values of the primitive V-functions 'h_get_cap(u,n) and 'h_distinguished(u,n). The derived variables of the directory module are those with the derived V-function values valid_dir(d), get_cap(d,n,k), dir(d), and dir_size(d), whose derivation employs u = get_slave(d) and appropriate exception conditions.

The set of effects in the specification of an O-function or an OV-function provides a set of constraints on the values of the write references of the module, by specifying their relations to read references, using only the primitives of the specification language. Thus the write references may have their values established as a result of execution of the specified function, but only consistent with the specified effects. However, any primitive V-function that is not mentioned in quoted form in a specification is constrained to have a value identical to its old value (the unquoted form). Thus the only changes in the values of the V-functions of the given level are those that are explicitly stated. (There are no side-effects to those V-function values, although those values may independently be changed at other times by other functions of the module or by different invocations of the same function.) Nesting of functions is permissible. Explicit recursive effects are not permissible, because of the fact that a write reference can take on only one value. However, implicit recursion is possible, as, for example, in the case of the function "resolve_reference" in the linker.

This primitive V-function is defined recursively in terms of the mutually recursive macro definitions "subpath" and "resolve_path", which permit the resolution of a symbolic name over multiple levels of directories. (Any implementation should of course be consistent with these specifications, as is to be proven in stages 3 and 4.)

Initial values of primitive V-functions can be specified only in terms of the read references. The exception conditions of derived V-functions and of O- and OV-functions must also be specified only in terms of read references.

Effects of O-functions and OV-functions can be written in various forms. Two forms are of particular value in terms of understandability of specifications, assurance of determinism, and simplicity of proof. These are as follows.

SEPARABLE FORM. Each effect contains exactly one primitive V-function mentioned in quoted form whose value is constrained by that effect.

CANONICAL FORM. Each effect is in separable form; further, the value of the primitive V-function that it constrains is explicitly specified by being of the form

```
'PV(x) = g(X),
```

where x is some sequence of read references, and where X is potentially the set of all read references of the module. Note that quantification over this form is permitted, e.g.

```
FORALL n INSET { n | PRED(n) }:
   'PV(x,n) = g(X,n);
```

Here PRED(n) is a predicate defining a set of n. Note that the function g(X) may be defined in terms of values of V-functions of the module, e.g., 'PV(x) = PV(x) + PV1(y), where y is a read reference of the module, and PV1 is another primitive V-function of the module. Note also that 'PV(x) = PV1(PV2(x)) is a legitimate effect in the specification of O(x) or OV(x), with the value of PV2(x) as the argument of the V-function PV1.

One of three types of transitions can arise in the specification of effects for any O- or OV-function. Given the state of a module, i.e., the set of values of all of the primitive V-functions of the module, the specification of a particular O- or OV-function can specify transition to

a) any one of a set of possible new states,

b) one particular new state, or

c) no possible new state.

These are the cases of nondeterminism, determinism, and unrealizability, respectively, which are discussed next.

a) ANY ONE OF A SET OF POSSIBLE NEW STATES. This is the case of nondeterminism. It may in fact be the correct desired effect, or may be the result of an error in specification. Nondeterminism in the specification of a visible function is potentially harmful, in that information can be passed between two supposedly isolated users by a sequence of calls to the function. However, the only possible instance of nondeterminism in the operating system specifications is in the mechanism for creating capabilities. However, this is nonintrinsic and can be avoided. The actual choice of the capability-generating algorithm is left as an implementation detail, although it could of course be resolved deterministically in the specifications by using (say) an add-one algorithm for the next unique identifier. This instance of nondeterminism is not harmful, since in this case the function get_uid is hidden, and since the design permits only comparisons of slave capabilities. (In general, the existence of any nondeterminism is easily detected if the specifications are in separable form, even more so if all specifications are in canonical form.)

b) ONE PARTICULAR NEW STATE. This is the normal case, except possibly for the case of creation of new objects.

c) NO POSSIBLE NEW STATE. This is the case of unrealizable specifications for which no consistent implementation is possible.

Note that in all three cases, the effects of an O- or OV-function are given by constraining the new values of the various V-functions and the return values. However, write references to higher-level functions are forbidden, as are write references to lower-level functions that are not a part of the given level. Unrealizable specifications may be detected, either syntactically (e.g., as in the case of conflicting effects defining the same V-function value), or by inability to implement, or by the failure of proofs of implementation involving mapping functions and implementations.

By virtue of the specification language, any specification for a particular O- or OV-function must be complete, in the following sense: values of V-functions of the given level can be changed only as stated; all other V-functions of the level are explicitly forbidden from having their values changed by the specified function. Therefore every V-function of the module is accounted for by each specification of an O- or OV-function of the module. Thus the specification language prevents incompleteness of individual specifications. Note that the functional completeness of a design (i.e., the adequacy of the specified functions) remains to be demonstrated subsequent to the specification stage, first via the ability to represent state

information consistently (stage 3), and then via the ability to realize the desired higher-level functions (stage 4). Thus the functional completeness of the design as a whole is not resolved at stage 2, although intelligence and carefulness in the design certainly can contribute to attaining a functionally complete design.

Every specification included in this report is in canonical form; all are deterministic, except for several functions of the capability module associated with new capabilities. (As noted above, the only potential nondeterminism is in the creation of new capabilities, and that can in fact be axiomatized in such a way as to be deterministic.)

The syntactic properties of the specification language discussed above are thus summarizable as follows.

*(1a) EFFECTS. Effects of an O- or OV-function of a module are defined by constraining the new values of the primitive V-functions of the given level (corresponding to occurrences of write references) in terms of the old values of those V-functions (corresponding to occurrences of read references of the specified O- or OV-function), using only primitives of the specification language. As a result of execution of any specified O- or OV-function of a particular module,

* values of functions corresponding to unconstrained write references of the given level must be identical to the corresponding read references. Thus 'PV(x) = PV(x) is implied, for all possible combinations of (quoted) primitive V-functions and arguments not explicitly constrained;

* values of functions corresponding to constrained write references may change only as specified.

*(1b) INITIAL VALUES AND EXCEPTIONS. Initial values (of PV) and exception conditions (of all other functions) are defined only in terms of the read references of the specified function.

*(1c) DERIVED V-FUNCTIONS. The value of each nonprimitive V-function DV is derived as a function of the read references of the function DV (and possibly certain HDV of the module), but not the write references.

*(1d) CONSISTENCY. All specifications should be locally self-consistent. Whether deterministic or nondeterministic, they should contain no constraints that are contradictory (such as a multiply-defined function PV).

Any implementation should be consistent with the (realizable) specifications. (Proof is of course required in stages 3 and 4 that such consistency is actually attained.)

From a logical point of view, the execution of each function is thought of as instantaneous, and hence indivisible and nonoverlapable with parallel execution of other functions. In actual implementation, considerable overlap with other functions is possible without violating consistency. See Part III-2. In general, overlap is permitted as long as no conflict arises in attempting multiple use of the same variables.

2. Properties of the Operating System Specifications

This section concerns extrinsic syntactic properties and semantic properties that must be satisfied by all specifications of the operating system functions (see Appendix B).

The basic notion involved is that of a type for each variable, including the type of each argument. Each type is supported by a particular module of the design, or particular primitives of the specification language. Once a variable has been declared to be of a particular type, all instances of it must conform in usage to that type. Thus type checking is required for every variable. Readability of the specifications is enhanced by using a global convention on argument naming, whereby the first letter of the argument name is descriptive of the type. For example, "c" denotes a capability, "u" a slave capability; "s" denotes a segment capability; "t" denotes a type-manager's capability; "d" denotes a directory capability, and so on. As a reminder for the reader, a capability for an object consists of a unique identifier and an access vector specifying which operations may be performed on that object. Access to every user-visible object in the system requires appropriate capabilities. The access vector is thus essential for protection (and hence security) in the operating system.

> *(2a) STRONG TYPING. Legitimate operations on a type at a given level are precisely those defined by legitimately visible functions (with arguments of the type).

As is seen below, capabilities are used as arguments to various functions. However, the design permits no operations that permit modification of the components of a capability. There are two operations for creating new capabilities, namely the OV-functions "c = create_capability" and "c = create_restricted_cap(mask)". There is one operation for creating a capability more restricted than an already existing capability (and having the same unique identifier), namely the V-function "c1 = restrict_access(c,mask)". Because of strong typing, these three operations are the only way by which a new capability may be created. (As elsewhere, proof is required that the implementation does not compromise this property. In fact, tagging of capabilities in the hardware is useful in assuring the proper use of capabilities.) Similarly, the only operation permitted in the specifications on slave capabilities is testing for their equality. This enables determination of whether two capabilities have the same unique identifier.

Certain kinds of functions are constrained to have particular types of variables in their specifications, for all specifications. In particular, there are constraints on the capability arguments of hidden functions and visible functions, and on the invisibility of primitive functions, as follows. (Arguments and return values which are neither capabilities nor slave capabilities may of course exist -- subject to the strong typing requirements -- but are ignored here.)

*(2b) HIDDEN V-FUNCTIONS. In the operating system, each hidden V-function (except for one) may have arguments of type "slave capability", but not of type "capability". The exceptional case is the hidden V-function "id = get_uid(c)", which has a capability argument (and which returns the integer unique identifier corresponding to the capability c). Except for "get_uid(c)", every hidden V-function has at least one slave capability argument. A hidden V-function may have a value of any type, e.g., slave capability, or capability, or aggregate thereof. Note that hidden functions have no exception conditions (e.g., no access violations); they require no protection, since they cannot be called from outside the module in which they occur. They are used only in specifications, in mapping functions, and in assertions.

*(2c) VISIBLE FUNCTIONS. An O-function, an OV-function, or a visible V-function of a module may have arguments of type capability but not of type slave capability, and may have a value which is of type slave capability or capability (or aggregate thereof). The requirement of capabilities and the exclusion of slave capabilities as arguments to visible functions is the basis for security in the operating system. It enforces access control on every operation on every object, avoiding circumvention of the desired protection. At least one capability argument is required for every visible function, including the function "create_capability" itself and the arithmetic operations, for which such a capability is implicit.

*(2d) INVISIBILITY OF PRIMITIVE V-FUNCTIONS. All visible V-functions in PSOS are derived (VDV). (This is an extrinsic syntactic property.) Further, all primitive V-functions in PSOS are hidden (HPV).

From properties (2b) and (2c), it follows that the general forms for hidden V-functions, visible V-functions, OV-functions, and O-functions, respectively, are (with the above-noted exception of the HV "get_uid(c)")

```
zi = HV(ui1, ui2, ...)
zi = VV(ci1, ci2, ...)
 z = OV(c1, c2, ...)
     O(c1, c2, ...),
```

ignoring arguments that are neither capabilities nor slave

capabilities. Here z denotes a return value which is arbitrary (i.e., may include a capability or a slave capability, or an aggregate thereof). An examination of Appendix B shows only a few hidden V-functions and corresponding derived visible V-functions that return capabilities, along with 11 capability-returning OV-functions, all of which involve creating a new object or creating a capability itself. The visible functions are summarized in Table 3-1.1. There is only one visible V-function that returns a slave capability, namely "get_slave" itself. Since a slave capability contains no access rights, the existence of this visible V-function returning slave capabilities presents no security violation.

Examples from the directory module which illustrate the above properties are included here as an aid to the reader. Let "d" be a directory capability, and "u" be its slave capability; n is a symbolic name. A hidden V-function HPV and a visible V-function VDV derived from it are exemplified by "c = h_get_cap(u,n)" and "c = get_cap(d,n,k)", respectively. The derivation of the latter uses the former as a read reference, as well as the visible lower-level V-function u = get_slave(d). O- and OV-functions are exemplified by "add_entry(d,n,c)" and "d = create_directory(cdt,slv)", respectively. No visible function has a slave capability argument; no hidden function has a capability argument (except get_uid). (Here slv is a boolean vector implying store-limitations, and is incidental to the example.)

One further requirement is useful in defining the notion of security. This concerns the relative strength of access codes and their appearance in exception conditions.

*(2e) ACCESS CODES. Two restrictions are imposed on the use of the specification language regarding access codes.

(i) Throughout the system, an access code bit of TRUE in some position of the access vector of a capability is more powerful than a FALSE in the same position. Exception conditions may be triggered only if an access code position contains a FALSE, never if it contains a TRUE. Thus, the only exception macro permitted is of the form "BOOLEAN no_ability(c,i)", defined as TRUE if and only if get_access(c)[i] = FALSE. An exception macro "ability(c,i)" defined as TRUE if and only if get_access(c)[i] = TRUE is not permitted.

(ii) Neither the unquoted form nor the quoted form of "get_access(c)" may appear in the specification of any function other than "restrict_access", except that the unquoted form may appear in exception macros. This prevents effects or derivations that are dependent on the access bits.

3. Properties of the Capability Module

Three basic properties are of interest in understanding the protection mechanisms provided by PSOS for the implementation of various security policies. Based on the specifications of the capability module, these are (1) that capabilities once created never change; (2) that access rights cannot be increased by any operations; and (3) that stronger access rights have at least equal power in terms of the results they may produce.

*(3a) INVARIANCE PROPERTY OF CAPABILITIES. A capability never changes its unique identifier or its access rights.

*(3b) MONOTONICITY OF ACCESS RIGHTS. There exists no function of the capability module that can make available a capability with access more powerful than all already available capabilities with the same unique identifier. That is, access rights for a given object may never be increased.

*(3c) COVERING PROPERTY FOR CAPABILITIES. Stronger access rights have at least equal power. Consider an O-, OV-, or visible V-function x=f(...,c1,...) that is successfully completable for the given arguments and some initial state. Suppose c1 = restrict_access(c2,k), where k is any mask, applied to the access code of c2. Then y=f(...,c2,...) (with otherwise identical arguments) is also successfully completable, assuming the same initial state.

Proofs of these three properties are straightforward. (3a) follows from the observation that get_uid and get_access never appear in quoted form. (3b) follows from the observation that only "restrict_access(c)" can create a capability with a unique identifier identical to an already in use. The newly created capability cannot have greater access rights, as is seen from the specification of "restrict_access".

(3c) is somewhat more complicated. If the function f(...,c1,...) is successfully completable for the given set of arguments and the given initial state, then none of the exception conditions is triggered. It is to be shown that none of the exception conditions can be triggered for f(...,c2,...). By (2e) (i and ii), no exception condition can be triggered if none was triggered by f(...,c1,...).

Note in (3c) that the effects of x=f(...,c1,...) and y=f(...,c2,...) on the V-functions of the given level are essentially the same. However, since no effect in any function of the system is dependent on the unquoted form of get_access, the difference in the two effects can only be in the value returned, e.g., x=c1 and y=c2.

The above properties relate to the protection mechanism provided by the operating system, not to any policy that is implemented using the operating system. The proofs indicated here are purposely intuitive and informal, although examination of the specifications clearly indicates that the desired

properties are satisfied. However, a more rigorous proof involves a formal axiomatization of the semantics of SPECIAL, which has not yet been done. Nevertheless, these properties provide the basis for the implementation proofs for the correctness of the policies implemented on top of PSOS. implemented.

4. Security of the Operating System Design

The basic properties to be satisfied by the specifications for PSOS are the Alteration and Detection Principles introduced in Section 1.1.4, and restated in Section 1.1.9. These properties are now formulated in terms of capabilities.

Security in the use of the system is based on capabilities and their immutability, as well as on the uniqueness of their identifiers. An object can be accessed only by presenting suitable capabilities as arguments to visible O-, OV-, and V-functions.

ACCESSIBILITY

Security is formalized here in terms of the (implicit) accessibility of capabilities, using the concept of ITERATIVE ACCESS. Intuitively, a capability is iteratively accessible from an existing set of capabilities if and only if that capability can be obtained as the value resulting from a sequence of calls on V-functions, whose capability arguments are themselves at each moment iteratively accessible. Iterative access is defined rigorously as follows.

ITERATIVE ACCESS: iterative_access(S) =

> Let S' be the set of all values of PV(args) that are capabilities (including elements of aggregated values composed of capabilities), where PV is any primitive V-function or parameter of the system (other than a hidden V-function of the capability module), and "args" is any argument list such that any capability in "args" is a member of S:
>
> > IF S' is contained in S THEN S
> > ELSE iterative_access(UNION (S,S'))

Since all visible V-, OV- and O-functions must have capabilities as arguments when referring to objects, it is sufficient to examine the accessibility of capabilities. It is now possible to state the two fundamental theorems that contribute to the security of the system design (as represented by the hierarchical structure (stage 1) and by the specifications for the system (stage 2)). These two theorems refer respectively to the absence of unauthorized acquisition and the absence of unauthorized modification of information, with respect to the use of capabilities, as follows.

*(4a) ALTERATION PRINCIPLE. With respect to the specification of any visible function f(args) of a module, let S' be the set of capability values assumed by write references of f(args). Then S' must be a subset of iterative_access(S).

*(4b) DETECTION PRINCIPLE. With respect to the specification of ay visible function f(args) of a module, let S' be the set of capability values of read references of f(args). Then S' must be a subset of iterative_access(S).

The satisfaction of the alteration and detection principles is aided by the properties of the previous sections, particularly by strong typing to guarantee the proper use of capabilities. Strong typing, combined with the restrictions on quantification imply that only accessible sets can be named. Thus, for example, it is impossible to cite the set of all capabilities not accessible, unless the set of all capabilities were accessible. Thus the satisfaction of (4a) and (4b) is seen to depend largely on the syntax of SPECIAL and on (2a), but also in some cases on the semantics of the specifications. Inspection of the specifications of Appendix B shows that (4a) and (4b) are in fact satisfied for all of those specifications. Above and beyond these properties, (2e), (3a), (3b) and (3c) offer further insights into the security provided by the system.

(Note that tagging of machine words supports the proper use of capabilities in hardware. It is assumed at this point that the hardware operates correctly. However, the design should be defensive on this point, and should be prepared to recover from malfunctions. It is also assumed that resource errors are handled properly, e.g., the exhaustion of unique identifiers (over decades). Overall, it must also be assumed that there are adequate safeguards to prevent unauthorized modification of the implementation, e.g., by an operator or system programmer. Some of this can be handled by the operating system itself.)

5. Additional Security Properties

The properties described above are useful in describing the security and protection of PSOS. In addition, they are useful in proving the security properties of subsystems implemented using PSOS, such as the Secure Object Manager described later. However, some of these properties are described in terms of particulars of the specification language, and are difficult to relate to concepts of security and protection as they are commonly understood. Such concepts include the ability to control access to data by different users and to control communication between different users. Properties that are expressed in terms of these more abstract concepts (rather than in terms of particulars of the specification) are more generally applicable, easier to apply to proofs of properties of subsystems, and easier to understand. Two useful properties are the Accountability and Isolation Principles given below. These principles are described only informally here, as their formalization is not yet complete; as

a result of the incomplete formalization, they have not yet been applied to PSOS. They are discussed here in order to further demonstrate the utility of the techniques of this report.

As an illustration, we first introduce the notion of a suitably defined "similarity" relation between capabilities that implies alternate ways of accessing the same data. Intuitively, two capabilities are similar if and only if they are intended to refer to the same data (e.g., the contents of a segment). For any system (such as PSOS), a precise definition would be given as to when two capabilities are (intended to be) similar, e.g., based on their unique identifiers. This is embodied in the Accountability Principle, which states that no two dissimilar capabilities can ever be used independently to access the same data (i.e., the same data object, since the data itself can always be copied into another object).

> *(5a) ACCOUNTABILITY PRINCIPLE. All capabilities that can be used to access a particular piece of data are actually similar to one another.

In PSOS, two capabilities (other than capabilities for windows) can be defined as similar if and only if they have the same unique identifier. In this case determination of the similarity of two capabilities is quite easy. However, the presence of windows in the design of PSOS given here complicates matters, in that the capability for a segment and the capability for one of its windows may both be used to access the same data (in the window), but those two capabilities have different unique identifiers. PSOS as specified in this second edition therefore requires a more complex definition of similarity than the version of the system defined in the 1977 report.

The Accountability Principle is important because it reduces the question of accessibility of data to one of the possession of certain capabilities. But, to be useful, it requires that the concept of similarity be well defined and easy to compute. This is not the case in the current PSOS design.

From the Accountability Principle the following principle can be derived.

> *(5b) ISOLATION PRINCIPLE. Two processes that do not have access to similar capabilities (i.e., none of the capabilities of one process are similar to any of the capabilities of the other process) cannot communicate (i.e., there is nothing one process can do that can affect the operation of the other).

This principle is particularly useful in security-related applications. One can derive variations of these principles that are useful in particular situations. For example, the total isolation between processes of the Isolation Principle is not always desired. One may wish instead to guarantee one-way

communication only or to guarantee communication of a particular type. Given that the underlying system obeys the Isolation Principle, it is straightforward to create processes that share some carefully chosen similar capabilities and prove that these shared capabilities permit only the desired form of communication.

Proof of the correctness of implementation is the subject of the next section. If the system is correctly specified, correctly implemented, and correctly initialized, it will then be secure -- at least with respect to the enforcement of the alteration and detection principles. However, the properties ultimately of interest are those of the applications. Multilevel security is considered in Chapter IV-1. Confinement is considered in Chapter IV-2. Denial of service is not considered, except superficially.

	module	OV-functions	V-functions
12	user-processes	create_process	
11	user-objects	create_directory create_segment create_object	
10	directories	directory_create (10)	get_cap
9	extended-types	object_create (10) create_type	impl_cap
8	segments	segment_create (10)	
7	pages	create_page (7)	
0	capabilities	create_capability create_restricted_cap	restrict_access get_slave [sl]

TABLE 3-1.1
VISIBLE FUNCTIONS RETURNING A CAPABILITY

(Note: if visibility is restricted, the highest level of visibility is indicated in parentheses. For simplicity, functions of level 0 are omitted that are restricted to level 3. "[sl]" denotes slave capability only.)

CHAPTER III-2

PROOFS OF SECURITY -- PROPERTIES OF IMPLEMENTATION

I Introduction

Proofs of implementation correctness are by far the most difficult stage of the program development process using the hierarchical methodology. This chapter discusses the proof of a subset of PSOS, including the assumptions made and the issues raised. There have been several years of experience to draw on in the proofs of specification properties, but there has been comparatively little experience in proofs of implementation (especially for systems of reasonable complexity). The experience gained from the work described in this chapter has shown that implementation proofs are orders of magnitude more difficult than proofs of design, because of the sheer complexity of the large number of states multiplied by the number of ways of sequencing the state transitions. This complexity can never be totally eliminated, but it can be reduced by placing restrictions on the implementations (e.g., hierarchical structure, call-by-value argument passing, type-checking). The remaining complexity can be made more pleasant to deal with by developing special notations to make formal statements easier to read and write, and by designing on-line tools to keep track of the large amount of material and to make the material easier to update. Some of the proposed notations, e.g., predicate transformations, are presented here. The implementation proofs presented here also involve a great many assumptions about the environment in which the programs must run. Such assumptions must be stated and accounted for either explicitly or implicitly in order for the proof to be meaningful in a real environment. Many formal methods have failed to be useful because of omissions of necessary assumptions.

The handling of complexity has already been done to a certain extent by the structuring inherent in the methodology. The system structure explicitly limits the effects of a state change to a particular part of the system. However, the hierarchical structure of the design is not sufficient in itself to make the proofs manageable. Additional constraints are: the way in which parallel programs can interact (Section II), and the constraints imposed by the programming language (Section IV). Some of the methods used for handling the remaining complexity are the specification language and the tools that support it (Section III), and certain notations used in the proof itself (Section V). Even with these complexity-handling methods, it shall eventually be necessary to build interactive tools to assist in the proof effort. One such tool is an interactive <u>proof manager</u> that (1) keeps track of what has been proved and what must be

proved, and (2) allows the prover to subdivide his proof into units that can be processed by an automatic deductive system.

The ultimate goal of the implementation proofs is to formally demonstrate, for each level of the system, that the implementation satisfies the specifications. Exactly what does this mean? If one proves programs in the applications environment of formal mathematics (e.g., factorial, multiplication by successive additions), where a complete set of defining axioms already exist, it is easy to say. For a large system in a physical environment that cannot be fully axiomatized, it is much more difficult to say. One can never be sure that one has axiomatized all of the properties of the environment that are relevant to the reliable behavior of the system. The first few attempts to do so, including this one, are certain to leave out some crucial properties. This chapter describes some of the most important properties and groups them into classes concerning parallelism (Section II), the specification language (Section III), and the programming language (Section IV).

The new results presented in this section are a method for handling parallelism (Section II), a programming language (Section IV), and a notation for describing the proof (Section V). The major additions to the methodology are the use of invariants (Section III) in the specifications, the mappings, and the proof. The major difference between the current semantics of SPECIAL (Section III) from those in previous descriptions occurs in the realm of exception handling. Many other points concerning the proof process, which have been discussed elsewhere in simplified form, are amplified and enlarged here.

There are several discrepancies between the complete PSOS design, presented in Appendix B, and the example system for implementation proofs, presented in Appendix C. Each module of the example system is a subset of the corresponding module in the complete system. The features not present in the example system are the following:

(1) Access codes in capabilities

(2) Store limitations for segments

(3) Creation and deletion of processes

(4) Arbitrary variation in the number of implementation capabilities for extended-type objects.

There are no formal obstacles to the incorporation of any of the above features into the system. However, it was determined that these features made the proofs more difficult to present, and they were left out in the interest of clarity.

Appendix C presents the syntax of ILPL (the abstract programming language) and the specifications, mappings. implementations, and a sample proof of the example system.

II Parallelism

Parallelism is the major issue that distinguishes the proof of an operating system from the proofs of other programs. The major problem attacked and solved here is to define what it means for the functions of an abstract machine (module) to be called concurrently by programs executing in multiple processes, so that it is possible to prove that an implementation for such a module is correct. This section

* Establishes a framework for discussion of the problem.
* Provides a definition of parallel execution.
* States -- and sketches the proof of -- a synchronization theorem stating that certain proofs of the implementation programs are sufficient to guarantee a correct implementation with parallel execution.
* Applies this definition to the methodology.
* Shows how the techniques are used in the example described in this section.

In the sequential model of program execution, there is a single program whose execution is manifested as a sequence of instructions $i_1,\ldots,i_n$ (or state transitions) on an abstract machine (Figure III-2.1(a)), The simplest way to describe program execution in the sequential model is to consider each instruction as an indivisible operation. An indivisible operation is defined with respect to an axis that designates time. An operation on an abstract machine is indivisible if and only if it appears as a point on the time axis that is distinct from the points occupied by all other indivisible operations. The state of an abstract machine is consistent for any point on the time axis not occupied by an indivisible operation. The module specification technique supports this view, because the specification for each O-function describes only the relationship between the old state and the new state of the abstract machine, without regard to any intermediate state. In program execution the new state of instruction x is the old state of instruction x+1. This model of program execution is illustrated in Figure III-2.1(b).

However, the model of indivisible operations does not hold in reality. Every realization of an abstract machine takes a finite nonzero time to complete each one of its instructions. During the time that an abstract machine is performing an operation, it may be in an inconsistent state (i.e., a state that is not part of the state space of the machine, or alternatively one in which the machine does not meet its specifications). Of course, there is a definite relationship between the two models, as shown in Figure III-2.1(c). But the model of indivisible operations is useful for describing

sequential execution of programs, because, for the purpose of analyzing these programs, it makes no difference whether the operations are indivisible or not. This is true because the model can be violated only by sampling the state while the abstract machine is in the midst of an instruction (when the state may be inconsistent). However, the state cannot be sampled except by the program doing the execution, which cannot execute an instruction until it has completed its previous one. It is thus possible to map the model of finite-length operations to the model of indivisible operations by supposing that an instantaneous state change takes place at some time during a given finite-length operation, as shown in Figure III-2.1(c).

In the parallel model of program execution, there is more than one program to be executed on a single abstract machine (that can include more than one physical processor). Each program in execution, represented by a sequence of instructions, is called a process, as shown in Figure III-2.2(a). However, the execution of multiple processes is realized as a single instruction stream (of indivisible operations) executing on an abstract machine. In this instruction stream instructions of different processes may be interleaved (Figure III-2.2(b)), with the constraint that, for every x and y, $i_{x,y}$ precedes $i_{x,y+1}$.

Again the model of indivisible operations is not observed in reality. Each instruction takes a finite nonzero amount of time to execute, and in addition the instructions may overlap in time (in the interest of efficiency) as shown in Figure III-2.2(c). In this case the discrepancy between the reality and the model of indivisible operations does matter, because the instructions in different processes can overlap. In the sequential model the indivisible operations could be mapped to the finite-length operations and still preserve the sequencing. In the parallel case in its full generality, there is no such mapping, because the semantics of overlapping instructions is not defined in the methodology, which does not axiomatize intermediate states in a module specification. As shown in the figure, the abstract machine may never be in a consistent state, because at least one instruction may always be in progress. However, the two models can be reconciled, by constraining the overlapping of execution so that its semantics can be defined in a manner that does not violate the spirit of the methodology. Without some definition of overlapping operations (i.e., parallel execution), there is no way to prove the correctness of shared implementations, because there is no formal system in which to carry out a proof.

One might ask, "Why allow overlapping execution at all if it causes so many problems in axiomatization and proof?" The answer is that at the heart of multiprogramming is the efficient sharing of physical resources. If the CPU were the only shared physical resource, it would not hurt efficiency to forbid the overlapping of abstract machine instructions, because it would always be possible to

achieve 100% CPU utilization, no matter how large the time slices were. However, the processes also share I/O devices, which run asynchronously with the CPU. Thus, if an abstract machine instruction involves an I/O operation, such as retrieving a page of memory from disk, it is more efficient to block the process and dispatch some other process, even though the current process is in the midst of an abstract machine instruction. If this is not done, the CPU becomes idle, and the response time for all processes suffers. On the other hand it is unacceptable to allow arbitrary overlapping of instructions. For example, instructions by two processes on the same object might have to be mutually excluded, because the object is in an inconsistent state during the instruction performed by a single process. The proper policy lies somewhere in between the two extremes.

Suppose it is possible to take a single abstract machine and generate a set of independent abstract machines that is collectively equivalent to the original machine (i.e., the original machine is the product machine of the set, or the set is a decomposition of the original machine). This can be done formally but is not presented here. This set of machines is called a partitioning of the original machine, and each member of this set is called a partition of the original machine. Each instruction of the original machine maps to a single instruction on a partition of the original machine. Often it is not possible to achieve a partitioning for an abstract machine, other than the trivial identity partitioning. However, it turns out that a nontrivial partitioning can often be accomplished, as is illustrated in the example below. Given a partitioning of an abstract machine, it is possible to forbid any overlapping in time of instructions executing on the same partition, and to allow unrestricted overlapping in time of instructions executing on different partitions. This restriction, called the partition exclusion, can be enforced in an implementation by synchronization primitives such as P and V, which can be used to form a single critical section for all code operating on a single partition. A graphical description of partitioning is given in Figure III- 2.3. The partition exclusion considerably simplifies the problem of describing the semantics of parallel execution, because each partition now has a well-defined sequence of instructions executing on it, as is true for the single abstract machine in the sequential model described above. Thus, any partition is in a consistent state during the time between the completion of one instruction and the commencement of the next. Note that even with finite-length operations, the beginning and ending of an instruction can each be thought of as instantaneous events.

The partitioning of an abstract machine allows a definition of the semantics of parallel execution, but by itself does not immediately provide a satisfactory mapping to the single sequence of indivisible operations of the sequential model. This is the case because, although each partition may be in a consistent state at some time, there may be no time at which all partitions are in a consistent state. However, just as in the sequential case the

discrepancy between the model of indivisible operations and the reality of finite-length operations does not matter, the discrepancy between the model of indivisible operations and the reality of parallel execution of instructions on a partitioned abstract machine with partition exclusion also does not matter. Since no process can determine the state of a particular partition, unless it executes an instruction on that partition (and such execution is regulated in time), it is possible to map the sequence of finite-length operations on each partition to an equivalent sequence of indivisible operations on that partition, as was done above for the sequential case. Now it remains to merge the sequences of indivisible operations for all processes in order to get a single sequence. The only problem in merging the many sequences of indivisible operations is what to do if two or more "indivisible" operations in different partitions occur at exactly the same time. If two operations A and B on different partitions occur at exactly the same time, they can be considered as happening in either of the two strict sequences "A;B" or "B;A". Since the two operations are independent (because they occur on different partitions), the effect of the two sequences on the original machine is exactly the same. Thus, the model of parallel execution with partitioning and partition exclusion can be mapped to the model of a single sequence of indivisible operations as is shown in Figure III-2.4.

The concept of a partitioning (explained above) must now be carried through to the implementation of an abstract machine, so that a definition of a correct implementation may be established. As has been described in the literature on the methodology, each function of an abstract machine is implemented by a program running on a lower-level abstract machine. In addition to the implementing programs, there is a mapping function that maps states of the lower-level abstract machine onto the states of the higher-level machine. An implementation p of an instruction i is correct with respect to a mapping function f if and only if it takes a lower-level state that maps up to an old state of i to a lower-level state that maps up to a new state of i. One difficulty of the partitioning approach is that a single partition of the higher-level state may not be mapped up to by a single partition in the lower-level state. In such cases a higher-level instruction might run on a single partition, but its implementation may not. This can happen in the case of distinct resources at the higher level being implemented by a shared resource at the lower level. The result of this problem was the imposing of a restriction on mapping functions that a partition at the higher level must be mapped to by a partition at the lower level. Adherence to this restriction can sometimes cause the addition of another level (as is done in the case of the table module in the example presented in Appendix C), but results in greatly simplified proofs. In addition to the restriction on lower-level partitions, the lower level must also provide synchronization primitives that enforce partition exclusion. These primitives are called "grab" and "release" in the example of Appendix C. Using these primitives, the protocol for the implementation of an instruction is

```
grab the partition;
perform the operation;
release the partition;
```

With this protocol, the operation can then be performed as if it were executing sequentially. Such protocol also eliminates the possibility of deadlock, because all resources for a higher-level instruction must be grabbed in a single indivisible lower-level operation. Of course this simple protocol does not work in certain routines in which complicated signaling must take place, such as I/O drivers, but is useful for resource sharing at most levels of the operating system.

It is now possible to state, and develop a proof for, a theorem stating that if certain properties are proved about a set of implementing programs, then it is a correct parallel implementation of an abstract machine. The theorem is called the indivisiblity theorem:

> If (1) the lower-level machine behaves as an indivisible sequence of operations with multiple processes, (2) each implementation program adheres to the grab-release protocol described above, (3) each program runs on the partition grabbed by it (which maps up to a single higher-level partition), and (4) each program is a correct sequential implementation of its corresponding higher-level instruction, then the set of programs is a correct parallel implementation of the higher-level machine, i.e., behaves as a sequence of indivisible operations on the higher level.

In order to develop a proof for this theorem, it is first necessary to define precisely what it means "to behave as a sequence of indivisible operations." To do this involves a formalization of the correspondence between the model of indivisible operations and the reality of finite-length operations. In the sequential model each instruction at the higher level is implemented by some sequence of instructions at the lower level, as illustrated in Figure III-2.5(a). Assume that the instructions at the lower level are indivisible. However, only the initial and final states at the lower level map up to distinct states at the higher level. When the lower-level abstract machine is between its initial and final states, the upper-level abstract machine may be in an inconsistent state. This model must be changed in order to guarantee that the instructions of upper-level abstract machine are indivisible.

In the new model, the mapping function and the lower-level instructions are changed as follows: the state of the lower-level machine always maps up to a valid higher-level state; and the entire sequence of operations at the lower level maps into a single instruction at the higher level. In the new mapping function definition, all intermediate states of a sequence of lower-level instructions implementing a higher-level instruction map up to the

initial state of the higher-level instruction (a phenomenon known as state clustering), as shown in Figure III-2.5(b). Suppose the original mapping function is the following:

```
higher_level_state: f( lower_level_state)
```

Then the new mapping function is the following:

```
higher_level_state: IF flag THEN f( lower_level_state)
                            ELSE f( lower_level_state_copy)
```

Initially the value of flag is FALSE. The first lower-level instruction sets flag to TRUE and copies the data in lower_level_state into lower_level_state_copy. This action is known as state copying. All intermediate lower-level instructions change only the data in lower_level_state, which leaves the value of the higher-level state unchanged. The last lower-level instruction resets flag to FALSE, causing the higher-level state to be derived from the data in lower_level_state, which should by then have been changed to reflect the upper-level state change. This operation is called changing the map. The mapping does not really change, only the incarnation of the lower-level state from which the higher-level state is computed. The transformation of the mapping function and the operations of state copying and changing the map are all formal manipulations that do not have to be implemented in practice. This is because the state cannot be sampled in the time between the first and last lower-level operations. Thus these transformations do not have to be explicitly written out, because State copying can thus formalize the correspondence between finite-length operations and indivisible operations, at least in the sequential case.

The state copying that must take place in the parallel case is a bit more complicated. Here the first lower-level operation makes a copy of only the partition of the lower-level state that it is operating on, and changes the map from the real partition to the copy. The last operation changes the map from the copy back to the real state, which by this time should map up to the new higher-level state. This is quite appropriate in the parallel case, because the first and last lower-level operations are "grab" and "release" respectively, which can be formally defined to perform the state copying and the changing of the map.

A proof for this theorem can now be sketched. It must be shown that if the higher-level machine starts out in a consistent state, it always remains in a consistent state, having arrived there from a sequence of transitions corresponding to instructions invoked by processes running programs on that machine. The four assumptions outlined above are postulated for the proof. The proof is in an induction on sequences of lower-level operations (a sequence of indivisible operations, according to assumption 1), which have no constraints on them, except for instructions within the same process (which are strictly ordered in time) and instructions in processes blocked attempting to perform a grab (which are not allowed to

execute until the a the process is unblocked by a release). Thus, the induction hypothesis is that the current lower-level state maps up to a valid higher-level state as described in the section on state clustering. The next lower-level operation can be any one of the following:

* A grab, in which case the partition indicated by the grab is copied but the higher-level state is not changed. Note that if that partition of the state has already been grabbed by another process, the attempted grab causes the calling process to block and is not officially executed. Thus it is impossible to execute a grab on a partition that has already been grabbed.

* Some intermediate lower-level operation (any instruction other than a grab or a release), which must operate only on a grabbed partition (by assumptions 2 and 3) and thus does not affect the higher-level state, which is mapped up to by a copy of the grabbed partition.

* A release, which changes the map and the higher-level state but does so in a way that corresponds to the correct implementation of some higher-level instruction, by assumption 4.

This completes the sketch of the proof.

In relating this parallel model of execution to the methodology, the most important issue is how to define a partition. Of course abstract machines in this description correspond to interfaces or levels in the methodology. A partition on a state of an abstract machine is a set of V-function positions that have certain properties, where a V-function position is a tuple consisting of a name for a primitive V-function of a level followed by a sequence of argument values. A V-function position is the most primitive repository of information provided by the methodology. Thus a partitioning is a set of disjoint sets of V-function positions with certain properties, whose union is the entire set of V-function positions of the abstract machines. The properties referred to above depend on the specifications for the O-, OV-, and visible V-functions of the abstract machine. Consider an instruction of an abstract machine to be a function instantiation, where a function instantiation is a tuple consisting of the name for an O-, OV-, or visible V-function, followed by a sequence of argument values. Each function instantiation must reference only the V-function positions contained in a single partition. A reference to a V-function position in an O-, OV-, or visible V-function specification is an appearance of that V-function position in either the exceptions, effects, or derivation of the function's specification. In implementation proofs it is necessary to generate the lower-level state partitions, by mapping each V-function position at the upper level to a corresponding set of V-function positions at the lower level. The system must provide a module with the synchronization

primitives "grab" and "release", and the user must provide some mapping from the arguments of grab to the lower-level partition, to carry out the proof.

The partitions for the small operating system example of Appendix C can now be described. In describing a partition, the following notation will be used:

```
< f(x), < g(x, y), for all y > >.
```

Here f and g are V-functions, and x and y are variables that can be used as arguments to the V-functions. The above expression means that there is a different partition for each value of x (because x is a free variable -- not quantified in the expression), consisting of the position f(x) and the set of positions g(x, y), for all y. Partitions need only be described for the non-primitive modules of the system, i.e., extended-types and tables. For extended-types, the partition is by type:

```
< is_type_manager(t), impl_length(t)
  < is_type(c, t), < impl_cap(c, t, i),
    original(c, t, i), for all i >, for all c > >.
```

The only purpose of the table module is to provide a partitioning that could be used by the implementing programs of the extended-type module. For tables there is only one partition:

```
< table(c), for all c >.
```

The mapped partition for extended-types is

```
< table(t), seg_exists( table(t)),
  < read( table(t), i), for all i > >.
```

For tables the implementation partition is

```
< seg_exists(st), < read(st, i), for all i > >,
```

where st is a parameter of the implementation for tables.

For the synchronization primitives "grab" and "release" in the example system, a partition is denoted by a vector of capabilities. For protection purposes, a capability is useful in denoting an object, which can also be described by the methodology as a set of V-function positions containing that capability as an argument. Grab and release were also specified to allow a process to grab more than one partition in a single indivisible operation (the operation blocks if any of the partitions have already been grabbed), by having "grab" take a vector of capability vectors as arguments. A partition must be uniquely defined by the capabilities passed as arguments to the implementation program, in cases where an abstract machine has more than one partition. However, the partition must also be protected from unauthorized grabbing by higher-level programs. Thus, because a

type defines a partition of the extended-type manager, the extended-type implementations could conceivably grab VECTOR (VECTOR(t)) and not endanger mutual exclusion. However, some higher-level program could also do the same thing, since both grab and t are available to the higher level programs. This could endanger the correctness of the implementation, because a higher-level program could cause a process to become infinitely blocked by executing a grab on an argument already grabbed by a higher-level program. This occurrence may not even be accidental, because there is nothing stopping two different levels from having partitions denoted by exactly the same argument! Fortunately the operating system structure has already presented a solution to this problem in the form of the capability mechanism. The integrity of lower-level data structures in the system has always been guaranteed by preventing higher-level programs from getting access to the capabilities for those data structures. This can be accomplished by requiring that the lower-level V-functions never return the capability as a value. Thus, the capability is a secret capability of the lower level. In order to prevent higher-level programs from grabbing its partitions, it is necessary for an implementation program to include a secret capability in its partition designation. It must also be proved, as part of the implementation proofs, that the implementation programs do not "give away" any secret capabilities. The secret capabilities used in the grab operations of the example system are "ex_secret" for the extended-type module and "st" for the table module. Thus, the arguments to grab for the extended-type and table modules are, respectively:

VECTOR(VECTOR (t, ex_secret))

and

VECTOR(VECTOR(st)).

The use of grab and release in the implementation programs can be seen in Appendix C.

III Semantics of the Specification Language

The semantics of SPECIAL has been characterized, but not formally defined. To some extent, this has not been necessary, because the assertions in SPECIAL are written in a notation based heavily on mathematical constructs. However, some of the constructs, especially those related to the methodology, are sorely in need of definition. Such definition is needed before any meaningful proofs can take place. This section defines the most important of these constructs. The following issues are presented, along with their implications for proof:

* Type and argument checking

* Mapping of modules repeated at different levels

* Mapping of types
* Exception signalling and returning of values
* Resource errors
* Assertions and invariants

A brief word on the significance of type checking with regard to proof is in order. For formal arguments to a function, a type declaration is equivalent to an input assertion regarding the type of the arguments (this assumes that type checking is performed on the programs that call the function); for results of functions, a type declaration is equivalent to an output assertion regarding the type of the result; for actual arguments to functions, type checking means a guarantee of the input assertion regarding the types of the arguments. Automatic type checking is done in the processing of both the specification language and the programming language. The presence of automatic type checking in both ILPL and SPECIAL minimizes both run-time type checking and the need for explicitly inserting assertions concerning an object's type. However, if a function is visible at the user interface, its code must perform run-time type checking on the arguments. This checking can be done by a level inserted between the highest system level and the lowest user level, whose sole purpose is to perform run-time type checking. Checking for the correct number of arguments can also be done in the same manner.

In the operating system a module can be visible at many levels and has the same specification for all levels at which it appears. Although the specification of a module is the same for different instances of the module (and even the code for implementation is usually the same), the state of the module usually does differ among its instances. For example, consider a simplified system structure, in which the higher level contains directories, segments, and capabilities, and the lower level contains segments and capabilities. Directories are implemented in terms of segments and capabilities. The set of segments that are part of the top level state must exclude the set of segments that implement directories; if this is not done, any operations on such segments would have to be specified at the top level (because a module specification must include all effects on the state). Then, the specifications of the directory module would have to include assertions about the effects of the directory functions on the segments that implement the directories. This is clearly undesirable (a violation of abstraction), so a method was devised for describing the "hiding" of parts of the lower-level state in the mapping functions. This can be done in the example here in the mapping between the higher-level and the lower-level instances of the capability module. The designator "capability" at the higher level is mapped to the set of capabilities at the lower level, minus those capabilities for segments that implement directories. Those capabilities hidden by the mapping are known as secret capabilities,

and it must be proved that programs at the higher level cannon get possession of any of them. Note that no special action is needed for the mapping of the higher-level segment module in terms of the lower level, because all V-functions of the higher-level module are defined only over the domain of the higher-level capabilities. The values of V-function positions (see Section II) whose arguments are not within the domain of the higher-level types, have a value of UNDEFINED. This is a constraint imposed by th methodology, for which specific statements in the specifications and mappings do not have to be made.

In a programming language with a built-in type mechanism that includes abstract data types (e.g., CLU, ALPHARD, GYPSY), there is an implicit mapping of types among the levels of a system programmed in such a language. Suppose that in such a language stacks were implemented in terms of arrays and integer pointers. The code for implementing stacks would automatically consider arrays and pointers at the lower level, when given an object of type stack at the higher level (a stack s might be operated on in terms of the array s.a and the pointer s.i). However, the methodology requires that such a mapping be explicitly made, because there is no assumption of a language facility to describe the mapping explicitly. Note that the mapping of types is not necessarily connected to the representation of abstract objects in terms of concrete objects. In the operating system, this is done by the extended-type mechanism, because the mapping is variable and must be under control of the operating system. One kind of type mapping, which is not done in the example, is that of types manipulated at the program level to their bit-string representations, in a more primitive description of the hardware. For capabilities this mapping could be

```
capability:  {VECTOR_OF {1,0} bs |
                        LENGTH (bs) = bit_string_length + 1 AND
                         bs[1] = 1}
```

where "bit_string_length" is as specified in the previous section, and "bs[1]" represents the tag bit distinguishing between capabilities and all other kinds of machine words. The "abstract programs" that might implement operations on capabilities (if such programs existed), could use this representation. Of course, integers, reals, etc., could also be mapped down in this way. The mapping of secret capabilities, as described in Section II, must also be performed.

The signalling of exception conditions and the returning of values by functions of a module are closely related, because both actions are ways in which the abstract machine communicates with the program that runs on it. That is, the calling program must know whether an exception has been signalled (and if so, which one), and whether a value has been returned (and if so, what that value is). No value is returned in the case of an O-function, or in any case where an exception occurs. The most useful formal method is consider both the signalling of exceptions and the returning of values as state changes in the lower-level abstract machine, in the values of

the meta V-functions VALUE() and ERRORCODE(). If the specification of an OV-function that returns the value v looks like this,

```
EXCEPTIONS
        ex1;
        ex2;
         .
         .
         .
        exn;
EFFECTS
        ef1;
        ef2;
         .
         .
         .
        efm;
```

then its output assertion (before mapping) is

```
IF ex1 THEN
     'ERRORCODE() = "ex1" AND 'VALUE() = ?
   ELSE IF ex2 THEN
     'ERRORCODE() = "ex2" AND 'VALUE() = ?
       .
       .
       .
   ELSE IF exn THEN
     'ERRORCODE() = "exn" AND 'VALUE() = ?
   ELSE 'ERRORCODE() = "NORMAL" AND
     'VALUE() = v AND
     ef1 AND ef2 AND ... AND efm;
```

The value of the V-function "ERRORCODE()" is tested by the exception-handling mechanism of the calling program in ILPL, so that changes in control or returned values can be based on the detection of lower-level exceptions. The value of the V-function VALUE() is used by the expression evaluation mechanism of ILPL. Note that the above notation can also be used for O- and V-functions, by assuming that v is UNDEFINED (in the case of O-functions) or that m is 0 (in the case of V-functions).

However, a certain kind of exception condition, called a resource error, is handled slightly differently. The problem with a resource error is that the conditions under which it occurs are invisible to the higher level; if such conditions were visible, many "implementation details" would become part of the abstraction. For example, if the maximum number of segments allowed were a complicated, function of the implementation of segments on primary and secondary memory, it might be necessary to include this information in the specification of the segment module. However, the inclusion of this information would make it difficult to use this specification of the segment module with a different implementation.

Instead, an exception named "RESOURCE_ERROR" could be placed in the specification of the function that creates a segment, leaving unspecified the exact conditions under which there are too few resources to create a new segment. Thus, the exact conditions that causes resource error are hidden, as a price for preserving abstraction. If the EXCEPTIONS section of an OV-function with value v looks like this:

```
EXCEPTIONS
        .
        .
        .
        exi;
        RESOURCE_ERROR;
        exi+1;
        .
        .
        .
```

then the output assertion of the function looks like this:

```
        .
        .
        .
ELSE IF exi THEN
   'ERRORCODE() = "exi" AND 'VALUE() = ?
ELSE ('ERRORCODE() = "RESOURCE_ERROR"
         AND 'VALUE() = ?) OR
     (IF exi+1 THEN
   'ERRORCODE() = "exi+1" AND 'VALUE() = ?
        ELSE ...
        .
        .
        .)
```

Usually the resource error is the last one checked, because any attempt to get resources should be delayed until all other requirements are satisfied.

Considerations of proof made it necessary to add the concepts of initial assertions and invariants to the methodology. The former construct has been added to the specifications, and the latter to the mapping functions.

The purpose of the ASSERTIONS section in the specifications is to enable formal statements to be made about the properties of a module's parameters. Assertions can be made about the values of a single parameter, or about the relationships among parameters. In the implementation of a module, the initialization program must satisfy the module's assertions on output, as well as the initial conditions of all V-functions. An example of module assertions is found in the capabilities module of Appendix C.

The purpose of the INVARIANTS section of the mapping functions is to place additional constraints on the lower-level state, to aid in the proof of the implementation programs. A program is most effective when it can assume on input restrictive properties of the state of the machine on which it runs. The invariants are precisely those properties. If an implementation program is allowed to assume that the invariants are true on input, all implementation programs (including the initialization program) must guarantee that the invariants are true on output. For example, in the representation of tables in Appendix C, the invariants state that

(1) The size of the implementing segment is even.

(2) An entry occurs at most once in the segment.

(3) A null entry is designated by the capability for the implementing segment.

Part of the implementation proofs involves demonstrating that the invariants are satisfied.

IV Semantics of the Programming Language

In order to write abstract implementations for some of the levels of PSOS, it became necessary to find a descriptive medium in which to write the programs. All existing programming languages were deemed unsuitable, because they lacked features to relate directly to the methodology and were too complex to allow verification condition generation and proof to be simply done. The language developed is not much more complex than assembly language (this is not too surprising because capability-based systems, especially those which support abstraction, easily lend themselves to assembly language programming), and for this reason is called an Intermediate Level Programming Language or ILPL.

Whenever a programming language is used to write a verified program, the compiler for that language onto the target machine must be verified. However, two related issues should be noted:

* ILPL is sufficiently simple that verifying a compiler would be easier than for most higher-level languages.

* Reliability in compilers can be achieved much more easily than reliability in operating systems, because compilers are amenable to exhaustive testing, whereas operating systems (because of their highly parallel nature) are not.

The entire grammar for ILPL is given in Appendix C, and the implementations for the table and extended-type modules are written in ILPL. Several features distinguish ILPL.

* Restriction of expressions that change the state. Places in which state changes can occur are in an C-function call (a single statement), an assignment to a V-function reference (in which only the assigned variable changes), and an assignment to an O-function reference (in which the assigned variable changes and the effects of the OV-function take place). Thus, no state changes can take place in evaluating the BOOLEAN expressions for IF...THEN...ELSE and WHILE statements or in evaluating the arguments for any C-, V-, or OV-function. This restriction is enforced by prohibiting references to OV-functions in any expression that is an argument to a function or an operator of ILPL (e.g., +).

* Protection according to module boundaries. All programs that implement a module are included in a block of code called the program module. Besides the programs that implement the visible functions of the module, there are an initialization program, declarations of read-only parameters, and code for subroutines shared by the implementation programs.

* Restricted usage of variables. ILPL allows local variables that are declared for each subroutine or program implementing a visible function. A local variable survives only for a single activation of a procedure call. Local variables are used mainly for indexing or temporary storage, since most data structuring is performed by the V-functions of lower-level modules. A variable can be of any valid type supported by SPECIAL, except sets. Parameters, i.e., those shared by more than one procedure of the module and that survive between activations, are read-only. The initialization program must initialize all such variables. The loader can then copy the values for these shared variables in-line into the code for the implementations. Shared variables are handled in this way to restrict the part of the lower-level state that can be changed to V-function values.

* Facilities for exception detection and handling. The facility for exception detection is the predefined O-function "RETURN(<exception name>)", which sets the identity of the exception detected by the function implementation to be "<exception name>". An exception name is either the string corresponding to any exception of the function that is being implemented or the string "NORMAL". For example, if an exception condition for a function is

```
ptr(s) = 0
```

then the implementation program can execute the statement

```
RETURN( "ptr(s) = 0")
```

to signal this exception. The statement

```
RETURN( "NORMAL")
```

indicates to the calling program that no exception has been detected. At least one RETURN statement must be executed in every implementation activation. The RETURN statement sets the value of the meta V-function ERRORCODE() (explained earlier) for use by the exception-handling mechanism of the higher-level program. The facilities for exception handling state precisely what is to be done in the event that an exception is detected during the execution of an C-, V-, or OV-function. The first construct, called an exception case statement, specifies a sequence of statements to be executed when an exception is detected for a reference to an O- or OV-function, or when an exception is detected in a V-function reference that is the right-hand side of an assignment statement. An exception case statement is a simple statement (either an O-function reference or an assignment statement), followed by list of exception names, each associated with a sequence of statements to be executed if that exception is detected. The following (taken from the code of "insert_table", presented in Appendix C) is an example of an exception case statement as part of an O-function reference:

```
DO change_seg_size(st, seg_size(st)+2) WITH
  "RESOURCE_ERROR": RETURN("RESOURCE_ERROR");
  "bad_size(j)": RETURN("RESOURCE_ERROR");
  "NORMAL": BEGIN
    write(st, i, c);
    write(st, i+1, w);
    RETURN("NORMAL")
  END
OD
```

The semantics of the exception case statement is that, in the event that the lower-level exceptions "RESOURCE_ERROR" or "bad_size(j)" are detected in the call to "change_seg_size", no effects are to be executed and the statement "RETURN("RESOURCE_ERROR")" is to be executed, signifying that the higher-level exception "RESOURCE_ERROR" is to be signalled. It must also be proved that the exceptions not mentioned in an exception case statement do not occur. The other construct, called the exception case expression, replaces a V- or OV-function reference with a specific value when the reference raises an exception condition. An exception case expression is a list of exception names (except "NORMAL"), each associated with an expression (without side effects) to be substituted for the value returned by the V or OV-function reference. For example, in the following expression

```
read(st, i) {"address_bounds(s, i)" : c} ~= c
```

c would be substituted for "read(st, i)" if the exception "address_bounds(u,i)" were detected in the function reference, resulting in a value of FALSE for the entire expression.

* Call-by-value and multiple results. The methodology requires that formal arguments to all functions be call-by-value. This restriction is enforced in ILPL, with the additional requirements that arguments to all subroutines (i.e., procedures internal to the module implementation) be call-by-value and that formal arguments may not be assigned to. The latter restriction simplifies verification, because arguments (and read-only variables) can now be referred to as mathematical variables, whose values do not change throughout execution of the program, rather than as program variables, whose values do change. The call-by-value restriction for subroutines makes it difficult to allow for a subroutine that returns multiple values, and for the assignment of the values to more than one variable. The solution employed here is to write a subroutine that returns an object of a structured type of SPECIAL. The result of this subroutine need not be assigned to a similar structure in the program, but may instead be assigned to a tuple of individual program variables. For example, if a subroutine f with one INTEGER argument returns an object of the following type

```
STRUCT( BOOLEAN x; INTEGER y)
```

then the following assignment statement could be written

```
b, i <-  f(j)
```

where b is BOOLEAN, and i and j are INTEGER. Note that b and i need not be part of a structure in the calling program.

At the top structure of ILPL is a program module, corresponding to the module specifications of the methodology. A program module contains the following:

* Declarations of shared variables.

* An initialization program for the module.

* Procedures called function implementations that implement each of the visible functions of the module specification. They are labeled OFUN_PROG, VFUN_PROG, and OVFUN_PROG, for O-, V-, and OV-functions, respectively.

* Subroutines that can be called only from programs inside the module. These are often needed for reasons of space efficiency, and because programs within the module will often share large sections of code that need only be proved once. They are labeled OFUN_SUBR, VFUN_SUBR, AND OVFUN_SUBR, for O-,

derived V-, and OV-functions respectively (note that a subroutine cannot implement a primitive V-function, because subroutines have no state of their own). For subroutines, the specifications are included with the implementations, as assertions for verification purposes.

The structure of a typical program module looks like this

```
PROGRAM MODULE <symbol>
  TYPES . . .
  DECLARATIONS . . .
  PARAMETERS . . .
  DEFINITIONS . . .
  EXTERNALREFS. . .
  INITIALIZATION
    <code for initialization program>
  IMPLEMENTATIONS
    <code for function implementations and subroutines>
END_MODULE
```

The first three paragraphs are similar to their corresponding parts in a module specification.

The next level of ILPL concerns the syntax of a function implementation. A function implementation starts with a header that looks almost exactly like the function header of a specification, e.g.,

```
OVFUN_PROG create_segment() --> capability c;
OFUN_PROG  write(capability s; INTEGER i; machine_word w);
VFUN_PROG  impl_cap(capability c, t; INTEGER i)
               --> capability c1;
```

Next comes a list of declarations of variables that are local to the procedure, followed by the executable statements:

```
DECLARATIONS d1; d2;...; dn;
<program>;
```

where a program is defined in extended BNF* as

```
<statement> | BEGIN <statement list> END
```

Each statement in a statement list can be any of: (1) a simple

---

* In extended BNF, <...> means that the enclosed symbol is a nonterminal of the grammar; [...] means that the enclosed construct is optional; {...}* means that the enclosed construct can occur 0 or more times, {...}+ means 1 or more times, and {...|...|...} means an alternative among the enclosed constructs. All special characters that are terminal symbols have been enclosed in single quotes (e.g., ':').

statement, i.e., an assignment or an O-function reference (including RETURN); (2) an exception case statement, explained above; (3) a control statement, i.e., an IF..THEN...ELSE, FOR or WHILE statement; or (4) a TYPECASE statement, similar to the TYPECASE expression in SPECIAL.

The syntax for the IF...THEN...ELSE statement is as follows

```
IF <expression> THEN <statement list>
                [ELSE <statement list>] FI
```

where the expression may not contain any OV-function reference. The syntax for a WHILE statement is

```
WHILE <expression> [ASSERT <assertion>]
      DO <statement list> OD
```

where the expression may not contain any OV-function reference and the optional assertion (necessary for verification) is an expression at the assertion level of SPECIAL. The syntax for the FOR statement is similar to that of the WHILE statement.

V Protocol for Proof

This section discusses both the steps involved in the proof of implementation, and the technique used for each of the steps. The steps include proofs about parallelism, and the techniques include some notational changes that make Floyd's method of verification easier to understand.

The following are inputs to the proof effort:

* The specifications for the module to be implemented
* The specifications for the lower-level modules
* The mapping functions (including invariants)
* The lower-level programs (including the initialization program)
* The lower-level partition and the mapping between the argument to grab and partitions.

The proof effort involves the following aspects:

* Prove that the invariants are maintained by each function implementation.
* Prove that the lower-level programs are correct with respect to the invariants (as input assertions), and to the mapped specifications and exception conditions (as output assertions).

* Prove that each lower-level program has grabbed its partition in the correct way and "runs" on the partition corresponding to the capability tuples that it has grabbed (the concept of "running" on a partition has been defined in Section II).

* Prove that the initialization program is correct with respect to the initial values of the lower-level specifications (as input assertions), and to the invariants and the mapped initial values of the higher-level specifications (as output assertions).

Some of the proofs are "textual" in nature. A textual proof is a proof that appeals to examination of the program text to show that an assertion must be satisfied. Textual proofs are often used to show that invariants are satisfied and that a program "runs" on its partition. The following is an example taken from the proof of implementation of the table module, presented in Appendix C. If the invariant to be satisfied is "seg_exists(st)", then a textual proof might appeal to the fact that a program does not execute the function "delete_seg(s)" for any s. Because "delete_seg(s)" is the only operation that could change "seg_exists(s)" from TRUE to FALSE, this would prove that if "seg_exists(st)" were TRUE at the beginning of the program, then it is TRUE afterward (thus satisfying the invariant).

The method used for correctness of programs is based on the inductive assertion method of Floyd. Floyd's method has two major variants. In verifying arbitrary flowchart programs, it is customary to break the flowchart of the program up into "simple paths", each simple path having an input assertion, an output assertion, and a fixed number of program statements in between. A separate verification condition is generated for each path. On the other hand, in a program with a nested control structure (e.g., that provided by ILPL), it is customary to "push" the output assertion backward through the program text. This second method produces a single verification condition, along with some auxiliary conditions to be proved. The second method is used here.

Most program verification methods totally separate the process of verification condition generation from the process of deduction (or proof of the verification conditions). In these methods the generation of verification conditions is done mechanically, often producing an extremely long formula (perhaps several pages long) which is then attacked by the deductive system, with some help from the user. In examining these verification conditions, it can be readily observed that about 90% of the proof consists of trivial simplification and 10% consists of the "interesting" deductions to be performed. Much of the needless length in such a formula comes from the fact that unsimplified expressions are repeatedly substituted for terms in the formula as the formula is being "pushed back" through the program. These unsimplified expressions expand even further as the formula continues to be pushed back. This results in the

repeated presence of long expressions that can be written as very simple expressions. For example, suppose the assertion

```
FORALL i | i >= 1 AND i <= char_length():
        char(i) <= char(i+1)
```

is pushed back through the O-function reference

```
insert(j, k)
```

whose specification is as follows:

```
insert(x, y)
    INPUT:  0 <= x AND x <= char_length();
    OUTPUT:
            'char_length() = char_length() + 1;
             FORALL z: 'char(z) =
                (IF z <= x THEN char(z)
                   ELSE IF z = x + 1 THEN y
                   ELSE char(z-1))
```

The resulting unsimplified output assertion becomes the following:

```
0 <= j AND j <= char_length() AND
FORALL i | i >= 1 AND i <= char_length() + 1:
     (IF i <= j THEN char(i)
        ELSE IF i = j+1 THEN k
        ELSE char(i-1))
  <= (IF i+1 <= j THEN char(i+1)
        ELSE IF i+1 = j+1 THEN k
        ELSE char(i+1-1))
```

This is not only difficult to understand but is excessively long. Upon simplification of the nine possible cases, this assertion becomes

```
char(j) <= k AND k <= char(j+1) AND
FORALL i | i >= 1 AND i <= char_length():
      char(i) <= char(i+1)
```

which is both simpler and easier to understand.

In the conventional method the deductive system and the human being guiding it are presented with a needlessly long formula to prove. The task of deduction becomes more of a chore, thus decreasing the reliability and believability of the proofs, even taxing the ability of current automatic theorem provers, which are already considered marginal for the logical formulae involved in program verification.

The method used in the proofs presented here is to combine the tasks of verification condition generation and deduction. It contains the following specific advantages over the conventional method:

* Incremental simplification of the verification conditions as they are being generated, vastly reducing the amount of text to be manipulated (as shown above).

* Manipulation of unexpanded substitution rules, which reduces text and allows the delaying of expansion until maximum simplification can be performed. This further reduces the amount of text to be manipulated.

* Closer association of the proof with the program. By combining verification condition generation and proof, the program text is now closely associated with the deduction process. This improves the understandability of the proofs.

Verification condition generation in the example proofs is done by expressing the effects of O-functions and the semantics of statements in the programming language as substitution rules. An assignment statement is a substitution rule for a variable, and an O-function is a substitution rule for V-functions (in this case, entire functions receive new values). Assertions to be pushed back are expressed as abstract predicates, taking any V-functions and variables that may change during the course of the program as formal arguments. In the example above, the assertion about the values of "char(i)" could be defined by the abstract predicate "P(j, char, char_length)". The effects of the O-function "insert(x,y)" could be expressed as the two substitution rules

```
char_length -> LAMBDA . char_length() + 1
char -> LAMBDA z . IF z <= x THEN char(z)
                   ELSE IF z = x+1 THEN y
                   ELSE char(z-1)
```

The LAMBDA notation is used to express the definition of a new function according to the well-known notation of lambda calculus (McCarthy [60]). When an assertion is pushed back, the substitutions are made and the input assertions are conjoined to the transformed predicate. When P is pushed back through the O-function call "insert(j,k)", the abstract predicate becomes

```
0 <= j AND j <= char_length AND
P(j,
  (LAMBDA z . IF z <= j THEN char(z)
              ELSE IF z = j+1 THEN k
              ELSE char(z-1)),
  (LAMBDA . char_length() + 1))
```

When an assertion is pushed back through a sequence of O-function calls, the argument expressions can often be simplified before the predicate is expanded.

The semantics of ILPL is now described in terms of verification condition generation. The notation of Dijkstra's predicate

transformers (Dijkstra [75]) can be used to abstractly describe a substitution rule. For example, the effect on the abstract predicate "P(f)" of the O-function call "O(args)" can be written "T[O(args)]P(f)". The semantics of the statement "RETURN(exp)" on P is

```
T[ ERRORCODE --> LAMBDA . exp ]P
```

The semantics of

```
IF b THEN c ELSE d FI
```

on P is

```
(b AND T[c]P) OR (NOT b AND T[d]P)
```

The semantics of

```
WHILE b ASSERT a DO c OD
```

on P is a, where the two formulae

```
NOT b AND a => P
```

and

```
b AND a => T[c]a
```

must be proved separately. If a statement s contains the following V-function reference (with an exception case expression):

```
V(args) {"ex1": exp1}
```

but V also contains exceptions ex2 and ex3 that are not handled, then the semantics of the V-function reference are

```
IF ex1 THEN exp1 ELSE V(args)
```

and the semantics of "T[S]" on the predicate P are

```
(NOT ex2) AND (NOT ex3) AND T'[s]P
```

where "T'[s]" is the effect of s without the consideration of exceptions. Consider the following statement

```
DO O(args) WITH
  "ex1": s1;
  "ex2": s2;
      ... ;
  "exn": sn;
  "NORMAL": s0
OD
```

where the si are statement lists. The semantics of this statement on P are

```
IF ex1 THEN T[s1]P
  ELSE IF ex2 THEN T[s2]P
  ELSE...
  .
  .
  .
  ELSE IF exn THEN T[sn]P
  ELSE T'[ O(args) ](T[s0]P)
```

where T' is the transformation on an O-function reference without the consideration of exceptions.

These rules are applied in the proof of the example.

VI Description of the Example

The example presented in Appendix C is an implementation of the extended-type module in terms of the segment and capability modules. The design presented here is one in which an extended-type object can have only a fixed maximum number of implementation capabilities, which is the same for every object of a given type. This is a subset of the module in the operating system design presented in this report; the subset loses almost no generality, and is much easier to implement. A diagram of the system is shown in Appendix C. Appendix C contains all of the specifications and mapping functions for the sample system in SPECIAL (these have been systactically checked by the specification handling program), and the implementations for the table and the extended-type modules in ILPL (these have not been syntactically checked because the processor for ILPL has not yet been implemented).

The representation consists of one segment, called the type segment, for each type maintained by the extended-type manager. The first location of the type segment contains the entry size (i.e., two plus the number of implementation for each object of that type). This is followed by a sequence of entries, each of which is headed by a capability for an extended-type object, followed by a fixed number of slots, which each contains either an implementation capability if one exists or a 0 if none exists. A null entry is headed by a 0. The end of the entry is a bit vector, for which a 1 in the ith position indicates that the ith implementation capability is "original", i.e., created by the extended-type manager particularly for this extended-type object.

There is also a segment called the central table, whose keys are type manager's capabilities and whose values are capabilities for their respective type segments. The maintenance of the central table creates difficulties in the handling of parallelism. This is because it was decided for efficiency reasons to have a separate partition for each type maintained by the extended-type manager. However, all type managers must reference the central table, so the partitions of

each type manager overlap (in the V-function values pertaining to the segment of the central table). To ameliorate this difficulty, a new level, containing the central table, has been inserted between the segment and extended-type levels. This level has a single partition containing the entire table. This change makes all references to the central table into indivisible operations so that segment operations by different processes on the representation of the central table do not overlap in time. This design decision is an example of how considerations of parallelism dictate the placement of levels in a system.

Further discussion of the example appears in-line in Appendix C,

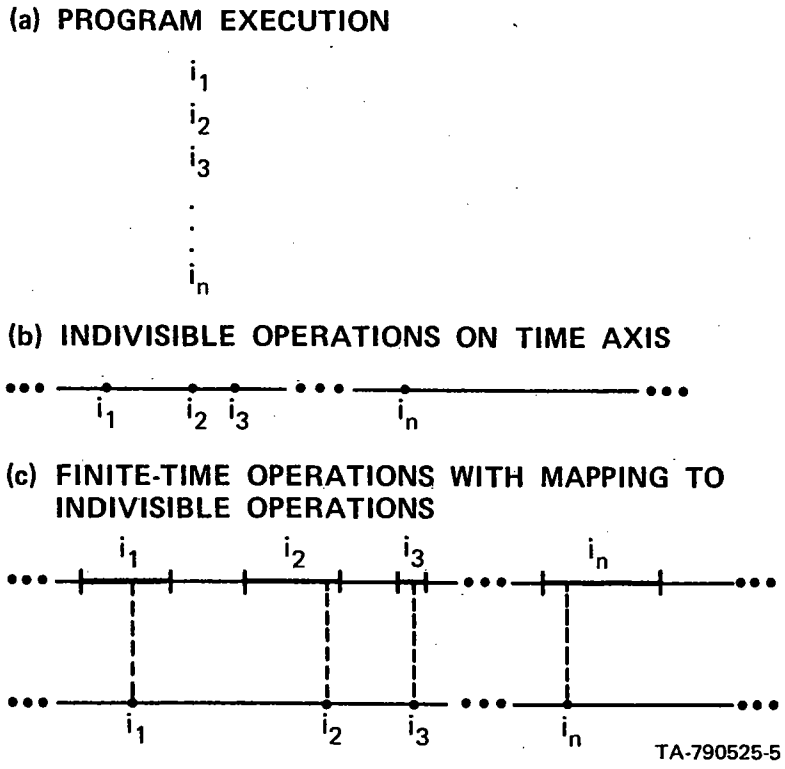


FIGURE III-2.1 PROGRAM EXECUTION—SINGLE PROCESS CASE

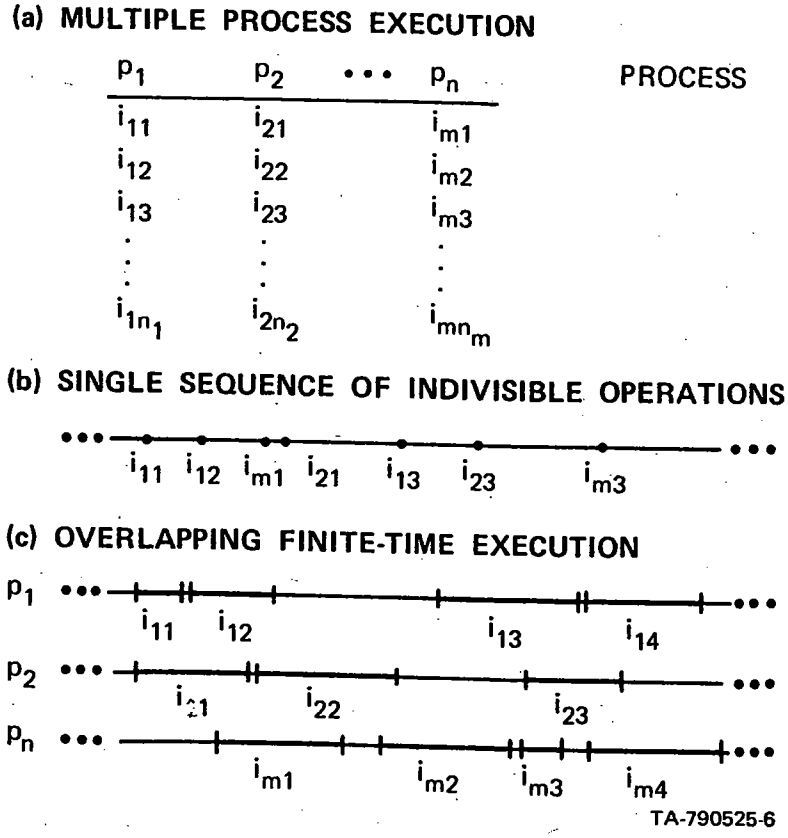
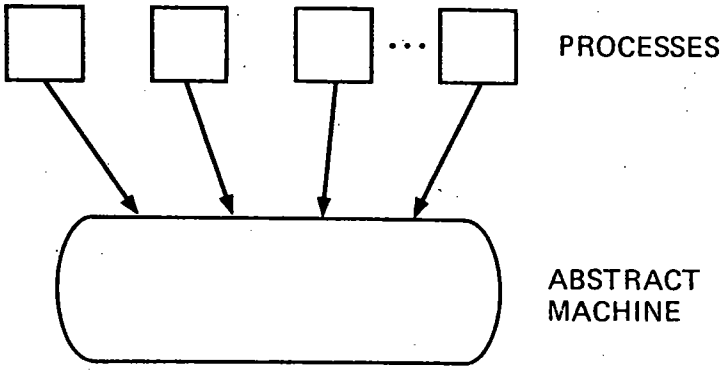
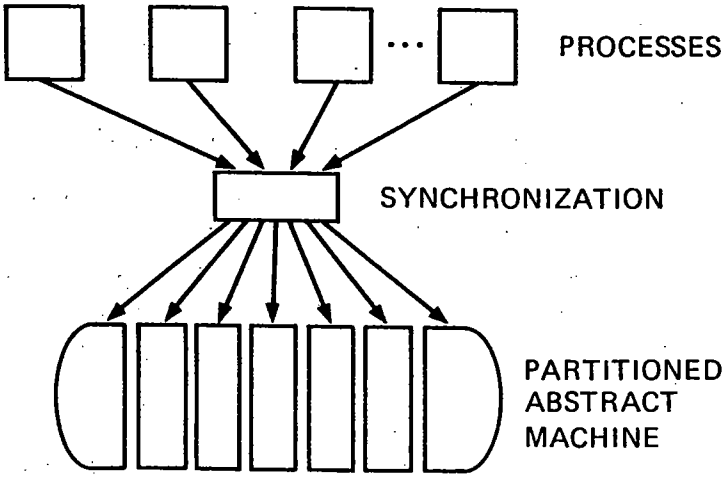


FIGURE III-2.2 PROGRAM EXECUTION—MULTIPLE PROCESS CASE



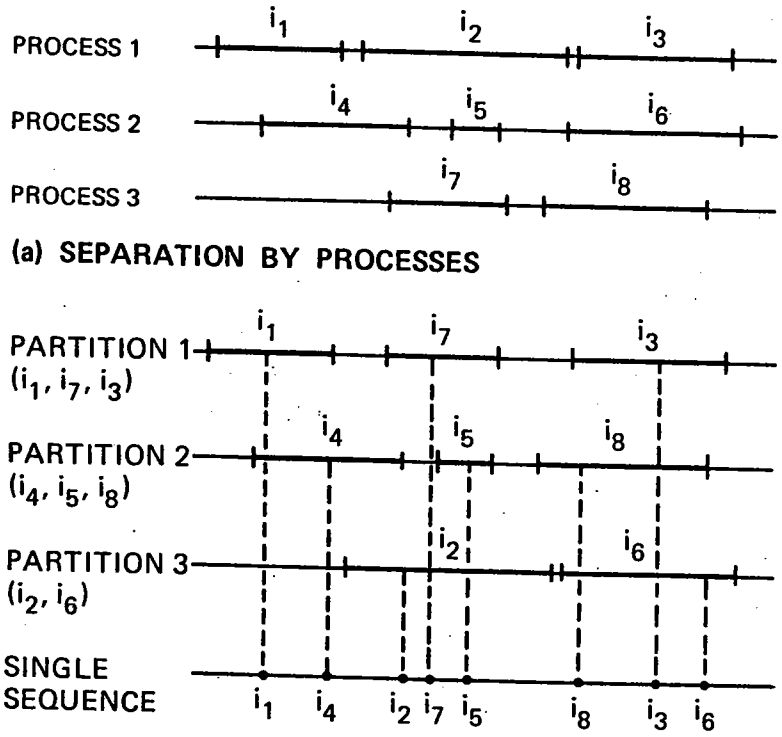
(a) MULTIPLE PROCESSES SHARING A SINGLE ABSTRACT MACHINE



(b) MULTIPLE PROCESSES SHARING A PARTITIONED ABSTRACT MACHINE

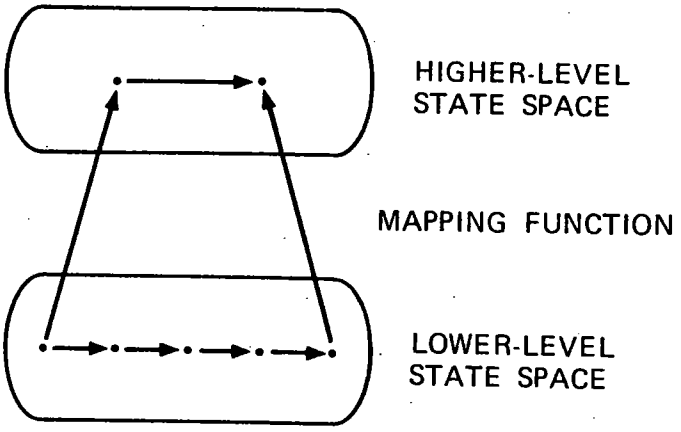
TA-790525-7

FIGURE III-2.3 PARTITIONING

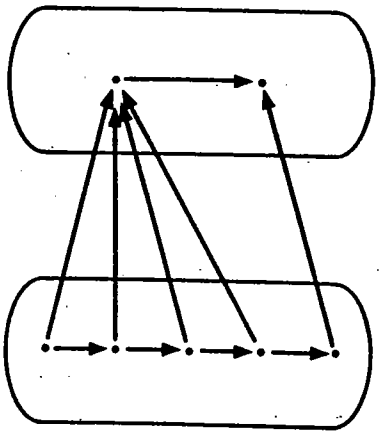


TA-790525-8

FIGURE III-2.4 PARALLEL EXECUTION WITH PARTITIONING



(a) MODEL OF FINITE-LENGTH OPERATIONS (THE HORIZONTAL ARROWS ARE INSTRUCTIONS)



(b) MODEL OF INDIVISIBLE OPERATIONS

TA-790525-9

FIGURE III-2.5 ABSTRACT IMPLEMENTATIONS

PART IV
APPLICATIONS

CHAPTER IV-1
MANAGEMENT OF OBJECTS WITH MULTILEVEL SECURITY

INTRODUCTION

This chapter is concerned with the sharing of the operating system among users whose security levels (defined precisely below) are not necessarily identical. Although the system is being shared, information should not be able to flow from a user at a given security level to one at a lower security level. The basic assumptions are that (1) users at a low security level might attempt to use the system as a channel for acquiring information from a user at a higher security level, (2) users at a high security level might attempt to pass information to a user at a lower security level. The first assumption is intuitively appealing, but the second is not -- since surely a human user could give a document to another user outside the boundaries of the system. However, each user executing on the system is trusted as a human being, but is extended by programs that could have been written by untrustworthy programmers who might collude with users at low security levels. Thus the system must not be used as a channel for transmitting information downward to lower security levels.

The current DoD practice is to permit a system to be shared, at any given instant, only among users all operating at the same security level. The "system" boundaries enclose the cpu, I/O devices and all memory accessible via the calling of some system function. Before users of a different security level are allowed access to the system, all of the system state is cleared. This conservative approach is undesirable because

* the loss of efficiency associated with flushing a system can be severe;
* users can not expect to get immediate access to the system;
* legitimate sharing of information is cumbersome.

For example, there is no security violation in a user reading a document produced by a different user at a lower security level; however, with the current practice, a distinct copy of a document must exist for each security level at which it can be read.

In the system discussed here, at any instant processes can exist for users at different security levels. The system will prevent any "downward" flow of information, initiated by a user doing the transmitting or the receiving. The purpose here is two-fold, namely

* to show that a system satisfying this multilevel security goal can be specified, and proven with respect to a formal statement of of the model developed by Bell and La Padula [74]. (An extended version of the model is introduced here.) For this model, Millen [75] has specified and proven an operating system kernel. The work described here differs from Millen's in the following respects: (1) the approach given here more closely approximates a complete operating system, (2) the formal statement of the security goals given here is more general, and (3) the specification format and the design decisions given here permit more easily developed proofs.

* to show that the multilevel security system can be easily and efficiently implemented with the basic provably secure operating system (PSOS). A particular challenge here has been to retain the capability as the mechanism for the read and write access of segments (documents) in order to achieve high efficiency.

This chapter is intended to be illustrative of how multilevel security can be efficiently implemented on top of PSOS (or efficiently embedded within PSOS, if the system itself were to support multilevel security for all applications). It is not intended to imply acceptability of the particular model for multilevel security, but rather as an indication of how such a model can be supported.

The following sections present the meaning of multilevel security in general terms, an informal statement of the multilevel security principles, the design decisions implied by the system interface, a formal model of multilevel security, a formal mapping from the model to the specification language, illustrative proofs of the specifications, the implementations of the system on PSOS, additional issues involving design and proof, and limitations of the model. The specifications are included in Appendix D.

MULTILEVEL SECURITY

Each user of the system has one or more independent processes operating solely on his behalf. Each process has associated with it a CLEARANCE and a CATEGORY SET. The system has a fixed finite number of clearances that are totally ordered by the relation "greater than". For example, the clearance TOP

SECRET is greater than SECRET, which is greater than CONFIDENTIAL, which is greater than UNCLASSIFIED. For convenience, it is assumed that the clearances are a set of integers.

A category set is any subset of the set of all possible categories. Examples of categories might be ATOMIC and NATO. The combination of a clearance and a category set is called a SECURITY LEVEL or equivalently ACCESS LEVEL; for simplicity, it is often called just a LEVEL, when ambiguity is not likely to arise. A security level L1 is said to be greater than or equal to a security level L2 whenever the clearance of L1 is greater than or equal to the clearance of L2, and the category set of L1 is an improper superset of the category set of L2. L1 is greater than L2 whenever L1 is greater than or equal to L2 and L1 is not identical to L2. Thus the security levels can be partially ordered as a lattice. A process operating at a security level of <SECRET, {ATOMIC, NATO}> has more POWER than one operating at <SECRET, {ATOMIC}>, but two processes with respective security levels <SECRET, {ATOMIC}> and <SECRET, {NATO}> are not comparable.

A system is MULTILEVEL SECURE if and only if, for any two processes P1 and P2, unless the security level of P1 is higher than or equal to the security level of P2, then there is nothing that P2 can do to in any way affect the operation of P1. That is, P1 is not able to know anything about P2, not even the existence of P2. This constraint implies that P2 cannot affect the operation of P1 using an intermediate process P3 that is incomparable with both P1 and P2 (i.e., neither higher nor lower in the lattice). Because a process at a particular security level cannot in any way affect a process at a security level that is not at least as high as that of the process, it is not possible for the process at the higher level to transmit information to the process at the lower level, either directly or via an intermediate process. Therefore, INFORMATION CAN ONLY FLOW UP OR REMAIN WHERE IT IS, i.e, can only flow to processes of higher or equal security level.

The above constraint on information flow prohibits the passage of information from P1 to P2 if the security level of P2 is not greater than or equal to that of P1. Thus it prohibits information flow between two processes whose security levels are incomparable in the lattice sense. This constraint is consistent with the real military security situation, since -- for example -- an individual whose category set contains only ATOMIC cannot pass information to an individual whose category set does not contain ATOMIC, independent of the latter's clearance or the other components of his category set.

In order to guarantee the upward flow of information, Bell and LaPadula assigned secuirty levels to all data channels and

data repositories of the system. (Hereafter, data channels and data repositories are called OBJECTS). A process P can READ (extract) data from an object O only if the security level of P is at least that of O. A process P can WRITE (insert) data into an object O only if the security level of O is higher than or equal to the security level of P. These are the rules formulated by Bell and La Padula [74] that guarantee the upward flow of information. Unfortunately, these rules are overly restrictive. A process P writing data into an object O with a lower security level clearly violates these rules, but if no process with a security level lower than that of P can read the data, the system is still multilevel secure by the above general definition. At first glance, such an object appears to be "write-only", and hence pathological. Although a "write-only" document is absurd, our specifications actually contain other objects whose states cannot be used as an information channel. A further example of the restrictiveness of the Bell and La Padula rules is given below when demonstrating the proof techniques.

For each user in the system, there is a maximum security level at which he can operate, i.e., as the result of the login routine assigning a process at that level to execute on his behalf. The user can select at login time any security level that is lower than this maximum level. Once he chooses the particular operating security level, it is possible to assume that all information he generates is at that level. Thus a user cleared to TOP SECRET must login at a CONFIDENTIAL level if he wishes to write a CONFIDENTIAL document. (It should be noted that the model and the system design are not intended to prevent a user from generating TOP SECRET information when logged in at CONFIDENTIAL, or from showing a TOP SECRET document to an uncleared colleague.)

MULTILEVEL SECURITY PROPERTIES -- INFORMAL DESCRIPTION

In order to prove that a specification is multilevel secure, it is necessary to have a definition of security. The definition must be in terms of the specification language. The formulation of this definition is part of stage 1 of the methodology. The definition of multilevel security in terms of SPECIAL is begun in this section, and is treated more formally in a subsequent section. After giving this definition, some principles are stated whose satisfaction by the specifications of each function guarantees that the overall specifications are multilevel secure.

In the specification of a multilevel secure system, each visible function must have as an IMPLICIT argument an identification of the process invoking the function. An implicit argument is one whose value is supplied by the system rather than by the process calling the function. Therefore, the process identifier being an implicit argument guarantees that the process

identifier cannot be forged. Each process operates at a single level. A function together with a set of argument values for the parameters of the function is called a FUNCTION INSTANTIATION, whose security level is that of the process. The security level of a process is fixed for the life of the process.

The RESULT of an invocation of a V-function or OV-function instantiation contains two pieces of information. The first is a boolean value indicating whether or not the invocation caused an exception to arise. If an exception did arise, the second piece of information consists of an indication of which exception arose. If an exception did not arise, the second piece of information is the return value of the function. The results of invocations are the only information considered visible to a process. (Thus the use of the time required to complete the invocation is not considered here as a channel of information.)

The MULTILEVEL SECURITY property can now be defined as follows: A system is multilevel secure if the invocation of any visible O-function or OV-function at security level L1 has no effect on the results of any future invocation of any visible V or OV-function, except those at security levels L2 such that L2 is greater than or equal to L1.

This is a precise statement of the meaning of multilevel security. Unfortunately, it is difficult to prove that the specifications satisfy this property, since the proof (potentially) requires an induction over all function invocations. The proof process is significantly simplified by utilizing principles which are proven to be satisfied individually by each visible function.

Each primitive V-function specification contains an argument position defining the security level of the V-function values. Thus each instance of a V-function value (i.e., a V-function for a particular argument tuple and output value) has a security level associated with it.

The following definitions are useful:

* A primitive V-function value is MODIFIED by the specified function if and only if it appears as a new value in an effect.

* A primitive V-function value is CITED by the specified function if and only if it appears as an old value in either an effect, an exception, or a derivation.

* A WRITE REFERENCE is a modified V-function value, or the return value of a visible V-function or of an OV-function, or the value of an exception condition.

* A READ REFERENCE is a cited V-function value of the specified function, an argument to the invocation of that function, a parameter of the module, or a bound variable.

* A read reference is TYPE-LEGITIMATE if it corresponds to (1) a possible value for a V-function as given in its type declaration or (2) a possible value for an argument, parameter or bound variable satisfying its declared type constraints. Note that this definition is conservative with respect to a V-function value, since it does not exclude such values that cannot be achieved by any sequence of O-or OV-function invocations.

* A write reference is DEPENDENT on a read reference in a specification if and only if there exist two different type-legitimate values for the read reference that would cause the write reference to assume correspondingly different values.

The STRONG SECURITY PRINCIPLE can now be given, which if satisfied by each visible function specification guarantees that the multilevel security property is satisfied:

> In the specification of a visible function, the security level of each write reference must be
>
> (a) at least the security level of the function invocation,
>
> (b) at least the security level of each read reference upon which the write reference is dependent.

Part (a) of the strong security principle is easy to prove for the specifications of the secure object manager. The write references that are returned values of a function invocation's trivially satisfy (a) since they are, by definition, at the security level of the invocation. It thus remains to show that all V-functions that are modified are at a security level at least that of the invocation. In the given specifications, this is easily seen since all appearances of quoted V-functions in a function specification have an associated security level that is at least that of the function invocation.

Part (b) of the strong security principle is more difficult to establish for the specifications. In a later section are presented some rules that can be applied to each function specification to establish its conformity with principles. These rules provide an informal axiomatizaton of SPECIAL.

Part (a) of the strong security principle prevents the caller of an O- or OV-function from changing a V-function at a lower security level, and thus from transmitting information to a

user at a lower security level. Part (b) prevents (1) the caller of a V- or OV-function from being returned a value that is dependent on a V-function (or a parameter) at a higher security level, (2) the caller of any visible function from receiving an exception value that is dependent on a V-function of a higher security level, and (3) the caller of an O- or OV-function from acting as an "intermediary" in causing a V-function of a security level L1 (with L1 greater than the security level of the caller) from acquiring a value dependent on a V-function at a security level L2 where L2 is greater than L1. Note that Part (b) does not exclude all read references to higher level V-functions, but only those on which lower-level V-functions are dependent.

The strong security principle is stronger than necessary since it excludes the modification of a V-function that is at a lower security level than the caller, even if no visible function invocation will return a value or exception that is sensitive to the modification. However, it is possible, as seen below to specify a useful system that satisfies the strong security principle, and then, as discussed subsequently, to transform the specifications in order to eliminate needless security level arguments.

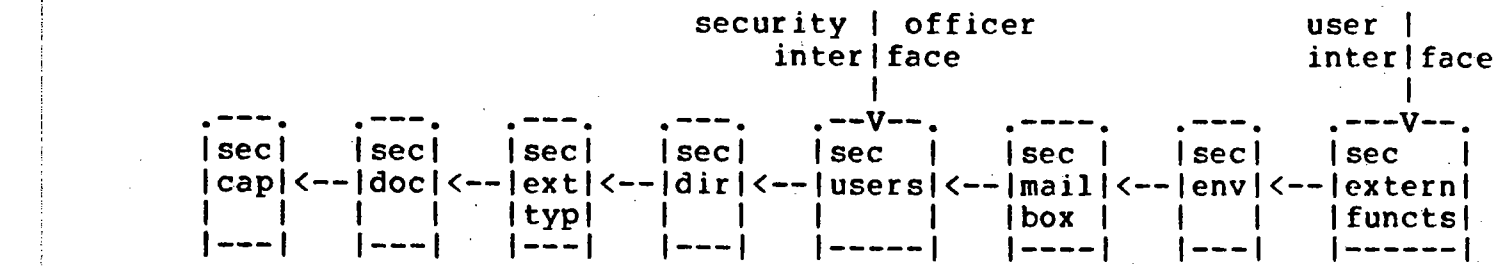
The requirement that each primitive V-function possesses an argument position corresponding to its security level clearly enhances proof. In particular, it is almost obvious from the specifications that, for example, a write onto a document causes a change in the document version at the writer's security level and at levels above that. However, the distinction of V-function values at different security levels is just apparent in the specifications, to enhance proof, but does not appear in the implementation. For example, assume that a V-function V(arg1, arg2,..., security level) that is initially undefined later attains a defined value only at security levels such that V(arg 1, arg2,..., L1) = V(arg1, arg2, ...,Li) for all Li greater than or equal to L1, and possibly later becomes undefined for all security levels. (A V-function satisfying this property is designated as a MINIMUM-LEVEL-INVARIANT V-function). The implementation of such a V-function can be efficient since it is not necessary to assign distinct representation states to all of the security levels at which the function is defined.

In some cases the security-level argument position in a V-function can be eliminated entirely, if it can be shown that the sharing of the V-function among different levels does not lead to any unwarranted inter-level information transfer. Here it is shown that the V-functions that relate to capabilities do not require security-level argument positions.

DESIGN DECISIONS

The secure object manager consists of an eight-module interface, with modules as follows.

secure external functions
secure environments
secure mailboxes
secure users
secure directories
secure extended types
secure documents
secure capabilities



Here module A being to the left of module B indicates that the specifications of B could reference functions of A. The manager could have been specified as a single module, but the decomposition enables design decisions to be localized. The decomposition as shown above is only for specification purposes, and does not imply that, for example, secure directories must be implemented in terms of secure documents; in the implementation, as discussed later, these objects are represented in terms of PSOS objects, not each other. However, as we discuss later, the current modules of PSOS could be replaced by the corresponding secure versions.

Each of these modules is described briefly below, as a guide for comprehending the specifications in Appendix D. However, before presenting the modules it is appropriate to discuss the major decisions motivating the design.

All V-functions of the system have an argument position for security level. If the function is hidden, the argument is explicit, otherwise it is implicit and hence "unforgeable". Each O-function of the system is one of three types.

1. An O-function called only by the security officer, which requires the presentation of the security officer capability cdsm. All such functions are restricted to the module SECURE_USERS.

2. An O-function that is NOT available externally at the

interface. Such O-functions are restricted to the six modules SECURE_CAPABILITIES, SECURE_DOCUMENTS, SECURE_EXTENDED_TYPES, SECURE_DIRECTORIES, SECURE_MAILBOXES, and SECURE_ENVIRONMENTS, and have an implicit argument that identifies the security level of an invocation. It is then possible to prove independently for each of these modules that it satisfies the strong security principle.

3. An O-function that is available externally at the interface. Such O-functions are restricted to the module SECURE_EXTERNAL_FUNCTIONS, and have an implicit argument corresponding to the process capability. The only V-function of this module is the hidden function that stores the uids corresponding to the various process and instances of each user. As noted below, the strong security principle for the SECURE_EXTERNAL_FUNCTIONS module is proved, assuming its satisfaction for the other modules (except for SECURE_USERS, as noted.)

The use of capabilities has no bearing on multi-level security. Their use in the system design is to provide a discretionary protection mechanism and to provide a design that is directly implementable in terms of the objects of PSOS, all of which require capabilities for accessing.

With regard to the "discretionary" issue, the multi-level security model does not exclude the sharing of objects among all users at the appropriate security levels. We have provided an augmentation of the model so that a user gains access to an existing object only if (1) his security level is appropriate, and (2) someone has given him access rights. Condition (2) could be viewed as relating to "need-to-know".

The specifications that are given can be proved multi-level secure independent of the protection properties of capabilities. The specifications can also be proved to satisfy the alteration and detection principles, and thus are in conformance with the need-to-know principle.

However, the protection properties of capabilities are important when proving that any efficient implementation is consistent with the specifications. In order to simplify the proof of specifications, the following principles are useful.

1. Most V-functions, e.g. document_read, have instances of numerous security levels, in particular, those at all levels between that of the process creating the segment and the maximum security levels.

2. For specification purposes, a read reference in a function invocation is almost always to the particular

instance of the V-function at the level of the invocation.

3. For specification purposes a write reference is almost always to all instances of the V-function at all security levels at least that of the invocation. A write reference is never to a V-function at a level L1 that is not at least that of the invocation.

4. Principle (3) seems to imply that each instance of a V-function corresponding to a distinct level could have a distinct value. However, it is almost always the case that the writing will be initiated at the same level of creation of the V-function. Thus all instances of a V-function at different levels will have the same value and hence the value of the V-functions at its "creation" level is adequate to characterize it. This limitation is achieved by not allowing capabilities with "modify" (or equivalently "write") access to be moved upward in the security sense. That is, a capability with "modify" access is never moved from a V-function at L1 to another V-function at a different level. (Principle (3) precludes the "downward" transmittal of a capability.)

5. Documents are to be directly implemented as segments, and their access is not to be impeded by software interpretation. That is, the level of a process is not to be checked prior to the process achieving access to a document. The possession of a capability is necessary and sufficient for accessing the document. Thus capabilities with "write" access are not allowed in documents for the following reason, assuming a representation for documents that assigns a single level to a document -- namely the creation level. Assume P1 at L1 writes a "write" capability into a document created at L1. If P2 at L2 (L2 > L1) reads the "write" capability, it then can write into the document and hence violate the multi-level security principle. In other words, by not allowing "write" capabilities in documents it is possible to ensure that all instances of a document, at different levels, have the same state, and hence there is no violation of the security principles in dropping the multiple-level distinction. Note that there is no security violation arising from the appearance of "read" capabilities in documents. Indeed a procedure, represented as a document, could contain capabilities to call other procedures. The use of a hardware implemented permit flag, distinguishes between "read" and "write" capabilities.

6. Principle (5) prevents the transmission of "write" capabilities via documents. However, the system would

not be useful if users, at the same level, could not share write_access to documents. Other channels, i.e. secure_directories and secure_mailboxes are provided for the purpose of transmission of "write" capabilities. Here some software interpretation, reconciling the levels of the object and the process is required, but the penalty in access time can be borne in the case of such objects. Generally, objects that can store "write" capabilities will have two separate instances in representation: (1) for access by processes of the creation level of the object, and (2) for access by processes at higher levels.

7. The "control" effect of the exceptions has no bearing on the proof of the specifications. That is, although an exception condition evaluating to TRUE causes the remaining exceptions and the effects to be ignored, each exception condition and each effect can be considered independently in the proof.

The modules are now discussed in detail.

SECURE CAPABILITIES: A secure capability, abbreviated as SCAPABILITY, can be considered as containing three components: an integer-valued uid and two boolean-valued flags: access and permit. For each security level, the module is assumed to be initialized with a set of scapabilities. This assumption is for specification purposes only. In the implementation, scapabilities would be implemented in terms of capabilities, each of which is generated by a capability generator. There are four possible scapabilities for each uid, corresponding to TRUE and FALSE values for the two flags. Each of the functions contains argument positions corresponding to the clearance and category set (cat_set); for convenience the functions are described as if the level and cat_set were combined into a single argument position, security_level, or simply level. The OV-function "create_scapability (level)" returns a scapability whose uid component has never been previously returned at ANY security level and whose access flag is TRUE. The V-function "restrict_saccess (sc, level)" returns the scapability scl whose uid component and permit components are the same as that of sc but whose access flag is FALSE. The function "change_permit" can set the permit flag to TRUE or FALSE. Note that no visible function of the module returns the uid component of a scapability. The version of the scapability with FALSE access flags will be used (obtained by calling "get_slavel") instead of the uid as the argument that uniquely identifies an object, e.g. document or sdirectory. Note also that for each level there is an essentially infinite roster of uids that can be earmarked to scapabilities initially created at that level, and at all higher levels. As noted below, there are no security violations if the scapabilities are not

segregated by security level. The global distinction of uids (among distinct security levels) significantly simplifies the implementation even if the segregation of scapabilities is retained.

The properties of capabilities in PSOS to control the access to objects (i.e., as arguments to all functions) is an integral aspect of the security principles of PSOS. On the other hand the use of scapabilities in the SOM specification has no bearing on its multilevel security properties -- which are completely determined by security level arguments of the functions. However, scapabilities are still used to constrain object access, beyond that proscribed by the security level arguments, but in the case of document access the constraints imposed by the scapabilities are entirely adequate to satisfy the multi-level security properties.

SECURE DOCUMENTS: Documents in the secure object manager serve the same role as segments in the operating system. A document is created by calling the OV-function "create_sdocument (size, level)". After the call, a version of the document apparently exists -- at least with respect to the specifications -- at each security level greater than or equal to "level". In fact, only one version will exist in the implementation, namely that at "level". The call is rejected if the quota of documents at that security-level would be exceeded. Separate quotas are needed for each security level to prevent information flow via the exception mechanism. Any modifications to any existing document, (i.e., changing the size, writing a word, or deleting the document for a document initiated at a security level L) are carried out on all (apparent) versions of the document at security levels L1, L1 >= L. Each V-function returns a value that is dependent only on the document version at the security level of the call. Note that the O-function "write_document" returns an exception if the word to be written is a scapability with permit flag of FALSE. As observed below, the specifications are secure independent of the presence of this exception condition. However, its presence ensures that all versions of a document at security levels at least that of the initiating level L1 are all identical. For all document scapabilities, if the access flag is TRUE (indicating an ability to modify the referenced document), then the permit flag is FALSE. Conversely a document scapability with an access flag of FALSE will have a permit flag of TRUE, and hence is writeable into a document. Thus it is not possible for a write scapability for a document to be inserted into a document of L1, which is read at security level L2 (L2 > L1), and then is used to write into the same document at L2. In other words, the disallowing of the storage of write scapabilities for documents in documents is essential for ensuring that all versions of a document have the same contents. (Again, this constraint is not essential for

multilevel security). As shown below, other channels (directories, environments, etc.) can transfer write scapabilities upward, but only at the same level.

SECURE EXTENDED TYPES: The purpose of this module is to allow users to create secure extended type managers as a service, for other users. Such types are created while the user is operating within the SOM, compared with the CSM (Chapter IV-2), where the creation is accomplished outside, with the CSM certifying the type after the fact. For example, a process operating at L1 could create a type manager for a particular type T of extended objects. A process at a lower level could not make use of this service, simply because it is not to be aware of the existence of any operations of a higher-level process. However, any process at L2, L2 >= L1 can create objects of type T and can call on the type manager to process such objects. Any objects so created will have creation level L2.

It is important to note that although T is created at L1, and is possibly manipulating objects at L2 > L1, it is not possible for the programs of T to erroneously transmit any information about such objects. There are two channels potentially available for information transmission: segments and the representation vectors for extended objects. However, a segment used to implement the procedures of a type manager cannot be used as a channel since it will not store any "write" scapabilities. Thus an unscrupulous type manager could not transmit information about a higher level object it is processing to segments, directories, or other objects.

The other channel available to a type manager is the representation vector for any object that it manages. If such an object is at a higher level L2 than the creation level L1 of the type manager then the object is accessed only in processes operating at level at least L2, and hence no information about the object remains with the type manager. A slightly more difficult problem relates to objects at level L1 that the type manager created to be available to higher levels during their execution. Let us denote such an object as a CONSTANT OBJECT, co. The important question is, can the type manager procedures change the representation of co while manipulating a higher level object, and later while operating at level L1 read the representation? The answer is NO, as is now observed from the specifications.

An object, for example co, created and initialized (i.e. given a vector of representation scapabilities) at L1, is for specification purposes created and initialized at all levels Li, Li >= L1. Any modification of the representation vector, at L2 > L1, say by the function, delete_impl_scap, delete_impl_subj, or insert_impl_subj, affects only those

instances of the representation vector at levels L3 >= L2, and thus has no bearing on the representation vector at level L1.

As with secure documents, it appears from an initial observation of the specifications that the different instances of the representation vector for an object corresponding to different levels, could all have distinct values. However, this is not the case since, the representation vector is only modifiable upon presentation of a scapability with "write" access, i.e., access flag = TRUE. As observed above, such scapabilities could not be transmitted "upward" via documents. For the SECURE_EXTENDED_TYPE module, the only V-function that can store a scapability is h_impl_scap (u,level), the vector of representation scapabilities for object u. However, the functions insert_impl_scap, and create_impl_sobj, invoked at L1, insert scapabilities with "write" access only in those instances of h_impl_scap (u,level) such that level = L1. The remaining instances get only "read" access. Thus this module does not serve as a channel for the "upward" transmission of "write" scapabilities.

SECURE DIRECTORIES: A secure directory, abbreviated as SDIRECTORY in the specifications, stores scapabilities associated with a user-applied name. A sdirectory serves as a legitimate channel for the transmission of scapabilities among users. In particular, a user operating at security level L1 can give a scapability sc to a user at security level L2, L2 >= L1 by writing the scapability in a sdirectory which both share. If L2 > L1 and if sc is a document scapability, then the transferred scapability should have no write access. Consistent with the above notion of apparent existence, a sdirectory newly created by a user at security level L1 will potentially have versions at all security levels at least L1. When an entry is placed in the L1 version, it will also apparently be placed in all higher level versions, except those versions that already have an entry of that name. The user operating at L1 cannot discover anything about the disposition of his "insert_sentry" call at higher security levels. In particular, the specifications indicate that an entry might not be placed in a sdirectory instance at a certain higher level such that the sdirectory is full.

The implementation of sdirectories is discussed below. At this point, it suffices to say that the level distinctions can be dropped in favor of two sdirectory instances, the level L1 of each being that at which the sdirectory was created. One instance contains "write-read" scapabilities for access by processes at L1, while the other contains only "read" scapabilities for access by processes at higher levels.

In summary, modify operations on sdirectories potentially occur on all versions of a sdirectory whose security level is

at least that of the caller. Read operations occur only on that version of the level of the caller. Unlike documents, all versions of a sdirectory do not have identical states, but this distinction is handled by visualizing two different versions for each sdirectory.

SECURE USERS: The functions of this module provide an interface by which the security officers can initiate and delete users. These functions require the presentation of the special capability cdsm. A newly created user, authorized to operate at all levels up to Lm, will be given a sdirectory, The scapability for which is stored in the function initial_environment. However, "write" access will appear only in the instance at LO, the lowest access level in the system. The other instances of initial environment will store the scapability with only "read" access.

The function augment_initial_environment is used to augment the initial environment of a user in response to the creation of a process for the first time at a level Ll other than LO. When the user logs in for the first time at Ll, he will be given write access to a new sdirectory. However, in order to provide "read" access for objects that are created at Ll to other processes of the user at L2 > Ll, the other such processes will be given "read" access to the sdirectory of Ll. Such access to the sdirectory is provided by storing the "read" scapability for the sdirectory in the initial environment of user at all levels Li > Ll.

Note that the V-function initial_environment, as with document_read, h_impl_scap, and get_scap, is not a channel for the upward transmission of "write" scapabilities. However, differing from these other three functions, initial_environment can assume as many different values as there are distinct security levels. Nevertheless, the amount of real storage devoted to this function is not excessive, and hence this is no cause of concern.

An exception condition for delete_user prevents the security officer from removing a user until all objects that he has created have been deleted. The existence of such an object is easily detected as the appearance of a corresponding distinguished_sentry in a sdirectory that the security officer created for the user. A more realistic approach would allow the security officer to himself delete all such objects. Such a policy is easily incorporated by providing a function that returns all distinguished entries in a tree of sdirectories.

Also a realistic system would allow for the changing of a user's clearance. This function is trivially specified as affecting only the value of initial_environment.

SECURE MAILBOXES: The mailbox operations allow a process to send a vector of machine words to an identified user. As with write operations for documents and sdirectories, the message is sent at all levels at least that of the sender. The receiving process will read the message at its level. It is necessary to ensure that message buffers are not channels for the upward transmissions of scapabilities with write access. If a scapability is sent as part of a message, then at all security levels exceeding the sending level, the scapability is inserted with restricted access (i.e., no write). Note that the sending function receives an exception only if the quota would be exceeded by the lengths of the remaining messages that were sent by other processes on behalf of the same user. There is no unwarranted information transmission, but a process could saturate the transmitting message buffer of a higher-level version of the process. This relatively unimportant difficulty could be eliminated, but requiring a slight increase in design complexity.

SECURE ENVIRONMENTS: This module provides functions for the maintenance of processes, similar to the processes module of PSOS. An important aspect of this module, that distinguishes it from those presented above, is that its V-functions, environment_stack and environments_stack_size exist only at one level -- the security level of the process. This simplification would be eliminated if we provided facilities for a secure form of process sharing among different levels. The module maintains a stack of environments, or equivalently stack frames, with functions provided for writing, reading, pushing a vector of smachine words, and popping a vector of smachine words, all relative to the current, i.e. topmost environment. A new frame is created by the function new_context, and can be initialized with values. The external function call would be implemented by new_context. Similarly old_context removes the current stack frame. Ultimately the specification would be augmented with control primitives appropriate to a process interpreter.

SECURE EXTERNAL FUNCTIONS: This module provides the interface to the user of the SOM. It has only the simple V-function process_exists, thus achieving its state mostly by externally referenciing the other modules. The specifications are quite simple, involving references to functions of other modules for exceptions and effects.

We now present a formal model of multilevel security in order to demonstrate that proofs of the specifications can be carried out.

FORMAL MODEL OF MULTILEVEL SECURITY

A multilevel system is defined to be the following ordered n-tuple:

$$<S, s_0, L, "<", I, K, R, N_r, N_s>$$

where the elements of the system can be intuitively interpreted as follows:

- S - States: the set of states of the system
- s_0 - Initial state: the initial state of the system; $s_0 \in S$
- L - Security levels: the set of security levels of the system
- "<" - Security relation: a relation on the elements of L that partially orders the elements of L
- I - Visible function instantiations: the set of specifications of all the visible functions and operations; if a function or operation requires arguments then the function specification along with each possible set of arguments is a separate element of I
- K - Function instantiation level: a function from I to L giving the security level associated with each visible function instantiation; K:I->L
- R - Results: the set of possible values of the visible function instantiations
- N_r,N_s - Interpreter: functions from I^XS to R and S that define how a given visible function instantiation invoked when the system is in given state produces a result and a new state; N_r:I^XS->R and N_s:I^XS->S.

In order to define the model of multilevel security, it is useful to define the following functions:

- Z(t) the value of the function Z is the last element of the ordered n-tuple t
- B(t) the value of the function B is the ordered n-tuple t with the last element removed
- C(t,e) the value of the function C is the ordered n-tuple t with the element e added at the end.

The following parts of the model can now be defined:

T the set of all finite ordered n-tuples of visible function instantiations or, in other words, all possible sequences of operations

$T = I^*$

M the state resulting from the given sequence of operations starting at some given state

$M: S \times T \rightarrow S$
$M(s,t) = N_s(Z(t), M(s, B(t)))$

E the sequence of operations that results when all the operations whose level is not less than the given level are removed from the given sequence of operations

$E: T \times L \rightarrow T$
$E(t,l) = (K(Z(t)) < l \vee K(Z(t)) = l) \Rightarrow C(E(B(t),l), Z(t))$
$\vee \sim (K(Z(t)) < l \vee K(Z(t)) = l) \Rightarrow E(B(t),l)$

Multilevel security can now be defined as follows:

$$(\forall i \in I, s \in S, t_1, t_2 \in T)$$

$$E(t_1, K(i)) = E(t_2, K(i)) \qquad \text{(P1)}$$

$$\Rightarrow N_r(i, M(s,t_1)) = N_r(i, M(s,t_2))$$

This says that if two sequences of operations are each applied to a system in the same state and if these sequences differ only in operations whose level is not less than or equal to some level, then any operation of that level that is invoked immediately following the two sequences will return the same result. In other words, the operations whose level is not less than or equal to this level can not effect results visible to the level.

STRONG MULTILEVEL SECURITY PROPERTIES

Unfortunately, it is difficult to prove that any specification meets this definition because any direct proof would require some induction on all possible sequences of function instantiations. The number of such sequences is generally very large. For this reason the following slightly more restrictive set of properties is more useful for proof because it does not involve sequences of function instantiations.

It is first necessary to introduce the notion of a partial state. There is a partial state set S^l for each security level l of the system. The cross product of all the partial state sets $(\underset{\forall l \in L}{\times} S^l)$ is isomorphic to the set of states (S). Therefore, each state $s \in S$ can be represented by the ordered n-tuple consisting of one element s^l from each of the partial state sets S^l. Intuitively, one can think of a partial state set as all the state variables assigned a given security level and a partial state as one set of values for these state variables.

The following useful functions can now be defined:

$Q_l : S \rightarrow S^l$	has as its value the partial state of each $s \in S$ for the level l
$Q^l : S \rightarrow \underset{\forall k \in L \mid k <= l}{\times} S^k$	has as its value the partial state of each $s \in S$ for all levels less than or equal to l
$D_l : S \rightarrow \underset{\forall k \in L \mid \sim(l <= k)}{\times} S^k$	has as its value the partial state of each $s \in S$ for all levels not greater than or equal to l

It is now possible to define three new security properties whose conjunction is stronger than P1 above:

$$(\forall i \in I)\ (\exists j)(\forall s \in S)\ N_r(i,s) = j(Q^{K(i)}(s)) \qquad \text{(P2a)}$$

$$(\forall i \in I, l \in L)\ (\exists j)(\forall s \in S)\ Q_l(N_s(i,s)) = j(Q^l(s)) \qquad \text{(P2b)}$$

$$(\forall i \in I, s \in S)\ D_{K(i)}(s) = D_{K(i)}(N_s(i,s)) \qquad \text{(P2c)}$$

The first property (P2a) states that the result of a function instantiation at some level can be dependent only upon state variables of a lower or equal level. The second property (P2b) states that the value assumed by a state variable at some level due to the action of some function invocation can be dependent

only upon state variables at a lower or equal level. The third property (P2c) states that a function invocation at some level can only change the values of state variables at a greater or equal level.

PROOF OF STRONG PROPERTIES

The following is an outline of the proof that the strong multilevel security properties (P2a, P2b, P2c) imply the general multilevel security property (P1); in other words that

****
*
* P2a & P2b & P2c => P1 (T1) *
*
****

Using P2a in the last part of P1 yields:

$$(\forall i \in I, t_1, t_2 \in T)\ (\exists j)(\forall s \in S)$$

$$j(Q^{K(i)}(M(s,t_1))) = j(Q^{K(i)}(M(s,t_2)))$$

$$\Rightarrow N_r(i,M(s,t_1)) = N_r(i,M(s,t_2))$$

and by eliminating the function j, the formula to be proven becomes:

P2a & P2b & P2c

=>

$$(\forall i \in I, s \in S, t_1, t_2 \in T) \qquad (F1)$$

$$E(t_1,K(i)) = E(t_2,K(i))$$

$$\Rightarrow Q^{K(i)}(M(s,t_1)) = Q^{K(i)}(M(s,t_2))$$

Now consider the cases in P1 when $E(t_1,K(i)) = E(t_2,K(i))$ is false. In these cases the theorem T1 is trivially true. Next

consider the cases where $E(t_1,K(i)) = E(t_2,K(i))$ is true. These cases require an inductive proof. The induction will be over the length of the reduced sequence E(t,K(i)). Since only the cases where the reduced form of the two strings t_1 and t_2 are equal are being considered, it is known that the lengths of the two reduced strings $E(t_1,K(i))$ and $E(t_2,K(i))$ will be equal.

The basis of the induction is a reduced length of 0. In this case the sequences t_1 and t_2 can contain only function instantiations whose level is not less than or equal to K(i). From property P2c one can observe that a function instantiation whose level is not less than or equal to K(I) cannot change the partial state of the system at levels less than or equal to K(i). Therefore, the partial state at levels less than or equal to K(i) must remain the same for sequences whose reduced length is 0. For these sequences:

$$Q^{K(i)}(M(s,t)) = Q^{K(i)}(s)$$

and, therefore, F1 is true.

For the purpose of accomplishing the inductive step in the proof, define the function G_1:T->T to map a sequence of function instantiations onto the beginning of that same sequence up to but not including the last function instantiation whose level is less than or equal to 1. Also define the function H_1:T->I to map a sequence of function instantiations onto the last function instantiation in the sequence whose level is less than or equal

to 1. If a sequence t has reduced length n with respect to some level 1 then the sequence $G_1(t)$ has reduced length n-1 with respect to 1. The induction hypothesis states that $Q^{K(i)}(M(s,G_{K(i)}(t_1)))=Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ for any two sequences, t_1 and t_2, whose reduced sequences are equal. Now it is necessary to show that the last parts of sequences t_1 and t_2 make identical changes to the partial states for levels less than or equal to K(i). If $H_{K(i)}(t_1)$ is not equal to $H_{K(i)}(t_2)$ then $E(t_1,K(i))=E(t_2,K(i))$ is false and F1 is trivially true. Recall that property P2b states that any partial state at some level that results from the invocation of a function instantiation must be a function of partial states with lower or equal level. Therefore, the partial states with level less than or equal to K(i) resulting from the invocation of $H_{K(i)}(t_1)$ and $H_{K(i)}(t_2)$ must be functions of $Q^{K(i)}(M(s,G_{K(i)}(t_1)))$ and $Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ respectively. If $H_{K(i)}(t_1)$ is equal to $H_{K(i)}(t_2)$ and since $Q^{K(i)}(M(s,G_{K(i)}(t_1))) = Q^{K(i)}(M(s,G_{K(i)}(t_2)))$ from the induction hypothesis then the partial states resulting from the invocations of $H_{K(i)}(t_1)$ and $H_{K(i)}(t_2)$ must be identical. All that can be left in the sequences t_1 and t_2 after the last function instantiation whose level is less than or equal to K(i) are obviously function instantiations whose level is not less than or equal to K(i). From P2c it is known that such function instantiations cannot change partial states with levels less than or equal to K(i). This completes the outline of the proof.

INTERPRETATION OF THE MODEL

In order to apply the security properties defined above to a particular system design, it is necessary to relate the elements of the model of a multilevel secure system to the specification language and to the particular system. Recall that the model is the following 9-tuple:

$$\langle S, s_0, L, "<", I, K, R, N_r, N_s \rangle$$

The elements of the model can be interpreted as follows for the SOM specifications:

- S - States: all possible collective values of all the primitive V-functions of the specification; each state can be represented by a particular set of values that the primitive V-functions can assume
- s_0 - Initial state: the initial values of all the primitive V-functions as given in the specifications
- L - Security levels: each security level is defined by two values, the clearance and the category set; the clearances are totally ordered
- "<" - Security relation: the security relation is a partial ordering on the security levels; a security level is less than (<) another security level if the clearance of the security level is less than the clearance of the other security level and the category set of the security level is a subset of the category set of the other security level
- I - Visible function instantiations: each visible function of the specifications together with a set of possible argument values to that function is a visible function instantiation
- K - Function instantiation level: this is the level of the visible function instantiation and must be defined for each visible function instantiation
- R - Results: a result is the return value of a visible V- and OV-function invocation and the number of the first exception, if any, in a visible function invocation whose value is true; i.e. a result are the visible effects of the visible functions
- N_r, N_s - Interpreter: the semantics of the specification language

The partial states S^1 are represented by subdividing the

primitive V-function instantiations (i.e. primitive V-functions together with a particular set of argument values) into disjoint sets, one set for each security level. The partial state is determined by the value of the primitive V-function instantiations that are members of the partial state set.

STRONG SECURITY PROPERTIES IN TERMS OF SPECIFICATION LANGUAGE

The purpose of this section is to state the strong security properties P2a, P2b, and P2c in terms of constructs of the specification language. In order to formally relate the strong security properties as given above in terms of the formal model to the specification language it is necessary to have a formal description of the semantics of the specification language. Since such a formal description of the language has not been completed, this section will discuss the strong security properties in an informal manner. An English language description of SPECIAL is given in Appendix A. The following definitions, given earlier, are repeated here for the reader's convenience:

* A primitive V-function instantiation is said to be modified by a particular visible function instantiation iff the primitive V-function instantiation appears as a new (quoted) value in the effects section of the specification of the visible function and the value of the primitive V-function instantiation could be changed by invoking the visible function instantiation.

* A primitive V-function instantiation is said to be cited by a particular visible function instantiation iff the primitive V-function instantiation appears as an old (unquoted) value in the specification of the visible function.

* A write reference in a visible function instantiation is a primitive V-function instantiation, the return value of a V- or OV-function, or the exceptions.

* A read reference in a function instantiation is a cited primitive V-function instantiation.

* The value of a read reference is legitimate iff it can be assumed by the cited primitive V-function instantiation after some sequence of O- or OV- functions applied to the system in its initial state.

* The value of a read reference is type legitimate iff it is of the type of the cited primitive V-function.

* A write reference is dependent upon a read reference with respect to a particular function instantiation iff there

exists two different legitimate values for the read reference that would cause the write reference to assume correspondingly different values as the result of the invocation of the function instantiation.

A slightly stronger form of the definition of dependency can be obtained by substituting "type legitimate" for "legitimate". It is easier to determine the type legitimate values of a read reference than it is to determine the legitimate values since type legitimacy is a property of the language whereas legitimacy is a property of a particular set of specifications. It is, therefore, easier to identify dependencies if the type legitimate version of the definition is used; however, for the purposes of this discussion either version of the definition of dependent suffices.

Given the above definitions it is possible to easily state the strong security properties in terms of the specification language. Note first that the above definition of dependence simply defines a functional relationship, i.e., if a write reference is dependent upon a read reference then the value of the write reference is simply a function of the value of the read reference. Recall that property P2a states that the result of the invocation of a function instantiation of some level is a function of (i.e., is dependent upon) the values of the state variables (i.e., the primitive V-function instantiations) of lower or equal levels. The results are the return values of V- and OV-functions and the exception conditions of all visible functions. Therefore, property P2a can be restated as:

P2a The return value of a V- or OV-function and the exceptions of a visible function instantiation can be dependent, with respect to that visible function instantiation, only upon read references of lower or equal level.

Property P2b states that the value assumed by a state variable (i.e., modified primitive V-function instantiation) at some level can be dependent, with respect to a visible function instantiation, only upon state variables (i.e., cited primitive V-function instantiations) at a lower or equal level. Restated this is:

P2b The value assumed by a modified primitive V-function instantiation at some level can be dependent, with respect to a visible function instantiation, only upon read references at a lower or equal level.

The similarity in the restatements of properties P2a and P2b and the fact that the return value, exceptions, and modified primitive V-function instantiations of a visible function are simply the write references of the function allows the following combination of the statements of the two properties into:

P2a,b For each visible function instantiation, the security level of each write reference must be at least the security level of each read reference upon which the write reference is dependent.

Property P2c states that the invocation of a function instantiation at some level can change only the values of state variables (i.e., modified primitive V-functions) at greater or equal levels. If the return value and the exceptions are defined to be at the level of the function instantiation of which they are a part then this property can be restated as:

P2c For each visible function instantiation, the security level of each write reference must be at least the security level of the function instantiation.

Combining this statement and P2a,b above gives a general restatement of the strong security properties in terms of SPECIAL:

```
**********************************************************************
*                                                                    *
* P3 For each visible function instantiation, the security           *
*    level of each write reference must be at least the              *
*    security level of:                                              *
*                                                                    *
*    (a) the function instantiation, and                             *
*                                                                    *
*    (b) each read reference upon which the write reference is       *
*        dependent.                                                  *
*                                                                    *
**********************************************************************
```

Given a formal description of the semantics of SPECIAL, property P3 can be formally stated and the logical statement P3 => P2 can be rigorously proven true.

DETERMINING DEPENDENCIES

This section discusses means for identifying dependencies. The objective is to find some simple algorithm for identifying dependencies. Unfortunately, determining if some write reference is dependent upon some read reference is, in general, undecidable. The approach taken here is to identify potential dependencies. If the set of all write references of a specification is W and the set of all read references is R, then the dependency relation DR is a subset of W R and the potential dependency relation PDR is a subset of W R and a superset of DR. If property P3 can be proven for potential dependencies rather than for dependencies, then clearly P3 must be true for

dependencies. Property P3 for potential dependencies rather than dependencies will be termed P4. The problem then becomes to identify the set of potential dependencies and show that all dependencies are included in this set. However, the cardinality of the set of potential dependencies must be kept as small as possible to make the proof of P4 tractable.

There are two approaches to handling of DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF constructs. First, it is possible to produce an expanded specification, i.e. one in which the substitutions resulting from DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF expressions have been performed. These substitutions are straightforward. In and expanded specification all read and write references relevant to a visible function instantiation will be explicitly present in the body of that visible function's specification. Specifications may still be written in unexpanded form, it is simply easier to describe the proof technique in terms of expanded specifications. Second, it is possible to prove multi-level security properties separately for each of the EXCEPTIONS_OF and EFFECTS_OF expressions, and then assume the correctness of these properties in proving the specifications that invoke these expressions. This is an attractive approach in that it decomposes the proof effort.

There are certain types of expressions that are legal in SPECIAL, but make it very difficult to determine if dependencies or potential dependencies exist. To eliminate the necessity of dealing with such expressions a canonical form for specifications is introduced. The canonical form is a restriction of SPECIAL. In the canonical form, the grammar of SPECIAL is modified and augmented as follows. An <expression> in the body of a function specification cannot contain the symbol which is the identifier for the return value of the function. The definition of <call> is modified to be:

```
<call> ::= <symbol> '(' [<expression> { ',' <expression>}*] ')'
```

The purpose of these two changes is to eliminate the possibility of a write reference in an <expression>. A <write reference> is either a quoted V-function or the identifier of the return value for visible function in which the <write reference> occurs. The following definitions are added (note that in the TYPECASE alternative of <canonical expression> below that <symbol> must not be the identifier of the return value):

```
<canonical expression>
  ::= <write reference> '=' <expression>
      | <canonical expression> AND <canonical expression>
      | <expression> '=>' <canonical expression>
      | {FORALL | EXISTS} <qualif\declarationlist>
        ":" <canonical expression>
      | IF <expression> THEN <canonical expression>
        ELSE <canonical expression>
```

```
        | LET <qualification> { ';' <qualification> }*
          IN <canonical expression>
        | TYPECASE <symbol> OF
          { <canonical case> ';' }+ END

  <canonical case>
    ::= <typespecification> ':' <canonical expression>
```

and finally the definition of <effects> is changed to:

```
  <effects> ::= EFFECTS { <canonical expression> ';' }+
```

The purpose of the canonical form is to restrict how write references can occur in specifications. This canonical form was arrived at through experience with writing specifications and attempts to prove the multilevel security of specifications. Our experience shows no specifications that do not fit into this canonical form.

In order to get some idea of how dependencies are indicated by function specifications, it is necessary to have some rough idea of the semantics of a function specification. For all visible functions the semantics of exceptions can be stated as:

$$(\forall i \mid 0<i<=n)\ ((\underset{0<j<i}{\text{AND}}\ \sim EX_j)\ \text{AND}\ EX_i) = (EV=i) \qquad \text{(S1a)}$$

$$(\underset{0<j<=n}{\text{AND}}\ \sim EX_j) = EV=0 \qquad \text{(S1b)}$$

where EX_i is the ith exception, n is the number of exceptions, and EV is the exception value. EV is the number of the first exception whose value is true. If all the exceptions are false, then EV is 0. In an O- or OV-function the semantics of effects are:

$$(\underset{0<j<=n}{\text{AND}}\ \sim EX_j) = (\underset{0<j<=m}{\text{AND}}\ EF_j) \qquad \text{(S2)}$$

where EX_i, n, and EV are as above and EF_i is the ith effect and m is the number of effects. Note that in an OV-function the return value is specified by the identifier given in the function header. In a V-function the semantics of the derivation is:

$$\left(\underset{0<j<=n}{\text{AND}} \ \sim EX_j \right) = (RV=DE) \qquad (S3)$$

where EX_i, n, and EV are as above and RV is the value returned by the function and DE is the derivation.

Consider now where potential dependencies can exist. As a first approximation assume that a potential dependency exists between all write references of a visible function instantiation and all of its read references. This is clearly a superset of all the dependencies that exist with respect to the function since the semantics of SPECIAL does not allow the value of primitive V-functions not appearing in the specification of a function to be changed by the function and does not allow any new values to be dependent on nonappearing primitive V-functions. Unfortunately, this rather simple identification of potential dependencies includes too many potential dependencies and it is not possible to construct useful systems that are consistent with property P4.

Consider the three types of write references separately. First consider the value of the exceptions, EV. EV is clearly potentially dependent only upon read references in the exceptions section of the function specification. In fact, in some circumstances it may be possible to prove that for some instantiations of a visible function, that a particular exception is always true. In this case EV is potentially dependent only upon read references in exceptions coming before the one that is always true for these instantiations of the visible function.

Now consider those write references that are modified primitive V-functions. Modified primitive V-functions can only occur in the effects section of an O- or OV-function. A write reference in an effect can only be potentially dependent upon read references in that same effect and read references in the exceptions. This follows from S2 above and the canonical form. If a write reference appears in a series of conjoined expressions then the write reference is <u>not</u> potentially dependent on read references in any of the other conjoined expressions. This follows from the definition of conjunction and the canonical form.

Finally consider write references that are return values. If the visible function is an OV-function then the rules for modified primitive V-functions apply. If the visible function is a V-function then the return value is potentially dependent upon the read references in the exceptions and in the derivation.

In summary, the rules for potential dependency are as follows:

PDR1 The exceptions value is potentially dependent upon read references in all exceptions up to the first exception that is always true for the visible function instantiation

PDR2 Each modified primitive V-function in an O- or OV-function and each return value in an OV-function is potentially dependent upon read references in exceptions and all read references in the same effect as the write reference with the exception of read references in expressions conjoined with the expression containing the write reference

PDR3 The return value of a visible V-function is potentially dependent upon read references in the derivation and read references in exceptions.

The following provide interesting and important exceptions to the above rules:

```
FALSE => exp_a
IF FALSE THEN exp_a ELSE exp_b
IF TRUE THEN exp_b ELSE exp_a
FORALL x INSET {}: exp_a
FORALL x | FALSE: exp_a
EXISTS x INSET {}: exp_a
EXISTS x | FALSE: exp_a
LET x INSET {} IN exp_a
LET x | FALSE IN exp_a
```

No write reference can be dependent upon any read reference in exp_a of these expressions. This is evident from the semantics of these expressions. Although it is unlikely to see expressions precisely like these in well written specifications, it is possible that such expressions effectively exist for some instantiations of visible functions. Some examples of these will be given below.

THE PROOF TECHNIQUE

Before summarizing the steps in the proof technique, one further observation is useful. Not all quoted primitive V-functions necessarily represent modified primitive V-functions and, therefore, do not necessarily represent write references. For example in an expression of the form

```
FALSE => 'pvf(args) = exp
```

the quoted primitive V-function pvf does not represent a write reference because the expression does not constrain the value of pvf(args) to change. This situation arises in all the expressions listed in the previous paragraph as exceptions to the potential dependency rules. Similarly, a quoted primitive V-function in the effects section of a visible function instantiation in which some exception is always true is never a write reference.

The proof of multilevel security of a given specification is quite straightforward. For each visible function specification it must be shown that each instantiation of that function is consistent with property P4. This can be accomplished by proving that P4 holds for all possible argument values to the function or it can be accomplished by dividing the possible sets of arguments into collectively exhaustive subsets and then proving P4 for each of the subsets. For each subset the write references must be identified and then it must be shown that for each write reference than is a modified V-function, that the level of that V-function if greater than or equal to the level of the visible function instantiation. Finally, it must be shown that for each write reference, each read reference upon which the write reference is potentially dependent has a level less than or equal to the level of the write reference.

Unfortunately, it is not always possible to determine the level of a read or write reference in a particular function instantiation by inspection of the specification. For example, an argument to some primitive V-function might be the value of some other primitive V-function. In this case it is necessary to know what values the other primitive V-function might have in order to know what the level of the read reference is. Such information may be deducible from the specification of the visible function in question (local assertion) or it may require proving some invariant of the specification of the system as a whole (global assertion). In either case it is necessary to prove P4 for all possible values that the other primitive V-function above may assume.

ILLUSTRATIVE PROOF

As an example consider the external O-function "write (idoc;i;wl)[se]". The security level of the invocation is given by proc_stat(se).s_level (which we abbreviate below as LP), the level of the environment. First consider the read reference dependencies as embodied in the three exceptions.

EX1 expanded becomes

```
~process_exists(proc_stat(se).s_sue,proc_stat(se).s_username, LP)
```

which involves a read reference to a V-function at LP. EX2 expanded becomes

```
~environment_exists(arg1,LP) OR
~get_saccess(arg1,LP),
```

which only involves read references to V-function at LP.

The expansion of EX3 leads to a complicated expression, but the reader can deduce that it only contains read references to V-functions at LP.

The expansion of the EFFECTS section becomes:

```
FORALL levell | gte(levell,LP)
  'document_read
    (get_slavel(current_environment(arg1,idoc,levell),
                levell), i,levell) = wl
```

First of all observe that all write references are to V-functions at levels with levell >= LP. It remains to consider separately each expression within the FORALL statement such that levell >= LP. For each such expression, the write reference is at levell, and the read references are either w at level LP, or the arguments of get_slavel and current environment which are at levell. Thus all write references are at a level at least as high as the read references on which they depend.

TRANSFORMATION OF SPECIFICATIONS TO ELIMINATE UID PARTITIONING

In the specifications of the SOM, all primitive V-functions have been partitioned according to security levels. In effect there is a separate machine for each security level, with only limited communication between machines. If there is to be no information leakage via the exhausting of resources, then it is clear that the resources must be partitioned. It would also, at first glance, seem that the uids would also have to be partitioned. (Indeed the strong security principle requires it). In the absence of partitioning, it appears that a process could

acquire information about a process of a higher security level by creating an object and observing the uid associated with the returned scapability. However, this channel does not exist, since there is no SOM function that returns the uid to a user.

A more convincing proof that uids need not be partitioned would proceed as follows. Consider two abstract machines: M1 described by the specifications as given, and M2 for which the level argument position is removed from all functions of the scapability module; except for these functions, M2 has the same specifications as M1. Note that for both machines there is assumed to be an unbounded supply of scapabilities; M1 has an unbounded supply in each partition. Thus an attempt to create a scapability in either machine never precipitates an exception. Further, M1 and M2 are isomorphic as perceived by a user. There exists a one-to-one onto mapping that will convert uids of one machine to the other. Any function that a user calls in M1 will have the same effect as calling the corresponding function in M2, modulo the uid mapping. Hence a set of users executing on M2 with some sequence of calls would get no more information than a set of users executing on M1 with the same sequence of calls.

Thus, by hiding the value of uids from the user, it is possible to eliminate the uid function as a channel. Unfortunately the complete hiding of uids results in implementation inefficiencies. The implementation of the secure object manager could use the uids of documents and directories for efficient searching. If the only legitimate operation on uids is an equality comparison, then a linear search is required. One solution is to allow the SOM implementation to perform other operations on uids, but to exclude these operations from the processes that use the SOM. Another solution is to give the SOM and the processes it serves access only to a transformation of the uids. In effect the processes would perceive the uids as random integers from which observation of one transformed uid would yield very little information about other existing uids.

REMOVAL OF SECURITY-LEVEL ARGUMENTS FROM DOCUMENTS

The specifications of the secure document module seem to indicate that distinct versions of a document exist, corresponding to distinct security levels. This distinctness is exists only in module's specifications. Indeed, if the specifications are to conform to the common military view of secure documents, all versions of the document should possess the same state. The concept of a minimum-level-invariant V-function is defined above. In this section it is shown that as a consequence of the constraints on the movement of document scapabilities, the three primitive V-functions of the document module (document-exists, document-size and document-read) are all minimum-level-invariant, and for a given document slave

capability sudoc, their values all have the same minimum level of definition. Once this property has been established, the security-level argument positions can be removed from the document specifications, in which case the slave capability argument is sufficient to uniquely identify the document, and presentation of the capability is sufficient to access the document. Thus there is no checking for correspondence of the process' security level with that of the document.

The primitive V-functions of an existing document are changed only by calling one of the O-functions (document-write, change-document-size, and delete-document) with a document scapability scdoc that has write-access. It is now shown that such a scapability is available only to processes that are operating at the same security level as the level at which the document was created. This involves showing that (1) at document creation the document-write scapabilities are acquired only by V-function values at the creation security level; (2) for those V-functions that have security level argument positions, values at a higher security level never acquire a document-write scapability -- i.e. document scapabilities with write access never move upward; (3) V-functions that do not have security level argument positions are never modified to yield a document-write scapability.

For case (1), only the sdirectory V-function h_get_scap acquires the newly issued scapability, and although potentially all sdirectory versions can acquire the scapability, only the version at the creation level acquires the non-restricted scapability.

For case (2), it first must be shown that a scapability with FALSE access flag cannot be changed to one with true access flag. There does not exist a function that can do this. Second, document scapabilities can be moved, between or to (a) sdirectories, (b) environments, (c) message buffers, (D) representation vectors of extended objects, or (e) initial environments. For (a) the relevant O-function is copy_sentry or move_distinguished_sentry, which for document scapabilities always does a restrict_document_access prior to moving the scapability upward. For (c) the relevant O-function is send_mail, which likewise transfers upward only the restricted version of the scapabilities. For (b) the transfer of any word to an environment is at the same level as the source V-function, i.e., there is no transfer upward. For (d), insertion and deletion operations behave the same as the corresponding sdirectory operations. For (e), only the security officer can manipulate the scapabilities.

For case (3), those V-functions which can return scapabilities must be considered whenever the security-level argument positions have been elided. These are "document_read",

"restrict_saccess", and "change_permit". The latter two can be quickly disposed of since they return a scapability with the same slave scapability as presented, and never increase access rights. The V-function "document_read" is modified by the O-function "document_write(scdoc; i; w)". If w is a scapability, the write is allowed only if the permit flag of the scapability is true. However, a newly created document scapability (that of course has write access) has a permit flag of false, and is thus not storable in documents. No visible function of the SOM causes the permit flag of a document scapability to become true, except the functions that restrict the access rights of the scapability.

Thus it is shown that all versions of a document are identical, a property that will bear on the implementation of documents.

IMPLEMENTATION OF THE SOM USING PSOS

A major objective of this work is to demonstrate that the secure object manager can be efficiently implemented using the visible functions of PSOS. Although, at the time of writing, implementation is not complete, it can be discussed in enough detail to demonstrate the feasibility. It is convenient to discuss separately each of the SOM modules.

SCAPABILITIES: The scapabilities, assuming the eliding of security-level argument positions, are implemented in terms of capabilities, with a natural correspondence between slave scapabilities and slave capabilities. The access-flag and permit-flag would be represented as bit positions in the access vector. The scapability module visible functions "create_scapability" and "restrict_access" are implemented immediately by their corresponding capability functions. Only the scapability function "change_permit(sc,TRUE,level) -> scl" poses some difficulties because it sometimes requires the return of a capability scl with a 1 in the access vector of sc replacing a 0. By intention, no capability function can accomplish this. (Note that the function "change_permit" would not be required if no scapabilities were storable in documents. If document read scapabilities are storable, but not document write scapabilities, then by restricting the access rights of a scapability its storing rights are increased--a situation not handled by the capability module.) One simple solution is to use a segment to implement the instances of change-permit, where the permit flag is changed from false to true. The segment holds the scapability versions with true permit flag. Since change-permit will only be called infrequently, the penalty involved in the look-up can be safely endured.

DOCUMENTS: Most of the instructions on secure objects will involve combinations of reads and writes on documents. Hence

it is essential that these instructions be efficiently implemented. It is shown here that such operations can be interpreted directly as reads and writes on segments, and hence no penalty is introduced by an interpretation through the SOM. It is shown above that the functions of the document module do not require security-level argument positions. (The SOM will record the security-level of each document, but this does not influence the read and write operations on documents.) The creation of a document is then interpreted as the creation of a segment. The segment will have an associated store permission, and two versions of the capability -- one which is store limited, i.e. not store-permitted, and the other which is not -- are generated. The non-store limited is retained in the segment that implements the scapability module; the store limited versions, which also contains write access is returned to the document creator. Reads and writes on documents are then performed directly as read and writes on the corresponding segment. An attempt to store into a segment a scapability (which is really a capability) that is store limited will be rejected.

SECURE DIRECTORIES AND SECURE EXTENDED TYPES: For these modules, the primary V-functions are get_scap and h_impl_scap. It is shown above that the instances of these V-functions can be collapsed into two instances: one that is available to an invocation at the creation level L1, and one that is available to all invocations at level L2 >= L1. Thus the implementation of each of these modules is easily carried out in terms of their corresponding PSOS modules. In the case of secure directories, each directory is represented as a directory with two sets of entries, corresponding to the sdirectory entires of L1 and L2 >= L1, respectively. Each in the first such set could be given a name L1 <name>, for each entry <name>. Similarly, each in the second set could be given a name gtL1<name>. With this particular approach, a level check is required at insertion but not at the time of entry retrieval. Another approach that is probably more efficient is to provide only one instance of these relevant V-functions, but to return either the stored scapability or the restricted version, depending on the level of the invocation.

DISCUSSION

An initial effort towards the design, proof of design, and implementation of a subsystem that manages objects of multilevel security has been completed. The global conclusions are as follows.

* Multilevel security can be easily modeled using the methodology.

* The specifications of the secure object manager are easily proved to be in conformance with the multilevel models.

* The secure object manager can be easily and efficiently implemented as a PSOS subsystem, although some improvements are foreseen in the implementation of secure directories.

The following are related issues that should be pursued.

Hybrid Documents: Many military documents are not uniform in security level, i.e. sections, even paragraphs, of the document can be individually classified. Such hybrid documents could be handled here if each such individually classified section is a distinct segment. A better solution, not requiring such a potential proliferation of segments, would be to associate subsegments with each individually classified section. This solution, in order to be efficiently implemented, requires the efficient handling of subsegments in PSOS. This latter solution might still be unacceptable since it requires the document creator always to log in at the security level at which he expects to generate information. Thus the nuisance of changing log in levels might bother him.

Secure Objects as Primitive System Objects: For an application wherein only multilevel secure objects are to be available to users, it is not necessary to have two sets of modules -- one for general objects and one for secure objects. That is, the PSOS segment module could be replaced by the document module, the directory module by the sdirectory module, etc. For such a system all objects are implemented ultimately in terms of documents, rather than segments. There is some added system overhead associated with the assignment of security levels to objects that need not be secure, but this should only impact a small fraction of the instructions that are executed.

Role of the Security Officer: There are certain SOM operations that must be restricted to the security officer: initializing a user, changing a user's clearance, and deleting a user. In addition, there are operations that, in practice, have been the responsibility of the security officer, e.g. changing the security level of a document, changing the name of a document, and changing the clearance of the name. These latter operations are not incorporated explicitly. Instead, in order to change the clearance of a document, the security officer would first create a new document at the intended security level, copy the contents of the old document to the new one, and delete the old document. He could then give selected users access to the new document. The SOM does not assign names to documents as in the military world. The names the users associate with the directory entries for documents

could serve this role, but their level is constrained to be that of the corresponding document. The security officer could retain a file of document names and their security level. Thus the security level of the names could be at the discretion of the security officer.

The programs that implement the security officer operations could be verified with respect to the specifications. However, it is clear that the security officer's specifications do not conform to the multilevel security model. It is thus possible that the security officer could be used maliciously as a channel. Rules of operation need to be established for a security officer to minimize the bandwidth.

Integrity: The multilevel security model does not prohibit writing up in security level, even if it involves the destruction of a TOP SECRET document by an uncleared user. The specifications here are carefully constructed to exclude upward writing by a process, except to those objects under its control. However, it is not realistic to exclude all writing upward. For example, an uncleared programmer could compose, outside of the SOM, a Fortran system which will operate on TOP SECRET data. If it can be assured that the system will operate in a confined manner (Chapter IV-2), then it is reasonable to allow the system (confined) access to the data -- provided it is accepted that the programmer is "trusted" enough not to willingly (or unwillingly) destroy the data. In order to formally handle the problem the MITRE group has developed the concept of integrity as the formal dual to multilevel security. Processes and objects are given integrity levels, and a process can write only on objects whose integrity level is no higher than its own.

Initial State of the SOM: In the presentation of the proofs of the the specification, the security of the initial state is not discussed. For the specifications that require each V-function to contain security-level argument positions, any initial state is secure. That is, if there exist some initial documents, then provided the security officer has classified them according to their contents, there is no possible security violation in future references. The situation is not as clear for the transformed specifications (with level arguments deleted from documents and scapabilities) which rely on the properties of capabilities to guarantee security. For the transformed system a security violation could occur if for some initial segment two environments at different security levels were given scapabilities with write access. Even though this situation can be considered beyond the scope of present concern -- the security officer is violating security by improperly initializing two users -- it is of interest to detect such violations. Since there are no initial objects in the system, except empty sdirectories for each user, the

initial state is seen to be secure. However, in an actual system those initial sdirectories would contain some scapabilities, e.g. to call system functions and library functions, and consequently a precise definition of secure initial state is required.

AUTOMATING THE PROOFS

The proof of the specification of Fig. 1 is simple but quite lengthy when fully documented. Proofs of complex systems will be extremely lengthy. In general, the proof of multilevel security of a specification is many times longer than the specification. If these proofs are written manually, the probability of their correctness is very small. Unfortunately, even a small error in the design of a system can result in a large breach of security. It is, therefore, necessary that there be a high degree of confidence in the total correctness of the security proofs. Such a high degree of confidence in the correctness of the proofs cannot be effectively gained by manual generation and checking of the proofs. The necessary degree of confidence can only be gained by automatic generation or checking of proofs or some combination of automatic and manual techniques.

The proof technique described above has been designed to permit automatic generation of proof. The identification of read references, write references, and potential dependencies of the desired property P4 can be done very simply with knowledge of the syntax and a little of the semantics of SPECIAL. The proof of the relationships between the security levels of the read and write references requires some theorem proving, but the types of theorems involved are all of the same simple kind and most of them can be handled by a simplifier. Those systems that require global assertions in order to perform the proofs will probably require human assistance in deriving the global assertions, however, the proof of the global assertions can probably be automated. For a given system specification, the same theorems arise many times in proving the security of the different visible functions. Once the security of a few of the visible functions has been proven, the proofs of the remaining functions follow similar patterns. Highly efficient operation can be achieved if the automated prover is directed by a human operator for the proofs of a few of the visible functions and then uses the same techniques to automatically prove the security of the remaining functions. Also, after the automated prover has proved the security of a system specification once and is aware of the necessary global assertions, it should be able to prove the security of modified versions of the system with human assistance. The use of such a semi-automated prover is essential to having a high degree of confidence in the proofs, is within the current state of the art of automated verification, and will be more cost effective than manual proof techniques for large systems even with the high initial cost.

Most of the tools necessary for constructing a semi-automated prover for the security of specifications already exist. There exists several theorem provers and simplifiers which should be adequate for the types of theorems that will be generated. A program exists to parse specifications written in SPECIAL and to convert to a form suitable for processing. The necessary additional programs are a verification condition generator that formulates the theorems that express the desired relationships between the security levels of the read and write references and a suitable human interface. Verification condition generators and human interfaces have been written to aid in the proof of properties in several other languages and the ideas in these programs can be used to create programs suitable for proving the multilevel security of specifications.

CHAPTER IV-2
CONFINED SYSTEMS MANAGER

INTRODUCTION

There are application areas in which members of a set of users do not trust each other, but are forced to depend on each other for data, service, etc. Each such user desires assurance that the damage causeable by others to him is limited. The mutual dependency and mistrust between an income tax service, e.g., H. & R. Block, and a customer is typical of the situation. The customer calls on the tax service to prepare a return for him, passing the service information that might be proprietary. Once the return is complete, it is the customer's desire that the service retain no memory of any of the propietary information. The customer thus desires that the service execute in a CONFINED ENVIRONMENT, wherein when it is processing the customer's data it cannot communicate with any other process.

Note that the service could attempt to use the customer himself as a channel. For example, the service and the Internal Revenue Service could be colluding parties, and thus the IRS could transmit information regarding the customer's tax return back to the service. In addition, the charge for preparing the return could be dependent on the customer's data. The constraint that the service not communicate with any process during execution of the program precludes the service retaining such a billing record. However, the service could attach a bill to the return, which the service does not record for itself. This bill, if paid by the customer, gives information to the service. In this discussion, such channels are ignored -- since the customer can in principle shut them off. Instead, elimination of those channels over which the customer has no control is considered.

The situation is more complex if the service, being protective of its programs, does not wish the customer to be able to detect any information about the programs. This more general case has been called the MUTUALLY SUSPICIOUS SUBSYSTEM problem. One would-be solution to this problem is to return no information to the customer; however, this clearly contradicts the hypothesis that the service performs a task for the customer. The full generality does not appear to be needed. Thus this chapter considers only the ONE-WAY confinement problem, although the given solution generalizes to the TWO-WAY case.

The trade-offs between an interpretive and compiled solution to the problem are discussed below. In contrast to the solution for the secure object manager (Chapter IV-1), the compiled solution is selected here. The specifications of the confined systems manager are discussed here, along with a few limitations of the approach. The design methodology is particularly attractive for specifying a confined environment for use with PSOS, since storage channels are easily identified with

V-function values, and since the access to such channels is achieved only by possession of an appropriate capability set for O- or V-functions.

INTERPRETIVE AND COMPILED SOLUTIONS

The secure object manager can be viewed as interpretive, since all operations on secure objects are via calls on the functions at the interface of the subsystem. A compiler-like system could alternatively have been specified, as follows. Users create and modify all objects off-line. An object can be certified as a secure object by being presented to the secure object manager (compiled version), which accepts it if it does not contain any erroneous capabilities. Essentially, the compiled SOM enforces the same rules as the interpreter version, but on the object as a whole rather than on each operation. The interpreter solution is attractive for the SOM, since the dynamic manipulation of secure objects seemed more useful. Also the clear identification of all interface operations greatly facilitates the proof.

On the other hand, the confined system manager (CSM) is better suited to a compiled solution. For the income tax example, the customer would call a program provided by the service, passing the tax data as an argument. The service would be guaranteed to execute in a confined manner if its programs have no access to any storage channels that survive the invocation. Of course, the customer does not trust the service's programs to be so constructed. Thus our solution requires that the service present his programs to a mutually trusted third party, namely the CSM, which will ensure that the service's programs will operate in a confined manner, and then declare them to be confined. In order to prevent the service from maliciously modifying a previously certified program, the CSM will copy the programs, retain the copies, and prevent their modification -- except via the CSM. Thus the CSM "compiles" the programs, and certifies that the compiled versions will operate in a confined manner. The meaning of confinement, with respect to the methodology and the operating system, is discussed in the next section.

THE MEANING OF CONFINEMENT

When a user calls on the confined versions of a service's programs, the execution will be confined if all "writes" are done on objects that are returned only to the user, or on objects that are local to the particular invocation of the procedure. The execution will not be confined if in execution the service's programs perform any of the following operations:

* write on a segment that is not local to the invocation;

* change the size of such a segment;

* delete such a segment;

* perform any modification of a directory, other than the user's process directory or a directory passed by the user;

* modify any extended object that is not local to the invocation.

All such operations could be perceived by a process other than the one running on behalf of the customer.

The above operations are illustrative of the "obvious" storage channels that a malicious service might use for information leakage. A more subtle one is the following: Assume that the service requires "read" access to a previously created extended object in processing the user's data. This object must be a CONSTANT OBJECT, not modifiable in any way by execution of the service's confined programs. All extended objects possess a representation, i.e., the correspondence between a capability for the extended object and capabilities for the objects that implement the extended object, maintained by the extended-type manager. The representation for constant objects cannot be changed by an instruction in a confined procedure, since such a modification could later be perceived by a user who had type-manager's access to the object. The solution here shuts off this channel.

An even more subtle form of leakage from the execution of a confined procedure involves system resources. Here the supposedly confined procedure could attempt to exhaust a resource, e.g., segments, such that a colluding process would detect information by being denied a new segment. This form of leakage, in addition to that via time channels, is not addressed here. Only leakage involving the V-functions of the operating system modules is considered.

With these assumptions, it is necessary to consider only the SEGMENT and EXTENDED TYPE modules. As noted below, a service can create segments and extended types using the functions provided by these modules. When the service has completed preparation of these objects, it passes them to the CSM for certification. The service is not allowed to pass to the CSM any capabilities for existing directories, I/O devices, processes, mail-boxes, or any other primitive storage medium.

A statement of the meaning of compiled confinement is cast in terms of the following two principles.

Internal Confined Object Principle--Assume the existence of a set S of capabilities (which can be viewed as corresponding to segments, extended-types, or extended

objects). The set S is INTERNALLY CONFINED if the use of any subset of S as an argument to (1) any accessible O- or OV-function does not result in a modification of the value of any V-function that has as argument the slave version of a member of S, and (2) any accessible V- or OV-function does not result in any capability being returned other than a capability in S, possibly with reduced access rights, or a previously unseen capability.

External Confined Object Principle-- A set S of capabilities is EXTERNALLY CONFINED if the use of any subset of S as arguments to any accessible O- or OV-function does not result in any hidden V-function that takes as argument a slave version of a member of S having its value modified so as to return a capability other than in S.

The Internal Confined Object Principle, if satisfied by a set of objects, ensures that the objects do not contain any capabilities that can be used to leak information out of a confined environment. The External Confined Object Principle relates to the capabilities that are returned to the service for the copies of objects that it requested to be certified to be confined. The principle, if satisfied, guarantees that the service cannot modify the certified copies of the objects, (in particular to insert capabilites for storage channels into the objects) except to delete the objects. The deletion is accomplished only via the CSM.

DISCUSSION OF THE SPECIFICATIONS

The specifications for the CSM are given in Appendix E. On the basis of the above discussion, definitions can be provided for a confined segment and a confined object. A segment is confined iff any embedded capability is for one of the following:

1. a system function that is confined.

2. another confined segment, but without the special access right "confined-delete" that allows the deletion of the segment via the CSM. (It is not necessary to test for write or delete access rights [via the segment module], since such rights are not returned by the CSM.)

3. a confined object, again without the access right "confined-delete".

An object is confined iff each capability entry in its representation vector is for one of the following:

1. a confined object, but without the "confined-delete" access right.

2. a confined segment, but without the "confined-delete" access right.

The CSM maintains a set of objects -- segments, extended type objects -- that have been certified to be confined. The parameter system_capabilities is the set of capabilities for system functions that are confined, including the basic machine instructions, in addition to the interface functions of the operating system. The V-functions: h_is_confined_segment, h_is_confined_type, and h_is_confined_object return TRUE if their argument is respectively a segment, type, and extended object that has been certified to be confined. The visible versions of these functions can be called by a user to determine if, for example, a procedure capability passed to the user by a service is indeed confined. The OV-functions: create_confined_type, make_segments_confined, and make_object_confined are called (by a service) to establish the respective objects as confined.

For example, the function make_segments_confined takes as argument a vector of capabilities, cv. (Note that each of a set of mutually recursive segments cannot be individually certified.) In conformance with the Internal Confined Object Principle, the specifications indicate that the call is rejected if any of the segments in cv contains a capability other than for a confined system function, a previously certified object, or a member of cv. If this principle is satisfied, then a copy of each of these segments is established, and again in conformance with the Internal Confined Object Principle, all access rights for modification, via the SEGMENT module or the CSM, are removed from the capabilities that are stored in the segments. In conformance with the External Confined-Object Principle, the returned capabilities do not permit write access or delete access via the SEGMENT module. However, the returned capabilities do contain the confined-delete access right which permits the deletion of confined-segments. In implementation the CSM will delete the concrete segment corresponding to a confined segment, using the capabilities with delete access that it retains.

The function make_object_confined takes as arguments a capability t1 for a confined type and a vector of capabilites cv, each member of which corresponds to a certified extended object or segment. A new extended object c1 is created whose representation is the capability vector cv, although the access right to delete (via the CSM function delete_confined_object) is removed from the stored version of cv[i], for all i. Note that the capability c1 has no access rights corresponding to add_rep and delete_rep, thus precluding leakage through the extended type V_function_impl_cap.

In this chapter a simple, although practical, solution to a particular confinement problem is presented. The solution prevents leakage from an environment via any storage channel. A slight extension could handle the case of leakage via resource

exhaustion. The problem of leakage via time channels remains as an extremely difficult problem.

CHAPTER IV-3
A SECURE RELATIONAL DATA MANAGEMENT SYSTEM

INTRODUCTION

This chapter summarizes the design of a secure data management system suitable for implementation on PSOS. The design is hierarchical and follows the methodology used for the operating system and for the other applications discussed here.

The goals of the data management system discussed here are familiar. These are to provide storage and retrieval facilities for large data collections, with a high degree of generality in data description, in query definition, and in the user interface. Also included are the goals of efficiency of operation and ease of use.

The main goal of this effort is not the data management system itself, but rather the demonstration of the suitability of the methodology for developing an application subsystem with appropriate security. To this end, the concept of a relational data base was chosen as being representative of suitable generality, while presenting realistic problems of efficiency of operation and ease of use. It is recognized that there is much controversy in the data management world over the various types of data management systems. The choice of a relational model here should not be construed as an advocacy of that approach.

The main emphasis here is on the mechanisms of the lower levels of the data management subsystem, namely, the notions of relations and views and the access authorization that they provide. The higher-level issues of providing appropriate user interfaces are considered to be important, but secondary for present purposes.

OVERVIEW OF THE DESIGN

The design given here is fairly simple, but general. It is relatively easy to embellish so as to increase efficiency or ease of use. However, where such embellishments do not add to the illustration of the methodology, their presence has been eschewed.

In accordance with the use of the methodology, the design is decomposed into levels of abstraction. The fundamental abstraction used here is that of relations. A relation contains data organized as a set of tuples that can be inserted and retreived, and whose values can be read and modified. The individual fields can be named and manipulated separately. The next higher-level abstraction is that of views, which provide authorization for selective accessing of the data in a relation,

including reading and writing of data in particular data fields by context. Views permit access to relations in a higher-level language than that provided by relations themselves. The next higher-level abstraction is that of queries, which permit requests about the data in relations that are more user-oriented. These three levels are part of the design, and can be augmented with other levels, e.g., lower levels for retrieval efficiency, and higher levels for user convenience. These levels of abstraction are discussed below.

RELATIONS

The literature on the relational model of data has been growing rapidly in the last few years, beginning with Codd [70]. For recent developments, the reader is referred to Chamberlin et al. [75], Mylopoulos et al. [75], Held et al. [75], and Schmid and Bernstein [75], for example. An excellent survey is given by Chamberlin [76].

For present purposes, the terms DOMAIN, SCHEMA, RELATION, and TUPLE are defined first, in order to develop the concept of a relational data base. In informal terms, a DOMAIN is a basic semantic variable of a body of data. More formally, a domain d is a variable of a particular type (here considered as a character string) whose range is a set of possible values. A convenient example, given by Chamberlin et al. [75], involves domains such as EMPLOYEE, SALARY, MANAGER, DEPARTMENT, ITEM, VOLUME, and FLOOR. A SCHEMA s is a vector of domains chosen to model some body of data. Three schemas are given.

s1: EMPLOYEE, SALARY, MANAGER, DEPARTMENT

s2: DEPARTMENT, ITEM, VOLUME

s3: DEPARTMENT, FLOOR

In formal terms, a RELATION r for a schema s with domains (d1, ..., dn) is a subset of the cartesian product d1 X d2 X ... X dn, i.e., a particular set of instances of a general schema. Three relations r1, r2, and r3 are considered here, based on the schemas s1, s2, and s3, respectively. These are given the symbolic names EMPLOYMENT, SALES, and LOCATIONS by which they may be identified at the query level. The symbol w (w1, w2, etc.) is used to denote a value.

EMPLOYMENT is a set of tuples {[w1, w2, w3, w4]} : r1
SALES is a set of tuples {[w4, w5, w6]} : r2
LOCATIONS is a set of tuples {[w4, w7]} : r3

A TUPLE t of a relation r is an instance of the schema for that relation. That is, it is a vector tw of values, one from

each of the domains in the schema for which the relation is defined. A relation is then seen to be a set of tuples for its schema. For example, the relation r1 (EMPLOYMENT) may contain tuples such as [Smith,9000,Kelly,personnel].

A RELATIONAL DATA BASE for a set of domains is a collection of relations each of whose schemas is a subset of that set of domains. In the design given here, the collection of relations in any data base rdb is catalogued as a directory of relations for that data base, r_set = get_relations(rdb). (The functions pertaining to relations are summarized in Table IV-3.1.) In the above example, the three relations r1, r2, r3 form a relational data base rdb (called DEPARTMENT_STORE) that may be one of many data bases known to the system. A catalog of data bases is provided by the function rdb_set = get_data_bases().

Additional functions provide the ability to create relations and to modify their contents. For example, t_set = get_tuples(rdb,r) provides the set of tuples forming the relation r. The function tw = get_values(rdb,r,t) provides the vector of values tw[i] in a particular tuple t. The function update_tuple(rdb,r,t,tw) permits the assignment of new values to a tuple. The set of functions associated with the maintenance of relations forms the module RELATIONS (see Table IV-3.1). (Terminologies in the literature differ slightly from one to another. The notion of relations given here is also slightly different from the others. What is sometimes called an attribute is here called a domain. Within a schema there may be differently identified domains with the same range, but the identically identified domain is not allowed to appear repeatedly in the same relation.)

(For simplicity, it is assumed here that all relations are in "third normal form" -- see Chamberlin [76]. Intuitively this means that each relation deals with a single concept and contains at least one unique key. This assumption greatly simplifies inserting, deleting, and updating. It avoids ambiguity and minimizes inconsistency.)

VIEWS

Access to a data base is governed by the use of VIEWS on the relations of that data base. A view acts as a selector, or mask, which, when applied to a particular relation, selects specific tuples from that relation, and extracts appropriate values from those tuples. Queries, discussed below, are used to access data by invoking views on different relations of the same data base.

In the given design, a view is applied to a relation by means of the function t1_set = extract(rdb,view). In the sense that the result of this extraction is itself a relation (with the same schema), a view may conveniently be considered as itself being a relation (i.e., as the result of extraction). However,

this can be confusing unless careful distinction is made between the vectors defining a view and the tuples of either the original relation or the extracted relation. This distinction is helpful, since the definition of a view itself looks like that of a relation.

In the sense that an authorized view is required in order to extract information from a relation, a view also acts as a capability for selective access to the relation. In the given design, the set of authorized views for each relation is catalogued in a directory of views, view_set = get_views(rdb,r). Access to these views may be granted as desired.

PRIMITIVE VIEWS

A view is either primitive or nonprimitive. A PRIMITIVE VIEW v for a given relation r contains a single vector vw of values for the schema of the relation, where vw = get_view_vector(rdb,v). However, each value vw[i] may be either a value from its own domain, or one of two special values "*" and "%". Here "*" denotes the universal value, and "%" denotes the null value. (For simplicity, "*" and "%" are considered to belong to every domain. However, note that each domain could also contain a null value of its own, other that "%".) In essence, "*" extracts any given value, while "%" extracts only itself, irrespective of the given value. Each view also contains appropriate access authorization, discussed below.

More precisely, consider the value vw[i] (for domain s[i]) for a view v, and the value tw[i] of a tuple t to be considered for selection, where tw = get_values(rdb,t). A value tw[i] is EXTRACTABLE by vw[i] if and only if vw[i] is "*" or "%" or the value tw[i] itself. A tuple is SELECTED if and only if each of its values is extractable. If a tuple t of relation r is selected, extraction takes place as follows, as a part of extract(rdb,v). If vw[i] = "*" or vw[i] = tw[i], then tw[i] is extracted; if vw[i] = "%", then "%" is extracted. (Note that if neither of these cases applies, then by definition the tuple is not selected.) Thus "*" acts as a "don't-care" in the selection, while "%" acts as a filter (mask) in the extraction. No values from domain s[i] may be viewed when vw[i] = "%". The null value "%" is returned in place of the actual value in position i of any selected tuple. The extraction of the tuple value tw[i] by the view value vw[i] is summarized in Table IV-3.2.

Specifications for the modules RELATIONS and PRIMITIVE-VIEWS are given in Appendix F. These represent a detailed design for a prototype set of functions supporting the relational concept. The following discussion of queries is purposely not specified in detail, in that the relations and views modules are intended to be general enough to support different query languages.

As an example, consider the schema s1 and the relation r1 (called EMPLOYMENT) on that schema. A primitive view on that relation r1 has the view vector vw = [*,*,*,*], which provides access to each domain of each tuple in the relation. A restricted primitive view on r1 is v1 with vw1 = [*,%,*,*], which provides access to all tuples, but with no SALARY information. A still more restricted primitive view on r1 is v2 with vw2 = [*,%,Kelly,%], which provides access to just those tuples in r1 with Kelly as the manager, returning the employee information, but filtering out the salary and department information. The result of "extract(rdb,v1)" is thus a set of tuples [w1,%,Kelly,%].

Since the result of t1_set = extract(rdb,view) is a set of tuples over the same domain as that for r, this result is itself a relation (with the convention that the values "%" and "*" are included in every domain). Thus, one view for a given relation may have a second view applied to it, and so on.

The above discussion concerns reading data through a view. Similarly, operations exist to update, insert, and delete through a view. In the case of updating, the function "v_update" permits a value to be updated for each tuple in a relation selected by the view. The function "v_insert" permits a new tuple to be inserted. The function "v_delete" permits tuples selected by the view to be deleted. Additional functions permit new views to be created for existing relations and more restricted views to be created for existing views. The set of functions provided by the module PRIMITIVE-VIEWS is summarized in Table IV-3.1.

NONPRIMITIVE VIEWS

A NONPRIMITIVE VIEW for a relation r is a set of two or more primitive views for r; it may be built up by set UNIONs of primitive views. In general, then, a VIEW is either a nonprimitive view or a set consisting of just one primitive view. Although externally it is thought of as just a set of primitive views, e.g., {[*,%,Kelly,*], [*,*,Jones,*]} on r1, a nonprimitive view could be represented in some compressed internal form (see below).

QUERIES

A QUERY is a request to obtain information from a data base and is stated in some well-defined language called a query language. Queries operate on views and may provide more flexibility than views with respect to formatting and searching. They are often also far more concise. Whereas each view above is for a single relation (although this is not a necessary distinction -- see below), queries may involve multiple relations. (The distinction between queries and views is somewhat similar to interpreted versus compiled access, although a query may in fact be compiled into extractions using particular

views.)

As an example, consider the primitive view v3 with vw3 = [*,*,Kelly,%] on the relation r1 above. Suppose it is desired to obtain the set of those employees whose salary is at most $8000. This could be conceived of as a set of primitive views v4 with vw4 = {[*,8000,Kelly,%], [*,7999,Kelly,%], ...} on r1, spanning the entire salary range up to $8000. However, from an efficiency viewpoint, such a representation would be ridiculous. Thus some notation such as vw4 = [*,<8000,Kelly,%] would seem appropriate, if this set were to be represented directly as a nonprimitive view. An alternative is the construction of a query.

Many query languages can be implemented on top of the mechanisms provided here. For illustrative purposes, SEQUEL is used here. (See Boyce and Chamberlin [74], and Chamberlin et al. [75].) Relative to the running example of the relations r1, r2, r3 given above, SEQUEL permits queries of varying complexity, such as the following.

```
Q1: SELECT EMPLOYEE,SALARY
    FROM EMPLOYMENT
    WHERE DEPARTMENT = personnel;

Q2: SELECT EMPLOYEE,FLOOR
    FROM EMPLOYMENT,LOCATIONS
    WHERE EMPLOYMENT.DEPARTMENT = LOCATIONS.DEPARTMENT;
```

Similarly, the example of the view v4 above would be handled by a query

```
Q3: SELECT EMPLOYEE,SALARY
    FROM EMPLOYMENT
    WHERE MANAGER = Kelly
    AND SALARY <= 8000
```

In attempting to honor such a query, the data management system must first check that the query is consistent with views authorized to the user (see below) and then retrieve and format the desired information. Using the query language, it is possible to rename or permute the order of the domains, to convert the units or representation of a domain, and to eliminate domains altogether.

As an example of the honoring of a query, consider the query Q2. Two views are required, such as va = [*,%,%,*] on r1 and vb = [*,*] on r3. The views [*,<8000,%,*] and [*,*], respectively, would also suffice, although the query would then provide the desired information only for those employees with salaries under $8000.

If such a query were to be used repeatedly, it would be desirable to state it directly or compile it into a sequence of

statements in the view language. However, the accesses defined by each view must be reevaluated on each such reuse to ensure the extraction of the most recent version of the data and to guarantee that any revoked access will be correctly revoked. In the case of Q2, the query corresponds to the set "result" of ordered pairs [EMPLOYEE , FLOOR] given by

```
LET ta_set = extract (rdb,r1,va);
LET tb_set = extract (rdb,r3,vb);
FORALL ta INSET ta_set
  FORALL tb INSET tb_set
    IF ta[4] = tb[1] THEN [ta[1], tb[2]] INSET result;
```

Note that queries could be named and parameterized. For example, Q1 could be generalized for an arbitrary department "dept", and cited as Q1(dept) -- with the given example being Q1(personnel).

If the data management system ensures consistency of all data entered into it, then it is possible to eliminate mention of specific relations as well as specific views from the queries, and thus leave the choice of view and relation up to the data management system. For example, consider the more abstract statement of Q2, as follows.

Q2: SELECT EMPLOYEE, FLOOR

This query could be interpreted as before. However, if a (redundant) fourth relation existed that included both EMPLOYEE and FLOOR, then ambiguity would exist as to how to interpret the query. Nevertheless, the correct interpretation would be made, so long as the data base was consistent.

ENRICHMENTS OF THE VIEW LANGUAGE

Since a view is the unit of authorization for selective access, it may be desirable to provide increased granularity of protection. One way to do this is to treat queries as complex views and to protect them in a similar manner. Another (possibly equivalent) approach is to enrich the scope of the view representations, as suggested above by v4. In addition to the "<" notation mentioned above and the related comparisons (e.g., ">", ">="), it would be desirable to write arbitrary expressions on the values of the tuples in a relation, such as

vw[1] > "Q", assuming vw[1] is a character (ASCII collating sequence implied),
vw[2] < vw[3] + vw[4], assuming vw[i] are integers,
vw[5] AND vw[6] = TRUE, assuming vw[i] boolean,
vw[7] = vw[8], assuming vw[i] of the same type,

and so on. These constraints are easily expressible in the query language. However, there could be advantages to expressing them explicitly as views.

Another enrichment would be to require the vectors defining views to contain sets rather than values, and to redefine "%" and "*" to be the null set and the universal set, respectively. Then extraction can be specified on the basis of set inclusion. This seems somewhat more natural than the formulation presented here in terms of the extraction; however, it significantly complicates the view definition language. (On the other hand, the view definitions would no longer look like relations.)

It is straightforward to make each relation and each view order-independent, e.g., by requiring ordered pairs of domain-typed descriptors and corresponding values. Then tuples would look like [d1:w1, d2:w2, ...]. This tends to complicate the calls given here, although it may be desirable for other reasons. However, note that the query language is order-independent.

Another possible enrichment entails the creation and naming of multirelational views. However, with the notion of views used here, these are unnecessary if multi-relational queries are compiled into efficient sequences of requests on views.

Some of these elaborations of the view definition language could be included in a production relational data management system. For present purposes, however, they are beyond the scope of the desired illustration. Therefore they are not included in the specifications. In general, views are expected to be fairly simple, indicating essentially just what domains are accessible, and how they may be restricted. The more sophisticated selections are expected to be handled by the query language, although frequently used queries should presumably be explicitly stated -- either written in or compiled into the view language.

ACCESS AUTHORIZATION

As noted above, authorization to access values of certain domains in a relation requires possession of a suitable view for that relation. Moreover, it requires authorization for the intended use of the information accessible via the given view. In particular, specific authorization is required to read (extract) or update a relation, as well as to perform other operations. Such authorization is associated with each relation and with views on relations. Authorization to access a relation is undifferentiated by domains. That is, if a relation is directly readable (or writable), each value of every tuple in that relation is readable (or writable). In general, however, each relation is directly accessible only to its creator. He may in turn grant selective access to others by means of views.

Specific authorization is required (and may be granted) to destroy a relation, to grant and revoke access to a relation with specified authorization, and to read, insert, delete, and update

tuples within a relation. There is also control over whether a user may create relations at all within a given data base.

Access to a view of a relation is controlled by explicit authorization associated with that view. However, this access is differentiated by domains, based on the occurrences of "%" and "*" within the view. For a view of a particular relation, different domains of the relation may have different accessibility. For example, the maintainer of a relation may disperse different views to different users, each reflecting access to different domains within the tuples of that relation.

As an example, possession of the view v1 = [*,%,*,*] on r1 with read authorization permits read access to the entire relation except for the salary field, which is invisible to this view. If this view is the only one provided for this relation to a particular user, he can never read any salary information, either explicitly or implicitly. He may in turn construct from v1 on r1 (and pass to other users) more constrained views -- for example, v5 = [Green,%,*,*] on r1, or v6 = [*,%,Jones,*] on r1 -- but can never construct more powerful views than what he has available. Note that the absence of read authorization prohibits the use of any operation (at the view level and at the query level) that requires the reading of the prohibited domain information.

Permission to update the values of any tuple within a relation requires a view with "*" in the positions to be modified, as well as "update" authorization associated with that view. A view with a particular value, e.g., v3 = [*,*,Kelly,%] on r1 with "update" authorization would permit updating of just those tuples containing the cited value for the appropriate domain. Similarly, authorization to insert, delete, and indeed to destroy the relation itself requires a view with the appropriate authorization.

Note that a user may have one view for a relation with some particular authorization, as well as a different view for that relation with a different authorization. An example for views on r1 would be v1 with read authorization and v6 with update authorization.

A view, like a capability, never changes once it is created. It may be given to other users and may be used to generate more restricted views. However, a revocable view may be created for a given view; the revocable view retains the usefulness of the original view until it is revoked -- at which time it loses all usefulness.

HIERARCHICAL STRUCTURE OF A RELATIONAL DATA MANAGEMENT SYSTEM

A data management system can be structured around these abstractions, for example with five conceptual levels built

successively upon the primitives of an operating system interface. From the top level of the data management system downward, these levels are as follows.

COMMANDS (including queries)
VIEWS
PRIMITIVE VIEWS
RELATIONS
RETRIEVAL
THE OPERATING SYSTEM (including virtual memory)

The functions of each of these levels should now be fairly self-evident. The COMMANDS level of the data management system accepts requests in some convenient query language to define relations, to instantiate them, and to retrieve data according to available views. It is capable of interpreting multirelation requests. The VIEWS level maintains sets of primitive views for the same relation. The PRIMITIVE VIEWS level interprets primitive views, and enforces the desired access authorization. (It could be absorbed into the VIEWS level.) The RELATIONS level maintains data in the form of various relations. The RETRIEVAL level aids in obtaining an efficient implementation of relations in terms of the primitives of the operating system. It is particularly concerned with the efficient management of virtual memory. It could conceivably be absorbed into the RELATIONS level, but is kept separate here so as to expose some issues that may have major impact on system performance. Table IV-3.1 provides a summary of specific functions of each level.

The design is conceptually similar in concept to those of Mylopoulos et al. [75], Schmid and Bernstein [75], and Astrahan et al. [76].

THE DETAILED DESIGN

The specifications for the RELATIONS module and for the PRIMITIVE VIEWS level are given in Appendix F. Tables IV-3.2 and 3 illustrate the functions "extract" and "and_views" of the PRIMITIVE VIEWS module.

Various nonprimitive functions have been omitted that can easily be implemented by using the available functions. For example, in the module RELATIONS, it may be desired to add a new domain (initially with null values for each tuple) to a given relation and thence to update the data for that domain. Similarly, operations for flagging ("marking") certain tuples or linking tuples of different relations can be conceived; however, they can be conceptually handled by including extra domains within a relation, to hold the marking or linking data by means of the functions specified here. Other special mechanisms may be readily added if desired for efficiency or ease of use.

Similarly, set operations on relations are desirable, such as set union, set intersection, set difference, Cartesian product, and projection (i.e., domain elimination). However, in a data base in which updates are made continually, the derivative relations thus obtained must also be maintained dynamically. Thus, it is natural to use views to perform such operations.

As noted above, for simplicity, the order of domains within a relation and within a view is made visible at the interfaces to the relations and views modules, but not at the query level. This is not essential. Note also that update by domain rather than by index is in fact provided by the function "update_tuple_value".

IMPLEMENTATION

The specifications are given in terms of abstract designators for domains, schemas, relations, tuples, relational data bases, and views. The access authorization inherent in data bases, relations, and views could be implemented either in terms of capabilities, or in terms of access control lists, or perhaps even as a combination of both. The designators for domains, schemas, and tuples can be represented simply as integers or symbolic names, as desired.

Considerable attention is paid in the literature to whether relational data base systems can be as efficient as conventional systems. The latter typically have lower-level language interfaces, and devote greater attention to accessing details -- although in many cases putting greater burdens on the users. The consensus seems to be that, whereas there are certain applications for which a relational interface is less efficient, clever optimization (e.g., at the RETRIEVAL level and in the choice of normal form) can make relational systems essentially as efficient for a very large and realistic class of applications. Besides, where efficiency is critical, a user may be permitted to use primitives of the views and relations modules directly. (Additional primitives desirable for increasing efficiency are omitted here.)

Numerous other implementation issues also arise naturally in this design. such as efficient handling of multi-relation queries, associative bypasses to deferred-update (e.g., batched-update) data bases. For an interesting discussion of various implementation issues, see Codd [75].

EXTENSIONS FOR MULTILEVEL SECURITY

Multilevel security is supported by the secure object manager described in Part IV-2. The design of the data management system presented here may be readily extended to support the desired multilevel security and integrity properties

via one of several approaches. One approach is to treat relations and views as secure objects, with all values of all domains within a given relation or view being at the same access level (security and integrity levels combined). In this case, the data management system can be implemented directly on top of the secure object manager.

A second approach is to permit each domain in a relation to have a different access level. In this case, efficiency reasons make it undesirable to treat the values of each domain as a separate secure object. However, it is straightforward to add access levels to the access information already present on each domain of each view.

A third approach is to permit each value of each domain within each relation applied to a relation to have its own access level.

In the first approach, each relation or view on a relation has an access level. That is, the functions "get_access(rdb,r)" and "get_view_access(rdb,v)" are supplemented by two new functions, "get_access_level(rdb,r)" and "get_view_access_level(rdb,v)", that return the access level of the relation and the access level of the view, respectively.

A relation or view has READ AUTHORIZATION if the access code for read is present. A relation or view has WRITE AUTHORIZATION if the access code for update, insert, delete, grant, invoke, or destroy is present. For all read and write authorizations, the desired multilevel access properties must be satisfied. That is,

* For read authorizations, the user's access level must be at least that of the relation or view.

* For write authorizations, the user's access level must be at most that of the relation or view.

* For read and write authorizations, the user's access level must be exactly that of the relation or view.

A view for a particular relation must always have a suitable access level compared to that of the relation or view for which it is created. In particular, the functions

"v = create_view(rdb,r,bv)" and
"v = create_restricted_view(rdb,vl,vwl,bv)"

must ensure that this is the case. Thus,

* If v is a view with a read authorization for the relation r (or for the view vl), then the access level of v must be at least that of r (or vl).

* If v is a view with a write authorization for the relation r (or for the view v1), then the access level of v must be at most that of r (or v1).

Also, by combination,

* If v is a view with a read and a write authorization for the relation r (or for the view v1), then the access level of v must be exactly that of r (or v1).

The second approach mentioned above is essentially the same, except that the access level information returned by "get_access_level(rdb,r)" and "get_view_access_level(rdb,v)" is a vector of access levels, with the ith element corresponding to the access level for the ith domain in the relation or view, respectively. In this way, the first approach is made a special case of the second, in which all elements of each vector are identical.

The third approach appears to be significantly more complex, both to specify and to implement. It also seems frought with pitfalls concerning inferential leakage of information. To a first approximation, the second approach seems to be the one worth pursuing.

CONCLUSIONS

This chapter presents the outline of a simplified relational data management system. The simple design specified here is intended to illustrate the applicability of the methodology to the design of secure applications subsystems. It is noted that the machine-independence of the relational interfaces makes them appropriate for implementation on various systems. However, the capability-based addressing underlying the operating system is particularly appropriate here, even though capabilities do not appear at all in the query language. Thus PSOS would be an appropriate system upon which to implement such a data management system.

TABLE 4-3.1. SUMMARY OF THE LOWER-LEVEL DATA MANAGEMENT SYSTEM FUNCTIONS

```
-----------------------------------------------
PRIMITIVE VIEWS:
extract
create_view
destroy_view
grant_access      (create_restricted_view)
revoke_access     (revoke_restricted_view)
v_update
v_insert
v_delete

RELATIONS:
create_data_base
create_domain
create_schema
create_relation
create_tuple
delete_tuple
update_tuple_value
destroy_relation
rename_relation
-----------------------------------------------
```

TABLE 4-3.2. RESULT OF extract(rdb,v) ON TUPLE tw[i], if selected; "-" indicates no selection; w' ~= w.

EXTRACT:	tw[i]			
tw1[i]=	%	*	w	w'
%	%	%	%	%
vw[i] *	%	*	w	w'
w	-	-	w	-

TABLE 4-3.3. RESULT OF and_views(vw[i],vw1[i]); w' ~= w.

ANDVIEWS:	vw[i]			
vw2[i]=	%	*	w	w'
%	%	%	%	%
vw1[i] *	%	*	w	w'
w	%	w	w	%

CHAPTER IV-4
MONITORING

This section considers the role of monitoring in PSOS. Monitoring of security and monitoring of performance are both discussed. They are interrelated by the need to prevent denial of service. They also must fit into the system in similar ways, without compromising security. Thus, it is advantageous to treat both kinds of monitoring within the same framework. However, primary emphasis is placed on monitoring of security.

Monitoring in this system is similar in purpose to its counterpart in typical operating systems, but is different in its design and implementation. First, it is integrated with the operating system design, but is logically external to the operating system. Most of the functionality is outside of what would normally be called the security kernel. Security of the monitoring functions themselves is thus obtained by the isolation of these functions in the design and by the judicious administrative control of the distribution of the capabilities for these functions. (Capabilities distributed dynamically can be process-limited so as to prevent their propagation.) Second, proofs of the intrinsic security of the monitoring functions can be given in exactly the same way as the proofs concerning the security of the operating system functions or of other applications functions. Third, the monitoring subsystem can be fairly general. It is possible to monitor most interesting events directly, without extensive changes to the system design. Since monitoring is basically a policy matter, it should not require continual changes to the system mechanisms upon which that policy is implemented -- assuming the system is well designed. The basic mechanisms are considered here.

Administrative functions such as metering and billing of resource usage and the enforcement of quotas on resource usage are also related to monitoring. Quotas are explicitly a part of the design, appearing as exception conditions in all relevant specifications. Similarly, the detection and correction of errors for fault-tolerance are related. Fault-tolerance, for example, requires monitoring of the reliability of system behavior and invoking appropriate recovery when faults are detected. Although the hardware can be made sufficiently redundant to avoid critical errors in addressing or in input-output channel isolation resulting in security compromises, there is also a need for software monitoring to ensure the consistency of critical software tables. Thus, the view taken here is sufficiently general to provide a unified view of all such monitoring activities, but is specific enough to permit efficient and secure handling of security monitoring.

1. CONSISTENCY WITH THE DESIGN

In most systems, monitoring is potentially detrimental to system security. In other words, the ability to monitor system behavior is potentially tantamount to the ability to compromise the security of the system. An example is provided by the visibility of page-fault activity that is the basis for the now-classical TENEX password detection scheme (no longer a problem in TENEX). In this scheme, it was possible to do a simple exhaustion of each character position, one at a time, gaining knowledge of the correct character on the basis of whether the next character (residing across a page boundary) caused a page fault. By shifting the would-be password one character after each such correctly determined character, the effort to obtain a password became linear instead of exponential. To minimize the deleterious effects of monitoring, it is desirable that all primitive monitoring functions be integrated into the system design and proofs, rather than be subsequently added onto the system design without being subject to proof.

1a. THE DESIRE FOR PASSIVE MONITORING

The use of all information resulting from monitoring is controlled by the capability-based addressing of the system. No user or process can have access to monitoring programs or monitoring data except via authorization by monitoring officers and their explicit delegates. All monitoring functions that return values exist at a level of abstraction appropriate to the functions being monitored -- i.e., high enough to avoid undesired release of low-level, nonvisible V-function values. In other words, implementation detail is hidden throughout and is compatible with the data abstractions involved. Thus, handling of monitoring information is completely consistent with the system design. For example, a monitoring routine can never directly alter lower-level V-functions consistent with all O- and OV-functions of the system. (This constraint is imposed by the specification language.)

Furthermore, an even stronger constraint is imposed on the monitoring functions. The term "NORMAL CONDITIONS" indicates that no threat to security has yet been detected. Under normal conditions, monitoring functions are constrained not to alter any system V-functions, not even indirectly via invoking O- or OV-functions. This constraint (called the PASSIVITY CONSTRAINT) thus ensures that monitoring is PASSIVE UNDER NORMAL CONDITIONS, where "PASSIVE" implies no changes to any visible V-functions of the operating system. Note that the V-functions of the monitoring subsystem that may change under normal conditions are considered to be an extension of the system V-functions, not a part of them. Some monitoring functions are able to

append to existing data bases of the monitoring subsystem, but are unable to overwrite the monitoring data or any other data. An example of this is the case of maintaining an audit trail. Other monitoring functions are essentially derived V-functions and cannot write at all -- as in the case of a monitoring program that determines which users are currently logged in and what they are doing. Thus, in these two cases there is READ-AND-APPEND-ONLY monitoring and READ-ONLY monitoring respectively, although both are passive (with respect to the operating system data bases).

1b. ACTIVE MONITORING UNDER ABNORMAL CONDITIONS

When the monitoring subsystem detects a potential security threat, it may wish to take some sort of action. This action could result either in a message to a security officer or in some immediate action, such as logging a user off of the system automatically, destroying a process, or locking up a critical segment. Thus, the monitoring system may become ACTIVE UNDER ABNORMAL CONDITIONS. Although it may still not directly modify any lower-level V-functions directly, it may modify them indirectly through use of existing O- and OV-functions explicitly authorized to the monitoring subsystem. A few special functions (such as forced_logout) may be required that would be restricted to use by the active monitor only. In each such case, however, appropriate capabilities must be explicitly made available. For example, the program that creates processes must transmit to the monitoring subsystem the capability for each process, including "delete" ability.

1c. INTRINSICALLY SECURE MONITORING

It is vital to recognize that any desired active monitoring function not already supported by the operating system is a potential violation of security. Thus, active monitoring should be eschewed, except where the absence of undesirable side-effects can be demonstrated. If it is in fact ensured that the monitoring subsystem obeys the desired constraints of the operating system even under abnormal conditions, it follows that the monitoring subsystem itself does not violate the security of the system and that it can cause no harmful side-effects or interaction with parts of the operating system (assuming that the subsystem is used nonmaliciously by the security officers).

In connection with the secure object manager (SOM), a monitoring subsystem for usage of the SOM must reside at or above the implementation level of the SOM. It should obey the security model (e.g., the *-property and the simple security condition). Thus, a monitor with top-secret security classification is needed. There may also be monitors with lower classification levels that are unable to

get access to monitoring information of higher classification levels.

2. APPROACHES TO MONITORING

There are two modes of monitoring, system-wide (GLOBAL), and per-user (LOCAL). Per-user monitoring that includes monitoring of several related processes for a single user is considered to be local.

There are essentially three approaches to monitoring, involving embedding, synchronous interposition, and asynchronous monitoring. These may be categorized as follows.

(A1) Embedding of monitoring functions into the modules being monitored (synchronously);

(A2) Synchronous interposition of monitoring functions
* on calling,
* on invocation of an exception condition, or
* on returning; and

(A3) Asynchronous use of monitoring data bases

The first of these approaches is internal to the functions being monitored. An example is the login monitor that records incorrect attempts to log in (perhaps recording terminal id and source -- if via a network). On a correct login, or perhaps by external notification, the authorized user should then be notified of all such attempts. The second approach is external to these functions, but internal to the calls. The third approach is external to the use of the functions being monitored. The first two approaches are synchronous to the use of the function, and the third is asynchronous to the invocation of the function being monitored. For each of these approaches (Ai), i= 1, 2, 3, both global monitoring (AiG) and local monitoring (AiL) are generally meaningful. However, in case (A1G), there is a danger of the embedded monitoring facilitating a Trojan horse, unless the module is either proved (e.g., if it is part of the operating system), or if it is forced to execute as confined subsystem. In case (A2G), global monitoring requires global data bases in which to store its results. Such data bases must either be append-only (no read), or else must be such that no information leakage can occur. These approaches are discussed next.

EMBEDDED MONITORING FUNCTIONS

The traditional approach to monitoring is to permit all sorts of monitoring activities to execute along with the operating system, in many cases introducing ad hoc hooks and

patches into the operating system. In the past, few controls have been placed on the relative power of monitoring operations within a system. Since prevention of Trojan horses is important here, such a situation would be intolerable. However, since every monitoring function must be formally specified, proofs of the design and of the implementation will also include the embedded monitoring functions. Note that any embedded function must obey the constraints of the specification language with respect to accessible variables, e.g., using V-functions only of the module in which it is embedded.

Because of the notion of information hiding, appropriate information must be available via visible V-functions sufficient to support the desired monitoring. Thus, embedded monitoring functions are essential in certain cases, such as in performance monitoring of page-fault activity that would otherwise be unavailable outside of the PAGES module. However, such performance information may be strongly in violation of security requirements unless its propagation is stongly limited (e.g., by administrative policy). Thus, potentially critical interactions between performance monitoring and security monitoring must be resolved by administrative controls that limit who can have access to which pieces of monitoring information.

In many cases, embedded monitoring may entail nothing more than appending monitoring information to a data base. Note that this must be done with great care below the SEGMENTS module because paging is not available. Moreover, care must be exerted above that module because paging activity due to excessive monitoring can lead to system thrashing.

Security monitoring occurs essentially only on information explicitly available to the monitor operating at the user interface. The most general interface includes the CAPABILITIES, SEGMENTS, EXTENDED-TYPES, DIRECTORIES, USER-OBJECTS, and USER-PROCESSES modules. However, subsystems may mask the interfaces to these modules and provide other more restricted interfaces. They may also provide their own higher-level monitoring primitives. The CAPABILITIES module is extremely simple and probably needs no monitoring directly (apart from some check to ensure that the generator of unique identifiers is behaving properly). Thus, most security monitoring is in terms of higher-level concepts such as user objects (including directories) and user processes. The only embedded security monitoring is expected to involve the use of these objects. Performance monitoring, on the other hand, could require some embedded functions of lower levels (e.g., for page activity), although a strict enforcement of security may severely limit the availability of performance information.

SYNCHRONOUS INTERPOSITION

Much local monitoring can be achieved by the synchronous interposition of a monitoring program immediately preceding or as a part of the execution of the exception handler for any exception condition. Such a monitoring program should have access to all accessible variables of the process (including the calling arguments), but to nothing else. For reader convenience, a list of illustrative exception conditions that may result from invocations of user-accessible operating system functions is shown in Table 4-4.1.

As an example of an exception condition of Table 4-4.1, excessive occurrences of the exception condition "no_ability" could be of interest in monitoring attempts to access objects in ways other than those that are authorized. This can be used to monitor every unsuccessful attempted access. It could also be used in connection with the login procedure to monitor all unsuccessful attempts to log in. Excessive occurrences of the exception conditions "too_many_objects", "too_many", and "invalid_type_correspondence" may imply attempts to cause denial of service. The exception conditions from the secure object manager, or other user applications subsystem, could be of similar value.

ASYNCHRONOUS MONITORING

Much of the actual monitoring in the system is expected to be done asynchronously, even though the data bases used for such monitoring may have been established by embedded functions or by synchronous interposition. The major reason for this is that the monitoring operations at any operating system level or user level cannot write into data bases at higher levels. Thus, higher-level monitoring functions must interrogate the visible V-functions corresponding to the desired monitoring information maintained by lower-levels. Note that the "monitor" mechanism in the user interface may be used to set up events whose occurrence will wake up monitoring processes designed to act upon those events.

3. EXAMPLES OF SECURITY MONITORING

The specific monitoring functions depend somewhat on the actual choices of the command level and on the interfaces of the various user subsystems. Thus, the monitoring of PSOS deals with the objects of the PSOS interface, while the monitoring of the SOM deals with the objects of that interface. For present purposes, three generic monitoring commands are given here. These commands first require defining the desired monitoring information (to be appended to a segment maintained at some appropriate

level) and then reading that information. Reading may be accomplished synchronously or asynchronously with respect to the occurrence of an event to be monitored. The generic commands are:

```
record(events, options, synchmode)
wait_on(events)
observe(events)
```

In these functions, the argument "event" indicates a set of circumstances to be monitored; "options" indicates how this is to be done. The "wait_on" function implies synchronous monitoring, with immediate notification. It is assumed to be a passive monitoring operation, unless an active monitoring protocol has been administratively established. Note that the active protocol must execute at, and be certified as a part of, the appropriate lower level to avoid the situation in which an uncertified higher-level program can affect the correctness of lower levels. The "observe" function implies asynchronous monitoring and is clearly passive in itself. It may, however, provoke active monitoring, e.g., by human intervention. A collection of illustrative options for these commands is given in Table 4-4.2.

For example, using the event "procedure call login" and the exception condition "no_ability" (or perhaps just the exception condition "illegal_password" of the login procedure itself), it is possible to monitor all unsuccessful login attempts due to the presentation of a bad password. (After several such attempts, various actions could be taken.)

4. AUDITING

With any event that can be monitored, it is also possible to leave an audit trail of the occurrences of any such event. This merely requires the copying of the desired information from the record made during monitoring into a longer-lived auditing data base, by the generic command "audit(events)". In this way, an audit trail can be obtained of all unsuccessful logins, calls to certain procedures, all operator requests, or all requests by system administrators. All audit trail entries are automaticaly identified by user or process identifier, along with other relevant identifying information.

Extensive monitoring and auditing are expected to be a routine part of system operation. For example, all data leaving the system should be logged, including printed documents, external tapes being written, and data being transferred via a network interface to another system. Similarly, all logins and logouts should be logged, perhaps both internally and on a dedicated terminal.

Extensive monitoring and auditing are expected to accompany the secure document manager and the confined subsystem manager. Every change of classification level (perhaps implemented via a fresh login) should be recorded. Upgradings or downgradings of classification level should also be recorded, whether automatic or manual. Attempts to read up or write down could also be audited. Finally, it may be desirable to provide routinely an audit trail for every access to a document above some classification level such as secret, including the user identification and other relevant information.

5. ADMINISTRATION OF THE SYSTEM

Various software modules for accounting, billing, and enforcing quotas will be associated with the system. These modules must obey the security of the system. Their usage should also be monitored and probably audited. Thus, all of the monitoring and auditing tools should also apply to the system administration functions.

TABLE 4-4.1. EXCEPTION CONDITIONS OF VISIBLE FUNCTIONS

EXCEPTION CONDITIONS	MODULE
not_security_officer	SECURE ENVIRONMENTS
no_user	
bad_level	
bad_password	
bad_cat_set	
exists_user	
too_many_directories	SECURE DIRECTORIES
no_directory	
name_exists	
too_many_entries	
bad_access	
no_entries	
no_entry	
too_many_sdirectories	
initiated	
not_initiated	
too_many_documents	SECURE DOCUMENTS
bad_document_access	
bad_scapability_write	
device_in_use(u)	INPUT-OUTPUT, level 10
too_many_commands(u)	
unitialized_device(u)	
too_much_output(u)	
no_input(u)	
no_user_process(up)	USER PROCESSES: level 9
not_suspended(up)	
too_many_processes()	
no_procedure(u)	
out_of_bounds(up,n)	
bad_stack_offset(up,n)	
bad_call(up,u,o)	
bad_return(up)	
too_many_segments	USER OBJECTS: level 8
too_many_directories	
too_many_objects	
no_create(cdt)	
impl_objs_present(u)	
no_dir(u)	DIRECTORIES: level 7
name_used(u,n)	
max_num(u)	
no_entry(u,n)	
not_distinguished(u,n)	

invalid_type(u)	EXTENDED-TYPES: level 6
invalid_object(u)	
too_many_objects(utv)	
invalid_type_correspondence(u,ut)	
no_ability(s,i)	SEGMENTS: level 5
no_seg(u)	
bad_size(i)	
not_storable(u,slv)	
address_bounds(u,i)	
bad_length(bv)	CAPABILITIES: level 0

TABLE 4-4.1. EXCEPTION CONDITIONS OF VISIBLE FUNCTIONS, cont'd.

TABLE 4-4.2. EXAMPLES OF MONITORABLE EVENTS.

MONITORABLE "events"	"options" (EXAMPLES)
login	who, time, terminal_id or type
logout	who
create_process	
create_directory	subsets of directories
change_working_dir	who
procedure_calls	subsets of procedures
exception conditions	subsets of exceptions (see Table 4-4.1)

(Note: Essentially all monitoring may be local or global and may be used in either passive or active monitoring, subject to the desired security requirements.)

CHAPTER IV-5
DISTRIBUTED SYSTEM AND NETWORK CONSIDERATIONS

BACKGROUND

The foregoing sections have dealt with environments within the scope of a single operating system, either as a part of the operating system or as applications using the operating system. This section examines the implications of distributed realizations of PSOS and its inclusion into a network.

Generally speaking, a multiprocessing system is a system in which different processors communicate through a common memory. The communication can be of high bandwidth (e.g., when the operating system itself resides in memory shared by multiple processors). The communication among processors sharing common memory may be read-write, as in typical multiprocessors (e.g., Multics), or read-only (e.g., as in SIFT). Processors are often essentially interchangeable, although that is not essential.

When the interconnections are external to the system, it is generally called a distributed system of processors, or a network of systems. These two terms are used almost interchangeably, and therefore no hard distinction is made here. However, the former often implies somewhat tighter coupling of control or higher bandwidth communication. Either may be homogeneous (i.e., a distributed system of identical processing units, or a network of suitably similar operating systems) or inhomogeneous. Intercommunication may be via explicit signalling channels, shared disk, message passing, or external communication lines.

It is highly desirable that the differences among the component systems in a network be hidden from the network user interfaces, wherever this is feasible from an efficiency point of view. Furthermore, advantage should be taken of the similarities, wherever possible. In most existing networks, detailed knowledge is required of any system to be used, including its login protocols, command language syntax, error messages, etc. The task of abstracting essentially irrelevant detail out of the user interface is the task of the operating system and of its language translators. However, most operating systems are highly nonstandard, and enforce constraints that make inter-system compatibility difficult. Thus it is desired to have a network operating system interface that serves the unifying purpose of an operating system interface. Although this is reasonable in a homogeneous network, it presents many problems in an arbitrary nonhomogeneous network.

Common to systems and networks are various characteristics for evaluation. These include

- Ease of use
- Understandability
- Reliability
- Security

Functionality
Performance (e.g., responsiveness, throughput, etc.)
Efficiency and overall cost effectiveness
Flexibility, including evolvability and expandability
Certifiability of critical properties

To varying degrees, these characteristics are vital to successful operation. Unfortunately, some of these characteristics may be mutually antagonistic, especially when interconnecting systems not designed to accommodate networking. On the other hand, a reasonable balance can be achieved with careful design and implementation of both the component systems and the network, particularly if an overall unifying approach is followed, such as that of the SRI methodology for design, implementation, and verification.

RESIDENCE OF DATA AND PROCEDURE

In both distributed systems and networks, it is important to characterize precisely the flow of control and the flow of data. In general, it is possible (from a given processor in a distributed system, or from a given system in a network) to request that an operation that is defined locally or remotely be performed locally or remotely on data that is defined locally or remotely with results being delivered locally or remotely. In fact, since the source of the request can itself be either local or remote, there are six location variables to be specified, as follows:

(1) OPERATION SOURCE (command initiation)
(2) OPERATION DEFINITION
(3) OPERATION SITE
(4) DATA DEFINITION
(5) DATA SITE
(6) DATA DESTINATION (command completion)

Each of these location variables may be local or remote. Thus there are sixty-four pure cases, with multitudinous hybrid cases involving operations and data distributed among multiple systems. Many of the 64 pure cases are common (all are meaningful), covering the standard notions of program sharing, data sharing, and load sharing in distributed systems and in networks, as well as message store-and-forward applications.

PROBLEMS OF HOMOGENEITY AND HETEROGENEITY

In designing and implementing networks, numerous difficulties have been encountered. Some of these are intrinsic even in completely homogeneous structures, while others arise only because of the differences among the individual components.

In order to improve our understanding of distributed systems and networks, it thus seems desirable first to isolate those issues of the homogeneous category -- arising even in homogeneity -- and then to isolate those of the nonhomogeneous category. Issues arising in

homogeneous structures include addressing, protection, data distribution, and intercommunication. Issues arising in heterogeneous structures include data representations (formatting, word-lengths, etc.) and language uniformity.

The resulting isolation of issues contributes significantly to a unified and structured design view. This view in turn can greatly enhance the fulfillment of many of the characteristics noted above, and would be widely applicable over a wide range of designs for distributed systems and networks. For example, in this way we could conceive of a virtual interface including a homogeneous structure, and then a virtual interface including a heterogeneous structure of homogeneous substructures. Thus the approach discussed here divides issues of network design into two classes. When these issues are suitably isolated and understood, each of these classes will contribute important and useful results.

HOMOGENEOUS DISTRIBUTED SYSTEM REALIZATIONS OF PSOS

The notion of distributed realizations of PSOS are considered next. The basic hierarchical design of PSOS (see Table 0.1) appears to represent a simple general-purpose nondistributed computer system. Nevertheless, PSOS lends itself readily to distributed implementations precisely because of its hierarchically structured collections of cleanly encapsulated type managers.

Table 4.5.1 illustrates how the PSOS system concept can lead to distributed versions of the system, first in the design, and then in the implementation. The table shows various levels at which potential intercommunications and protocols may meaningfully exist, and some of the forms the distribution can take. In general, virtual distribution and explicit distribution may both be accomplished by the introduction of new levels into the design. Virtual distribution may also be accomplished invisibly within an existing level of the design.

As noted above, each level in PSOS acts as the type manager for some object type. This conceptual distribution of type managers suggests one mode of distributed implementation for PSOS, in which the implementations of the various type managers can be distributed separately. The complete encapsulation of typed objects for any particular type also suggests a mode of distribution, in which a type manager may itself have a distributed implementation, coordinated by communication at that level or a lower level. These two modes are discussed next, at each of several levels of interest, beginning at the lowest level.

The creation of capabilities is centralized conceptually in the lowest level of the PSOS basic design. It is thus used by each of the higher levels. Nevertheless, capability creation can be distributed with the aid of a simple system convention (see below).

The interpretation of capabilities is intentionally already distributed in the design. Each type manager is responsible for interpreting the capabilities for all objects of its type. For

example, the segment manager -- implemented largely in hardware -- knows about all of the segments in the system. It could, however, be distributed so that each of several segment managers knows only about its local segments -- using either virtual distribution or explicit distribution. On the other hand, a collection of distributed PSOSes could be conceived in which the distribution is done at the user object level and in which the segment level of each component system is left intact.

In any distributed version of PSOS, universal uniqueness of capabilities must be maintained. In a homogeneous distributed system of PSOS systems, global uniqueness is trivial to achieve, if each unique identifier contains a SITE_UID and a LOCAL_UID. (The LOCAL_UID can be implicit in locally interpretable capabilities, or could be explicit.) Since SITE_UIDs can be fixed forever at site creation, it is relatively easy to ensure that they are unique. Thus it is easy to ensure that the distributed-system-wide capabilities are unique. Consequently a central mechanism for generating capabilities is not needed, and each component can do its own creation. Recognition of foreign capabilities at this level is thus easy, based on the SITE_UID. Note that in such a scheme the SITE_UIDs must also be trusted and thus nonforgeable.

The most commonly executed instructions in any PSOS implementation are expected to be those that read or write at some location in a segment designated by a capability and an offset. Virtual distribution can be achieved by having a distributed segment manager that knows about nonlocal segments and that can redirect the access to the appropriate local segment manager (e.g., basing its action solely on the SITE_UID). Explicit redirection can also be handled at the user object level by redirecting a symbolic but globally meaningful segment name to the appropriate local segment manager. (Permitting explicit requests for foreign segments at the level of the segment manager is probably a bad idea, although it could be implemented.)

Given a distributed segment manager, the abstract-type manager could be extended to take into account multisegment object implementations in which the various segments forming an abstract object are themselves distributed. Note that this effect can also be achieved by extensions to the user object level.

The distinction between the abstract type manager and the particular type managers (e.g., the directory manager) is important. Each type manager may use virtual and/or explicit distribution of its implementation, depending on its needs. Each type manager is responsible for encapsulating the implementation of objects of its type, although the abstract-type manager facilitates the isolation of the implementation capabilities from the abstract object capability.

The directory manager lends itself nicely to a virtual directory system in which there is distributed implementation of various directory structures. It would also be easy to provide redundant (e.g., distributed) directory information. However,

explicit separate directory roots are better accomplished at the user object manager level. At the user object manager level, both virtual and explicit distribution of object creation and deletion is possible, with the options of distributed directories and distributed segment managers noted above.

At the user process level, both virtual interprocess communication (across distributed system boundaries) and explicit signalling are possible. At the user input-output level, both virtual input-output and explicit foreign names are possible. Similarly, both options exist for user environments (virtual name distribution and explicit foreign names) and for the user request interpreter (virtual command distribution or explicit remote logins).

HOMOGENEOUS NETWORKS OF PSOS

Consider a homogeneous network of PSOS systems. There are various strategies for using the network. For example, there may be

* explicit logins, as in the current ARPANET practice, e.g., using TELNET and FILE_TRANSFER_PROTOCOL. Here detailed knowledge is required of each system used, and long sequences of commands must typically be executed, such as

```
(on sys1:)
ftp sys2
login username password account
send local_name -to remote_name

get name1 -from name2
logout
quit
```

* implicit logins, in which a virtual interface is layered on top of the explicit logins, hiding them, but retaining the inefficiency. That is, each time a command is executed, a sequence such as the above must be executed, with the inefficiency of repeated logins.

* a virtual interface that supports efficient direct access.

There are at least three options with respect to naming of location variables.

* an all-inclusive unified network-wide naming scheme in which the network is able to keep track of all named objects and their locations;

* an extended naming scheme that includes the system identification as an explicit part of each location variable, e.g.,

```
copy sys1>local_name sys2>remote_name
```

with system-maintained authorization information to provide the default username, password, and account information;

* a natural naming scheme that presupposes an explicit or default declaration for each location variable, e.g.,

```
attach data_destination sys2 [authorization]
copy old_name new_name
```

where the default is the local system.

If the attachment implies opening a connection (e.g., the login within the ftp above), or if there is an explicit connection, then sequences of network operations may be made efficient in a virtual interface without requiring repeated logins. In this way, a program or sequence of commands can execute in any of the above 64 cases without modification, requiring only the appropriate attachments.

FAULT TOLERANCE AND RELIABILITY

Relatively little has been said in this report about the attainment of high reliability and high availability in an implementation of PSOS. Certain aspects result from good engineering practice in the hardware. However, the design must include appropriate hardware and software measures in order for a reliable and fault-tolerant system to be attained. Fortunately the hierarchical design of PSOS lends itself naturally to certain enhancements that can aid in attaining these goals. Some of these are particularly aided by a distributed realization, while others fit in nicely with a network. A brief outline of some of these that result from virtualization of reliability and fault-tolerance mechanisms within the various abstract type managers is given in Table 4.5.2.

CONCLUSIONS

Combinations of these options are possible within a particular network. It is significant to note that the PSOS specifications permit great flexibility as to how distributed implementations and networking can be handled. There could be a simple network-operating system interface layered on top, or there could be different interfaces incorporated into various levels, as appropriate. The latter approach is intuitively more appealing from an efficiency point of view, while the former may be more attractive with respect to ease of use. In any event, PSOS provides an excellent basis for further research in distributed systems and networks. In addition, the design of a network in which homogeneous and nonhomogeneous aspects are cleanly decoupled at different hierarchical levels would be a significant step toward a unified approach to networks.

Table 4.5.1
Levels of Potential Intercommunications and Protocols

user request interpreter (virtual command distribution; explicit remote logins)
user environments (virtual name distibution; explicit foreign names)
user input-output (virtual i-o; foreign names)
user processes (virtual ipc; explicit signalling)
user objects (virtual; explicit remote creation)
directories (virtual directory system; explicit roots)
abstract types (virtual distributed implementation; explicit distribution of multisegment objects)
segmentation (virtual and explicit distribution based on SITE_UID)
primitive input/output (controller signalling)
capabilities (creation distributed; no dynamic protocol)

Table 4.5.2

Some Examples of Virtualization of Fault-Tolerance in a Hierarchical System Design.

Visible functions	Hidden functions	Fault-tolerance mechanisms. [Protocols]. DISTRIBUTION IS INDICATED IN UPPER CASE.
Application	Network, nodes	Application rollback features and DATA DISTRIBUTION (redundant or not). [APPLICATION-TO-APPLICATION PROTOCOLS.]
Virtual network	Other nodes	DISTRIBUTED CONTROL OF COMMUNICATION,coding; REDUNDANT DISTRIBUTION OF DATA AND PROGRAMS [NODE-TO-NODE PROTOCOLS.]
Virtual system	Other subsystems	Compartmentalization, data security, system integrity; COOPERATING SUBSYSTEMS [SUBSYSTEM TO SUBSYSTEM PROTOCOLS.]
Virtual process	Process scheduling	REPLICATED PROCESSES; INDEPENDENT ALTERNATE PROCESSES; automatic rollback; process directories. [HIGH-LEVEL INTERPROCESS PROTOCOLS.]
Virtual i-o	Asynchrony Buffering	DISTRIBUTED I-O ARCHITECTURES, REMOTE CONTROLLERS; safe asynchrony; extensive handshaking and cross-checking. [I-O DEVICE PROTOCOLS.]
Virtual file system	Inaccessible directories, archiving	REPLICATION OF CRITICAL DATA, e.g., on on different media; FILE ARCHIVING AND ROLLBACK/RETRIEVAL; cross-checking. [FILE TRANSFER PROTOCOLS.]
Virtual memory	Storage addresses	Redundant physical address calculations; REPLICATION OF CRITICAL DATA. [INTERDEVICE PROTOCOLS.]
Virtual uniprocessing	Multiprogramming: dispatching	Reliable interrupt mechanisms; process isolation (e.g., domains of protection). [LOW-LEVEL INTERPROCESS PROTOCOLS.]
Multiprocessing	Processor coordination	Redundant interprocessor signalling [INTERPROCESSOR PROTOCOLS.]

Highest level is at the top of the table. Each level tends to hide some internal functionality from lower levels. Each level depends exclusively on lower levels for its implementation.

PART V
PERSPECTIVE

CHAPTER V-1
SUMMARY OF RECENT PROGRESS IN RELATED SRI WORK

1. METHODOLOGY -- DESIGN, IMPLEMENTATION, AND VERIFICATION

Boyer and Moore have rigorously formalized a subset of the version of HDM described in Robinson and Levitt [75]. Their formalization (Boyer and Moore [78]) in their theory (see Boyer and Moore [79]) describes what it means to prove an implementation consistent with its specification. This formalization excludes syntactic typing and explicit quantification. Specifications are written in a LISP-like internal form (VSSL). The implementation language formalized -- now called CIF (Common Internal Form) -- is a simple sequential language resembling assembly language. It provides the conventional control primitives plus instructions for invoking the operations (OV-functions) of the lower module and for causing and fielding exceptions. An extension of the verification system exists for proving that a program (in CIF) is consistent with its specs (in VSSL) for upper and lower level modules and the mapping functions that relate the states (V-functions) of the two levels of specifications.

2. VERIFICATION CONDITION GENERATION

Boyer and Moore have recently developed a verification condition generator (VCG) for most of ANSI FORTRAN, while Elspas has developed one for most of Jovial J73. In addition there are VCGs for CIF and for the versions of Modula (which is translated into CIF) and Pascal that are compatible with HDM. (Modula and Pascal are being used for KSOS and SIFT, respectively -- see below.)

Recently we have succeeded in developing a meta-VCG described in Flon and Moriconi [79]. This program is provided with a grammar for a language L and an axiomatic definition of L, and automatically produces a VCG for L. This meta-VCG has now been used to generate a VCG for the HDM Pascal, and is expected to be used for other languages as well.

3. TECHNIQUES FOR HANDLING CONCURRENCY

The basic method for proving correctness properties of nondistributed "shared-variable" concurrent programs was introduced by Owicki and extended by Lamport. Their ideas have been extended to apply to temporal logic specifications of concurrent behavior. (See their contribution.) Recent work by

Lamport and Schwartz has shown that temporal logic can be used to specify both safety (partial correctness) and liveness (total correctness) properties of concurrent systems. They have applied their methods to the specification of the Alternating Bit Protocol.

We believe that it is now possible axiomatically to define a simple concurrent programming language using shared variables, and to use this definition to prove correctness properties of programs written in this language.

4. THEOREM PROVERS

The Boyer-Moore Theorem Prover has evolved throughout the last decade. It is well described in their book (Boyer and Moore [79]), and is not discussed further here. Rob Shostak has been studying the problem of developing efficient simplifiers for decidable domains, e.g., Presburger arithmetic.

5. ENGINEERING OF A DEVELOPMENT AND VERIFICATION ENVIRONMENT

Developing and maintaining verified designs and programs, especially large ones, is an evolutionary activity. Specifications, programs, and proofs are gradually creationed and frequently revised. Consequently developers and verifiers are continually faced with the problem of determining the effects of changes. Mark Moriconi (when at UTexas) developed a program called the Designer/Verifier's Assistant that assists in reasoning about the effects of changes. This is being extended and incorporated into the HDM/Pascal verification system.

6. MODELS FOR SYSTEM REQUIREMENTS AND DESIGN PROOFS

With regard to applications of the theory and supporting tools, we are currently involved in design proofs and code proofs of three systems in addition to PSOS (all using HDM), namely the fault-tolerant system SIFT (in Pascal), and the kernel-secure UNIX-compatible KSOS systems (one by Ford, one by Honeywell, as noted in the next section). Our goal in such applications is to verify correctness of executable code with respect to a model of system requirements.

MODELS FOR SECURITY

Feiertag has developed a formal model for multilevel security based on the model of Bell and LaPadula, but somewhat easier to relate to SPECIAL and the theorem prover (Feiertag [80]). The model states in effect that information can flow only to a higher level in a lattice, never lower and never laterally. Feiertag has also developed a set of tools for proving that a set of specs written in SPECIAL is consistent with that formal model. Formulas are generated whose proof is sufficent (and in most

cases necessary) for the model to be satisfied. Those formulas that cannot be proved trivially are given to the Boyer-Moore theorem prover. (The tools are powerful enough to handle the entire KSOS kernel.)

MODELS FOR FAULT TOLERANCE

We have developed a Markov model that characterizes the conditions under which a multicomputer (e.g., SIFT) that employs (1) voting to detect and mask failures and (2) reconfiguration to eliminate faulty computers. This model can be instantiated with rates for anticipated component failures and rates for detection and reconfiguration as the basis for analyzing the reliability (e.g., mean time to failure) of the modeled system.

7. APPLICATIONS -- PROVING SYSTEMS CONSISTENT WITH MODELS

KSOS (Ford's KSOS/11 and Honeywell's KSOS/6)

KSOS (Kernelized Secure Operating System) is an operating system based on a security kernel responsible for the maintenance of a multilevel security policy. Its user interface is precisely the UNIX (TM) user interface (as long as a user lives within a single security partition).

KSOS exists in two versions, KSOS-11 (designed and implemented on a DEC 11/70 by Ford Aerospace) and KSOS-6 (designed and implemented on a Honeywell Level 6 machine by Honeywell).

On top of this kernel [which in the Ford version runs in 11/70 kernel mode] are successively (1) a collection of trusted processes that are permitted to violate the standard kernel user security in carefully controlled ways [in supervisor mode], (2) an emulator that creates the UNIX user interface [in supervisor mode], and (3) any desired UNIX-based software (including the C compiler) [in user mode].

Both designs are specified in SPECIAL and are being subjected to the Feiertag Multilevel Security specification prover. KSOS-11 (McCauley et al. [79]) has had its entire kernel interface run through the prover. The result of that effort is a large number of successful proofs that the kernel functions successfully satisfy the multilevel security model, together with some nonproofs that indicate various flaws in the design. Some of these flaws have been or are being fixed, while others will remain in the kernel as KNOWN potential low-bandwidth signalling paths. The trusted processes (which are privileged to violate the security of the kernel) are also being subjected to the same tools, although in this case the resulting nonproofs must be analyzed carefully to see if they actually imply flaws -- as many of the potential violations are actually masked by the trusted processses.

A few illustrative code proofs that Modula code is consistent with SPECIAL specifications have been carried out. Several of the limitations of the existing axiomatization of HDM have been surmounted [e.g., in dealing with sets, structures, existential quantification], but the there is no intent of carrying out extensive code proofs for KSOS.

The same approach is also being applied to KSOS-6, which is being implemented in UCLA-Pascal.

SIFT -- A Fault-Tolerant Avionics Computer

SIFT (Wensley et al. [78]) is an experimental, fault-tolerant, multi-processor computer intended to provide extremely reliable flight control for advanced air transports. As the central computer system in an aircraft, SIFT is intended to (1) handle around twenty computational tasks associated with modern computer aircraft and (2) to have a mean time to failure of at least a million years in the presence of hardware faults (i.e., a failure probability of 10 to the minus 10 per hour). The architecture of SIFT is a collection of Bendix BDX930 minicomputers interconnected by a bus system. SIFT is connected to the sensors and actuators of an aircraft by standard 1553 channels. The fault tolerance is implemented in software, hence the name SIFT (Software Implemented Fault Tolerance).

Reliability is achieved by replication and voting. The outputs of tasks (where a task is executed by 3 or more processors) are voted on, and discrepencies are processed by a special executive task. Repeated errors by a processor are treated as permanent faults, and the processor deemed to be faulty is logically disconnected.

The reliability analysis is accomplished relative to the Markov model in which state transitions correspond to fault occurrence, fault detection, and reconfiguration. The design proof for SIFT demonstrates the consistency of the model and the specifications of the SIFT operating system. Several models have been interposed between the Markov model and the operating system (specified in SPECIAL and implemented in Pascal) to decompose the proof into manageable size subproofs.

TACEXEC

SRI has designed and specified a family of secure, real-time systems, known as TACEXEC (Feiertag et al. [79b]). The needs of real-time applications were of particular concern. The hierarchical structure and the system design concept of TACEXEC are such that specifications may be readily configured for any desired member of the family. For example, a resulting system may or may not implement multilevel security, as desired. Similarly, it may or may not have a virtual memory. At present,

TACEXEC is a paper system only.

8. FUTURE PLANS

Work is also unfolding on a new specification language which is expected to overcome many of the deficiencies of SPECIAL, to incorporate some of the theoretical bases of algebraic specification languages, and to include the temporal logic. It is expected to be applicable to the specification and proof of concurrent systems, e.g., distributed systems and network protocols.

Further work is also anticipated on extending the development environment to include incremental evolution, the meta-VCG approach, and adding more features of more real programming languages (possibly including an extended subset of Ada!).

CHAPTER V-2
CONCLUSIONS

INTRODUCTION

This report has presented the design of PSOS (a Provably Secure Operating System), including formal specifications, and has considered several representative application environments. PSOS is a general-purpose computer system capable of meeting advanced security requirements. Among such requirements satisfied by the design are basic security, multilevel security, confinement of storage channels, and provability of the security of both the design and its implementations. The report also gives statements of the desired security properties that the specifications should meet, and outlines their proofs. In addition, it gives illustrative implementations and proofs of those implementations.

The conclusions given in this final chapter represent the opinions of the authors, based on their experience and on their work with PSOS. The main conclusion is that PSOS has the potential of meeting advanced security requirements significantly more dependably than any existing general-purpose system. Further, it appears that the design can be efficiently implemented. More generally, the methodology used here is thought to have great potential for the development of future computer systems. However, much work is still required to demonstrate and evaluate the usefulness of the methodology and of the design. The most effective demonstration would be the development of a useful prototype implementation, along with proofs of the correctness of that implementation. In addition, other issues concerning PSOS, e.g., relating to fault-tolerance, performance evaluation, and networking, are considered potentially rich areas of future research.

1. EVALUATION OF PSOS, ITS POTENTIAL IMPACT, AND ITS RELATION WITH OTHER WORK

This section considers an evaluation of the work described here and its potential impact on the computer field. It also provides some general comparisons of this work with related work. It treats, in order, the methodology, the operating system, the role of specifications, and the role of proving.

THE METHODOLOGY

In recent years, approaches to the development of computer systems have achieved varying degrees of success. The approach taken here attempts to unify the entire development process, to decouple design and implementation into stages, to enforce structure in the resulting system, to provide a formal representation for the design and a formal basis for implementation and proof. This approach is the only one known to the authors that considers the entire development process in a formal way and permits formal proofs at each stage in the

process. Even in the absence of proofs, this approach seems to greatly increase the understandability and precision with which a design can be expressed, and the reasonableness of such a design with respect to stated desired properties of the system. The methodology should have considerable utility in future large system developments requiring some sort of extremely reliable service (e.g., security, fault-tolerance). It should also have considerable utility where proofs of design and implementation properties are desired.

THE OPERATING SYSTEM

The design of PSOS is of interest for several reasons. PSOS has been designed from the outset with advanced security requirements in mind, in such a way that security properties of the operating system and its applications could be formally proved. It is not restricted to any particular security policy. It is the first general-purpose capability-based operating system that has been designed hierarchically throughout. It has been designed together with various security-specific applications, such as the secure object manager and the confined subsystem manager. It has been designed using the staged formal design methodology noted above. PSOS was the first such system to be formally specified, although MITRE, Honeywell, and Ford have subsequently specified security kernels. It has substantial potential for realizing advanced security requirements expected in the next decade.

The choice of tagged capabilities is seen to be very natural in PSOS for achieving a highly secure system, particularly with the use of limited-store capabilities to prevent the uncontrolled passage of capabilities. First, the use of tagged capabilities in the design permits a conceptually simple design and greatly simplifies the proofs of security, both for design properties and for implementation. In particular, the direct relationship between designators in the specification language and capabilities in the design is beneficial to the proofs. This puts a precise boundary on the protection mechanism within the operating system. Second, various problems customarily associated with capability-based systems have been resolved. For example, the use of limited-store capabilities introduced here greatly extends the usefulness of capabilities, without complicating their implementation. In so doing, it surmounts the common worry that capabilities might be intrinsically uncontrollable. Third, the complexity of implementation in hardware and software is seen to be competitive with conventional systems.

This report represents the design of a family of operating systems together with several important applications, with respect to both software and hardware. For example, the specifications can be simplified in various ways to produce subsets of PSOS as specified, e.g., omitting paging, or omitting user processes. Similarly, the specifications can be implemented in various ways, on widely differing hardware. Thus PSOS is not

limited to just a relatively small kernel of an operating system or to any particular hardware. The system could be the basis for a large general-purpose time-sharing and batch system, or for an efficient distributed system or network of minicomputers. Thus PSOS and the approach taken to develop it are relevant to a wide class of applications. This approach is both more general and more powerful than the approach of designing a kernel of an operating system with highly constrained functionality. It avoids the inordinate dependence on a higher-level policy embedded within a low-level mechanism. Note that in this kernel approach, it is usually necessary to have some security-critical code in trusted processes outside of the kernel. In such cases, the operation of the trusted processes must of course also be secure. The unified approach taken here seems to be superior to the kernel approach in the long run, leading to a general-purpose operating system capable of efficiently supporting arbitary security needs. (Note that PSOS itself can be viewed as a hierarchical collection of kernels, with each type manager encapsulating any policy that it may enforce.)

It should be noted that PSOS, or a subset of it, could serve as a secure kernel for a commercial operating system such as TOPS-20 or TENEX. The command set that can be built on top of PSOS is essentially arbitrary.

The closest comparable operating system effort in existence seems to be the Multics system. However, the existing Multics system is unprovable by contemporary techniques. Further, it is precisely describable only in terms of the particular implementation that currently exists. The proposal (which was not funded) to retrofit Multics with a multilevel security kernel might have produced a provable kernel. However, that effort would still have been basically a short-term solution to the multilevel security problem, whereas PSOS is a general solution to a much wider class of long-range problems.

SECURITY

Some of the desired security properties have been precisely stated. For PSOS, these are the Alteration Principle and the Detection Principle, which state that there is no unauthorized modification or acquisition of information. (It remains to axiomatize these properties sufficiently for purposes of automatic proof.) For the multilevel security subsystem, the desired property is that information cannot move downward to a lower security level. Because of the use of SPECIAL for specifications, the proofs of some of these properties become largely syntactic checks of the specifications.

THE ROLE OF SPECIFICATIONS

Formal specifications by themselves provide a significant advance in the state of the art of software system development. They provide a concise and precise functional statement of exactly what any external or internal interface is expected to

do. They enforce abstraction on the design that hides many details typically visible to the interfaces of a system, consequently simplifying implementation, debugging, system integration, and maintenance. They greatly enhance the understandability of a design. They also make possible the intuitive verification of certain desired properties that the design should satisfy. As noted in the previous paragraph, they support proof of specification properties.

THE ROLE OF PROVING

The ability to prove properties about a design (as represented by a set of specifications) before that design is ever implemented should have a significant impact on the computer industry. Nevertheless, no system can justifiably be thought to be secure unless appropriate properties of its implementation can also be proved. As seen from the KSOS work, proving that a specified design satisfies certain critical design properties is straightforward and largely automatable by tools. This will also apply to the PSOS security properties. In addition, proving the consistency of implementation with respect to the specifications is now becoming a realistic task, especially with the emergence of recent theoretical advances and advances in suitable on-line tools. Furthermore, the potential role of a language like ILPL is considerable, either directly or combined with a preprocessor translating to a desired programming language. In the latter of these approaches, programs could be written, proved, and maintained in the simple proof-oriented language (ILPL), although their documentation versions could be either in that language or in a more widely used language (e.g., Euclid or Modula). An attractive alternative is to take a modern language like Euclid or Modula, to extend it slightly in order to accommodate the methodology, and to write programs directly in that language.

2. RECOMMENDATIONS FOR FUTURE WORK

The work reported here demonstrates the feasibility of designing an operating system to meet advanced security requirements, and of proving properties about that design. It also indicates that the system can be efficiently implemented on hardware that seems realistically obtainable in the near future. However, much work remains to achieve such an implementation and to carry out the proofs of correctness of that implementation -- including proofs that the hardware and microcode correctly implement the PSOS specifications (e.g., those for the capability module).

Many topics related to PSOS and to the methodology seem worthy of future study. A summary of the most important of these topics is given in Table 5-2.1. These topics include issues concerning the language used in writing programs to realize PSOS and its applications, system issues involved in choosing appropriate hardware and suitable software configurations for a prototype system development, further tool development to support proofs, and further proving efforts. In addition, there are

significant issues relating to fault-tolerance, networking, and performance issues, and to the support of the development of incrementally changing systems and families of related systems. These additional issues are important for future enhancements of PSOS, but are not essential to achieving an early and useful prototype of PSOS.

Preliminary consideration has been given to finding hardware suitable for the development of a prototype, and to selecting a viable subset of PSOS as specified here. A summary of these considerations is as follows.

HARDWARE (AND/OR MICROCODE) SHOULD SUPPORT (AT LEAST)
* tagging of capabilities in memory
* support for limited-store capabilities
* some support for segment windows
* hardware memory mapping of segment addresses
* small associative memory for segment accesses
* support for the capability instructions
* general address registers and stack register
* level register
* interval timer and clock
* input-output using capabilities

SOFTWARE SHOULD SUPPORT (AT LEAST)
* one processor (although more are desirable)
* segmentation (but not necessarily paging)
* extended types
* directories
* input-output
* processes
* some representative applications

Various existing machines could be used for the development of a prototype, with reasonable modification. Alterable microcode is particularly helpful. The complete operating system as specified is conservatively estimated to be at most 8,000 lines of source code (e.g., ILPL). A useful subset of PSOS would be smaller.

Some further development of the methodology and supporting tools has been carried on in other projects in which the authors of this report and other members of the SRI Computer Science Lab are also involved (see Chapter V-1). This includes work on the development of families of related systems, with applications to message processing and to hardware architecture; real-time operating systems, including highly fault-tolerant systems; proofs of small, special-purpose, secure operating systems or kernels thereof; and on-line tools to support proofs -- of both design properties and implementation correctness. Much of this work is expected to have a significant impact on further PSOS developments.

TABLE 5-2.1
TOPICS FOR FUTURE STUDY RELATING TO PSOS AND THE METHODOLOGY

Research to further develop the methodology:
- Formalization of more of HDM and of the SPECIAL semantics.
- Development of a new specification language better suited to proof.
- Further work on modeling of specification properties other than multilevel security, e.g., PSOS capabilities.
- Further axiomatization of Euclid/Modula/Ada in the Boyer-Moore theory.
- Further work on proofs of implementation correctness:
  - Initialization correctness.
  - Process implementation and register set visibility details.
  - Parallelism issues to enhance proof below system process level.
  - Abstract machine interpreters. Compiler correctness.
  - Use of second-order logic in proof methods.

Further tool development to support specification proofs other than for MLS, e.g., PSOS capability properties.

Further tool development to support implementation and its proofs:
- ILPL preprocessors for various commonly used languages?
- Extend the power of the META-VCG environment.
- Mapping function expander.
- Interactive "proof supervisor". Definition expander.
- Logical simplifiers. Assertion generation for types.
- Other on-line consistency checkers; proof checker.

Implementation language issues: Choice of implementation languages contributing to provability and security, compatible with the methodology; directly compilable into executable code; direct use of EUCLID, Modula, Ada, GYPSY, etc., or use of ILPL with a preprocessor to the desired source language.

Abstract implementations for the system design, and for the
- Secure Object Manager,
- Confined Environment Manager,
- Secure Data Management System.

Realization of the system and applications on actual hardware:
- Choice of software configuration (including support for multiprocessing and for the general-purpose interface) with implementation sufficiently efficient to provide credibility.
- Choice of hardware, prototype implementation, and proofs as desired.

Mechanical proofs of specification properties for PSOS and applications.

Mechanical proofs of implementation for the above design portions.

Network and distributed system issues, and fault-tolerance issues:
- Design of a fault-tolerant provably secure distributed PSOS.
- Monitoring of security in such a system.
- Design to avoid crash-induced security violations:
  - Detection of and tolerance to security-related faults.
  - Proofs of fault-isolation and recovery properties, a la SIFT.

Performance issues, including:
- Extending the methodology to handle analysis of performance.
- Simulation of portions of the software and/or hardware.
- Measures (e.g., information-theoretic) of security, confinement, denial of service, fault-tolerance.

Extended utility issues, including portability of specs, support for specifications and implementation of related families of systems and their incremental evolution.

PART VI
REFERENCES

Ambler [76a] A. L. Ambler, D. I. Good and W. F. Burger, Report on the language GYPSY, University of Texas at Austin, Department of Computer Sciences (June 1976).

Ambler [76b] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch and R. E. Wells, GYPSY: A language for specification and implementation of verifiable programs, University of Texas at Austin, Department of Computer Sciences (August 1976). (Published in the proceedings of a DoD Conference on Verification and Validation held 3-5 August 1976, Syracuse, N. Y.)

Astrahan et al. [76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaren, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, IBM Research Memo RJ 1738 (25356), 27 February 1976 (San Jose).

Bell and La Padula[74] D. E. Bell and L. J. La Padula, Secure computer systems: Mathematical foundations and model, Vols. I, II, III, MITRE Corp., Bedford, MA (November 1973 - June 1974).

Bisbey and Popek[74] R. L. Bisbey II and G. J. Popek, Encapsulation: An approach to operating system security, Proc. ACM Annual Conf., pp. 666- 675 (1974).

Boyce and Chamberlin [74] R. F. Boyce and D. D. Chamberlin, A structured english query language, Proc. ACM SIGFIDET, Ann Arbor, Michigan, May 1974.

Boyer and Moore [75] R. S. Boyer and J S. Moore, Proving theorems about LISP functions, J. ACM 22, no. 1, pp. 129-144 (January 1965).

Boyer and Moore [78] R. S. Boyer and J S. Moore, "A Formal Semantics for the SRI Hierarchical Development Methodology", SRI Computer Science Lab Report, (November 1978).

Boyer and Moore [79] R. S. Boyer and J S. Moore, "A Computational Logic", Academic Press (1979).

Burke[74] E. L. Burke, Synthesis of a software security system, Proc. ACM Annual Conf., pp. 648-50 (November 1974).

Dahl et al.[70] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, The SIMULA 67 common base language, Publication S-22, Norwegian Computing Center, Oslo (1970).

Chamberlin et al. [75] D. D. Chamberlin, J. N. Gray, and I. L. Traiger, Views, authorization, and locking in a relational data base system, Proc. National Computing Conference, pp. 425-430, 1975.

Chamberlin [76] D. D. Chamberlin, Relational data-base management systems, ACM Computing Surveys, vol. 8, no. 1, pp. 43-66, March 1976.

Codd [70] E. F. Codd, A relational model of data for large shared data banks, Comm. ACM, vol. 13, no. 6, pp. 377-387, June 1970.

Codd [75] E. F. Codd, Implementation of relational data base management systems, Panel Discussion from the 1975 NCC, Bulletin of ACM SIGMOD, vol. 7, nos. 3-4, pp. 3-22, 1975.

Dennis and Van Horn[66] J. B. Dennis, and E. C. Van Horn, Programming semantics for multiprogrammed computations, Comm. ACM 9, pp. 143-155 (March 1966).

Dijkstra [68] E. W. Dijkstra, Complexity controlled by hierarchical ordering of function and variability, in Report on a Conference on Software Engineering (Randell and Naur, eds.), NATO (1968).

Dijkstra[72] E. W. Dijkstra, Notes on structured programming, in Structured Programming (O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare), Academic Press, Academic Press, N. Y., pp.1-82 (1972).

Dijkstra [75] E. W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs, Comm. ACM 18, pp. 453 - 457, August 1975.

Fabry[74] R. S. Fabry, Capability-based addressing, Comm. ACM 17, pp. 403-412 (July 1974).

Feiertag [78] R. J. Feiertag, "A Formal Technique for Designing Secure Communications Systems", National Telecommunications Conference, Birmingham, AL (December 1978).

Feiertag et al. [79] R. Feiertag, K. Levitt, P. Melliar-Smith, "Tactical Executive (TACEXEC): A Real-Time Secure Operating System for Tactical Applications", Final Report, SRI Project 5545 (July 1979).

Feiertag and Neumann [79] R.J. Feiertag and P.G. Neumann, "The Foundations of a Provably Secure Operating System (PSOS)", NCC '79, AFIPS Conf. Proc., New York, NY, pp. 329-334 (June 1979).

Feiertag [80] R. J. Feiertag, "A Technique for Proving Specifications are Multilevel Secure", SRI Computer Science Lab Report CSL-109, 10 January 1980.

Flon and Moriconi [79] L. Flon, M. S. Moriconi, "Automatic Generation of Verification Condition Generators -- An Experiment in Metaprogramming" (November 1979).

Floyd[67] R. W. Floyd, Assigning meaning to programs,

Mathematical Aspects of Computer Science, vol. 19 (J. T. Schwartz, ed.), American Mathematical Society, Providence, RI, pp. 19-32 (1968).

Goldberg[74] R. P. Goldberg, A survey of virtual machine research, IEEE Computer, pp. 34-45 (June 1974).

Goldberg et al. [79] J. Goldberg, W. Kautz, L. Lamport, P. Neumann, "Formal Techniques for Fault Tolerance in Distributed Data Processing (DDP)", Final Report, SRI Project 7242 (April 1979).

Good[75] D. I. Good, Provable programming, Proceedings 1975 International Conference on Reliable Software, Los Angeles, pp. 411-419 (April 1975).

Graham and Denning[72] G. S. Graham and P. J. Denning, Protection--principles and practice, Proc. AFIPS SJCC 40, pp. 417-429 (1972).

Griffiths and Wade [76] P. P. Griffiths and B. W. Wade, An authorization mechanism for a relational data base system, IBM Research Memo RJ 1721 (25154), 11 February 1976 (San Jose).

Guttag et al. [76] J. Guttag, E. Horowitz, and D. Musser. The Design of Data Structure Specifications, Proc. Second International Conference on Software Engineering, San Francisco, California (13-15 October 1976).

Held et al. [75] G. Held, M. Stonebraker, and E. Wong, INGRES-- A relational data base system, loc. cit., pp. 409-416.

Hoare[72a] C. A. R. Hoare, Proof of correctness of data representations, ACTA Informatica 1, pp. 271-281 (1972).

Hoare[72b] C. A. R. Hoare, Proof of a structured program: the sieve of Eratosthenes, Computer Journal, 15, pp. 321-325, (Nov. 1972).

Hoare and Wirth [73] C. A. R. Hoare and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Informatica 2, pp. 335-355 (1973).

Ichbiah et al.[74] J. D. Ichbiah, J. P. Rissen, and J. C. Heliard, The two-level approach to data independent programming in the LIS system implementation language, in Machine Oriented Higher Level Languages (v. d. Poel and Maarsen, eds.), North-Holland Publishing Company, pp. 161-174 (1974).

Lamport [78a] L. Lamport, "The Specification and Proof of Correctness of Interactive Programs", Proc. of the Int'l Conf. on Mathematical Studies of Information Processing, Koyoto, Japan (August 1978).

Lamport [78b] L. Lamport, "The Implementation of Reliable

Distributed Multiprocess Systems", Computer Networks 2 (1978).

Lamport [79] L. Lamport, "A New Approach to Proving the Correctness of Multiprocess Programs", ACM Trans. on Progr. Lang. and Systems, Vol 1, No 1 (July 1979).

Lamport [??] L. Lamport, "How to Make a Multiprocessor Computer Which Correctly Executes Multiprocess Programs", IEEE Trans. on Computers (Jan? 1980).

Lampson[69] B. W. Lampson, Dynamic protection structures, Proc. AFIPS 1969 FJCC 35, AFIPS Press, Montvale, N.J. pp. 27-38 (1969).

Lampson[73] B. W. Lampson, A note on the confinement problem, Comm. ACM 16, pp. 613-614 (October 1973).

Levitt [78] K. N. Levitt, "A Panel Session--Formal methods in programming--When will they be practical?", NCC 1978, AFIPS Press Vol. 47 (1978).

Levitt et al. [80] K. Levitt, L. Robinson, B. Silverberg, The HDM Handbook, Volume III: A Detailed Example in the Use of HDM", SRI Project 4828 (June 1979)

Lipner[74] S. B. Lipner, A minicomputer security control system, COMPCON. pp. 26-28 (1974).

Lipner [75] S. B. Lipner, A comment on the confinement problem, Proc. Fifth Symposium on Operating Systems Principles, ACM SIGOPS Review, vol. 9, no. 5, pp. 192 - 196 (19-21 November 1975).

Liskov and Zilles[74] B. Liskov and S. Zilles, Programming with abstract data types, SIGPLAN Notices 9, 4, pp. 50-59 (April 1974).

Liskov and Zilles[75] B. Liskov and S. Zilles, Specification techniques for data abstraction, IEEE Trans. Software Engineering, SE-1, pp. 7-19 (March 1975).

McCarthy [60] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Comm. ACM 3 (April 1960).

Millen [75] J. K. Millen, Security kernel validation in practice, CACM vol. 19 no. 5, pp. 243-250 (May 1976).

Moore [76] J S. Moore, The INTERLISP virtual machine specification, Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California (September 1976).

Moriconi [79a] M. S. Moriconi, "Interactive Design and Verification: A Message Switching Network Example", Technical Report CSL-90, SRI International, (June 1979). (To appear in

Springer-Verlag Lecture Notes in Computer Science)

Moriconi [79b] M. S. Moriconi, "A Designer/Verifier's Assistant", published in IEEE Transactions on Software Engineering, Vol SE-5, No 4, pp 387-401 (July 1979).

Mylopoulos et al. [75] J. Mylopoulos, S. Schuster and D. Tsichritzis, A multi-level relational system, loc. cit., pp. 403-408, 1975.

Neumann et al.[72] P. G. Neumann, J. Goldberg, K. Levitt, and J. Wensley, A Study of Fault-Tolerance Computing, SRI (July 1973). AD 766 974.

Neumann et al.[74] P. G. Neumann, R. S. Fabry, K. N. Levitt, L. Robinson, and J. H. Wensley, On the design of a provably secure operating system, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 1161-175 (August 1974).

Neumann [74] P. G. Neumann, Toward a methodology for designing large systems and verifying their properties, 4. Jahrestagung, Gesellschaft fur Informatik, Berlin, October 9-12, 1974, in Lecture Notes in Computer Science, vol. 26, Springer Verlag, Berlin, pp. 52-67 (1974).

Neumann et al. [75] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena, A provably secure operating system, SRI Final Report, Project 2581 (13 June 1975).

Neumann et al. [76] P. G. Neumann, R. J. Feiertag, K. N. Levitt, L. Robinson, Software development and proofs of multi-level security, Second Int. Conf. on Software Engineering, San Francisco CA (13-15 October 1976).

Neumann et al [77], P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, L. Robinson, "A Provably Secure Operating System: The System, Its Applications, and Proofs", SRI Final Report, Project 4332 (11 February 1977).

Neumann [78] P. G. Neumann, "Computer System Security Evaluation", NCC '78, AFIPS Conf. Proc., Anaheim, CA, pp. 1087-1095 (June 1978).

Neumann [79] P. G. Neumann, "The Use of Formal Specifications in the Design, Implementation and Verification of Large Computer Systems", in Research Directions in Software Technology, P. Wegner (Ed.) pp 190-198, MIT Press (1979).

Neumann et al [80], P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, L. Robinson, "A Provably Secure Operating System: The System, Its Applications, and Proofs -- Second Edition", SRI Computer Science Laboratory Report CSL-116 (7 May 1980).

Organick[72] E. I. Organick, The Multics system: an examination

of its structure, MIT Press, Cambridge, MA (1972).

Parnas[72a] D. L. Parnas, A technique for software module specification with examples, Comm. ACM 15, pp. 330-336 (May 1972).

Parnas[72b] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Comm. ACM 15, pp. 1053-58 (December 1972).

Parnas[72c] D. L. Parnas, Some conclusions from an experiment in software engineering techniques, Proc. FJCC, pp. 325-329 (1972).

Parnas[72d] D. L. Parnas, Response to detected errors in well-structured programs, Technical Report, Department of Computer Science, Carnegie-Mellon University (July 1972).

Parnas[74] D. L. Parnas, On a buzzword: hierarchical structure, Proceedings IFIP 74, Stockholm, pp. 336-339 (August 1974).

Popek and Kline[74] G. J. Popek and C. Kline, The design of a verified protection system, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 183-196 (August 1974).

Price[73] W. R. Price, Implications of a virtual memory mechanism for implementing protection in a family of operating systems, Ph.D. thesis, Carnegie-Mellon University, Department of Computer Science (June 1973).

Redell and Fabry[74] D. D. Redell and R. S. Fabry, Selective revocation of capabilities, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 197-209 (August 1974).

Robinson and Levitt[75] L. Robinson and K. N. Levitt, Proof techniques for hierarchically structured programs, SRI Report (January 1975). To appear, Comm. ACM (March 1977).

Robinson et al. [75] L. Robinson, K. N. Levitt, P. G. Neumann, and A. K. Saxena, On attaining reliable software for a secure operating system, Proc. International Conf. on Reliable Software, SIGPLAN Notices, vol. 10 no. 6, pp. 267-284 (June 1975). A revised and extended version is being published under the title, "A Formal Methodology for the Design of Operating System Software," in R. T. Yeh (ed.), Current Trends in Programming Methodology, vol. 1, Prentice-Hall (1977).

Robinson [76], L. Robinson, Specification techniques, Proc. 13th Design Automation Conference, IEEE cat. 76-CH1098-3C, pp. 470 - 478 (28-30 June 1976).

Robinson [79] L. Robinson, "The HDM Handbook, Volume I: The Foundations of HDM", SRI Project 4828 (June 1979).

Roubine and Robinson[76] O. Roubine and L. Robinson, SPECIAL (SPECIfication and Assertion Language): reference manual, SRI memo (August 1976).

Saltzer[74] J. H. Saltzer, Ongoing research and development or information protection, ACM Operating Systems Review 8, pp. 8-24 (July 1974).

Schmid and Bernstein [75] H. A. Schmid and P. A. Bernstein, A multi-level architecture for relational data base systems, University of Toronto memo (1975).

Schorre[74] D. V. Schorre, Example of a module specification and implementation for automatic verification, Working Paper TM-5310, System Development Corporation, Santa Monica, California (May 1974)

Schroeder[72] M. D. Schroeder, Cooperation of mutually suspicious subsystems in a computer utility, Ph.D thesis, MIT (1972). MAC TR-104.

Schroeder [75] M. D. Schroeder, Engineering a security kernel for multics, Proc. Fifth Symposium on Operating Systems Principles, ACM SIGOPS Review, vol. 9 no. 5, pp. 25-32 (19-21 November 1975).

Silverberg et al. [80] B. Silverberg, L. Robinson, K. Levitt, "The HDM Handbook, Volume II: The Languages and Tools of HDM", SRI Project 4828 (June 1979).

Spitzen et al. [76] J. M. Spitzen, K. N. Levitt, and L. Robinson, An example of hierarchical design and proof, Technical Report CSL-30, Computer Science Laboratory, Stanford Research Institute, Menlo Park, California (also submitted for publication) (March 1976).

Spitzen and Wegbreit[75] J. M. Spitzen and B. Wegbreit, The verification and synthesis of data structures, ACTA Informatica, 4 (1975)

van Wijngaarten et al.[69] A. V. van Wijngaarten et al., Report on the algorithmic language Algol 68, Numerische Mathematik, pp. 79-218 (1969).

Wegbreit et al.[74] B. Wegbreit et al., ECL programmer's manual, Center for research in computing technology, Harvard University, Cambridge, Mass. (1974).

Wensley et al. [76] J. H. Wensley, M. W. Green, K. N. Levitt, R. E. Shostak, The Design, Analysis, and Verification of the SIFT Fault-Tolerant System, Second Int. Conf. on Software Engineering, San Francisco CA (13-15 October 1976).

Wensley et al. [78] J. H. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", IEEE Proceedings (October 1978).

Wirth and Hoare [66] N. Wirth and C. A. R. Hoare, A contribution to the development of Algol, Comm. ACM 9, 6, pp. 413-432 (June 1966).

Wulf[74] W. A. Wulf, ALPHARD: Toward a language to support structured programs, unpublished paper (April 1974).

Wulf et al.[74] W. A. Wulf et al., HYDRA: The kernel of a multiprocessor operating system, Comm. ACM 17, pp. 337-345 (July 1974).

APPENDIX A
SPECIAL -- a SPECIfication and Assertion Language

ABSTRACT

SPECIAL is a specification language developed in conjunction with the SRI methodology for design, implementation, and formal verification of software systems. Some of the language features are specific to the SRI methodology. Others, such as its non-procedural nature, concept of type, and aggregate data types, are generally useful for software specification and verification. A description of the features of the language is supplied, along with several examples of its use. The language has proved useful in the design of several large software systems, including an operating system. A discussion of the issues in the design of SPECIAL is presented, followed by a description of its features and some examples.

I. INTRODUCTION

SPECIAL (SPECIfication and Assertion Language) is a language for specifying software systems and was created with the following goals in mind:

* To specify systems in conjunction with a particular methodology for design, implementation, and proof of computer systems. The methodology, called the SRI Hierarchical Design Methodology, is described elsewhere [6, 7].
* To be powerful enough to specify a large class of systems, including systems containing both hardware and software, yet restrictive enough to permit syntactic checks of well-formedness of a specification.
* To be usable directly in the statement and formal proof of properties of software systems.
* To foster abstraction through the hiding of unnecessary data and algorithms of implementation.
* To specify systems to be implemented in any programming language.

SPECIAL describes properties of the data contained in a software system, but not properties concerning the time required to perform an operation.

In order to fulfill these goals, we have developed a language with the following features:

* An inner level that allows the writing of non-procedural expressions, using predicate calculus and set theory. This level is called the assertion level of the language. Expressions at the assertion level are used to describe the behavior of software systems and their abstract properties in a precise way.
* An outer level, called the specification level, that enables the description of a software system as a hierarchy of modules, in which each module is an abstract machine having a state and operations to change the state. The behavior of a module is described in terms of a formal specification, containing functions (1) that can be called by programs using the module. The state of a module is represented by the outputs of its V-functions (functions that return values). The state transformations of a module are called O-functions (functions that perform operations) of the module. Each transformation is described as a set of assertions relating the values of V-functions before the function call to the values of V-functions after the call. OV-functions are functions that both return values and transform the state. In addition to module specifications, assertions written in SPECIAL are used to specify relations among the states (V-function values) at different levels of the hierarchy. These relations are called mapping functions.
* Its own notion of type and facilities for describing abstract data types.
* Aggregate objects such as sets, vectors, and structures.
* An expression-based macro facility.
* Features to facilitate the characterization of objects without overconstraining them, and the detection of abnormal conditions.

Section II of this paper describes some background in the design of SPECIAL. Section III describes the assertion level. Section IV describes the specification level. Section V presents two complete examples of module specifications.

II. BACKGROUND AND GENERAL CONCEPTS

A. Background

Formal specification languages of some kind have always been necessary for research in programming language semantics and program verification. Recently there has been some emphasis on specification

---

(1) The term "function" hereafter refers to a V-, O-, or OV-function of a module and not to a mathematical function.

languages themselves -- as design tools for software systems. Parnas [4, 5] first suggsted the idea of designing a software system as a collection of formally specified modules. Parnas' language was more precise than the informal methods of software specification that preceded it, but its syntax and semantics were not formally stated. Recently there have been several efforts [1, 2] aimed at specifying the formal properties of data structures and other software systems. Yet oher specification languages have emphasized the theory of primitive recursive functions [10] and the formal English of a mathematics textbook [11] for describing properties of software.

The SRI Hierarchical Design Methodology suggests a particular (hierarchical) way of structuring these modules in order to improve the reliability of large software systems by:

* Formally stating all design decisions.
* Allowing a complex design to be structured so that it can easily be understood.
* Allowing proofs of formally stated properties of the design.
* Allowing a proof of syntactic consistency of the implementation with the design specifications.

Several small proofs have been completed (e.g. [6, 9]), and proofs for the design and implementation of a general-purpose operating system whose major design goal is security [3, 7], are now in progress.

SPECIAL was developed through numerous attempts to write module specifications in the style of Parnas ([4]) for many different kinds of software systems. This paper presents a self-consistent and working version of the language, although it may change as more experience is gained.

In the description of SPECIAL that follows, some of the examples have been over-simplified for ease of explanation. One feature of SPECIAL that is not discussed deals with the handling of parallelism. However, the full grammar appears in the Appendix, and a complete description of the language appears in the SPECIAL Reference Manual [8].

B. General Concepts

The problem of designing a good specification language is immense. There are perhaps as many factors as there are in designing a good programming language, and there is not much experience of others to draw on. The language should be able to characterize the objects of the system it is specifying (e.g., the variables, procedures, and data structures), and may employ objects of its own to facilitate that description -- thus the distinction between specification objects and implementation objects. Specification

objects are not computed, and may not be computable, in the implementation. The language must be well suited to the particular kinds of systems it hopes to specify (e.g., operating systems, data base systems), and at the same time must adhere to the constraints of the design methodology being used with the language. The language must allow as much machine checking of specifications as possible, and also be amenable to formal mathematical proof. The language must be powerful, but without a proliferation of specialized syntactic features to suit every possible need. The language must be conceptually elegant, but must have features that make specifications in the language easy to read and write.

We shall now outline some of the desirable properties of a specification language, with the intent of later motivating each of the features of SPECIAL according to the list.

* Mathematical basis: A specification language must have some features that relate directly to the mathematics on which the language is based. In the case of SPECIAL, the mathematics involved is that of set theory and first-order logic (including quantification), with integer and real arithmetic. SPECIAL allows the writing of arbitrary expressions in these domains. Other specification languages, (e.g., [1] and [10]) may not permit such generality.
* Powerful, concise expressions: A specification language should be able to state, in a straightforward way, properties shared by different objects. Sets, vectors, and structures (as well as their constructors) are supported by SPECIAL in order to assist the writing of these expressions.
* Well-suited to the specified systems: A specification language should be able to accomodate in a natural way the properties of the systems being specified. For example, in operating systems, it is desirable to be able to look at a machine word as being of different types under different circumstances. The concept of type in SPECIAL provides this facility (united types) without restricting the kind of machine checking that can take place.
* Formal statements: It is desirable in a specification language to be able to make as many formal statements about an object as possible. In SPECIAL one statement that can always be made about an object is its type (i.e., the set of values that it can assume). The macro facility in SPECIAL provides a formal substitution rule based on expressions. O-functions in SPECIAL are defined completely by the specification: every V-function value not mentioned as changing in the specification must stay the same.
* Characterizing an object: Although the above goal (formal statements) is sometimes desirable, some specification language objects must not be overconstrained for fear of

dictating a particular implementation (e.g., a particular ordering among elements of a vector). Thus, a specification language should be able to introduce new objects without uniquely defining them. The existential quantifier in SPECIAL allows this characterization, and we have introduced two other constraints (LET and SOME) that make this process more straightforward and readable.

* Abstraction and protection: One original goal of SPECIAL was to foster abstraction, i.e, the definition, maintenance, and protection of objects of an abstract data type [2]. Some specification languages (e.g., [1]) deal explicitly with the objects themselves, and write axioms about functions defined on the objects in order to specify the abstraction. In SPECIAL, an object is defined by the V-function values that take the name for the abstract object as an argument. The operations on an abstract object are defined by the O-functions that take the name for the abstract object as an argument. SPECIAL provides a facility for defining protected names (called designators) for these abstract objects, such that the creation and modification of these names is limited.

* Support a methodology: It is extremely useful to couple a specification language with a methodology for designing, structuring, and implementing systems. Not only does it allow the language to help constrain the systems to conform to the methodology (a desirable trait), but it helps restrict the possible alternatives in specification language design, which are many. This has characterized the relationship of SPECIAL to the SRI Hierarchical Design Methodology.

* Support readability and writability: In certain cases there is no possible justification for the inclusion of a feature in a specification language other than that the feature makes the specifications either more readable or writable. These features come under the general category of shorthands and mnemonics. Excessive length has been a problem with this specification technique in the past, so shorthands, such as macros and global declarations, are welcome for that reason. These shorthands also make it possible for the user to establish mnemonic names for macros and variables that are used globally. Concerning readability, other specification methods may produce more concise specifications (e.g., [1]); but we believe that they are more difficult to understand than those written in SPECIAL, especially when trying to implement a complex system based on them or when trying to ascertain whether or not they conform to the intentions of the specification writer.

* Machine checking: Besides the usual grammatical checking, it is also desirable to be able to check properties of variables such as their types, declarations, and bindings. Thus, SPECIAL provides mechanisms for separating the declaration of

a variable from its use, and scope rules that limit naming ambiguities (see Section IV-B). Machine checking of these rules is easily done. In fact we have implemented several on-line tools at SRI to perform syntactic checking. The use of such tools has reduced the frequency of errors (including logical errors) in writing specifications, by immediately flagging syntactically meaningless statements.

* Verification: The specifications in SPECIAL have already been used in proofs of correctness (by hand) of small hierarchical systems [6,9]. A semi-automatic verification system based on SPECIAL as an assertion language is currently in the planning stages. All of the constructs at the assertion level of SPECIAL can be formally stated. The constructs at the specification level of SPECIAL are in the process of being defined, as part of a formal description of the SRI Hierarchical Design Methodology.

The next two sections should be read with the preceding criteria in mind.

III. THE ASSERTION LEVEL

A. Introduction

Assertions (or predicates) in SPECIAL are used to describe properties of systems, e.g., state transformations, error conditions, invariant properties, mapping functions, and conditions that must be true at a particular time in a program's execution. This section describes the objects of the assertion level, and then describes the various operators of the language.

B. Expressions

The primitive construct at the assertion level is an expression, defined as either 1) a constant or a variable, or 2) an ordered pair consisting of an operator and a sequence of expressions denoting its operands. A constant represents a single value, while a variable may represent several different values. Operators may be predefined (e.g., + or NOT) or user-defined (e.g., functions or macros defined by the user). The constants, variables, and operators of SPECIAL are called the objects of the language.

C. Types

Every object and expression in SPECIAL has a type. A type can be thought of as a set of values (or constants in the assertion language). With the exception of UNDEFINED as explained below, the set of constants in SPECIAL is partitioned. The type of a constant is of course the partition to which it belongs. For a variable, its type is the set of values that it can assume. For any operator, its operands as well as its result have a type. Thus, the type of an expression is either 1) if the expression is defined by a single

constant or variable, then the type of the constant or variable; or 2) if the expression is an operator and a sequence of operands, then the type of the result of its operator. The types of all objects are explicitly specified: the type of each constant and predefined operator is specified as part of the semantics of SPECIAL; the type of each variable and user-defined operator in a specification must be declared by the writer of the specification.

In SPECIAL, we have incorporated a more flexible attitude in restricting the types of the operands for the predefined operators, than has been done in most strongly typed programming languages (e.g., [12]). The only restrictions imposed are those that would prevent the writing of meaningless expressions. We have also supplied a mechanism in SPECIAL (the TYPECASE expression) for going back and forth between related types, so the term "coercion" has no meaning here.

The definition of a valid type in SPECIAL is related to the ways of naming sets of values in the language. Types in SPECIAL are of three varieties: primitive types, subtypes, and constructed types. A primitive type is a set of values that is disjoint from every other primitive type (with the exception of UNDEFINED) and such that no value in the set is defined in terms of any other values. There are three kinds of primitive types in SPECIAL:

* Those types whose values have well-known mathematical properties, called predefined types. The predefined types in SPECIAL are BOOLEAN, INTEGER, REAL, and CHAR. The usual kinds of operations apply to these types.

* Those types whose values are used to name objects of an abstract data type in specifications for abstract machines. These types are called designator types, and the objects of such types are called designators. The only operations that apply to designators are equality, inequality, and NEW(t), which returns a never-used designator of type t.

* Those types whose values are a set of symbolic constants, called scalar types. For example, the scalar type "primary color" could be the set {red, blue, yellow}. There is no added generality in including scalar types in SPECIAL, because the objects of a scalar type could always be represented by the integers. However, scalar types provide a useful mnemonic, and increase reliability by restricting the operations that can be performed on objects of scalar type to equality and inequality. Thus, for example, one could never have the expression "red + blue = yellow", which could occur in an integer representation.

The other types can be built using the primitive types as a basis. A subtype is an arbitrary subset of a given type. An example of a subtype is the even integers. A constructed type can be any one of the following:

* An aggregate type, i.e., a set or vector of objects of the same type.

* A united type, which is the union of several types.

* A structured type, which is the cartesian product of types. For example, a complex number can be thought of as a structure of "REAL X REAL".

Aggregate and structured types provide complex specification language objects, whose use often shortens the specifications. The types out of which a subtype or constructed type is made are called the constituent types of the subtype or constructed type. The operations on a subtype are the same as the operations on its constituent type. However, syntactic checking cannot be made to determine whether or not an expression of a given type is a member of a subtype of that type. The operations defined for constructed types will be discussed in later subsections.

In addition to the standard constants of all types, there is a constant, UNDEFINED (or ?), that is a member of any type, but different from any other constant of any type. The semantics of this constant are that an object whose value is UNDEFINED really has no value. For example, the value of a stack pointer of a stack that does not exist would be UNDEFINED.

D. Declarations

A variable or user-defined operation must be declared, or associated with a type, before using it. The syntax of a declaration, written in extended BNF (2), is

```
<declaration> ::= <type specification> <symbol>
                      {',' <symbol>}*
```

A type specification can be either an symbol that refers to a type or an explicit type specification. Here are some examples of explicit type specifications:

```
predefined:     INTEGER
                BOOLEAN

scalar:         {red, green, blue}

subtype:        {INTEGER x | x MOD 2 = 0}
                (the set of all integers x such that
```

---

(2) In extended BNF, <...> means that the enclosed symbol is a nonterminal of the grammar; [...] means that the enclosed construct is optional; {...}* means that the enclosed construct can occur 0 or more times, {...}+ means 1 or more times, and {...|...|...} means an alternative among the enclosed constructs. All special characters that are terminal symbols have been enclosed in single quotes (e.g., ';').

x MOD 2 is 0, i.e., the set of all even integers)

aggregate: SET_OF INTEGER
VECTOR_OF BOOLEAN

structured: STRUCT(REAL realpart, imagpart)

united: ONE_OF(INTEGER, VECTOR_OF CHAR)

A named type is a type that has been associated with a symbol for use in later declarations, e.g.,

```
STRUCT(REAL realpart, imagpart): complex_number
```

Then a new complex number xx can be declared as follows:

```
complex_number xx
```

A designator type must be a named type, so we write

```
stack_name: DESIGNATOR
```

to define the type, and later

```
stack_name st
```

to declare a variable of type "stack_name".

A function is declared with its function type, name, formal arguments, and result (in the case of a V-function). For example, a V-function declaration looks like this,

```
VFUN read(segment s, INTEGER i) -> machine_word w
```

where "segment" and "machine_word" are named types, s and i are the formal arguments, and w is the result. An O-function declaration looks like this,

```
OFUN write(segment s, INTEGER i, machine_word w)
```

The exact role of type specifications and declarations in a module specification is discussed in Section IV.

E. Simple Operations on Predefined Types

These include the logical, arithmetic, and relational operators, which apply to objects of types INTEGER, BOOLEAN, and REAL. The logical operators are AND, OR, NOT, and IMPLIES, all of which take BOOLEAN arguments and have BOOLEAN results. The basic arithmetic operators are +, - (unary or binary), *, and /. They operate on objects of types INTEGER or REAL, hereafter called numbers. Objects of both types can be arbitrarily intermixed as arguments to the arithmetic operators, with the following constraints on the results:

a binary operator having two INTEGER or two REAL arguments has an INTEGER or REAL result, respectively; and a binary operator having mixed arguments has a REAL result. The relational operators are =, ~=, >, >=, <, and <=, take numbers as arguments, and have a BOOLEAN result. For objects of type CHAR there are no distinguished operators; perhaps at a later time it will be advisable to define, as part of SPECIAL, a lexical ordering function, or a function that maps from a character to its integer code.

F. Operations on Sets, Vectors, and Structures

SPECIAL provides the conventional set operators -- union, intersection, set difference, elementhood, a subset predicate, and the number of elements in a set -- as UNION, INTER, DIFF, INSET, SUBSET, and CARDINALITY, respectively. The arguments to UNION can be sets of any type, and the type of the result is the set of the union of the constituent types of the arguments. For example, if the types of s1 and s2 are "SET_OF INTEGER" and "ONE_OF(SET_OF BOOLEAN, SET_OF CHAR)", respectively, then the type of "s1 UNION s2" is "ONE_OF(SET_OF ONE_OF(INTEGER, BOOLEAN), SET_OF ONE_OF(INTEGER, CHAR))". The arguments to INTER must be sets whose constituent types are not disjoint, and the type of the result is the set of the intersection of the constituent types of the arguments. For example, if the types of s1 and s2 are "SET_OF ONE_OF(INTEGER, BOOLEAN)" and "ONE_OF(SET_OF INTEGER, SET_OF CHAR)", respectively, then the type of "s1 INTER s2" is "SET_OF INTEGER". The arguments to DIFF must both be sets whose constituent types are not disjoint, and the result has the type of the first argument. The second argument to INSET must be a set, the first argument must be of a type that is not disjoint with the constituent type of the second argument, and the result is BOOLEAN. The arguments to SUBSET must be sets whose constituent types are not disjoint, and the result is BOOLEAN. CARDINALITY allows any set as an argument and returns an INTEGER result.

Constructors are expressions that define sets, vectors, and structures in terms of objects of their constituent types. An extensional constructor (used for all of the above types) requires the individual elements. An intensional constructor (used for sets and vectors only) supplies a necessary and sufficient property of the elements. The syntax of the extensional constructor for sets is as follows,

```
{1, 3, 5, 7}
```

The syntax of the intensional constructor for sets looks like this,

```
{INTEGER i | 0<i AND i<9 AND  i MOD 2 = 1}
```

This reads, "the set of all integers i such that 0 is less than i and i is less than 9 and i MOD 2 is equal to 1," or "the set of all odd integers on the open interval (0, 9)." Only the intensional constructor can be used to specify infinite sets, e.g.,

```
{INTEGER i | i MOD 2 = 1}
```

There are only two operations on vectors: length and extraction. The length operation is written LENGTH(v), where v is a vector expression. The result is of type INTEGER. The extraction operation is written v[i], where v is an expression of type "VECTOR_OF x" and i is an expression of type INTEGER, whose value is between 1 and LENGTH(v). The result is of type x. The extensional vector constructor is written as follows,

```
VECTOR(1, 3, 5, 7)
```

The intensional constructor for the same vector as above is written

```
VECTOR(FOR i FROM 1 TO 4:  2*i - 1)
```

Structures have an extractor. If a structured type has the following declaration,

```
STRUCT(REAL realpart, imagpart): complex_number,
```

then the two extractors for the type are "x.realpart" and "x.imagpart", where x is an object of type "complex_number." The extensional constructor for an object of type "complex_number" whose realpart is 1 and whose imagpart is 2 would be "<1,2>".

G. Quantified Expressions and Characterization Expressions

The syntax of a quantified expression is as follows:

```
{FORALL|EXISTS} <qualification> {';' <qualification>}* ':'
    <expression>
```

where

```
<qualification> ::= <symbol>
                    [{INSET <expression> | '|' <expression>}]
```

The purpose of a qualification is to optionally restrict the domain of a quantified expression. This adds no generality but improves readability. For example, we write

```
FORALL x INSET s : p(x)
```

to mean, "for all x in set s, p(x) is TRUE." This is equivalent to

```
FORALL x : x INSET s => p(x)
```

We write


```
FORALL x | q(x) : p(x)
```

to mean

```
FORALL x : q(x) => p(x)
```

Sometimes it is desirable to write expressions in which an object with a particular value or property is used repeatedly. To do this we have devised a construct called a characterization expression, in which a variable is first characterized and then used in an expression containing it. One of these, the LET expression, has a syntax as follows:

```
LET <qualification1> {';' <qualification1>}* IN <expression>
```

where

```
<qualification1> ::= <symbol>
                     {INSET <expression> | '|' <expression>}
```

In the LET expression the domain restriction is mandatory. As an example of a LET construct, suppose we have a table implemented as a function t, of one integer argument, where the key is in an even position and the value is stored in the subsequent position. We would like to calculate "f(x) + g(x)", where x is the value corresponding to the key y in the table. We write

```
LET x | EXISTS z : z >= 0 AND z MOD 2 = 0 AND t(z) = y
                   AND x = t(z+1)
    IN f(x) + g(x)
```

Note that the function t may not define a table (i.e., have more than one z whose key is y). Then the semantics is for the expression to be calculated for any x (and z) satisfying the predicate, but it is not known which ahead of time. If there is no x (or z), then the value of the expression is UNDEFINED.

A restricted form of the LET expression is called a SOME expression, and is written

```
SOME <qualification1>
```

An expression of the form

```
SOME x | p(x)
```

is equivalent to

```
LET x | p(x) IN x
```

The variables defined by quantified expressions and characterization expressions are called indicial variables.

B. Miscellaneous Operators

The remaining operations are the equality and inequality operations, the conditional expression, and the TYPECASE expression. Equality and inequality have already been discussed for numbers. For other types, equality and inequality will permit objects of any two non-disjoint types as arguments.

The conditional expression is of the form

```
IF b THEN e1 ELSE e2
```

where b is a BOOLEAN expression. If expressions e1 and e2 have types t1 and t2, respectively, then the type of the result is the union of t1 and t2.

Suppose we have a variable x of type "ONE_OF(t1,t2)", and wish the result of an expression to be f(x) if x is of type t1 and g(x) if x is of type t2. The most obvious solution to this problem is to provide a function "TYPE(x)" in SPECIAL to return the type of a variable x and write

```
IF TYPE(x) = t1 THEN f(x) ELSE g(x)
```

This would work in some cases, but would still produce a type error if f required an argument of type t1. The error occurs because automatic type checking cannot identify (without semantic checks) that the context of the call has restricted the type of x. Instead we provide the TYPECASE expression, which provides a context in which the automatic type checking facility can detect that an object of a united type has a particular constituent type. To solve the above problem, we write

```
TYPECASE x OF
  t1 : f(x);
  t2 : g(x);
END
```

The type labels (t1 and t2 in the example) must refer to disjoint types, the type of the object variable (x in the example) must be the union of all the type labels, and the type of the entire expression is the union of the types of the component expressions (f(x) and g(x) in the example).

IV. THE SPECIFICATION LEVEL

A. Introduction

In the discussion of the specification level, we discuss how to configure expressions at the assertion level in order to write module

specifications. The specification of a module is divided into six optional paragraphs so that its top-level structure looks like this:

```
MODULE <symbol>
    TYPES
        .
        .
        .
    DECLARATIONS
        .
        .
        .
    PARAMETERS
        .
        .
        .
    DEFINITIONS
        .
        .
        .
    EXTERNALREFS
        .
        .
        .
    FUNCTIONS
        .
        .
        .
END_MODULE
```

<symbol> is the name of the module. The TYPES paragraph contains the declarations for all named types (including designators). The DECLARATIONS paragraph contains all global declarations for variables (see the next subsection). The PARAMETERS paragraph contains the declarations for symbolic constants (called parameters) that become bound at some time before the module is used and that cannot be changed. Module parameters are used to characterize a resource (e.g., the maximum size of a stack) or the values of initialization that are not bound to particular constants. The DEFINITIONS paragraph contains the definitions of macros whose scope is global to the module. The EXTERNALREFS paragraph contains the declarations of objects of other modules (i.e., designator types, functions with their arguments and results, scalar types, and parameters) that are referenced in the specification. The FUNCTIONS paragraph contains the definitions (and declarations) for all the V-, O-, and OV-functions of the module.

Following a discussion of some general issues (e.g., as binding, declaration, scope rules, macros, and external references) is a description of function definition.

B. Binding, Declaration, and Scope

Every name in SPECIAL must have a binding, or a place where it

is associated with a particular object. The scope of a binding is the text in a module specification over which that particular binding is in force. The scope of a binding depends on the object being bound, and may be any one of the following:

* The entire module, in the case of module parameters, function names, names for types, global macros, and constants of scalar types.
* The function definition or macro definition, in the case of formal arguments and results.
* The expression over which the variable is an index, in the case of indicial variables.

The binding of a name for which a binding is already in force is not allowed, thus eliminating the overlapping of scopes for the same name.

In most programming languages the declaration of a variable is inseparable from its binding. In SPECIAL this is the case for names of functions, macros, module parameters, and constants of scalar types, which are all bound when they are declared. However, the declaration of a variable in SPECIAL (i.e., an indicial variable or formal argument or result) may be separated from its binding. This is done to allow a single global declaration for a variable name (in the DECLARATIONS paragraph) to apply to many different bindings. Bindings for indicial variables occur in the expressions in which they are introduced; formal arguments and results are bound either in the function definition, the external reference, or the macro definition. The motivation for global declarations is to save writing on the part of the module specifier, and to enable the establishment of mnemonics in the choice of names for globally declared variables. For example, this allows conventions such as having the variable i be of type INTEGER in all bindings. A local declaration supersedes a global one.

C. Macros

The concept of macros in SPECIAL is different from that of most programming languages. In programming languages a macro definition is generally a string substitution rule of of some complexity. The string to which the macro expands need not be a particular syntactic entity. All variables in the macro definition that are not formal arguments are bound the context of the expansion, so that the macro writer must be careful about using macro definitions with unbound variables. In SPECIAL, a macro definition has a body which is an expression and thus has a type that must be the same as the declared type of the macro definition. All macro references, which look syntactically like function references (excepting macros without arguments, which look like variables), have the same type as the declared type of the macro definition. In addition a macro

definition may not use any names except its own formal arguments, and the type names and parameters of the module. Thus, there is no chance of misusing the macro because of the context in which it is referenced.

Macros in SPECIAL are a formal shorthand (corresponding to the definitions used by a mathematician) and are not expanded as part of the syntactic processing of a module specification. They would be expanded only for proving properties based on the specification.

The syntax of a macro definition is as follows:

```
<definition> ::= <typespecification>
                 <symbol> [<formalargs>] IS <expression>
```

where

```
<formalargs> ::= '(' <declaration> {';' <declaration>}* ')'
```

Local macros are defined within the DEFINITIONS section of a function definition, and have a scope of that function definition only. Global macros are defined in the DEFINITIONS paragraph of the entire module and have global scope.

Further work on macros might include the ability to define macros with complex type checking rules (such as the set operations of SPECIAL described above). This would probably require some restricted notion of a type variable. Also interesting would be the ability to define new types of specification language objects for which functions are part of the definition (e.g., bags as defined in [6]) and intensional constructors for these types. This kind of definitional facility would put the put the complete power of a mathematician (to define new mathematical concepts) in the hands of the specification writer, but might require so much mechanism as not to be worthwhile.

D. V-function Definition

V-functions have two purposes: to describe the state of the module and to provide information about the module's state to programs using the module. The current value of a V-function that defines a module's state is not explicitly described as part of the module specification. Instead, its current value is defined by induction: its initial value appears in its specification, and subsequent values are determined by the sequence of O-functions calls up to that point (since each O-function call relates the values of V-functions before the call to values of V-functions after the call).

A V-function may be either hidden or visible, and either primitive or derived. A hidden V-function is one that cannot be called by programs using the module, whereas a visible V-function can be. A derived V-function is one whose value is an expression derived

from other V-functions of the module, whereas a primitive V-function contains part of the state definition. The form of a V-function's definition depends on its status. The syntax for a V-function definition is as follows:

```
VFUN <symbol> <formalargs> '->' <declaration> ';'
  [DEFINITIONS {,definition> ';'}+]
  [HIDDEN | EXCEPTIONS {<expression> ';'}+]
  [{ INITIALLY | DERIVATION} <expression> ';']
```

The first line, the function header, declares the formal arguments and result of the V-function. The second line declares any macros local to the function. The third line establishes whether the function is hidden or visible. If the function is hidden, the keyword "HIDDEN" appears. If the function is visible, the third line contains the keyword EXCEPTIONS followed by a list of exception conditions for the function. An exception condition is a BOOLEAN expression which, if TRUE for a given call to the function, means that the function is not executed (thus no value is returned in the case of a V-function). Instead, control is returned to the calling program with a notification of the exception that was detected. The exceptions have an ordering, meaning that a single exception is be signalled for a given function call, even if more than one exception condition is satisfied. If the EXCEPTIONS section of a function looks like this:

```
EXCEPTIONS e1; e2; ... ; en;
```

then its semantics are

```
IF e1 THEN ERRORCODE = 1
  ELSE IF e2 THEN ERRORCODE = 2
    .
    .
    .
  ELSE IF en THEN ERRORCODE = n
  ELSE ERRORCODE = UNDEFINED
```

where ERRORCODE is an abstract variable used in passing the identity of the exception beck to the calling program. Exception conditions are the same for O- and OV-functions, described in the next subsection.

If the V-function is primitive, the fourth line has the word INITIALLY followed by an assertion characterizing the function's initial value. If the V-function is derived, the fourth line has the word DERIVATION followed by an expression (of the same type as the V-function's result) that denotes its initial value.

As an example of V-function definition, consider a stack of integers maintained by a module. The state information for the module is contained in the V-functions "stack" and "ptr", signifying the elements of the stack and the stack pointer, respectively. Their definitions are as follows:


```
VFUN ptr() -> INTEGER i;
     INITIALLY i = 0;


VFUN stack(INTEGER i) -> INTEGER j;
     HIDDEN;
     INITIALLY j = ?;
```

Note that "ptr" has no exceptions, that "stack" is hidden (to prevent programs from examining intermediate values of the stack), and that the initial values of "ptr" and "stack" are 0 and UNDEFINED (or ?), respectively. The value of "ptr" signifies the number of values on the stack (also the number of integers i for which "stack(i)" is defined). We could also define a derived V-function, "top", that returns the value of the top of the stack, as follows:

```
VFUN top() -> INTEGER j;
     EXCEPTIONS ptr() = 0;
     DERIVATION stack(ptr());
```

Note that "ptr" could also have been written as a derived V-function whose derivation would be as follows,

```
CARDINALITY({INTEGER k | stack(k) ~= ?})
```

This would do away with redundancy at the expense of some clarity. Note also that asking for the top of an empty stack is meaningless, so an exception condition prevents such a call.

E. O- and OV-function Definition

The syntax of an O-function definition is as follows:

```
OFUN <symbol> <formalargs> ';'
     [DEFINITIONS { <definition> ';' }+]
     [EXCEPTIONS { <expression> ';' }+]
     [EFFECTS { <expression> ';' }+]
```

The syntax of an OV-function definition differs only in the header (the first line), which is

```
OVFUN <symbol> <formalargs> '->' <declaration> ';'
```

The EFFECTS section describes a state transformation by using assertions to relate values of V-functions before the call to values of V-functions after the call. Values of V-functions before and after the call are distinguished by preceding all references to values of V-functions after the call by a single quote ('). For example, to indicate that the value of a V-function f() is incremented by an O- or OV-function call, we write

```
'f() = f() + 1
```

Note that an effect is an assertion rather than an assignment, for we can write

```
'f() > f() + 1
```

or

```
'f() + 'g() = f() + g()
```

which do not uniquely redefine the values of f() and g(). Note that all V-function values in the EFFECTS of an O-function that do not appear with a single quote are left unchanged by the O-function.

O- and OV-functions of other modules can be referenced by means of the EFFECTS_OF construct. If "ol(args)" is a reference to an O-function of another module, then the effect "EFFECTS_OF ol(args)" would expand all of the effects of "ol(args)" in the place where it was written. OV-functions may also be referenced in this way. The expression "x = EFFECTS_OF ovl(args)", where "ovl(args)" is a reference to an OV-function of another module, means that, "x is equal to the result of ovl(args) and all effects of ovl(args) are TRUE."

An example of an O-function definition is the function "push", to be used with the V-functions of the stack, described above:

```
OFUN push(INTEGER j);
     EXCEPTIONS ptr() >= maxsize;
     EFFECTS 'stack('ptr()) = j;
          'ptr() = ptr() + 1;
```

"maxsize" is a module parameter of type INTEGER that designates the maximum permitted stack size.

An example of an OV-function definition is provided by "pop", which pops the stack and returns the popped value, as follows:

```
OVFUN pop() -> INTEGER j;
     EXCEPTIONS ptr() = 0;
     EFFECTS j = top();
          'ptr() = ptr() - 1;
          'stack(ptr()) = ?;
```

Note that the value of j, the result of "pop", is also specified as part of the EFFECTS.

V. EXAMPLES

Table I displays the specification of a module that maintains a set of stacks as an abstract data type. Its specification differs slightly from the examples presented above, having the following interesting properties:

* A module parameter, "maxstacks", to limit the number of stacks that can exist.
* Designators of type "stack_name", to name the individual stacks, and an extra argument in each function for the stack designator.
* The functions "create_stack" and "delete_stack".
* Two global macros: "nstacks", the number of stacks that currently exist (the existence predicate for stacks is "ptr(s) ~= ?"); and "empty(s)", the empty predicate for stack s. These are both examples of mnemonics. Using mnemonics for exception conditions has been used frequently for writing larger specifications.

Note the use of global declarations: s always refers to a stack; i to a pointer value; and j to a stack value. Note also that comments, denoted by "$(...)", can be inserted anywhere in a specification.

Table II is a specification for a telephone system of a single area code. This is not a software system; it is implemented in hardware and its O-functions correspond to physical acts performed by people. However, the telephone system is a good example because everyone understands it, as opposed to most complex software systems. The system contains the following general design decisions:

* Telephones are named by a designator type (phone_id) rather than a telephone number. This corresponds to real life, in which a telephone is physically protected, i.e., knowing the number of a phone is not sufficient to be able to pick up that phone and dial from it.
* The mapping between phone numbers and connected phones is an invertible function. This is not true in a phone system with more than one area code, because a single phone number may identify different phones in different areas and because a phone is known by a different number when dialed from within the area than when dialed from outside the area.
* The state of a phone is indicated by the scalar type "phone_state". Based on the state of the phone, different things happen when an O-function is called. These states can later be mapped to particular switch positions in the actual phone circuits.
* The phenomenon that a connection can be terminated only by the party that initiated the call. Thus if the called party hangs up, the connection still exists, and the hung up phone has state "hung_up_but_connected".

There are also several features of SPECIAL whose use is worth pointing out:

* Use of a subtype to characterize a digit. Note that in this case there is no chance of misusing the subtype, since digits are only used with the equality operator.

* Use of vectors to represent phone numbers. The intensional vector constructor is used in the O-function "dial".

* Use of the SOME expression in the O-function "pick_up_phone". In this case there is only one value of "phone1" satisfying the given assertion.

The reader can see other ways of writing module specifications with the same properties as those above. The style to be chosen in writing specifications depends on one's desire for conciseness, readability, or even provability of the specifications. It is also possible to see that specifications may differ considerably from their implementations. In fact, the difference between specification and implementation is often so great that specifications cannot be construed in any way as a guide to the programmer on how to write an efficient implementation. Such information must often be supplied separately from the module specification.

VI. CONCLUSIONS

SPECIAL has been shown extremely useful for designing certain classes of systems, especially operating systems. It enables some effects of crucial design decisions to be examined at an early stage in the design process, resulting in "tight" designs for systems specified in this way. Its usefulness for proof is currently being examined. However, SPECIAL is only one of several techniques currently in use for specifying software systems. Our experience has shown that specifications written in SPECIAL tend to be lengthy, because

* SPECIAL tries to be as general as possible, and encourages the writing of sufficient (as well as necessary) properties of software systems.

* SPECIAL tries to be both easy to read and easy to use.

More work must be done concerning tradeoffs and criteria for design of specification languages before the ultimate value of SPECIAL can be determined.

Table I


```
MODULE stacks


    TYPES

stack_name: DESIGNATOR;

    DECLARATIONS

INTEGER i, j;
stack_name s;

    PARAMETERS

INTEGER maxsize $( maximum size of a given stack) ,
       maxstacks $( maximum number of stacks allowed) ;


    DEFINITIONS

INTEGER nstacks IS CARDINALITY({ stack_name s | ptr(s) ~= ? });

BOOLEAN empty(stack_name s) IS ptr(s) = 0;

    FUNCTIONS


VFUN ptr(s) -> i;
    INITIALLY
       i = ?;

VFUN stack(s; i) -> j;
    HIDDEN;
    INITIALLY
       j = ?;

VFUN top(s) -> j;
    EXCEPTIONS
       ptr(s) = ?;
       empty(s);
    DERIVATION
       stack(s, ptr(s));

OVFUN create_stack() -> s;
    EXCEPTIONS
       nstacks >= maxstacks;
    EFFECTS
       s = NEW(stack_name);
       'ptr(s) = 0;

OFUN delete_stack(s);
```

```
    EXCEPTIONS
       ptr(s) = ?;
    EFFECTS
       'ptr(s) = ?;
       FORALL i: 'stack(s, i) = ?;

OFUN push(s; j);
    EXCEPTIONS
       ptr(s) = ?;
       ptr(s) >= maxsize;
    EFFECTS
       'stack(s, 'ptr(s)) = j;
       'ptr(s) = ptr(s) + 1;

OVFUN pop(s) -> j;
    EXCEPTIONS
       ptr(s) = ?;
       empty(s);
    EFFECTS
       j = top(s);
       'ptr(s) = ptr(s) - 1;
       'stack(s, ptr(s)) = ?;

END_MODULE
```

Table II

```
MODULE telephone_system


    TYPES

phone_id: DESIGNATOR;
phone_state:
{ hung_up, hung_up_but_connected, dial_tone, dialing,
 dialed_unconnected_number, ringing_another_phone, being_rung,
 connected, busy } ;
digit: { INTEGER i | 0 <= i AND i <= 9 };
phone_number_id: VECTOR_OF digit;


    DECLARATIONS

BOOLEAN b;
phone_id phone, phone1;
digit d;
```



```
phone_number_id phone_number;
phone_state ps;


    FUNCTIONS


VFUN state(phone) -> ps; $( state of "phone")
    HIDDEN;
    INITIALLY ps = ?;

VFUN connection(phone) -> phone1;  $( "phone1" is the phone that
                                   "phone" has dialed and is either
                                   connected or ringing)

    HIDDEN;
    INITIALLY phone1 = ?;

VFUN buffer(phone) -> phone_number;  $( sequence of digits dialed
                                       by "phone")

    HIDDEN;
    INITIALLY phone_number = ?;

VFUN valid_phone_number(phone_number) -> b; $(TRUE for all valid
                                              phone numbers)

    HIDDEN;
    INITIALLY TRUE $(initialized by the phone company);

VFUN directory(phone_number) -> phone;  $( "phone_number" that
                                         corresponds to "phone")

    HIDDEN;
    INITIALLY TRUE $( initialized by the phone company) ;

OVFUN install(phone_number) -> phone;  $( creates a new designator
                                         "phone" that corresponds to
                                         "phone_number")

    EXCEPTIONS
       NOT valid_phone_number(phone_number);
       directory(phone_number) ~= ?;
    EFFECTS
       phone = NEW(phone_id);
       'directory(phone_number) = phone;
       'state(phone) = hung_up;
       'buffer(phone) = VECTOR();

OFUN disconnect(phone_number);  $( disconnects phone corresponding
                                  to "phone_number")

    DEFINITIONS
       phone_id phone IS directory(phone_number);
    EXCEPTIONS
       phone = ?;
       state(phone) ~= hung_up;
    EFFECTS
       'directory(phone_number) = ?;
       'state(phone) = ?;
```

```
        'buffer(phone) = ?;

OFUN pick_up_phone(phone); $( "phone" is picked up)
    EXCEPTIONS
       state(phone) = ?;
       NOT state(phone) INSET
        { hung_up, hung_up_but_connected, being_rung };
    EFFECTS
       IF state(phone) = hung_up
         THEN $( picking up to dial) 'state(phone) = dial_tone
       ELSE IF state(phone) = being_rung
         THEN $( answering phone)
            'state(SOME phonel | connection(phonel) = phone)
                = connected
              AND 'state(phone) = connected
       ELSE $(resuming existing connection)
         'state(phone) = connected;

OFUN dial(phone; d); $( dials a digit "d" from "phone")
    DEFINITIONS
       INTEGER j IS LENGTH(buffer(phone));
       phone_number_id newbuf
          IS VECTOR(FOR i FROM 1 TO j + 1
                    : IF i <= j THEN buffer(phone)[i] ELSE d);
       phone_id phonel IS directory(newbuf);
    EXCEPTIONS
       state(phone) = ?;
    EFFECTS
       state(phone) INSET {dial_tone, dialing}
         => $( update buffer and change state)
           'buffer(phone) = newbuf
         AND (IF phonel ~= ?
           THEN $( a valid number has been reached)
             IF state(phonel) = hung_up
               THEN  $( ringing begins)
                 'state(phone) = ringing_another_phone
                    AND 'state(phonel) = being_rung
                   AND 'connection(phone) = phonel
               ELSE $( busy signal) 'state(phone) = busy
           ELSE IF valid_phone_number(newbuf)
             THEN $( not a connected number)
                'state(phone) = dialed_unconnected_number
           ELSE 'state(phone) = dialing);

OFUN hang_up(phone); $( hangs up "phone")
    EXCEPTIONS
       state(phone) = ?;
       state(phone) INSET
         { hung_up, being_rung, hung_up_but_connected };
    EFFECTS
       IF EXISTS phonel : connection(phonel) = phone
         THEN $(connection NOT terminated)
              'state(phone) = hung_up_but_connected
       ELSE $(back to original state)
```



```
        'state(phone) = hung_up
       AND 'buffer(phone) = VECTOR()
       AND (connection(phone) ~= ? => $(connected to someone else)
            'connection(phone) = ?
            AND 'state(connection(phone)) =
             (IF state(phone) = ringing_another_phone
                THEN $(ringing stops) hung_up
              ELSE $(terminates connection) dial_tone));

END_MODULE
```

REFERENCES

[1] J. Guttag, E. Horowitz, and D. Musser. "The Design of Data Structure Specifications," Proc. Second International Conference on Software Engineering, San Francisco, California (October 1976).

[2] B. H. Liskov and S. Zilles. "Specification Techniques for Data Abstractions," Proc. International Conference on Reliable Software, Los Angeles, California, pp. 72-87 (April 1975).

[3] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. "A Provably Secure Operating System," Final Report, Project 2581, Stanford Research Institute, Menlo Park, California (June 1975).

[4] D. L. Parnas. "A Technique for Software Module Specifications with Examples," Comm. ACM 15, 5, pp. 330-336 (May 1972).

[5] D. L. Parnas. "On the Criteria to Be Used in Decompcsing Systems into Modules," Comm. ACM 15, 12, pp. 1053-1058 (December 1972).

[6] L. Robinson and K. N. Levitt. "Proof Techniques for Hierarchically Structured Programs," Technical Report CSL-27, Computer Science Laboratory, Stanford Research Institute, Menlo Park, California, to appear in Comm. ACM (November 1975).

[7] L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena. "On Attaining Reliable Software for a Secure Operating System," Proc. International Conference on Reliable Software, Los Angeles, California, pp. 267-284 (April 1975).

[8] O. M. Roubine and L. Robinson. "SPECIAL Reference Manual," Technical Report CSL-45, Computer Science Laboratory, Stanford Research Institute, Menlo Park, California (August 1976).

[9] J. M. Spitzen, K. N. Levitt, and L. Robinson. "An Example of Hierarchical Design and Proof," Technical Report CSL-30,

Computer Science Laboratory, Stanford Research Institute, Menlo Park, California (also submitted for publication) (March 1976).

[10] R. S. Boyer and J S. Moore. "Proving Theorems About LISP Functions," J. ACM 22, 1 pp. 129-144 (January 1976).

[11] J S. Moore. "The INTERLISP Virtual Machine Specification," Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California (September 1976).

[12] C. A. R. Hoare and N. Wirth. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, pp. 335-355 (1973).

APPENDIX: GRAMMAR OF SPECIAL

```
ROOT                    ::= MODULE <symbol> [<types>]
                            [<declarations>] [<parameters>]
                            [<definitions>] [<externalrefs>]
                            [<functions>]
                            END_MODULE
                        ::= MAP <symbol> TO
                            <symbol> { ',' <symbol> }* ';'
                            [<types>] [<declarations>]
                            [<parameters>] [<definitions>]
                            [<externalrefs>] [<mappings>]
                            END_MAP

<types>                 ::= TYPES { <typedeclaration> ';' }+

<typedeclaration>       ::= <symbol> { ',' <symbol> }* ':'
                            { DESIGNATOR | <typespecification>
                              | <setexpression> }

<typespecification>     ::= <symbol>
                        ::= INTEGER
                        ::= BOOLEAN
                        ::= REAL
                        ::= CHAR
                        ::= STRUCT '(' <declarationlist> ')'
                        ::= ONE_OF '(' <typespecification>
                            { ',' <typespecification> }+ ')'
                        ::= { SET_OF | VECTOR_OF }
                            <typespecification>

<simple declaration>    ::= <typespecification> <symbol>

<declaration>           ::= <simple declaration> { ',' <symbol> }*
                        ::= <symbol>

<declarations>          ::= DECLARATIONS { <declaration> ';' }+

<parameters>            ::= PARAMETERS
                            { <parameterdeclaration> ';' }+

<parameterdeclaration>  ::= <typespecification> <symbol>
                            [<formalargs>]
                            { ',' <symbol> [<formalargs>] }*

<formalargs>            ::= '(' [<declarationlist>] ')'
                            [ '[' <declarationlist> ']' ]
```

```
<declarationlist>     ::= <declaration> { ';' <declaration> }*

<definitions>         ::= DEFINITIONS { <definition> ';' }+

<definition>          ::= <typespecification> <symbol>
                              [<formalargs>] IS <expression>

<externalrefs>        ::= EXTERNALREFS { externalgroup }+

<externalgroup>       ::= FROM <symbol> ':' { <externalref> ';' }+

<externalref>         ::= <parameterdeclaration>
                      ::= <symbol> { ',' <symbol> }* ':' DESIGNATOR
                      ::= <symbol> ':' <setexpression>
                      ::= { VFUN | OVFUN } <symbol> <formalargs>
                              '->' <declaration>
                      ::= OFUN <symbol> <formalargs>

<functions>           ::= FUNCTIONS { <functionspec> }+

<functionspec>        ::= VFUN <symbol> <formalargs>
                              '->' <declaration> ';'
                              [<definitions>]
                              [{ HIDDEN ';' | <exceptions> }]
                              { INITIALLY | DERIVATION }
                                 <expression> ';'
                      ::= OVFUN <symbol> <formalargs>
                              '->' <declaration> ';'
                              [<definitions>] [<exceptions>]
                              { <delay> }* [<effects>]
                      ::= OFUN <symbol> <formalargs> ';'
                              [<definitions>] [<exceptions>]
                              { <delay> }* [<effects>]

<delay>               ::= DELAY UNTIL <expression> ';'

<exceptions>          ::= EXCEPTIONS { <expression> ';'
                              | EXCEPTIONS_OF <call> ';' }+

<effects>             ::= EFFECTS { <expression> ';' }+

<mappings>            ::= MAPPINGS { <mapping> ';' }+

<mapping>             ::= <symbol> [<formalargs>] ':' <expression>
                      ::= <symbol> ':' <typespecification>

<expression>          ::= IF <expression> THEN <expression>
                              ELSE <expression>
                      ::= LET <qualification>
                              { ';' <qualification> }*
                              IN <expression>
                      ::= SOME <qualification>
                      ::= { FORALL | EXISTS }
```



```
                              . <qualif\declarationlist> ':'
                                 <expression>
                      ::= TYPECASE <symbol> OF { <case> ';' }+ END
                      ::= <expression> <binaryoperator>
                              <expression>
                      ::= { NOT | '~' } <expression>
                      ::= '-' <expression>
                      ::= '(' <expression> ')'
                      ::= <symbol>
                      ::= <number>
                      ::= <character constant>
                      ::= <string constant>
                      ::= TRUE | FALSE | UNDEFINED | ?
                      ::= <expression> '[' <expression> ']'
                      ::= <expression> '.' <symbol>
                      ::= { CARDINALITY | LENGTH | MAX | MIN | SUM
                              | INTPART | FRACTPART }
                              '(' <expression> ')'
                      ::= NEW '(' <symbol> ')'
                      ::= [EFFECTS_OF] <call>
                      ::= <structureconstructor>
                      ::= <vectorconstructor>
                      ::= <setexpression>

<qualif\declarationlist>::= [<qualification> | <declaration>]
                              { ';' <qualification> |
                                 <declaration> }*

<qualification>       ::= [<typespecification>] <symbol>
                              { '|' | INSET } <expression>

<case>                ::= <typespecification> ':' <expression>

<binaryoperator>      ::= '*' | '/' | INTER | '+' | '-' | UNION |
                              DIFF | '=' | '~=' | '>' | '>=' |
                              '<' | '<=' | INSET | AND | OR |
                              SUBSET | MOD | '=>'

<call>                ::= ['''] <symbol> '(' [ <expression>
                              { ',' <expression> }*] ')'

<structureconstructor> ::= '<' [ <expression>
                              { ',' <expression> }* ] '>'
                      ::= '<' <range> ':' <expression> '>'

<vectorconstructor>   ::= VECTOR '(' [ <expression>
                              { ',' <expression> }* ] ')'
                      ::= VECTOR '(' <range> ':' <expression> ')'

<range>               ::= FOR <symbol> FROM <expression>
                              TO <expression>

<setexpression>       ::= '{' [ <expression>
                              { ',' <expression> }* ] '}'
```

```
::= '{' [<typespecification>] <symbol> '|'
        <expression> '}'
```

APPENDIX B
SPECIFICATIONS FOR THE BASIC DESIGN

This appendix contains the formal specifications for the design of the Provably Secure Operating System. The system is decomposed into 14 levels listed in Table 2-1. Each level further decomposed into modules as described in Table 2-2. The specifications of these modules form the body of this appendix.

At first glance the reader may believe that attempting to understand this many pages of specifications is a huge undertaking. The total number of pages of specifications is large because the system being specified is a large system and the specifications include not only descriptions of the system interface, but descriptions of all the internal interfaces of the system that are not visible. However, the specifications are not as complex as their size might suggest. The important ideas of the design are expressed in relatively few lines of the specifications. Much of the specifications are concerned with the syntax of the language, the requirements of formality, and details of the design. The details of the design are necessary for a precise formulation, but are not essential to an understanding of the important ideas. As with any language with which the reader is unfamiliar some experience with reading the language is necessary before it becomes possible to easily identify the important constructs.

The reader should not attempt to understand these specifications without first reading Section II of this report. Section II describes, in prose, the important ideas in the design. It is also assumed that the reader is familiar with the SPECIAL language in which the specifications are written. The reader interested only in the system interface, and not in the internal interfaces, should read only the interface and module specifications of level 13.

To a large degree the readability and understandability of a specification is dependent upon the style in which that specification is written. One can write an unreadable program in the best of programming languages and one can write an easily readable program in the worst of programming languages. The same can be said of specifications written in SPECIAL. Examples of differences in style that can effect readability are use of global or local declarations and definitions, liberal or limited use of definitions, and frequent or infrequent use of data abstraction. The specifications presented here are not consistent with respect to style. The style differs in the different modules because the specifications were written by different people at different times while the language was emerging and while the authors were learning how to use the language. This is unfortunate because a higher degree of uniformity of style would enhance the readability of the specifications as a whole. However, the variation in style does serve to illustrate different styles of specification writing. Some consistency has been enforced in the formatting of specifications.

APPENDIX B PSOS SPECIFICATIONS

This appendix is divided into four remaining sections:

* the specification of the interface for each of the levels 0 through 13,
* the specification of the modules of the levels 0 through 13,
* the specification of the example modules USER, MAIL, and LINKER,
* and the specification of an alternative module CHANNEL_IO.

The modules of levels 0 through 13 are presented beginning with the modules of level 0 and proceeding upward through the hierarchy to level 13. It is suggested that the reader examine them in this order. Modules that are included in the specification of more than one level are not repeated and are introduced as part of the lowest level in which they appear.

The interface specification for each level consists of a list of the modules that comprise that level. It is assumed that all the visible functions of all modules of the interface are visible at the interface unless the functions are itemized in the WITHOUT clause of that module in the interface specification.

APPENDIX B PSOS SPECIFICATIONS

CONTENTS

APPENDIX B INTERFACES

```
(INTERFACE LEVEL_0
    capabilities )

(INTERFACE LEVEL_1
    capabilities
    registers
    memory )

(INTERFACE LEVEL_2
    capabilities
    registers
    memory
    interrupts )

(INTERFACE LEVEL_3
    capabilities
    registers
    memory
    interrupts
    clock )

(INTERFACE LEVEL_4
    capabilities
    registers
    memory
    interrupts
    clock
    arithmetic )

(INTERFACE LEVEL_5
    capabilities
    registers
    memory
    interrupts
    clock
    arithmetic
    primitive_io )

(INTERFACE LEVEL_6
    capabilities
    registers
    memory
    arithmetic
    system_process
    system_invoke
    coordinator
    system_io
    timers )
```

```
(INTERFACE LEVEL_7
    capabilities
    registers
    memory
    arithmetic
    system_process
    coordinator
    system_invoke
    system_io
    timers
    pages )

(INTERFACE LEVEL_8
    capabilities
    registers
    arithmetic
    system_process
    coordinator
    system_invoke
    system_io
    timers
    segments
    windows )

(INTERFACE LEVEL_9
    capabilities
    registers
    arithmetic
    system_process
    coordinator
    system_invoke
    system_io
    timers
    segments
    windows
    extended_types )

(INTERFACE LEVEL_10
    capabilities
    registers
    arithmetic
    system_process
    coordinator
    system_invoke
    system_io
    timers
    segments
    windows
    extended_types
    directories )
```



```
(INTERFACE LEVEL_11
    capabilities
    registers
    arithmetic
    system_process
    coordinator
    system_invoke
    system_io
    timers
    segments
    windows
    (extended_types WITHOUT object_create object_delete)
    directories
    user_objects )

(INTERFACE LEVEL_12
    capabilities
    registers
    arithmetic
    (system_invoke WITHOUT system_call system_return)
    coordinator
    timers
    (segments WITHOUT segment_create segment_delete)
    (windows WITHOUT window_create window_delete)
    (extended_types WITHOUT object_create object_delete)
    (directories WITHOUT directory_create directory_delete
                         add_distinguished_entry
                         remove_distinguished_entry)
    user_objects
    user_process
    user_invoke
    visible_io )

(INTERFACE LEVEL_13
    capabilities
    registers
    arithmetic
    (system_invoke WITHOUT system_call system_return)
    coordinator
    timers
    (segments WITHOUT segment_create segment_delete)
    (windows WITHOUT window_create window_delete)
    (extended_types WITHOUT object_create object_delete)
    (directories WITHOUT directory_create directory_delete
                         add_distinguished_entry
                         remove_distinguished_entry)
    user_objects
    user_process
    user_invoke
    visible_io
    procedure_records )
```

```
MODULE capabilities

    TYPES

capability: DESIGNATOR;
slave_capability:
{ capability c | EXISTS capability cl: get_slave(c) = cl };
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length }
        $( access rights from a capability);
machine_word: ONE_OF(capability, INTEGER, CHAR, BOOLEAN);
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );

    DECLARATIONS

access_string as, asl;
capability c, cl;
slave_capability u;
INTEGER id, idl;
BOOLEAN b;

    PARAMETERS

INTEGER access_string_length  $( access strings must be at least 7
                                 bits long);
INTEGER n_store_permissions $( number of store permissions must be
                               less than access string length);
INTEGER read, write, modify, delete $( access rights) ;
machine_word zeroword $( initial value for machine words) ;
capability null $( capability guaranteed for nothing);
capability minimum_instructions, maximum_instructions
          $( capabilities for the smallest and largest instructio
             classes );
capability resident_instructions
          $( capability for instructions that interpret addresses
             as resident memory locations );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );

    DEFINITIONS

access_string all_true
   IS VECTOR(FOR I FROM 1 TO access_string_length: TRUE);
access_string all_false
   IS VECTOR(FOR I FROM 1 TO access_string_length: FALSE);
access_string and_as(as; asl)
   IS VECTOR(FOR I FROM 1 TO access_string_length
          : as[i] AND asl[i])  $( the bitwise and of two bit
                                  strings);

    FUNCTIONS
```



```
VFUN make_cap(id; as) -> c;  $( this function expresses the
                             relationship between uid's, access
                             rights, and capabilities)
    HIDDEN;
    INITIALLY
        (FORALL idl:
            FORALL asl:
                (make_cap(id, as) = make_cap(idl, asl))
              =(id = idl AND as = asl))
       AND(FORALL cl:
              EXISTS idl: EXISTS asl: cl = make_cap(idl, asl));

VFUN get_uid(c) -> id;  $( this function returns the uid for a
                        capability)
    HIDDEN;
    DERIVATION
       SOME idl | EXISTS as: c = make_cap(idl, as);

VFUN get_access(c) -> as;  $( this function returns the access
                           rights of a capability)
    DERIVATION
       SOME asl | EXISTS id: c = make_cap(id, asl);

VFUN old(id) -> b;  $( true if a capability with uid u has been
                    returned by create_capability or
                    create_restricted_cap)
    HIDDEN;
    INITIALLY
        b
       =(EXISTS slave_capability u
            | predefined_slave(u) ~= not_predefined:
          get_uid(u) = id);

VFUN restrict_access(c; as) -> cl;  $( returns a capability with
                                    the uid of c and the access
                                    rights of c reduced by as)
    DERIVATION
       make_cap(get_uid(c), and_as(as, get_access(c)));

VFUN get_slave(c) -> u; $( returns the slave capability for c)
    DERIVATION
       make_cap(get_uid(c), all_false);

OVFUN create_capability() -> c;  $( returns a capability with a uid
                                 with which no other capability
                                 has been returned)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       EXISTS id:
            old(id) = FALSE AND 'old(id) = TRUE
          AND c = make_cap(id, all_true)
         AND(FORALL idl: old(idl) => 'old(idl));
```

APPENDIX B CAPABILITIES

```
OVFUN create_restricted_cap(as) -> c;
          $( returns a capability with a previously unused uid and
           with the given access)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       EXISTS id:
            old(id) = FALSE AND 'old(id) = TRUE
          AND c = make_cap(id, as)
         AND(FORALL id1: old(id1) => 'old(id1));
END_MODULE
```


APPENDIX B REGISTERS

```
MODULE registers

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
address:
STRUCT_OF(capability add_cap; INTEGER add_offset)  $( address of a
                                                      word of storage);
state:
STRUCT_OF(capability inst_class_reg;
          VECTOR_OF machine_word general_reg;
          VECTOR_OF address address_reg)  $( the state of a processor,
                                             i.e., all its registers);

    DECLARATIONS

INTEGER i;
VECTOR_OF BOOLEAN bv;
capability c, pc;
slave_capability u, up;
machine_word w;
state st;
address addr;

    PARAMETERS

INTEGER max_offset $( largest possible offset in address) ,
       n_general_registers $( number of general registers) ,
       n_address_registers $( number of address registers) ;
INTEGER program_counter, stack_register, arguments_register,
        environment_register $( registers with special functions );
address starting_address $( initial contents of program counter );

    DEFINITIONS

BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN bad_state(st)
   IS LENGTH(st.general_reg) ~= n_general_registers
     OR LENGTH(st.address_reg) ~= n_address_registers;
BOOLEAN out_of_bounds(INTEGER o) IS o < 0 OR o >= max_offset;
BOOLEAN no_general_register(i)
   IS i < 0 OR i >= n_general_registers;
BOOLEAN no_address_register(i)
   IS i < 0 OR i >= n_address_registers;
BOOLEAN valid_inst_class(u) IS predefined_slave(u) ~= inst_class_slave;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
```

```
APPENDIX B                          REGISTERS

INTEGER read, modify $( access rights) ;
capability maximum_instructions $( capability for executing
                                   all instructions );
capability null $( capability for nothing);
machine_word zeroword;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;

    FUNCTIONS

VFUN get_registers(pc) -> st; $( get state of process or processor pc)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, read);
    DERIVATION
       STRUCT(h_inst_class_reg(up),
              VECTOR(FOR i FROM 1 TO n_general_registers
                     : h_gen_reg(up, i)),
              VECTOR(FOR i FROM 1 TO n_address_registers
                     : h_add_reg(up, i)) );

OFUN set_registers(pc; st);  $( change the state of the given process
                             or processor)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, modify);
       bad_state(st);
    EFFECTS
       'h_inst_class_reg(up) = st.inst_class_reg;
       FORALL i:
          ( i >= 0 AND i < n_general_registers
           => 'h_gen_reg(up, i) = st.general_reg[i]);
       FORALL i:
          ( i >= 0 AND i < n_address_registers
           => 'h_add_reg(up, i) = st.address_reg[i]);

VFUN h_inst_class_reg(up) -> c; $( instructions that can be executed )
    HIDDEN;
    INITIALLY
       c = maximum_instructions;

VFUN inst_class_register()[pc] -> c;  $( instructions that can be
                                         executed)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    DERIVATION
       h_inst_class_reg(up);

OFUN load_inst_class_register(c)[pc];  $( change level at which process
                                       is executing)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       valid_inst_class(get_slave(c));
    EFFECTS
```



```
APPENDIX B                          REGISTERS

       'h_inst_class_reg(up) = c;

VFUN h_gen_reg(up; i) -> w; $( value of general register i)
    HIDDEN;
    INITIALLY
       w = zeroword;

VFUN general_register(i)[pc] -> w; $( value of general register i)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_general_register(i);
    DERIVATION
       h_gen_reg(up, i);

OFUN load_general_register(i; w)[pc]; $( set general register i)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_general_register(i);
    EFFECTS
       'h_gen_reg(up, i) = w;

VFUN h_add_reg(up; i) -> addr;  $( capability part of address
                                   register i)
    HIDDEN;
    INITIALLY
       IF i = program_counter
        THEN addr = starting_address
        ELSE addr = STRUCT(null, 0);

VFUN address_register(i)[pc] -> addr;  $( capability part of
                                          address register i)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_address_register(i);
    DERIVATION
       h_add_reg(up, i);

OFUN load_address_register(INTEGER i; address addr)[capability pc];
     $( set address register i)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_address_register(i);
       out_of_bounds(addr.add_offset);
    EFFECTS
       'h_add_reg(up, i) = addr;

VFUN proc_cap()[pc] -> c;  $( returns capability for invoking
                              process or processor)
    DERIVATION
       pc;
END_MODULE
```

```
MODULE memory

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);

    DECLARATIONS

BOOLEAN b;
VECTOR_OF BOOLEAN bv;
slave_capability u;
capability c;
INTEGER i;
machine_word w;

    PARAMETERS

INTEGER block_length(u) $( length of existing memory block );

    DEFINITIONS

INTEGER integer_value(w)
   IS TYPECASE w OF
       INTEGER: w;
       capability: 0;
       BOOLEAN: 0;
       CHAR: 0;
     END;
BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN no_block(u) IS block_length(u) = ?;
BOOLEAN address_bounds(slave_capability u; INTEGER i)
   IS i < 0 OR i >= block_length(u);
BOOLEAN not_integer(w)
   IS TYPECASE w OF
       INTEGER: FALSE;
       capability: TRUE;
       BOOLEAN: TRUE;
       CHAR: TRUE;
     END;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
INTEGER read, write $( access rights) ;
machine_word zeroword;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;
```



```
    FUNCTIONS
VFUN h_read(u; i) -> w;  $( read a word at offset i in memory block
                            u)
    HIDDEN;
    INITIALLY
       w = zeroword;

VFUN block_read(c; i) -> w;  $( read a word at offset i in block
                              with capability c)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_block(u);
       address_bounds(u, i);
    DERIVATION
       h_read(u, i);

OFUN block_write(c; i; w);  $( change word at offset i in block
                             with capability c to w)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, write);
       no_block(u);
       address_bounds(u, i);
    EFFECTS
       'h_read(u, i) = w;

OVFUN block_decrement_and_test(c; i) -> b;  $( try a P operation on
                                              a specified word)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_block(u);
       address_bounds(u, i);
       not_integer(h_read(u, i));
    EFFECTS
       b =(integer_value(h_read(u, i)) < 1);
       'h_read(u, i) = integer_value(h_read(u, i)) - 1;

OVFUN block_increment_and_test(c; i) -> b;  $( do a V operation on
                                              a specified word)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_block(u);
       address_bounds(u, i);
       not_integer(h_read(u, i));
    EFFECTS
       b =(integer_value(h_read(u, i)) < 0);
       'h_read(u, i) = integer_value(h_read(u, i)) + 1;
```

```
OVFUN block_conditional_write(c; i; w) -> BOOLEAN written;
     $( modify word if contents were zero )
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_block(u);
       address_bounds(u, i);
    EFFECTS
       IF h_read(u, i) = 0
        THEN written AND 'h_read(u, i) = w
        ELSE ~written;

END_MODULE
```



```
MODULE interrupts

    TYPES

slave_capability: { capability u | EXISTS c: get_slave(c) = u };
address: STRUCT_OF(capability add_cap; INTEGER add_offset);

    DECLARATIONS

BOOLEAN b;
VECTOR_OF BOOLEAN bv;
capability c, pc;
capability int $( capability for interrupt) ;
slave_capability u, pu, ul;
VECTOR_OF capability iv $( list of interrupts) ;
VECTOR_OF slave_capability uv $( list of uid's for interrupts) ;
INTEGER o $( offset of word in memory) ;
INTEGER i, j;
address addr;

    PARAMETERS

INTEGER set_handler, mask, $( access rights: they assume values)
      set_int $( from 1 to access_length) ;
BOOLEAN init_interrupt_addr(addr) $(true if addr is address
            of the initial interrupt procedure );

    DEFINITIONS

BOOLEAN no_interrupt(u) IS h_interrupt_set(u) = FALSE;
BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN not_interrupt_vector(uv)
   IS EXISTS j:
        ( j >= 0 AND j <= LENGTH(uv)
         AND h_interrupt_set(uv[j]) = FALSE);
BOOLEAN no_ability_vector(iv; i)
   IS EXISTS j:
        ( j >= 0 AND j <= LENGTH(iv)
         AND get_access(iv[j])[i] = FALSE);
BOOLEAN not_maskable_vector(pu; uv)
   IS EXISTS j:
        ( j >= 0 AND j <= LENGTH(uv)
         AND h_masked(pu, uv[j]) = TRUE);
BOOLEAN not_unmaskable_vector(pu; uv)
   IS EXISTS j:
        ( j >= 0 AND j <= LENGTH(uv)
         AND h_masked(pu, uv[j]) = FALSE);
BOOLEAN no_processor(pu)
   IS predefined_slave(pu) ~= processor_slave;
BOOLEAN no_value(pu) IS h_old_pc(pu) = ?;

    EXTERNALREFS
```

```
    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_sla
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
INTEGER read, modify $( access rights) ;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;

    FROM registers:
INTEGER program_counter $( index for program counter register
VFUN h_add_reg(pu; i) -> addr;
OFUN load_address_register(i; addr)[pc];

    FUNCTIONS

VFUN h_old_pc(pu) -> addr;  $( value of program counter befor
                            interrupt)

    HIDDEN;
    INITIALLY
       addr = ?;

VFUN old_pc()[pc] -> addr; $( external form of h_old_pc)
    DEFINITIONS
       slave_capability pu IS get_slave(pc);
    EXCEPTIONS
       no_processor(pu);
       no_ability(pc, read);
       no_value(pu);
    DERIVATION
       h_old_pc(pu);

VFUN h_interrupt_mode(pu) -> b; $( true of interrupts can occ
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN h_masked(pu; u) -> b; $( true if interrupt u is masked)
    HIDDEN;
    INITIALLY
       b =(IF predefined_slave(u) = interrupt_slave
            THEN TRUE ELSE ?);

VFUN h_interrupt_pending(pu; u) -> b;  $( true if interrupt u
                                       pending in processor )

    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN h_interrupt_set(u) -> b;  $( true if u is uid for an inte
    HIDDEN;
    INITIALLY
       b =(predefined_slave(u) = interrupt_slave);
```



```
VFUN h_int_handler(u) -> addr;  $( location for handler for
                                   interrupt u)

    HIDDEN;
    INITIALLY
       IF predefined_slave(u) = interrupt_slave
          THEN init_interrupt_addr(addr) ELSE addr=?;

OFUN set_int_handler(int; c; o);  $( set the location of the
                                     handler for interrupt int)

    DEFINITIONS
       slave_capability u IS get_slave(int);
    EXCEPTIONS
       no_interrupt(u);
       no_ability(int, set_handler);
    EFFECTS
       'h_int_handler(u) = STRUCT(c, o);

OFUN set_mask(pc; iv);  $( mask interrupts for the given interrupt
                           capabilities)
    DEFINITIONS
       VECTOR_OF slave_capability uv
          IS VECTOR(FOR j FROM 1 TO LENGTH(iv)
                    : get_slave(iv[j]));
       slave_capability pu IS get_slave(pc);
    EXCEPTIONS
       not_interrupt_vector(uv);
       not_maskable_vector(pu, uv);
       no_ability_vector(iv, mask);
       no_ability(pc, modify);
       no_processor(pu);
    EFFECTS
       FORALL j:
          ( j >= 0 AND j <= LENGTH(uv)
           => 'h_masked(pu, uv[j]) = TRUE);

OFUN reset_mask(pc; iv);  $( unmask interrupts for the given
                             interrupt capabilities)
    DEFINITIONS
       VECTOR_OF slave_capability uv
          IS VECTOR(FOR j FROM 1 TO LENGTH(iv)
                    : get_slave(iv[j]));
       slave_capability pu IS get_slave(pc);
    EXCEPTIONS
       not_interrupt_vector(uv);
       not_unmaskable_vector(pu, uv);
       no_ability_vector(iv, mask);
       no_ability(pc, modify);
       no_processor(pu);
    EFFECTS
       FORALL j:
          ( j >= 0 AND j <= LENGTH(uv)
           => 'h_masked(pu, uv[j]) = FALSE);
```

```
OFUN set_mode_normal()[pc]; $( permits interrupts)
    DEFINITIONS
       slave_capability pu IS get_slave(pc);
    EFFECTS
       'h_interrupt_mode(pu) = TRUE;

VFUN h_int_priority(u) -> i;  $( higher integer means higher
                               priority, each interrupt must have
                               different priority)
    HIDDEN;
    INITIALLY
       IF predefined_slave(u) = interrupt_slave
         THEN i ~= ?
             AND ~(EXISTS ul
                     | predefined_slave(ul) = interrupt_slave:
                      h_int_priority(ul) = i)
         ELSE i = ?;

OFUN receive_interrupt()[pc];  $( now is the time to interrupt
                                  processor pc)
    DEFINITIONS
       slave_capability pu IS get_slave(pc);
    EFFECTS
        h_interrupt_mode(pu) = TRUE
       =>(FORALL u:
             (    predefined_slave(u) = interrupt_slave
                AND h_interrupt_pending(pu, u) = TRUE
               AND h_masked(pu, u) = FALSE
              AND ~(EXISTS ul:
                       (predefined_slave(ul) = interrupt_slave
                          AND ul ~= u
                         AND h_interrupt_pending(pu, ul) = TRUE
                        AND h_masked(pu, ul) = FALSE
                       AND h_int_priority(ul)
                         > h_int_priority(u)))
            =>  'h_interrupt_mode(pu) = FALSE
               AND 'h_old_pc(pu) = h_add_reg(pu, program_counte
                AND EFFECTS_OF load_address_register
                                (program_counter,
                                 h_int_handler(u),
                                 pc)));

OFUN set_interrupt(pc; int); $( cause an interrupt to occur)
    DEFINITIONS
       slave_capability u IS get_slave(int);
       slave_capability pu IS get_slave(pc);
    EXCEPTIONS
       no_interrupt(u);
       no_ability(int, set_int);
       no_ability(pc, set_int);
       no_processor(pu);
    EFFECTS
       'h_interrupt_pending(pu, u) = TRUE;

END_MODULE
```



```
MODULE clock

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };

    DECLARATIONS

VECTOR_OF BOOLEAN bv;
capability c, pc;
slave_capability u, pu;
INTEGER i;
INTEGER time $( clock time) ;

    PARAMETERS

INTEGER start_up_time $( time of clock initialization) ;

    DEFINITIONS

BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN no_processor(pu)
   IS predefined_slave(pu) ~= processor_slave;
BOOLEAN not_clock(u)
   IS predefined_slave(u) ~= clock_slave;
BOOLEAN negative(i) IS i < 0;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
             predefined slave capability );
INTEGER read, modify $( access rights) ;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;

    FROM interrupts:
INTEGER set_int $( access right) ;
OFUN set_interrupt(pc; c);

    FUNCTIONS

VFUN h_clock() -> time; $( real clock time)
    HIDDEN;
    INITIALLY
       time = start_up_time;

VFUN h_clock_int(c) -> BOOLEAN b;
    $( interrupt for timer runnout )
    HIDDEN;
    INITIALLY
       b =(predefined_slave(get_slave(c)) = interrupt_slave);
```

APPENDIX B CLOCK

```
VFUN h_timer(pu) -> time;  $( time when processor pu is to be
                              interrupted)
    HIDDEN;
    INITIALLY
       time = ?;

VFUN h_timer_processor(pu) -> pc;  $( capability for processor with
                                      unique identifier pu)
    HIDDEN;
    INITIALLY
       pc = ?;

VFUN read_clock(c) -> time; $( real time clock)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       not_clock(u);
    DERIVATION
       h_clock();

OFUN update_clock(c; i); $( maintains real time clock)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, modify);
       not_clock(u);
       negative(i);
    EFFECTS
       'h_clock() = h_clock() + i;
       FORALL pu | predefined_slave(pu) = processor_slave:
           h_timer(pu) ~= ? AND h_timer(pu) <= h_clock() + i
          => 'h_timer(pu) = ?
            AND EFFECTS_OF set_interrupt(h_timer_processor(pu),
                                         SOME c | h_clock_int(c));

OFUN set_timer(i)[pc]; $( set interval timer value)
    DEFINITIONS
       slave_capability pu IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, modify);
       no_ability(pc, set_int);
       no_processor(pu);
       negative(i);
    EFFECTS
       'h_timer(pu) = h_clock() + i;
       'h_timer_processor(pu) = pc;

END_MODULE
```


APPENDIX B ARITHMETIC

```
MODULE arithmetic
    TYPES

machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
character_string: VECTOR_OF CHAR;

    DECLARATIONS

INTEGER i1, i2, i;
BOOLEAN b1, b2, b;
machine_word w1, w2, w;

    PARAMETERS

INTEGER min_integer, max_integer $( interval of legal integers) ;

    DEFINITIONS

BOOLEAN overflow(i) IS i > max_integer OR i < min_integer;
character_string type_id(w)
   IS TYPECASE w OF
       capability: "capability";
       INTEGER: "INTEGER";
       BOOLEAN: "BOOLEAN";
       CHAR: "CHAR";
    END;
BOOLEAN type_mismatch(w1; w2) IS type_id(w1) ~= type_id(w2);

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;

    FUNCTIONS

VFUN add(i1; i2) -> i; $( returns the sum of two integers)
    EXCEPTIONS
       overflow(i1 + i2);
    DERIVATION
       i1 + i2;

VFUN subtract(i1; i2) -> i;  $( returns the difference between two
                                integers)
    EXCEPTIONS
       overflow(i1 - i2);
    DERIVATION
       i1 - i2;

VFUN minus(i1) -> i; $( returns the negation of an integer)
    DERIVATION
       - i1;

VFUN multiply(i1; i2) -> i; $( multiplies two integers)
    EXCEPTIONS
       overflow(i1 * i2);
    DERIVATION
       i1 * i2;
```

APPENDIX B ARITHMETIC

```
VFUN divide(i1; i2) -> i; $( divides i1 by i2)
    EXCEPTIONS
       i2 = 0;
       overflow(i1 / i2);
    DERIVATION
       i1 / i2;

VFUN logical_and(b1; b2) -> b; $( logical and of two booleans)
    DERIVATION
       b1 AND b2;

VFUN logical_or(b1; b2) -> b; $( logical or of two booleans)
    DERIVATION
       b1 OR b2;

VFUN logical_not(b1) -> b; $( logical negation of a boolean)
    DERIVATION
       ~ b1;

VFUN less_than(i1; i2) -> b; $( true if i1 < i2)
    DERIVATION
       i1 < i2;

VFUN greater_than(i1; i2) -> b; $( true if i1 > i2)
    DERIVATION
       i1 > i2;

VFUN equal(w1; w2) -> b; $( true if two values are equal)
    EXCEPTIONS
       type_mismatch(w1, w2);
    DERIVATION
       w1 = w2;

END_MODULE
```


APPENDIX B PRIMITIVE I/O

```
MODULE primitive_io

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);

    DECLARATIONS

BOOLEAN b, int;
VECTOR_OF BOOLEAN bv;
slave_capability u;
capability c, c1;
capability d $( capability for an I/O device) ;
INTEGER stat, comm;
machine_word data;

    PARAMETERS

INTEGER control,
       device  $( each of these assumes a different value from 1 to
                  access_length);

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] = FALSE;
BOOLEAN not_device(slave_capability u)
   IS h_device_set(u) = FALSE;
BOOLEAN no_input(slave_capability u) IS h_input(u) = ?;
BOOLEAN no_output(slave_capability u) IS h_output(u) = ?;
BOOLEAN no_command(slave_capability u) IS h_command(u) = ?;
BOOLEAN too_much_input(slave_capability u) IS h_input(u) ~= ?;
BOOLEAN too_much_output(slave_capability u) IS h_output(u) ~= ?;
BOOLEAN too_many_commands(slave_capability u)
   IS h_command(u) ~= ?;
BOOLEAN uninitialized_device(slave_capability u)
   IS h_status(u) = ?;
BOOLEAN no_interrupt(slave_capability u; BOOLEAN int)
   IS IF int THEN h_device_interrupt(u) = ? ELSE FALSE;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
INTEGER read, write $( access rights) ;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;
```

```
    FROM interrupts:
OFUN set_interrupt(c; c1);

    FUNCTIONS

VFUN h_device_set(u) -> b; $( true if uid u designates a device)
    HIDDEN;
    INITIALLY
       b =(predefined_slave(u) = device_slave);

VFUN h_device_interrupt(u) -> c;  $( interrupt capability for
                                    device with uid u)
    HIDDEN;
    INITIALLY
       predefined_slave(get_slave(c)) = interrupt_slave;

VFUN h_device_processor(u) -> c;  $( processor for handling
                                    interrupts from device u)
    HIDDEN;
    INITIALLY
       predefined_slave(get_slave(c)) = processor_slave;

VFUN h_input(u) -> data;  $( one word of input data from I/O device
                          with uid u)
    HIDDEN;
    INITIALLY
       data = ?;

OVFUN read_device(d) -> data; $( visible form of h_input)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       not_device(u);
       no_input(u);
    EFFECTS
       data = h_input(u);
       'h_input(u) = ?;

OFUN write_device(d; data);  $( output data to I/O device with
                             capability d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, write);
       not_device(u);
       too_much_output(u);
    EFFECTS
       'h_output(u) = data;

OFUN send_command(d; comm); $( send command to I/O device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, control);
       not_device(u);
       too_many_commands(u);
```



```
    EFFECTS
       'h_command(u) = comm;

VFUN h_status(u) -> stat; $( status of I/O device with uid u)
    HIDDEN;
    INITIALLY
       stat = ?;

VFUN receive_status(d) -> stat; $( visible form of h_status)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       not_device(u);
       uninitialized_device(u);
    DERIVATION
       h_status(u);

VFUN h_output(u) -> data;  $( data to be output by I/O device with
                           uid u)
    HIDDEN;
    INITIALLY
       data = ?;

OVFUN device_receive(d) -> data; $( visible form of h_output)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_output(u);
    EFFECTS
       data = h_output(u);
       'h_output(u) = ?;

OFUN device_send(d; data); $( input data to system)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       too_much_input(u);
    EFFECTS
       'h_input(u) = data;

VFUN h_command(u) -> comm; $( command for I/O device with uid u)
    HIDDEN;
    INITIALLY
       comm = ?;

OVFUN device_command(d) -> comm; $( gets a command for a device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_command(u);
```

```
    EFFECTS
       comm = h_command(u);
       'h_command(u) = ?;

OFUN change_status(d; stat; int);  $( change the status of a device
                                      and possibly send an interrupt)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_interrupt(u, int);
    EFFECTS
       'h_status(u) = stat;
        int
       => EFFECTS_OF set_interrupt(h_device_processor(u),
                                   h_device_interrupt(u));

END_MODULE
```



```
MODULE system_process
    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
address:
STRUCT_OF(capability add_cap; INTEGER add_offset)  $( address of a
                                                      word of storage);
state:
STRUCT_OF(capability inst_class_reg;
          VECTOR_OF machine_word general_reg;
          VECTOR_OF address address_reg) $( state of system process) ;

    DECLARATIONS

BOOLEAN b;
VECTOR_OF BOOLEAN bv;
capability pc, c;
slave_capability up, u;
state st, old_state, new_state;
INTEGER i;

    PARAMETERS

BOOLEAN h_sproc_exists(up) $( true if system process up exists) ;

    DEFINITIONS

BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN no_sproc(up) IS h_sproc_exists(up) ~= TRUE;
BOOLEAN bad_state(st)
   IS LENGTH(st.general_reg) ~= n_general_registers
     OR LENGTH(st.address_reg) ~= n_address_registers;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, modify $( access rights) ;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;

    FROM registers:
INTEGER n_general_registers,
        n_address_registers $( number of registers of each type) ;
VFUN get_registers(pc) -> st;
OFUN set_registers(pc; st);

    FUNCTIONS

VFUN h_user_level(up) -> b;
     $( true if system process up is executing user program )
    HIDDEN;
    INITIALLY
       b = TRUE;
```

```
OFUN indicate_user_level(b)[pc];
     $( is system process pc executing user program)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EFFECTS
       'h_user_level(up) = b;

OVFUN instantiate_sprocess(pc; new_state) -> old_state;
         $( bind new user process to system process pc)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, read);
       no_ability(pc, modify);
       no_sproc(up);
       bad_state(new_state);
    DELAY UNTIL h_user_level(up);
    EFFECTS
       old_state = get_registers(pc);
       EFFECTS_OF set_registers(pc, new_state);

END_MODULE
```



```
MODULE system_invoke

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
offset: {INTEGER i | 0 <= i AND i <= max_offset};
address:
STRUCT_OF(capability add_cap; offset add_offset)  $( address of a
                                                  word of storage);

    PARAMETERS

INTEGER callable $( access right) ;
address h_function_address(slave_capability u)
        $( addresses of all system functions);
address h_stack_address(slave_capability u, up)
        $( stacks for all system functions );
address high_level_return $(address of instruction that transfers
        to return location of nonresident procedure );
address low_level_return $(address of instruction that transfers
        to return location of resident procedure );

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i) IS get_access(c)[i] ~= TRUE;
BOOLEAN bad_call(slave_capability u; INTEGER o)
   IS h_function_address(u) = ?;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
INTEGER read $( access rights) ;
capability minimum_instructions $( capability for executing the
                                   smallest instruction class);
capability resident_instructions $( capability for interpreting
            addresses as resident memory addresses );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
VFUN get_slave(capability c) -> slave_capability u;
VFUN get_access(capability c) -> VECTOR_OF BOOLEAN bv;

    FROM registers:
INTEGER max_offset $( maximum offset allowed in a register) ,
        stack_register, arguments_register, program_counter
         $( index of address registers used for specific purpose );
OFUN load_address_register(INTEGER i; address a)[capability pc];
OFUN load_inst_class_register(capability c)[capability pc];
```

APPENDIX B SYSTEM INVOKE

```
    FUNCTIONS
OVFUN system_call(address procedure_address, return_address,
                  arguments, top_of_stack)
                [capability pc]
                -> address return_info;
     $( invoke a system procedure )
    DEFINITIONS
       slave_capability u IS get_slave(procedure_address.add_cap);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(procedure_address.add_cap, callable);
       bad_call(u, procedure_address.add_offset);
    EFFECTS
       return_info = return_address;
       EFFECTS_OF load_address_register
                  (arguments_register, arguments, pc);
       EFFECTS_OF load_address_register
                  (program_counter, h_function_address(u), pc);
       IF predefined_slave(h_stack_address(u, up).add_cap)
          = memory_block_slave
        THEN EFFECTS_OF load_inst_class_register
                  (resident_instructions, pc)
        ELSE EFFECTS_OF load_inst_class_register
                  (minimum_instructions, pc);
       IF no_ability(procedure_address.add_cap, read)
        THEN EFFECTS_OF load_address_register
                  (stack_register, h_stack_address(u, up), pc)
        ELSE EFFECTS_OF load_address_register
                  (stack_register, top_of_stack, pc);
OVFUN system_return(address return_info)[capability pc]
                -> address return_address;
     $( Return to calling activation )
    EFFECTS
       return_address = return_info;
       IF predefined_slave(return_info) = memory_block_slave
        THEN EFFECTS_OF load_inst_class_register
                  (resident_instructions, pc)
             AND EFFECTS_OF load_address_register
                        (program_counter,
                         low_level_return,
                         pc)
        ELSE EFFECTS_OF load_inst_class_register
                  (minimum_instructions, pc)
             AND EFFECTS_OF load_address_register
                        (program_counter,
                         high_level_return,
                         pc);

END_MODULE
```


APPENDIX B COORDINATOR

```
MODULE coordinator

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };

    PARAMETERS

INTEGER wait, wakeup $( access rights);

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] ~= TRUE;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
VFUN get_slave(capability c) -> slave_capability u;
VFUN get_access(capability c) -> VECTOR_OF BOOLEAN bv;

    FUNCTIONS

VFUN h_waiting_for(slave_capability u) -> slave_capability p;
     $( given process u is waiting for wakeup from process p)
    HIDDEN;
    INITIALLY
       p = ?;

OFUN queue_for_wakeup(capability p)[capability pc];
     $( place this process on list of processes waiting for
        wakeup from process p)
    DEFINITIONS
       slave_capability u IS get_slave(pc);
    EXCEPTIONS
       no_ability(p, wait);
       RESOURCE_ERROR;
    EFFECTS
        'h_waiting_for(u) = get_slave(p);

OFUN wait_for_wakeup()[capability pc]; $( wait for a wakeup )
    DEFINITIONS
       slave_capability u IS get_slave(pc);
    DELAY UNTIL h_waiting_for(u) = ?;

OFUN wakeup_waiters(BOOLEAN all)[capability pc];
     $( wake up one or more waiting processes )
    DEFINITIONS
       slave_capability u IS get_slave(pc);
    EFFECTS
       IF all
        THEN FORALL slave_capability up | h_waiting_for(up) = u:
                'h_waiting_for(up) = ?
        ELSE LET slave_capability un | h_waiting_for(un) = u
                IN FORALL slave_capability up | h_waiting_for(up) = u:
```

```
                IF  up = un
                 THEN 'h_waiting_for(up) = ?
                 ELSE 'h_waiting_for(up) = un;

OFUN wakeup_process(capability p);
     $( wakeup process p)
    DEFINITIONS
       slave_capability u IS get_slave(p);
    EXCEPTIONS
       no_ability(p, wakeup);
    EFFECTS
       'h_waiting_for(u) = ?;

END_MODULE
```



```
MODULE timers

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };

    DECLARATIONS

VECTOR_OF BOOLEAN bv;
capability c, pc;
slave_capability u, up;
INTEGER i;
INTEGER time $( clock time) ;

    PARAMETERS

INTEGER start_up_time $( time of clock initialization) ;

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] = FALSE;
BOOLEAN not_clock(slave_capability u)
   IS predefined_slave(u) ~= clock_slave;
BOOLEAN negative(INTEGER i) IS i < 0;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, modify $( access rights) ;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
             predefined slave capability );
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;

    FROM coordinator:
OFUN wakeup_process(capability pc);

    FUNCTIONS

VFUN h_clock() -> time; $( real clock time)
    HIDDEN;
    INITIALLY
       time = start_up_time;

VFUN h_process_time(up) -> time;  $( time process has spent
                                     executing)
    HIDDEN;
    INITIALLY
       time = 0;
```

```
VFUN h_clock_timer(up) -> time;  $( time when V operation to be
                                    done)
    HIDDEN;
    INITIALLY
       time = ?;

VFUN h_process_timer(up) -> time;  $( V operation to be performed
                                      when process up uses time)
    HIDDEN;
    INITIALLY
       time = ?;

VFUN read_clock(c) -> time; $( real time clock)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       not_clock(u);
    DERIVATION
       h_clock();

VFUN read_process_timer()[pc] -> time;  $( returns time used by
                                           process pc)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    DERIVATION
       h_process_timer(up);

OFUN update_clock(c; i); $( maintains real time clock)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, modify);
       not_clock(u);
       negative(i);
    EFFECTS
       'h_clock() = h_clock() + i;
       FORALL up:
            h_clock_timer(up) ~= ?
          AND h_clock_timer(up) <= h_clock() + i
         => 'h_clock_timer(up) = ?
           AND EFFECTS_OF wakeup_process(up);

OFUN update_process_timer(pc; i); $( maintains process timer)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, modify);
       negative(i);
    EFFECTS
       'h_process_time(up) = h_process_time(up) + i;
        h_process_timer(up) ~= ?
       AND h_process_timer(up) <= h_process_time(up) + i
       => 'h_process_timer(up) = ?
         AND EFFECTS_OF wakeup_process(up);
```



```
OFUN set_clock_timer(i)[pc];  $( set interval clock_timer
                                 value)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       negative(i);
    EFFECTS
       'h_clock_timer(up) = h_clock() + i;

OFUN set_process_timer(i)[pc]; $( set process timer value)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       negative(i);
    EFFECTS
       'h_process_timer(up) = h_process_time(up) + i;

END_MODULE
```

```
MODULE system_io

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);

    DECLARATIONS

BOOLEAN b, wake;
VECTOR_OF BOOLEAN bv;
slave_capability u;
capability c;
capability d $( capability for an I/O device) ;
INTEGER stat, comm;
machine_word data;

    PARAMETERS

INTEGER control,
       device  $( each of these assumes a different value from 1 to
                  access_length);

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] = FALSE;
BOOLEAN not_device(slave_capability u)
   IS h_device_set(u) = FALSE;
BOOLEAN no_input(slave_capability u) IS h_input(u) = ?;
BOOLEAN no_output(slave_capability u) IS h_output(u) = ?;
BOOLEAN no_command(slave_capability u) IS h_command(u) = ?;
BOOLEAN too_much_input(slave_capability u) IS h_input(u) ~= ?;
BOOLEAN too_much_output(slave_capability u) IS h_output(u) ~= ?;
BOOLEAN too_many_commands(slave_capability u)
   IS h_command(u) ~= ?;
BOOLEAN uninitialized_device(slave_capability u)
   IS h_status(u) = ?;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write $( access rights) ;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
            predefined slave capability );
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;
```



```
    FROM coordinator:
INTEGER wakeup $( access right );
OFUN wakeup_process(capability pc);

    FUNCTIONS

VFUN h_device_set(u) -> b; $( true if uid u designates a device)
    HIDDEN;
    INITIALLY
       b =(predefined_slave(u) = device_slave);

VFUN h_device_process(u) -> capability pc;
     $( process capability for device with uid u)
    HIDDEN;
    INITIALLY
       pc = ?;

VFUN h_input(u) -> data;  $( one word of input data from I/O device
                           with uid u)
    HIDDEN;
    INITIALLY
       data = ?;

OVFUN read_device(d) -> data; $( visible form of h_input)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       not_device(u);
       no_input(u);
    EFFECTS
       data = h_input(u);
       'h_input(u) = ?;

OFUN write_device(d; data);  $( output data to I/O device with
                              capability d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, write);
       not_device(u);
       too_much_output(u);
    EFFECTS
       'h_output(u) = data;

OFUN send_command(d; comm); $( send command to I/O device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, control);
       not_device(u);
       too_many_commands(u);
    EFFECTS
       'h_command(u) = comm;
```

```
VFUN h_status(u) -> stat; $( status of I/O device with uid u)
    HIDDEN;
    INITIALLY
       stat = ?;

VFUN receive_status(d) -> stat; $( visible form of h_status)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       not_device(u);
       uninitialized_device(u);
    DERIVATION
       h_status(u);

VFUN h_output(u) -> data;  $( data to be output by I/O device with
                              uid u)
    HIDDEN;
    INITIALLY
       data = ?;

OFUN set_device_process(d; capability pc);  $( indicate a process for
                                               device d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, control);
       no_ability(pc, wakeup);
       not_device(u);
    EFFECTS
       'h_device_process(u) = pc;

OVFUN device_receive(d) -> data; $( visible form of h_output)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_output(u);
    EFFECTS
       data = h_output(u);
       'h_output(u) = ?;

OFUN device_send(d; data); $( input data to system)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       too_much_input(u);
    EFFECTS
       'h_input(u) = data;
```



```
VFUN h_command(u) -> comm; $( command for I/O device with uid u)
    HIDDEN;
    INITIALLY
       comm = ?;

OVFUN device_command(d) -> comm; $( gets a command for a device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_command(u);
    EFFECTS
       comm = h_command(u);
       'h_command(u) = ?;

OFUN change_status(d; stat; wake);  $( change the status of a
                                       device and possibly send an
                                       interrupt)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
    EFFECTS
       'h_status(u) = stat;
        wake AND h_device_process(u) ~= ?
       => EFFECTS_OF wakeup_process(h_device_process(u));

END_MODULE
```

```
MODULE pages
    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);

    DECLARATIONS

BOOLEAN b;
VECTOR_OF BOOLEAN bv;
slave_capability u $( unique identifier of a page) ;
INTEGER i;
machine_word w;
capability c $( capability for a page) ;

    PARAMETERS

INTEGER max_pages $( maximum number of pages in system) ,
       page_size $( number of words in a page) ;

    DEFINITIONS

INTEGER npages IS CARDINALITY({ u | h_page_exists(u) = TRUE });
BOOLEAN no_ability(c; i) IS get_access(c)[i] = FALSE;
BOOLEAN no_page(u) IS h_page_exists(u) = FALSE;
BOOLEAN address_bounds(i) IS i < 0 OR i > page_size;
BOOLEAN too_many() IS npages >= max_pages;
BOOLEAN not_integer(w)
   IS TYPECASE w OF
       INTEGER: FALSE;
       capability: TRUE;
       BOOLEAN: TRUE;
       CHAR: TRUE;
     END;
INTEGER integer_value(w)
   IS TYPECASE w OF
       INTEGER: w;
       capability: 0;
       BOOLEAN: 0;
       CHAR: 0;
     END;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write, delete $( access rights) ;
machine_word zeroword $( initial value of words of storage) ;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;
OVFUN create_capability() -> c;
```



```
    FUNCTIONS
VFUN h_page_exists(u) -> b;  $( true for uid's of currently
                              existing pages)
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN h_page_read(u; i) -> w; $( returns the ith element of a page)
    HIDDEN;
    INITIALLY
       w = ?;

VFUN read_page(c; i) -> w; $( external form of h_page_read)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_page(u);
       address_bounds(i);
       no_ability(c, read);
    DERIVATION
       h_page_read(u, i);

OFUN write_page(c; i; w);  $( writes machine word w into ith
                            location of page)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, write);
       no_page(u);
       address_bounds(i);
    EFFECTS
       'h_page_read(u, i) = w;

OVFUN conditional_write(c; i; w) -> BOOLEAN written;
      $( modify word if contents were zero )
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_page(u);
       address_bounds(i);
    EFFECTS
       IF h_page_read(u, i) = 0
        THEN written AND 'h_page_read(u, i) = w
        ELSE ~written;

OVFUN page_decrement_and_test(c; i) -> b;  $( try a P operation on
                                           a specified word)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_page(u);
       address_bounds(i);
       not_integer(h_page_read(u, i));
```

```
    EFFECTS
       b =(integer_value(h_page_read(u, i)) < 1);
       'h_page_read(u, i) = integer_value(h_page_read(u, i)) - 1;

OVFUN page_increment_and_test(c; i) -> b;  $( do a V operation on a
                                            specified word)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, read);
       no_ability(c, write);
       no_page(u);
       address_bounds(i);
       not_integer(h_page_read(u, i));
    EFFECTS
       b =(integer_value(h_page_read(u, i)) < 0);
       'h_page_read(u, i) = integer_value(h_page_read(u, i)) + 1;

OVFUN create_page() -> c; $( create a new page)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       too_many();
    EFFECTS
       c = EFFECTS_OF create_capability();
       'h_page_exists(u) = TRUE;
       FORALL INTEGER i INSET { INTEGER j | j >= 0
                                          AND j <= page_size }:
          'h_page_read(u, i) = zeroword;

OFUN delete_page(c); $( delete a page)
    DEFINITIONS
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(c, delete);
       no_page(u);
    EFFECTS
       'h_page_exists(u) = FALSE;
       FORALL INTEGER i INSET { INTEGER j | j >= 0
                                          AND j <= page_size }:
          'h_page_read(u, i) = ?;

END_MODULE
```



```
MODULE segments

    TYPES

access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };
character_string: VECTOR_OF CHAR;

    PARAMETERS

INTEGER max_seg_size $( MAXIMUM SEGMENT SIZE) ;

    DEFINITIONS

BOOLEAN no_seg(slave_capability u) IS h_seg_exists(u) = FALSE;
BOOLEAN address_bounds(slave_capability u; INTEGER i)
   IS h_read(u, i) = ?;
BOOLEAN bad_size(INTEGER i) IS i < 0 OR i > max_seg_size;
BOOLEAN no_ability(capability s; INTEGER i)
   IS NOT get_access(s)[i] = TRUE;
character_string type_id(machine_word w)
   IS TYPECASE w OF
       capability:"capability";
       INTEGER:"INTEGER";
       BOOLEAN:"BOOLEAN";
       CHAR:"CHAR";
     END;
BOOLEAN not_integer(machine_word w)
   IS NOT type_id(w) ="INTEGER";
BOOLEAN not_writeable(slave_capability u; machine_word w)
   IS TYPECASE w OF
       capability:
          EXISTS INTEGER i | 0 < i AND i < n_store_permissions:
             ~ get_access(w)[i] AND ~ h_store_permits(u)[i];
       INTEGER: FALSE;
       BOOLEAN: FALSE;
       CHAR: FALSE;
     END;
INTEGER integer_value(machine_word w)
   IS TYPECASE w OF
       INTEGER: w;
       BOOLEAN: 0;
       CHAR: 0;
       capability: 0;
     END;
INTEGER seg_size(slave_capability u)
   IS CARDINALITY({INTEGER i | h_read(u, i) ~= ?});
```

```
    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER access_string_length $( number of potential access rights)
;
INTEGER n_store_permissions  $( number of store permissions must be
                               less than access string length);
INTEGER read, write, delete $( access rights) ;
machine_word zeroword $( initial value of a word of storage) ;
OVFUN create_restricted_cap(access_string as) -> capability s;
VFUN get_access(capability s) -> access_string as;
VFUN get_slave(capability s) -> capability u;

    FUNCTIONS

VFUN h_seg_exists(slave_capability u) -> BOOLEAN b;
        $( TRUE FOR UID'S OF CURRENTLY EXISTING SEGMENTS)
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN seg_exists(capability s) -> BOOLEAN b;  $( EXTERNAL FORM OF
                                               H_SEG_EXISTS)
    DERIVATION
       h_seg_exists(get_slave(s));

VFUN segment_size(capability s) -> INTEGER j;  $( EXTERNALFORM OF
                                                  H_SEG_SIZE U)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_seg(u);
    DERIVATION
       seg_size(u);

VFUN h_store_permits(slave_capability u) -> permit_string ps;
        $( store permits for segment with slave u)
    HIDDEN;
    INITIALLY
       ps = ?;

VFUN h_read(slave_capability u; INTEGER i) -> machine_word w;
        $( RETURNS THE ITH ELEMENT)
    HIDDEN;
    INITIALLY
       w = ?;

VFUN h_procedure(slave_capability u) -> capability s;
     $( capability for segment with slave u )
    HIDDEN;
    INITIALLY
       s = ?;
```



```
VFUN read_segment(capability s; INTEGER i) -> machine_word w;
        $( EXTERNAL FORM OF HREAD)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, read);
       no_seg(u);
       address_bounds(u, i);
    DERIVATION
       h_read(u, i);

OVFUN segment_create(permit_string ps) -> capability s;
        $( CREATES A NEW SEGMENT OF SIZE J)
    DEFINITIONS
       slave_capability u IS get_slave(s);
       access_string as
          IS VECTOR(FOR i FROM 1 TO access_string_length
                    : IF i <= n_store_permissions
                       THEN ~ ps[i]
                       ELSE TRUE);
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       s = EFFECTS_OF create_restricted_cap(as);
       'h_seg_exists(u) = TRUE;
       'h_store_permits(u) = ps;
       'h_procedure(u) = s;

OFUN segment_delete(capability s); $( DELETES SEGMENT S)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, delete);
       no_seg(u);
    EFFECTS
       'h_seg_exists(u) = FALSE;
       FORALL INTEGER i: 'h_read(u, i) = ?;

OFUN write_segment(capability s; INTEGER i; machine_word w);
        $( WRITES MACHINE WORD W INTO THE ITH LOCATION OF SEGMENT
         S)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, write);
       no_seg(u);
       bad_size(i);
       not_writeable(u, w);
       RESOURCE_ERROR;
    EFFECTS
       'h_read(u, i) = w;
       seg_size(u) < i
        => (FORALL INTEGER j | seg_size(u)-1 < j AND j < i:
            'h_read(u, j) = zeroword);
```

```
OVFUN conditional_write(capability s; INTEGER i; machine_word w)
                    -> BOOLEAN written;
      $( modify contents of a word if its contents were zero )
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, read);
       no_ability(s, write);
       no_seg(u);
       address_bounds(u, i);
       not_writeable(u, w);
    EFFECTS
       IF h_read(u, i) = 0
        THEN written AND 'h_read(u, i) = w
        ELSE ~written;

OVFUN destructive_read(capability s; INTEGER i) -> machine_word w;
      $( Read a word from a segment and destroy the contents of
         the word )
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, read);
       no_ability(s, write);
       no_seg(u);
       address_bounds(u, i);
    EFFECTS
       w = h_read(u, i);
       IF i = seg_size(u)-1
        THEN 'h_read(u, i) = ?
        ELSE 'h_read(u, i) = zeroword;

OFUN truncate_segment(capability s; INTEGER i);
     $( Reduce the size of the given segment to the indicated size
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, write);
       no_seg(u);
       address_bounds(u, i-1);
    EFFECTS
       FORALL INTEGER j | j >= i: 'h_read(u, j) = ?;
OVFUN decrement_and_test(capability s; INTEGER i) -> BOOLEAN b;
         $( try a P operation on a specified word)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, read);
       no_ability(s, write);
       no_seg(u);
       address_bounds(u, i);
       not_integer(h_read(u, i));
    EFFECTS
       b =(integer_value(h_read(u, i)) < 1);
       'h_read(u, i) = integer_value(h_read(u, i)) - 1;
```



```
OVFUN increment_and_test(capability s; INTEGER i) -> BOOLEAN b;
         $( try a V operation on a specified word)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, read);
       no_ability(s, write);
       no_seg(u);
       address_bounds(u, i);
       not_integer(h_read(u, i));
    EFFECTS
       b =(integer_value(h_read(u, i)) < 0);
       'h_read(u, i) = integer_value(h_read(u, i)) + 1;

VFUN h_procedure_entries(slave_capability u) -> INTEGER i;
         $( the number of entries in procedure with slave u)
    HIDDEN;
    INITIALLY
       i = ?;

VFUN procedure_entries(capability p) -> INTEGER i;
         $( the number of entries in procedure with capability p)
    DEFINITIONS
       slave_capability u IS get_slave(p);
    EXCEPTIONS
       no_ability(p, read);
       no_seg(u);
    DERIVATION
       h_procedure_entries(u);

OFUN declare_procedure_entries(capability s; INTEGER i);
         $( declare a new procedure p with procedure segment s and
          i entries)
    DEFINITIONS
       slave_capability u IS get_slave(s);
    EXCEPTIONS
       no_ability(s, write);
       no_seg(u);
    EFFECTS
       'h_procedure_entries(u) = i;

END_MODULE
```

```
MODULE windows

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
offset: {INTEGER i | 0 <= i AND i <= max_offset};
address:
STRUCT_OF(capability add_cap; offset add_offset)  $( address of a
                                                    word of storage);

access_string:
{VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length};
permit_string:
{VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions};

    PARAMETERS

INTEGER max_depth $( the maximum depth of nesting windows );
INTEGER examine $( access right to examine base information of window );

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i) IS get_access(c)[i] ~= TRUE;
address get_base(slave_capability u; INTEGER o, mode, depth)
   IS IF depth = 0
       THEN ?
       ELSE IF h_seg_exists(u)
             THEN IF h_read(u, o) = ?
                   THEN ?
                   ELSE STRUCT(u, o)
             ELSE IF h_base_object(u) = ?
                   THEN ?
                   ELSE IF    h_window_length(u) <= o
                           OR no_ability(h_base_object(u), mode)
                         THEN ?
                         ELSE get_base(get_slave(h_base_object(u)),
                                       h_base_offset(u) + o,
                                       mode,
                                       depth - 1);
BOOLEAN bad_permits(permit_string ps; machine_word w)
   IS TYPECASE w OF
       capability:
          EXISTS INTEGER i | 0 < i AND i < n_store_permissions:
           ~get_access(w)[i] AND ~ps[i];
       INTEGER: FALSE;
       BOOLEAN: FALSE;
       CHAR:    FALSE;
      END;
BOOLEAN no_window(slave_capability u)
   IS h_base_object(u) = ?;
```



```
    EXTERNALREFS
    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write, delete, modify $( access rights) ;
INTEGER n_store_permissions, access_string_length;
VFUN get_slave(capability c) -> slave_capability u;
VFUN get_access(capability c) -> VECTOR_OF BOOLEAN bv;
OVFUN create_restricted_cap(access_string as) -> capability c;

    FROM registers:
INTEGER max_offset $( maximum offset allowed in a register);

    FROM segments:
VFUN h_read(slave_capability u; INTEGER i) -> machine_word w;
VFUN h_seg_exists(slave_capability u) -> BOOLEAN b;
VFUN h_store_permits(slave_capability u) -> permit_string ps;
OFUN write_segment(capability s; INTEGER i; machine_word w);

    FUNCTIONS

VFUN h_base_object(slave_capability u) -> capability c;
     $( capability for object to which window points )
    HIDDEN;
    INITIALLY
       c = ?;

VFUN h_base_offset(slave_capability u) -> INTEGER o;
     $( offset of beginning of window in base object )
    HIDDEN;
    INITIALLY
       o = ?;

VFUN h_window_length(slave_capability u) -> INTEGER l;
     $( length of window )
    HIDDEN;
    INITIALLY
       l = ?;

VFUN h_window_permits(slave_capability u) -> permit_string ps;
     $( store permits for window u)
    HIDDEN;
    INITIALLY
       ps = ?;

VFUN read_window(capability s; offset o) -> machine_word w;
     $( Read a word from a window )
    DEFINITIONS
       slave_capability u IS get_slave(s);
       address base_address IS get_base(u, o, read, max_depth);
    EXCEPTIONS
       no_ability(s, read);
       base_address = ?;
    DERIVATION
       h_read(base_address.add_cap, base_address.add_offset);
```

```
OFUN write_window(capability s; offset o; machine_word w);
     $( Modify a word in a window )
    DEFINITIONS
       slave_capability u IS get_slave(s);
       address base_address IS get_base(u, o, write, max_depth);
    EXCEPTIONS
       no_ability(s, write);
       base_address = ?;
       bad_permits(h_store_permits(base_address.add_cap), w);
    EFFECTS
       EFFECTS_OF write_segment(base_address.add_cap,
                                base_address.add_offset,
                                w);

OVFUN window_create(capability base_object; offset base_offset;
                    offset window_length; permit_string ps)
                    -> capability window;
      $( create a new window )
    DEFINITIONS
       slave_capability u IS get_slave(window);
       access_string as
          IS VECTOR(FOR i FROM 1 TO access_string_length:
                    IF i <= n_store_permissions
                     THEN ps[i]
                     ELSE TRUE);
    EXCEPTIONS
       bad_permits(ps, base_object);
       RESOURCE_ERROR;
    EFFECTS
       window = EFFECTS_OF create_restricted_cap(as);
       'h_base_object(u) = base_object;
       'h_base_offset(u) = base_offset;
       'h_window_length (u) = window_length;
       'h_window_permits(u) = ps;

OFUN window_delete(capability window);
     $( delete an existing window )
    DEFINITIONS
       slave_capability u IS get_slave(window);
    EXCEPTIONS
       no_ability(window, delete);
       no_window(u);
    EFFECTS
       'h_base_object(u) = ?;
       'h_base_offset(u) = ?;
       'h_window_length(u) = ?;

VFUN examine_window(capability window)
      -> STRUCT_OF(capability bo; offset bof; offset l) window_info;
     $( get data about a window )
    DEFINITIONS
       slave_capability u IS get_slave(window);
    EXCEPTIONS
       no_ability(window, examine);
       no_window(u);
    DERIVATION
       STRUCT(h_base_object(u), h_base_offset(u), h_window_length(u));
```



```
OFUN modify_window(capability window; capability base_object;
                   offset base_offset; offset window_length);
     $( change the meaning of a window )
    DEFINITIONS
       slave_capability u IS get_slave(window);
    EXCEPTIONS
       no_ability(window, modify);
       no_window(u);
       bad_permits(h_window_permits(u), base_object);
    EFFECTS
       'h_base_object(u) = base_object;
       'h_base_offset(u) = base_offset;
       'h_window_length(u) = window_length;

END_MODULE
```

```
MODULE extended_types
    TYPES

access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };

    DECLARATIONS

slave_capability ut, u;
capability t;
BOOLEAN b;

    PARAMETERS

BOOLEAN type_type_slave(ut) $( true if ut is for type type),
        segment_type_slave(ut) $(true if ut is for segment type);
INTEGER max_impl_caps  $( maximum number of implementation
                       capabilities in the representation of an
                       object);
INTEGER create, manage, interrogate $( access rights) ,
        add_rep, delete_rep;

    DEFINITIONS

BOOLEAN bad_type_manager(ut) IS ~ h_is_type_manager(ut);
BOOLEAN invalid_object(slave_capability u, ul)
   IS ~ h_is_type(u, ul);
BOOLEAN invalid_impl_object(slave_capability u)
   IS ~ h_is_type_manager(u) AND ~segment_type_slave(u);
BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] ~= TRUE;
BOOLEAN orig_impl_objs_present(u)
   IS EXISTS INTEGER i: h_original(u)[i] = TRUE;
BOOLEAN bad_element(INTEGER i; ut)
   IS i < 1 OR i > h_impl_length(ut);
BOOLEAN exc_original(u; INTEGER i) IS h_original(u)[i] = TRUE;
BOOLEAN not_original(u; INTEGER i) IS h_original(u)[i] = FALSE;
BOOLEAN object_there_already(u; INTEGER i)
   IS h_impl_cap(u)[i] ~= ?;
BOOLEAN no_impl_obj(u; INTEGER i) IS h_impl_cap(u)[i] = ?;
access_string no_delete_as
   IS VECTOR(FOR i FROM 1 TO access_string_length
             : IF i = delete THEN FALSE ELSE TRUE);
BOOLEAN not_writeable(u; capability c)
   IS EXISTS INTEGER i | 0 < i AND i <= n_store_permissions:
       ~ get_access(c)[i] AND ~ h_permits(u)[i];
VECTOR_OF BOOLEAN insert_boolean(VECTOR_OF BOOLEAN bv;
                                 INTEGER i; BOOLEAN b)
   IS VECTOR(FOR j FROM 1
             TO(IF LENGTH(bv) >= i THEN LENGTH(bv) ELSE i):
            IF j = i THEN b
              ELSE IF j <= LENGTH(bv) THEN bv[j] ELSE ?)
```



```
          $( insert an element into a boolean vector);
VECTOR_OF capability insert_capability(VECTOR_OF capability cv;
                                       INTEGER i; capability c)
   IS VECTOR(FOR j FROM 1
             TO(IF LENGTH(cv) >= i THEN LENGTH(cv) ELSE i):
            IF j = i THEN c
              ELSE IF j <= LENGTH(cv) THEN cv[j] ELSE ?)
          $( insert an element into capability vector);

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER delete $( access right) ;
INTEGER access_string_length $( potential number of access rights);
INTEGER n_store_permissions $( number of store limitations) ;
OVFUN create_capability() -> capability s;
OVFUN create_restricted_cap(permit_string ps) -> capability s;
VFUN get_access(capability s) -> access_string as;
VFUN restrict_access(capability s; access_string as)
                    -> capability sl;
VFUN get_slave(capability s) -> slave_capability sl;

    FROM segments:
OVFUN segment_create(permit_string ps) -> capability s;
OFUN segment_delete(capability s);
VFUN h_seg_exists(slave_capability u) -> BOOLEAN b;

    FUNCTIONS

VFUN h_is_type_manager(ut) -> b; $( true if ut is a type manager)
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN is_type_manager(t) -> b;  $( visible version of
                                  h_is_type_manager)
    EXCEPTIONS
       no_ability(t, interrogate);
    DERIVATION
       h_is_type_manager(get_slave(t));

VFUN h_is_type(u; ut) -> b;  $( true iff the type of extended-type
                               object u is ut)
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN is_type(capability c; capability t) -> b;  $( external form of
                                                   h_is_type)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability ul IS get_slave(c);
    EXCEPTIONS
       no_ability(t, interrogate);
    DERIVATION
       h_is_type(ul, ut);
```

```
VFUN h_impl_cap(u) -> VECTOR_OF capability cv;
        $( RETURNS THE TUPLE OF CAPABILITIES IMPLEMENTING THE
         EXTENDED-TYPE OBJECT U)
    HIDDEN;
    INITIALLY
       cv = ?;

VFUN impl_cap(capability c2; capability t; INTEGER i)
          -> capability c1;  $( EXTERNAL FORM OF H_IMPL_CAP U --
                                T IS THE TYPE manageR'S capability)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c2);
    EXCEPTIONS
       no_ability(t, manage);
       invalid_object(u, ut);
       bad_element(i, ut);
       no_impl_obj(u, i);
    DERIVATION
       IF h_original(u)[i]
         THEN restrict_access(h_impl_cap(u)[i], no_delete_as)
         ELSE h_impl_cap(u)[i];

VFUN h_impl_length(ut) -> INTEGER i;  $( returns the length of the
                                        vector of implementation
                                        capabilities)
    HIDDEN;
    INITIALLY
       i = ?;

VFUN impl_length(t) -> INTEGER i;  $( visible version of
                                      h_impl_length)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
    EXCEPTIONS
       no_ability(t, interrogate);
       bad_type_manager(ut);
    DERIVATION
       h_impl_length(ut);

VFUN h_original(u) -> VECTOR_OF BOOLEAN bv;
        $( states whether a representation object has been
         created with a create_impl_object call on the
         extended-type object u)
    HIDDEN;
    INITIALLY
       bv = ?;

VFUN original(capability c; capability t; INTEGER i) -> BOOLEAN b;
        $( external form of h_original)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(t, manage);
       invalid_object(u, ut);
       bad_element(i, ut);
```



```
       no_impl_obj(u, i);
    DERIVATION
       h_original(u)[i];

VFUN h_permits(u) -> permit_string ps;  $( store limitations for
                                          object with slave u and
                                          type ut)
    HIDDEN;
    INITIALLY
       ps = ?;

OVFUN create_type(capability t; INTEGER i) -> capability c;
        $( CREATES an extended type type)
    EXCEPTIONS
       ~type_type_slave(get_slave(t));
       i < 1 OR i > max_impl_caps;
       no_ability(t, create);
       RESOURCE_ERROR;
    EFFECTS
       c = EFFECTS_OF create_capability();
       'h_is_type_manager(get_slave(c)) = TRUE;
       'h_impl_length(get_slave(c)) = i;

OVFUN object_create(capability t; permit_string ps)
                  -> capability c;
          $( CREATES AN EXTENDED-TYPE OBJECT OF TYPE THE SAME AS
           GET_UID T, RETURNS capability C, AND LEAVES THE OBJECT
           UNINITIALIZED)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(t, create);
       bad_type_manager(ut);
       RESOURCE_ERROR;
    EFFECTS
        c = EFFECTS_OF create_restricted_cap(
             VECTOR( FOR i FROM 1 TO access_string_length:
               IF i <= n_store_permissions THEN ~ ps[i] ELSE TRUE));
       'h_is_type(u, ut) = TRUE;
       'h_original(u) = VECTOR();
       'h_impl_cap(u) = VECTOR();
       'h_permits(u) = ps;

OFUN object_delete(capability c, t);
          $( deleteS AN OBJECT FROM THE EXTENDED-TYPE LEVEL, ITS
           ORIGINAL REPRESENTATION OBJECTS HAVING PREVIOUSLY BEEN
           deleteD;)
    DEFINITIONS
       slave_capability u IS get_slave(c);
       slave_capability ut IS get_slave(t);
    EXCEPTIONS
       no_ability(t, delete);
       invalid_object(u, ut);
       orig_impl_objs_present(u);
    EFFECTS
       'h_is_type(u, ut) = FALSE;
```

```
       'h_impl_cap(u) = ?;
       'h_original(u) = ?;

OVFUN create_impl_obj(capability c; capability t;
                      INTEGER i; capability tl) -> capability cl;
          $( creates an implementation object at position i of
           implementation vector for object c. tl is the
           type-creation capability for the type of the object
           being created)
    DEFINITIONS
       slave_capability ul IS get_slave(tl);
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(t, manage);
       no_ability(c, add_rep);
       invalid_object(u, ut);
       bad_element(i, ut);
       object_there_already(u, i);
       invalid_impl_object(ul);
       no_ability(tl, create);
       RESOURCE_ERROR;
    EFFECTS
       EXISTS capability c2:
             (IF segment_type_slave(ul)
                THEN c2 = EFFECTS_OF segment_create(h_permits(u))
                ELSE(c2 = EFFECTS_OF create_restricted_cap(
                   VECTOR(FOR i FROM 1 TO access_string_length:
                            IF i <= n_store_permissions
                             THEN ~ h_permits(u)[i] ELSE TRUE))
                          AND 'h_is_type(get_slave(c2), ul) = TRUE
                         AND 'h_original(get_slave(c2)) = VECTOR()
                        AND 'h_impl_cap(get_slave(c2)) = VECTOR()
                       AND 'h_permits(get_slave(c2)) = h_permits(u)))
           AND( 'h_impl_cap(u)
              = insert_capability(h_impl_cap(u), i, c2))
          AND( 'h_original(u)
             = insert_boolean(h_original(u), i, TRUE))
        AND(cl = restrict_access(c2, no_delete_as));

OFUN insert_impl_obj(capability c; capability t;
                     INTEGER i; capability cl);
          $( inserts the capability cl to be get_impl_cap (c,t) (i))
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(t, manage);
       no_ability(c, add_rep);
       invalid_object(u, ut);
       bad_element(i, ut);
       object_there_already(u, i);
       not_writeable(u, cl);
    EFFECTS
        'h_impl_cap(u) = insert_capability(h_impl_cap(u), i, cl);
        'h_original(u) = insert_boolean(h_original(u), i, FALSE);
```



```
OFUN delete_impl_cap(capability c; capability t; INTEGER i);
          $( deletes the ith element of the implementation vector
           of c, but only if the object is not original)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
    EXCEPTIONS
       no_ability(t, manage);
       no_ability(c, delete_rep);
       invalid_object(u, ut);
       bad_element(i, ut);
       no_impl_obj(u, i);
       exc_original(u, i);
    EFFECTS
       'h_impl_cap(u) = insert_capability(h_impl_cap(u), i, ?);
       'h_original(u) = insert_boolean(h_original(u), i, ?);

OFUN delete_impl_obj(capability c; capability t;
                     INTEGER i; capability tl);
          $( deletes the object and entry for the ith
           implementation object of capability c. object must be
           original)
    DEFINITIONS
       slave_capability ut IS get_slave(t);
       slave_capability u IS get_slave(c);
       slave_capability x IS h_impl_cap(u)[i];
       slave_capability ux IS get_slave(x);
       slave_capability utl IS get_slave(tl);
    EXCEPTIONS
       no_ability(t, manage);
       no_ability(c, delete_rep);
       invalid_object(u, ut);
       bad_element(i, ut);
       no_impl_obj(u, i);
       not_original(u, i);
       invalid_object(ux, utl);
       orig_impl_objs_present(ux);
    EFFECTS
       IF h_seg_exists(ux)
         THEN EFFECTS_OF segment_delete(x)
         ELSE(  'h_is_type(ux, utl) = FALSE
              AND 'h_impl_cap(ux) = ?
             AND 'h_original(ux) = ?);
       'h_impl_cap(u) = insert_capability(h_impl_cap(u), i, ?);
       'h_original(u) = insert_boolean(h_original(u), i, ?);

END_MODULE
```

```
MODULE directories

    TYPES

access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
character_string: VECTOR_OF CHAR;
machine_word: ONE_OF(capability, INTEGER, CHAR, BOOLEAN);
entry_name:
{ character_string cs | LENGTH(cs) <= entry_name_length };
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };

    PARAMETERS

INTEGER entry_name_length  $( number of machine words for an entry
                              name);
BOOLEAN root_slave(slave_capability u)
        $(true if u is for the root directory) ;
INTEGER remove, add_entries, load, list, add_locks;

    DEFINITIONS

BOOLEAN no_dir(slave_capability u) IS h_valid_dir(u) = FALSE;
BOOLEAN no_entry(slave_capability u; entry_name n)
   IS h_get_cap(u, n) = ?;
BOOLEAN no_ability(capability c; INTEGER i)
   IS NOT get_access(c)[i] = TRUE;
BOOLEAN name_used(slave_capability u; entry_name n)
   IS NOT h_get_cap(u, n) = ?;
BOOLEAN not_addable(slave_capability u; capability c)
   IS EXISTS INTEGER i | 0 < i AND i <= n_store_permissions:
        ~ get_access(c)[i] AND ~ h_dir_permits(u)[i];
BOOLEAN directory_occupied(slave_capability u)
   IS EXISTS entry_name n: h_get_cap(u, n) ~= ?;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER n_store_permissions $( number of store limitations) ;
INTEGER delete $( access right) ;
INTEGER access_string_length;
OVFUN create_restricted_cap(permit_string ps) -> capability c;
VFUN get_access(capability s) -> access_string as;
VFUN get_slave(capability s) -> slave_capability sl;

    FUNCTIONS

VFUN h_valid_dir(slave_capability u) -> BOOLEAN b;
        $( true for all directories currently in existence)
    HIDDEN;
    INITIALLY
       b = root_slave(u);
```



```
VFUN valid_dir(capability d) -> BOOLEAN b;  $( external form of
                                             h_valid_dir)
    DERIVATION
       h_valid_dir(get_slave(d));

VFUN h_get_cap(slave_capability u; entry_name n) -> capability c;
        $( returns the capability associated with the name n in
         the directory u)
    HIDDEN;
    INITIALLY
       c = ?;

VFUN h_locks(slave_capability u; entry_name n)
          -> SET_OF slave_capability ls;  $( set of locks for
                                             entry)
    HIDDEN;
    INITIALLY
       ls = { };

VFUN get_locks(capability d; entry_name n)
           -> SET_OF slave_capability ls;  $( returns locks for
                                              entry)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, load);
       no_dir(u);
       no_entry(u, n);
    DERIVATION
       h_locks(u, n);

VFUN get_cap(capability d; entry_name n; capability k)
          -> capability c; $( external form of h_get_cap)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, load);
       no_dir(u);
       no_entry(u, n);
       ~(get_slave(k) INSET h_locks(u, n));
    DERIVATION
       h_get_cap(u, n);

VFUN dir(capability d) -> SET_OF entry_name n;
    DEFINITIONS
       slave_capability u IS get_slave(d);
       SET_OF entry_name s
          IS { entry_name n | h_get_cap(u, n) ~= ? };
    EXCEPTIONS
       no_dir(u);
       no_ability(d, list);
    DERIVATION
       s;
```

```
VFUN dir_size(capability d) -> INTEGER i;  $( number of entries in
                                            directory d)
    EXCEPTIONS
       no_dir(get_slave(d));
    DERIVATION
       CARDINALITY({ entry_name n | NOT h_get_cap(get_slave(d), n)
                                       = ? });

VFUN h_distinguished(slave_capability u; entry_name n)
                    -> BOOLEAN b;  $( true if given entry is a
                                      distinguished entry)
    HIDDEN;
    INITIALLY
       b = ?;

VFUN h_dir_permits(slave_capability u) -> permit_string ps;
         $( store permits for directory with slave u)
    HIDDEN;
    INITIALLY
       ps =(IF root_slave(u)
             THEN VECTOR(FOR i FROM 1 TO n_store_permissions: FALSE)
             ELSE ?);

OVFUN directory_create(permit_string ps) -> capability d;
          $( returns the capability for an empty, newly created
           directory)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       d = EFFECTS_OF create_restricted_cap(
             VECTOR( FOR i FROM 1 TO access_string_length:
               IF i <= n_store_permissions THEN ~ ps[i] ELSE TRUE));
       'h_valid_dir(get_slave(d)) = TRUE;
       'h_dir_permits(get_slave(d)) = ps;

OFUN directory_delete(capability d); $( to delete directory d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, delete);
       no_dir(u);
       directory_occupied(u);
    EFFECTS
       'h_valid_dir(u) = FALSE;

OFUN add_entry(capability d; entry_name n; capability c);
         $( to add entry (n,c) to directory d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, add_entries);
       RESOURCE_ERROR;
       no_dir(u);
       name_used(u, n);
       not_addable(u, c);
    EFFECTS
       'h_get_cap(u, n) = c;
```



```
       'h_distinguished(u, n) = FALSE;

OFUN remove_entry(capability d; entry_name n);  $( to remove entry
                                                   n from directory d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_dir(u);
       no_entry(u, n);
       no_ability(d, remove);
       h_distinguished(u, n) = TRUE;
    EFFECTS
       'h_get_cap(u, n) = ?;

OFUN add_distinguished_entry(capability d; entry_name n;
                             capability c);
          $( add distinguished entry (n, c) to directory d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, add_entries);
       no_dir(u);
       name_used(u, n);
       not_addable(u, c);
       RESOURCE_ERROR;
    EFFECTS
       'h_get_cap(u, n) = c;
       'h_distinguished(u, n) = TRUE;

OFUN remove_distinguished_entry(capability d; entry_name n);
         $( remove distinguished entry n from directory d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, remove);
       no_dir(u);
       no_entry(u, n);
       h_distinguished(u, n) = FALSE;
    EFFECTS
       'h_get_cap(u, n) = ?;

OFUN add_lock(capability d; entry_name n; capability k);
          $( add lock to the entry)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, add_locks);
       no_dir(u);
       no_entry(u, n);
       not_addable(u, k);
       RESOURCE_ERROR;
    EFFECTS
       'h_locks(u, n) = h_locks(u, n) UNION { get_slave(k) };
```

```
OFUN remove_lock(capability d; entry_name n; capability k);
        $( remove lock k from the entry (d, n))
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, remove);
       no_dir(u);
       no_entry(u, n);
    EFFECTS
       'h_locks(u, n) = h_locks(u, n) DIFF { get_slave(k) };

END_MODULE
```



```
MODULE user_objects
    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
offset:
{INTEGER i | 0 <= i AND i <= max_offset};
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };
character_string: VECTOR_OF CHAR;
entry_name:
{ character_string n | LENGTH(n) <= entry_name_length };
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };

    DEFINITIONS

capability dummy_cap(permit_string ps)
   IS restrict_access(null,
                      VECTOR(FOR i FROM 1 TO access_string_length
                             : IF i <= n_store_permissions
                                THEN ~ ps[i]
                                ELSE TRUE));

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER n_store_permissions $( number of store limitations) ;
INTEGER access_string_length;
capability null $( capability for nothing) ;
VFUN get_slave(capability c) -> slave_capability u;
VFUN restrict_access(capability c; access_string as)
                     -> capability cl;

    FROM registers:
INTEGER max_offset;

    FROM segments:
OVFUN segment_create(permit_string ps) -> capability s;
OFUN segment_delete(capability s);

    FROM windows:
OVFUN window_create(capability base_object; offset base_offset;
                    offset window_length;permit_string ps)
                   -> capability window;
OFUN window_delete(capability window);

    FROM extended_types:
OVFUN object_create(capability t; permit_string ps)
                   -> capability c;
OFUN object_delete(capability c, t);
```

```
    FROM directories:
INTEGER entry_name_length  $( maximum number of characters in an
                              entry name);
VFUN h_get_cap(slave_capability u; entry_name n) -> capability c;
OVFUN directory_create(permit_string ps) -> capability d;
OFUN directory_delete(capability d);
OFUN add_distinguished_entry(capability d;
                             entry_name n;
                             capability c);
OFUN remove_distinguished_entry(capability d; entry_name n);

    FUNCTIONS

OVFUN create_segment(capability d; entry_name n;
                     permit_string ps) -> capability s;
        $( create a segment with a distinguished directory entry)
    EXCEPTIONS
       EXCEPTIONS_OF segment_create(ps);
       EXCEPTIONS_OF add_distinguished_entry(d, n, dummy_cap(ps));
    EFFECTS
       s = EFFECTS_OF segment_create(ps);
       EFFECTS_OF add_distinguished_entry(d, n, s);

OFUN delete_segment(capability d; entry_name n);
          $( delete a segment and its distinguished directory entry)
    EXCEPTIONS
       EXCEPTIONS_OF remove_distinguished_entry(d, n);
       EXCEPTIONS_OF segment_delete(h_get_cap(get_slave(d), n));
    EFFECTS
       EFFECTS_OF remove_distinguished_entry(d, n);
       EFFECTS_OF segment_delete(h_get_cap(get_slave(d), n));

OVFUN create_window(capability d; entry_name n;
                    capability base_object; offset base_offset;
                    offset window_length; permit_string ps)
                   -> capability window;
       $( create a window for some base object )
    EXCEPTIONS
       EXCEPTIONS_OF window_create(base_object,
                                   base_offset,
                                   window_length,
                                   ps);
       EXCEPTIONS_OF add_distinguished_entry(d, n, dummy_cap(ps));
    EFFECTS
       window = EFFECTS_OF window_create(base_object,
                                         base_offset,
                                         window_length,
                                         ps);
       EFFECTS_OF add_distinguished_entry(d, n, window);

OFUN delete_window(capability d; entry_name n; capability window);
     $( delete a window and its directory entry )
    EXCEPTIONS
       EXCEPTIONS_OF remove_distinguished_entry(d, n);
       EXCEPTIONS_OF window_delete(window);
    EFFECTS
       EFFECTS_OF remove_distinguished_entry(d, n);
```



```
       EFFECTS_OF window_delete(window);

OVFUN create_object(capability d; entry_name n; capability t;
                    permit_string ps) -> capability c;
        $( create an extended type object with a distinguished entry)
    EXCEPTIONS
       EXCEPTIONS_OF object_create(t, ps);
       EXCEPTIONS_OF add_distinguished_entry(d, n, dummy_cap(ps));
    EFFECTS
       c = EFFECTS_OF object_create(t, ps);
       EFFECTS_OF add_distinguished_entry(d, n, c);

OFUN delete_object(capability d; entry_name n; capability t);
          $( delete extended type object and its distinguished
           entry)
    EXCEPTIONS
       EXCEPTIONS_OF remove_distinguished_entry(d, n);
       EXCEPTIONS_OF object_delete(h_get_cap(get_slave(d), n), t);
    EFFECTS
       EFFECTS_OF remove_distinguished_entry(d, n);
       EFFECTS_OF object_delete(h_get_cap(get_slave(d), n), t);

OVFUN create_directory(capability d; entry_name n;
                       permit_string ps) -> capability dl;
        $( create a directory with a distinguished directory entry)
    EXCEPTIONS
       EXCEPTIONS_OF directory_create(ps);
       EXCEPTIONS_OF add_distinguished_entry(d, n, dummy_cap(ps));
    EFFECTS
       dl = EFFECTS_OF directory_create(ps);
       EFFECTS_OF add_distinguished_entry(d, n, dl);

OFUN delete_directory(capability d; entry_name n);
          $( delete directory and distinguished directory entry)
    EXCEPTIONS
       EXCEPTIONS_OF remove_distinguished_entry(d, n);
       EXCEPTIONS_OF directory_delete(h_get_cap(get_slave(d), n));
    EFFECTS
       EFFECTS_OF remove_distinguished_entry(d, n);
       EFFECTS_OF directory_delete(h_get_cap(get_slave(d), n));

END_MODULE
```

```
MODULE user_process

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
address:
STRUCT_OF(capability add_cap; INTEGER add_offset)  $( address of a
                                                    word of storage);

state:
STRUCT_OF(capability inst_class_reg;
          VECTOR_OF machine_word general_reg;
          VECTOR_OF address address_reg)  $( the state of a processor);
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };
character_string: VECTOR_OF CHAR;
entry_name:
{ character_string n | LENGTH(n) <= entry_name_length };
offset:
{INTEGER i | 0 <= i AND i <= max_offset};
    DECLARATIONS

BOOLEAN b;
INTEGER i, o;
capability c, pc, d, cl;
slave_capability u, up;
state st;
entry_name n;
access_string as;
permit_string ps;

    PARAMETERS

capability user_level $( level for user programs) ;
BOOLEAN initial_user_process(up)
         $(true if up is the process initially running),
        initial_activation_record(SET_OF capability sc)
          $(true if sc contains only the activation record
            for the initial user process),
        initial_proc_dir(d)
          $(true if d is process dir of initial user process) ;
access_string process_permissions
               $( access string for temporary segments and
                  windows);
    DEFINITIONS

BOOLEAN no_ability(c; i) IS get_access(c)[i] ~= TRUE;
BOOLEAN no_uproc(up) IS h_uproc_exists(up) ~= TRUE;
BOOLEAN not_suspended(up) IS h_uproc_suspended(up) ~= TRUE;
BOOLEAN bad_entry(u; o) IS o >= h_procedure_entries(u);
BOOLEAN not_temp_window(slave_capability u, up)
   IS ~(u INSET h_temp_windows(up));
BOOLEAN not_temp_seg(slave_capability u, up)
   IS ~(u INSET h_temp_segments(up));
```



```
    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, modify $( access rights) ;
INTEGER access_string_length;
INTEGER n_store_permissions;
capability null $( capability for nothing) ;
capability minimum_instructions;
machine_word zeroword;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> as;
OVFUN create_capability() -> c;
VFUN restrict_access(cl; as) -> c;

    FROM registers:
INTEGER n_address_registers,
      n_general_registers $( number of registers) ;
INTEGER program_counter, arguments_register, stack_register;
INTEGER max_offset  $( maximum possible offset in an offset
                    register);
OFUN set_registers(pc; st);

    FROM segments:
VFUN h_procedure_entries(slave_capability u) -> INTEGER i;
OVFUN segment_create(permit_string ps) -> capability s;
OFUN segment_delete(capability s);

    FROM windows:
OVFUN window_create(capability base_object; offset base_offset;
                    offset window_length; permit_string ps)
                   -> capability window;
OFUN window_delete(capability window);

    FROM directories:
INTEGER load $( access right) ;
INTEGER entry_name_length;
VFUN h_get_cap(u; n) -> c;
OVFUN directory_create(ps) -> d;
OFUN directory_delete(d);
OFUN add_distinguished_entry(d; n; c);
OFUN remove_distinguished_entry(d; n);

    FUNCTIONS

VFUN h_uproc_exists(up) -> b; $( true if process up exists)
    HIDDEN;
    INITIALLY
       b = initial_user_process(up);

VFUN h_uproc_suspended(up) -> b;  $( true if process up can not
                                     execute)
    HIDDEN;
    INITIALLY
       b =(IF initial_user_process(up) THEN FALSE ELSE TRUE);
```

```
VFUN h_temp_segments(up) -> SET_OF capability sc;
     $( all temporary segments for process up)
    HIDDEN;
    INITIALLY
       IF initial_user_process(up)
        THEN initial_activation_record(sc)
        ELSE sc = {};

VFUN h_temp_windows(up) -> SET_OF capability sc;
     $( all temporary windows for process up)
    HIDDEN;
    INITIALLY
       sc = {};

VFUN h_proc_dir(up) -> d;  $( capability for process directory of
                            process up)
    HIDDEN;
    INITIALLY
       IF initial_user_process(up)
        THEN initial_proc_dir(d) ELSE d=?;

OVFUN create_uproc(d; n; address init_proc, init_args) -> pc;
      $( create a new user process)
    DEFINITIONS
       slave_capability u IS get_slave(init_proc.add_cap);
       slave_capability up IS get_slave(pc);
       permit_string ps
          IS VECTOR(FOR i FROM 1 TO n_store_permissions: TRUE);
    EXCEPTIONS
       no_ability(init_proc.add_cap, read);
       bad_entry(u, init_proc.add_offset);
       EXCEPTIONS_OF add_distinguished_entry(d, n, null);
       RESOURCE_ERROR;
    EFFECTS
       pc = EFFECTS_OF create_capability();
       EFFECTS_OF add_distinguished_entry(d, n, pc);
       'h_uproc_exists(up) = TRUE;
       'h_proc_dir(up) = EFFECTS_OF directory_create(ps);
       EFFECTS_OF set_registers
                  (pc,
                   STRUCT(minimum_instructions,
                          VECTOR(FOR i FROM 1
                                    TO n_general_registers
                                 : zeroword),
                          VECTOR(FOR i FROM 1
                                    TO n_address_registers
                                 : IF i = program_counter
                                   THEN init_proc
                                   ELSE
                                    IF i = arguments_register
                                     THEN init_args
                                     ELSE
                                      IF i = stack_register
                                       THEN
                                        STRUCT
                                         (EFFECTS_OF
                                           create_temp_segment(pc),
```



```
                                            0)
                                       ELSE STRUCT(null, 0))));

OFUN delete_uproc(d; n); $( destroy a user process)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
       capability pc IS h_get_cap(get_slave(d), n);
    EXCEPTIONS
       EXCEPTIONS_OF remove_distinguished_entry(d, n);
       no_uproc(up);
       not_suspended(up);
       EXCEPTIONS_OF directory_delete(h_proc_dir(up));
    EFFECTS
       'h_uproc_exists(up) = FALSE;
       'h_proc_dir(up) = ?;
       FORALL capability c INSET h_temp_segments(up):
        EFFECTS_OF segment_delete(c);
       FORALL capability c INSET h_temp_windows(up):
        EFFECTS_OF window_delete(c);
       EFFECTS_OF directory_delete(h_proc_dir(up));
       EFFECTS_OF remove_distinguished_entry(d, n);

OFUN stop_uproc(pc); $( suspend execution of a user process)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, modify);
       no_uproc(up);
    EFFECTS
       'h_uproc_suspended(up) = TRUE;

OFUN start_uproc(pc); $( restart execution of a user process)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(pc, modify);
       no_uproc(up);
    EFFECTS
       'h_uproc_suspended(up) = FALSE;

VFUN get_process_dir()[pc] -> d;  $( get a capability for the
                                     process directory of process pc)
    DEFINITIONS
       slave_capability up IS get_slave(pc);
       access_string as
          IS VECTOR(FOR i FROM 1 TO access_string_length
                    : IF i = load THEN FALSE ELSE TRUE);
    EXCEPTIONS
       no_ability(pc, read);
       no_uproc(up);
    DERIVATION
       restrict_access(h_proc_dir(up), as);
```

```
OVFUN create_temp_segment()[capability pc] -> capability s;
      $( create a temporary segment, i.e., one attached to
         a process rather than a directory )
    DEFINITIONS
       slave_capability u IS get_slave(s);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       EXCEPTIONS_OF segment_create(process_permissions);
    EFFECTS
       s = EFFECTS_OF segment_create(process_permissions);
       'h_temp_segments(up)
        = h_temp_segments(up) UNION {u};

OFUN delete_temp_segment(capability s)[capability pc];
     $( delete a process's temporary segment )
    DEFINITIONS
       slave_capability u IS get_slave(s);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       not_temp_seg(u, up);
       EXCEPTIONS_OF segment_delete(s);
    EFFECTS
       EFFECTS_OF segment_delete(s);
       'h_temp_segments(up)
        = h_temp_segments(up) DIFF {u};

OVFUN create_temp_window(capability base_object;
                         offset base_offset, window_length)
                        [capability pc]
                        -> capability window;
      $( create a window associated withprocess pc)
    DEFINITIONS
       slave_capability u IS get_slave(window);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       EXCEPTIONS_OF window_create(base_object,
                                   base_offset,
                                   window_length,
                                   process_permissions);
    EFFECTS
       window = EFFECTS_OF window_create(base_object,
                                         base_offset,
                                         window_length,
                                         process_permissions);
       'h_temp_windows(up)
        = h_temp_windows(up) UNION {u};

OFUN delete_temp_window(capability window)[capability pc];
     $( delete a temporarily created window )
    DEFINITIONS
       slave_capability u IS get_slave(window);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       not_temp_window(u, up);
       EXCEPTIONS_OF window_delete(window);
    EFFECTS
       EFFECTS_OF window_delete(window);
```



```
       'h_temp_windows(up)
        = h_temp_windows(up) DIFF {u};

END_MODULE
```

```
MODULE user_invoke

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
offset: {INTEGER i | 0 <= i AND i <= max_offset};
address:
STRUCT_OF(capability add_cap; offset add_offset)  $( address of a
                                                    word of storage);

permit_string:
{VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions};

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i) IS get_access(c)[i] ~= TRUE;
BOOLEAN bad_call(slave_capability u; INTEGER o)
   IS    h_function_address(u) = ?
      OR h_procedure(u) = ?
      OR h_procedure_entries(u) <= o;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read $( access rights) ;
INTEGER n_store_permissions $( number of store permissions in a
                               capability );
capability minimum_instructions $( capability for executing the
                                   smallest instruction class);
VFUN get_slave(capability c) -> slave_capability u;
VFUN get_access(capability c) -> VECTOR_OF BOOLEAN bv;

    FROM registers:
INTEGER max_offset $( maximum offset allowed in a register) ,
        stack_register, arguments_register, program_counter
         $( index of address registers used for specific purpose );
OFUN load_address_register(INTEGER i; address a)[capability pc];
OFUN load_inst_class_register(capability c)[capability pc];

    FROM system_invoke:
INTEGER callable $( access right) ;
address h_function_address(slave_capability u)
         $( addresses of all system functions);
address h_stack_address(slave_capability u, up)
         $( stacks for all system functions );
address high_level_return $(address of instruction that
         returns to calling procedure);
OVFUN system_call(address procedure_address, return_address,
                  arguments, top_of_stack)
                [capability pc]
                -> address return_info;
```



```
    FROM segments:
VFUN h_procedure(slave_capability u) -> capability s;
VFUN h_procedure_entries(slave_capability u) -> INTEGER i;

    FROM user_process:
OVFUN create_temp_segment()[capability pc] -> capability s;
OFUN delete_temp_segment(capability s)[capability pc];

    FUNCTIONS

VFUN h_activation_stack(slave_capability up)
                    -> VECTOR_OF address return_address_stack;
     $( the stack of protected calls )
    HIDDEN;
    INITIALLY
       return_address_stack = VECTOR();

OVFUN call(address procedure_address, return_address,
                  arguments, top_of_stack)
          [capability pc]
          -> address return_info;
     $( invoke a procedure in a protected manner )
    DEFINITIONS
       slave_capability u IS get_slave(procedure_address.add_cap);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(procedure_address.add_cap, callable);
       bad_call(u, procedure_address.add_offset);
       RESOURCE_ERROR;
    EFFECTS
       IF h_stack_address(u, up) ~= ?
        THEN   return_info
             = EFFECTS_OF system_call(procedure_address, return_address,
                                      arguments, top_of_stack, pc)

        ELSE   return_info
             = STRUCT(EFFECTS_OF create_temp_segment(pc), 0)
         AND EFFECTS_OF load_address_register
                (arguments_register,
                 arguments,
                 pc)
         AND EFFECTS_OF load_address_register
                       (program_counter,
                        STRUCT(h_procedure(u),
                               procedure_address.add_offset),
                        pc)
         AND EFFECTS_OF load_inst_class_register
                       (minimum_instructions, pc)
         AND EFFECTS_OF load_address_register
                       (stack_register, return_info, pc)
         AND   'h_activation_stack(up)
             = VECTOR(FOR i FROM 1 TO LENGTH(h_activation_stack(up))+1:
                       IF i > 1
                        THEN h_activation_stack(up)[i-1]
                        ELSE return_address);
```

```
OVFUN unprotected_call(address procedure_address, return_address,
                                  arguments, top_of_stack)
                       [capability pc]
                       -> address return_info;
     $( invoke a procedure in a protected manner )
    DEFINITIONS
       slave_capability u IS get_slave(procedure_address.add_cap);
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       no_ability(procedure_address.add_cap, callable);
       bad_call(u, procedure_address.add_offset);
       RESOURCE_ERROR;
    EFFECTS
       IF no_ability(procedure_address.add_cap, read)
        THEN   return_info
             = EFFECTS_OF call(procedure_address, return_address,
                               arguments, top_of_stack, pc)
        ELSE IF h_stack_address(u, up) ~= ?
              THEN   return_info
                   = EFFECTS_OF system_call(procedure_address,
                                            return_address,
                                            arguments, top_of_stack,
                                            pc)
              ELSE return_info = return_address
               AND EFFECTS_OF load_address_register
                               (arguments_register,
                                arguments,
                                pc)
               AND EFFECTS_OF load_address_register
                               (program_counter,
                                procedure_address,
                                pc)
               AND EFFECTS_OF load_inst_class_register
                               (minimum_instructions, pc)
               AND EFFECTS_OF load_address_register
                          (stack_register, top_of_stack, pc);
OVFUN return(address return_info)[capability pc]
           -> address return_address;
     $( Return to calling activation )
    DEFINITIONS
       slave_capability up IS get_slave(pc);
    EXCEPTIONS
       IF return_info.add_offset = 0
        THEN EXCEPTIONS_OF delete_temp_segment
                           (return_info.add_cap, pc)
        ELSE FALSE;
    EFFECTS
       IF return_info.add_offset = 0
        THEN     return_address = h_activation_stack(up)[1]
             AND   'h_activation_stack(up)
                 = VECTOR(FOR i FROM 1
                              TO LENGTH(h_activation_stack(up))-1:
                          h_activation_stack(up)[i+1])
             AND EFFECTS_OF delete_temp_segment
                             (return_info.add_cap, pc)
        ELSE return_address = return_info;
       EFFECTS_OF load_inst_class_register
```



```
                       (minimum_instructions, pc);
       EFFECTS_OF load_address_register
                       (program_counter,
                        high_level_return,
                        pc);

END_MODULE
```

```
MODULE visible_io

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };

    DECLARATIONS

BOOLEAN b, wake;
VECTOR_OF BOOLEAN bv;
slave_capability u, ut, ul;
capability c;
capability d, dt $( capability for an I/O device) ;
INTEGER stat, comm, data;

    PARAMETERS

INTEGER control, device $( access rights) ;

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] = FALSE;
BOOLEAN not_device(slave_capability u) IS h_temp_device(u) = ?;
BOOLEAN not_perm_dev_cap(slave_capability u)
   IS h_device_exists(u) ~= TRUE;
BOOLEAN already_temp(slave_capability u)
   IS EXISTS ut: h_temp_device(ut) = u AND ut ~= u;
BOOLEAN no_input(slave_capability u) IS h_input(u) = ?;
BOOLEAN no_output(slave_capability u) IS h_output(u) = ?;
BOOLEAN no_command(slave_capability u) IS h_command(u) = ?;
BOOLEAN too_much_input(slave_capability u) IS h_input(u) ~= ?;
BOOLEAN too_much_output(slave_capability u) IS h_output(u) ~= ?;
BOOLEAN too_many_commands(slave_capability u)
   IS h_command(u) ~= ?;
BOOLEAN uninitialized_device(slave_capability u)
   IS h_status(u) = ?;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write, delete $( access rights) ;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
           predefined slave capability );
VFUN get_slave(c) -> u;
VFUN get_access(c) -> bv;
OVFUN create_restricted_cap(bv) -> c;
```



```
    FROM coordinator:
OFUN wakeup_process(capability pc);

    FUNCTIONS

VFUN h_device_exists(u) -> b;  $( true if uid u designates a device)
    HIDDEN;
    INITIALLY
       b =(predefined_slave(u) = device_slave);

VFUN h_temp_device(ut) -> u;  $( returns permanent unique
                               identifier of device u)
    HIDDEN;
    INITIALLY
       u =(IF h_device_exists(u) = TRUE THEN u ELSE ?);

VFUN h_device_process(u) -> capability pc;
     $( process capability for device with uid u)
    HIDDEN;
    INITIALLY
       pc = ?;

VFUN h_input(u) -> data;  $( one word of input data from I/O device
                          with uid u)
    HIDDEN;
    INITIALLY
       data = ?;

OVFUN assign_device(d) -> dt;  $( assigns a temporary capability
                               for device d)
    DEFINITIONS
       slave_capability u IS get_slave(d);
       slave_capability ut IS get_slave(dt);
    EXCEPTIONS
       not_perm_dev_cap(u);
       already_temp(u);
    EFFECTS
       dt = EFFECTS_OF create_restricted_cap(get_access(d));
       'h_temp_device(ut) = u;

OFUN deassign_device(d);  $( eliminates temporary capability for
                          device d)
    DEFINITIONS
       slave_capability ut IS get_slave(d);
       slave_capability u IS h_temp_device(ut);
    EXCEPTIONS
       no_ability(d, delete);
       not_device(ut);
    EFFECTS
       FORALL ul: h_temp_device(ul) = u => ul = u;

OVFUN read_device(d) -> data; $( visible form of h_input)
    DEFINITIONS
       slave_capability u IS h_temp_device(ut);
       slave_capability ut IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
```

```
        not_device(ut);
        no_input(u);
    EFFECTS
        data = h_input(u);
        'h_input(u) = ?;

OFUN write_device(d; data);  $( output data to I/O device with
                               capability d)
    DEFINITIONS
        slave_capability u IS h_temp_device(ut);
        slave_capability ut IS get_slave(d);
    EXCEPTIONS
        no_ability(d, write);
        not_device(ut);
        too_much_output(u);
    EFFECTS
        'h_output(u) = data;

OFUN send_command(d; comm); $( send command to I/O device)
    DEFINITIONS
        slave_capability u IS h_temp_device(ut);
        slave_capability ut IS get_slave(d);
    EXCEPTIONS
        no_ability(d, control);
        not_device(ut);
        too_many_commands(u);
    EFFECTS
        'h_command(u) = comm;

VFUN h_status(u) -> stat; $( status of I/O device with uid u)
    HIDDEN;
    INITIALLY
        stat = ?;

VFUN receive_status(d) -> stat; $( visible form of h_status)
    DEFINITIONS
        slave_capability u IS h_temp_device(ut);
        slave_capability ut IS get_slave(d);
    EXCEPTIONS
        no_ability(d, read);
        not_device(ut);
        uninitialized_device(u);
    DERIVATION
        h_status(u);

VFUN h_output(u) -> data;  $( data to be output by I/O device with
                              uid u)
    HIDDEN;
    INITIALLY
        data = ?;

OFUN set_device_process(d; capability pc);
     $( indicate a process for device d)
    DEFINITIONS
        slave_capability u IS h_temp_device(ut);
        slave_capability ut IS get_slave(d);
    EXCEPTIONS
```



```
        no_ability(d, control);
        not_device(ut);
    EFFECTS
        'h_device_process(u) = pc;

OVFUN device_receive(d) -> data; $( visible form of h_output)
    DEFINITIONS
        slave_capability u IS get_slave(d);
    EXCEPTIONS
        no_ability(d, device);
        not_perm_dev_cap(u);
        no_output(u);
    EFFECTS
        data = h_output(u);
        'h_output(u) = ?;

OFUN device_send(d; data); $( input data to system)
    DEFINITIONS
        slave_capability u IS get_slave(d);
    EXCEPTIONS
        no_ability(d, device);
        not_perm_dev_cap(u);
        too_much_input(u);
    EFFECTS
        'h_input(u) = data;

VFUN h_command(u) -> comm; $( command for I/O device with uid u)
    HIDDEN;
    INITIALLY
        comm = ?;

OVFUN device_command(d) -> comm; $( gets a command for a device)
    DEFINITIONS
        slave_capability u IS get_slave(d);
    EXCEPTIONS
        no_ability(d, device);
        not_perm_dev_cap(u);
        no_command(u);
    EFFECTS
        comm = h_command(u);
        'h_command(u) = ?;

OFUN change_status(d; stat; wake);  $( change the status of a
                                       device and possibly send an
                                       interrupt)
    DEFINITIONS
        slave_capability u IS get_slave(d);
    EXCEPTIONS
        no_ability(d, device);
        not_perm_dev_cap(u);
    EFFECTS
        'h_status(u) = stat;
        wake AND h_device_process(u) ~= ?
      => EFFECTS_OF wakeup_process(h_device_process(u));

    END_MODULE
```

```
MODULE procedure_records
    TYPES

machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
permit_string:
{ VECTOR_OF BOOLEAN ps | LENGTH(ps) = n_store_permissions };
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
address: STRUCT_OF(capability add_cap; INTEGER add_offset);
offset: {INTEGER i | 0 <= i AND i <= max_offset};

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] ~= TRUE;
BOOLEAN not_writeable(machine_word w)
   IS TYPECASE w OF capability:
         EXISTS INTEGER i | 1 <= i AND i <= n_store_permissions:
             ~ get_access(w)[i]
           AND ~ process_permissions[i];
      INTEGER: FALSE;
      BOOLEAN: FALSE;
      CHAR: FALSE;
     END;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write;
INTEGER n_store_permissions;
INTEGER access_string_length;
machine_word zeroword;
VFUN get_slave(capability c) -> slave_capability u;
VFUN get_access(capability c) -> access_string as;
OVFUN create_restricted_cap(access_string as)
                          -> capability c;

    FROM registers:
INTEGER max_offset;

    FROM user_process:
access_string process_permissions;

    FUNCTIONS

VFUN h_pr_read(slave_capability u; INTEGER i)
             -> machine_word w;
     $(contents of a procedure record)
    HIDDEN;
    INITIALLY
      w = ?;
```



```
VFUN h_pr_map(slave_capability u, up) -> capability pr;
     $(procedure record for given procedure u in process up)
    HIDDEN;
    INITIALLY
      pr = ?;

VFUN get_pr_address(capability proc)[capability pc]
                  -> capability pr;
     $(return capability for procedure record for given
       procedure)
    DEFINITIONS
      slave_capability u IS get_slave(proc);
      slave_capability up IS get_slave(pc);
    EXCEPTIONS
      no_ability(proc, read);
      h_pr_map(u, up) = ?;
    DERIVATION
      h_pr_map(u, up);

OVFUN create_pr(capability proc; offset l)[capability pc]
              -> capability pr;
     $(create a new procedure record for procedure proc
       in process pc)
    DEFINITIONS
      slave_capability u IS get_slave(proc);
      slave_capability up IS get_slave(pc);
    EXCEPTIONS
      no_ability(proc, read);
      h_pr_map(u, up) ~= ?;
      RESOURCE_ERROR;
    EFFECTS
      pr = EFFECTS_OF create_restricted_cap
                        (process_permissions);
      'h_pr_map(u, up) = pr;
      FORALL INTEGER i | 0 <= i AND i < l:
       'h_pr_read(get_slave(pr), i) = zeroword;

OFUN delete_pr(capability proc)[capability pc];
     $( delete the given procedure record )
    DEFINITIONS
      slave_capability u IS get_slave(proc);
      slave_capability up IS get_slave(pc);
    EXCEPTIONS
      no_ability(proc, read);
      h_pr_map(u, up) = ?;
    EFFECTS
      'h_pr_map(u, up) = ?;
      FORALL INTEGER i:
       'h_pr_read(get_slave(h_pr_map(u, up)), i) = ?;

VFUN read_pr(capability pr; offset i)
             -> machine_word w;
     $(read a word from a procedure record)
    DEFINITIONS
      slave_capability upr IS get_slave(pr);
    EXCEPTIONS
      no_ability(pr, read);
```

```
      h_pr_read(upr, i) = ?;
   DERIVATION
      h_pr_read(upr, i);

OFUN write_pr(capability pr; offset i; machine_word w);
    $(modify a wrd in a procedure record)
   DEFINITIONS
      slave_capability upr IS get_slave(pr);
   EXCEPTIONS
      no_ability(pr, write);
      h_pr_read(upr, i) = ?;
      not_writeable(w);
   EFFECTS
      'h_pr_read(upr, i) = w;

END_MODULE
```



```
MODULE linker

    TYPES

machine_word: ONE_OF(INTEGER, capability, BOOLEAN, CHAR);
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
character_string: VECTOR_OF CHAR;
entry_name:
{ character_string n | LENGTH(n) <= entry_name_length };

    DECLARATIONS

BOOLEAN b;
INTEGER i;
capability c, pg;
slave_capability u, ud, upg;
machine_word w;
entry_name n;
VECTOR_OF entry_name nv $( path name) ;
VECTOR_OF machine_word pgv $( a procedure) ;
SET_OF capability ls $( set of locks for an entry) ;

    PARAMETERS

VECTOR_OF entry_name h_linkage_definition(pgv; i)
         $( returns the path name for a given link i in procedure
          pgv);
VECTOR_OF capability h_keys(pgv; i)  $( returns keys for entries in
                                       path name of link i in
                                       procedure pgv);

    DEFINITIONS

capability subpath(slave_capability ud; VECTOR_OF entry_name nv;
                   INTEGER i)
   IS resolve_path(ud, VECTOR(FOR j FROM 1 TO i: nv[j]));
capability resolve_path(ud; nv) $( returns capability for path
              name nv beginning at directory ud)
   IS IF LENGTH(nv) = 1
       THEN h_get_cap(ud, nv[1])
       ELSE h_get_cap(get_slave(subpath(ud, nv, LENGTH(nv) - 1)),
                      nv[LENGTH(nv)]);
BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] ~= TRUE;
BOOLEAN bad_def(VECTOR_OF machine_word pgv; INTEGER i)
   IS h_linkage_definition(pgv, i) = ?
     OR LENGTH(h_linkage_definition(pgv, i)) = 0;
BOOLEAN bad_path(slave_capability ud; VECTOR_OF entry_name nv;
                 VECTOR_OF capability kv)
   IS h_valid_dir(ud) ~= TRUE OR LENGTH(nv) ~= LENGTH(kv)
     OR(EXISTS i:
           i > 0 AND i <= LENGTH(nv)
         =>  h_get_cap(IF i = 1 THEN ud
                       ELSE get_slave(subpath(ud, nv, i - 1)),
```

```
                        nv[i])
              = ?
             OR NOT(  kv[i]
                    INSET h_locks(IF i = 1 THEN ud
                                  ELSE get_slave(subpath(ud, nv,
                                                         i - 1)),
                                  nv[i])
                   OR( i ~= LENGTH(nv)
                      => h_valid_dir(get_slave(subpath(ud, nv, i)))
                           ~= TRUE
                         OR get_access(subpath(ud, nv, i))[load]
                           ~= TRUE)));

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read $( access right) ;
INTEGER access_string_length;
VFUN get_slave(c) -> u;
VFUN get_access(c) -> access_string as;

    FROM segments:
VFUN segment_size(pg) -> i;
VFUN h_read(upg; i) -> w;

    FROM directories:
INTEGER load $( access right) ;
INTEGER entry_name_length;
BOOLEAN root_slave(u) $(true if c is slave for root dir);
VFUN h_valid_dir(ud) -> b;
VFUN h_get_cap(ud; n) -> c;
VFUN h_locks(ud; n) -> ls;

    FUNCTIONS

VFUN resolve_reference(pg; i) -> c;  $( returns capability for
                                       object indicated by link i of
                                       procedure pg)
    DEFINITIONS
       VECTOR_OF machine_word pgv
          IS VECTOR(FOR I FROM 0 TO segment_size(pg) - 1
                    : h_read(upg, i));
       slave_capability upg IS get_slave(pg);
    EXCEPTIONS
       no_ability(pg, read);
       bad_def(pgv, i);
       bad_path(SOME u | root_slave(u),
                h_linkage_definition(pgv, i),
                h_keys(pgv, i));
    DERIVATION
       resolve_path(SOME u | root_slave(u),
                    h_linkage_definition(pgv, i));
    END_MODULE
```



```
MODULE user
   $ "This module introduces the concept of a user.  A user is a
      person or group of persons on whose behalf processes can be
      created to perform computations as directed by the person or
      persons.  All processes created on behalf of the same user
      will have the same initial conditions."

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };
character_string: VECTOR_OF CHAR;
name: { character_string n | LENGTH(n) <= entry_name_length };
offset: {INTEGER I | 0 <= i AND i <= max_offset};
address: STRUCT_OF(capability add_cap; offset add_offset);

    PARAMETERS

INTEGER create_proc; $ "Access rights."
capability user_dir;
       $ "Capability for directory containing process capabilities"

    DEFINITIONS

BOOLEAN name_duplication(name user_name)
   IS user_name INSET h_user_names();
BOOLEAN no_ability(capability c; INTEGER ability)
   IS get_access(c)[ability] ~= TRUE;
BOOLEAN no_user(slave_capability user_uid)
   IS h_user_name(user_uid) = ?;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER delete; $ "Access rights"
INTEGER access_string_length;
VFUN get_slave(capability c) -> slave_capability u;
        $ "Extracts the unique identifier of a capability."
VFUN get_access(capability c) -> access_string as;
        $ "Extracts the access bits from a capability."
OVFUN create_capability() -> capability c;
        $ "Creates a new capability."

    FROM registers:
INTEGER max_offset;

    FROM directories:
INTEGER entry_name_length;
        $ "Length of entry name is also length of user name"
```

```
    FROM user_process:
OVFUN create_uproc(capability d;
                   name n;
                   address procedure, args)
                  -> capability process_cap;
    $ "Creates a new process with capability process_cap.  The new
       process will begin execution in the procedure whose
       capability is procedure_cap at offset proc_offset with
       arguments given in init_stack."

    FUNCTIONS

VFUN h_user_name(slave_capability user_uid) -> name user_name;
        $ "Returns the name associated with the user."
    HIDDEN;
    INITIALLY
       user_name = ?;

VFUN h_user_procedure(slave_capability user_uid)
                     -> address procedure;
    $ "Returns the address for the initial procedure of processes
       created on behalf of this user."
    HIDDEN;
    INITIALLY
       procedure = ?;

VFUN h_user_args(slave_capability user_uid)
                -> address args;
    $ "Returns the aguments to the initial procedure of any process
       created on behalf of this user."
    HIDDEN;
    INITIALLY
       args = ?;

VFUN h_user_names() -> SET_OF name user_names;
      $ "Returns the set of all names ever associated with a user."
    HIDDEN;
    INITIALLY
       user_names = { };

VFUN h_process_user_name(slave_capability process_uid)
                        -> name user_name;
    $ "Returns the name of the user on whose behalf the process was
       created."
    HIDDEN;
    INITIALLY
       user_name = ?;

VFUN h_process_device(slave_capability process_uid)
                     -> capability device_cap;
    $ "Returns the capability for the device which the given process
       is to use to communicate with the user."
    HIDDEN;
    INITIALLY
       device_cap = ?;
```



```
VFUN get_user_name()[capability process_cap] -> name user_name;
    $ "Returns the name of the user on whose behalf the currently
       executing process was created."
    DEFINITIONS
       slave_capability process_uid IS get_slave(process_cap);
    DERIVATION
       h_process_user_name(process_uid);

VFUN get_device_cap()[capability process_cap]
                    -> capability device_cap;
    $ "Returns a capability for the device on which the executing
       process can communicate with the user."
    DEFINITIONS
       slave_capability process_uid IS get_slave(process_cap);
    DERIVATION
       h_process_device(process_uid);

OVFUN create_user(name user_name;
                  address procedure, args)
                 -> capability user_cap;
    $ "CREATE_USER creates a new user of the system.
         user_name - The name of the new user. The name must be
                unique for the lifetime of the system.
         procedure - The address for the procedure segment
                of the procedure that will first be invoked in any
                process created on behalf of this user.
         args - The arguments to the initial procedure.
         user_cap - The capability for the new user."
    DEFINITIONS
       slave_capability user_uid IS get_slave(user_cap);
    EXCEPTIONS
       name_duplication(user_name);
       RESOURCE_ERROR;
    EFFECTS
       user_cap = EFFECTS_OF create_capability();
       'h_user_name(user_uid) = user_name;
       'h_user_procedure(user_uid) = procedure;
       'h_user_args(user_uid) = args;
       'h_user_names() = h_user_names() UNION { user_name };

OFUN delete_user(capability user_cap);
        $ "Deletes the user with capability user_cap."
    DEFINITIONS
       slave_capability user_uid IS get_slave(user_cap);
    EXCEPTIONS
       no_ability(user_cap, delete);
       no_user(user_uid);
    EFFECTS
       'h_user_name(user_uid) = ?;
       'h_user_procedure(user_uid) = ?;
       'h_user_args(user_uid) = ?;

OVFUN create_user_process(capability user_cap; capability device_cap)
                         -> capability process_cap;
    $ "Creates a new process with capability PROCESS_CAP on behalf of
       the user with capability USER_CAP.  The new process can
       communicate with the user on the device whose capability
```

```
        is DEVICE_CAP."
    DEFINITIONS
       slave_capability user_uid IS get_slave(user_cap);
       slave_capability process_uid IS get_slave(process_cap);
    EXCEPTIONS
       no_ability(user_cap, create_proc);
       no_user(user_uid);
       EXCEPTIONS_OF create_uproc(user_dir, h_user_name(user_uid)
                                  h_user_procedure(user_uid),
                                  h_user_args(user_uid));
    EFFECTS
        process_cap
       = EFFECTS_OF create_uproc(user_dir, h_user_name(user_uid),
                                  h_user_procedure(user_uid),
                                  h_user_args(user_uid));
        'h_process_user_name(process_uid)
       = h_user_name(user_uid);
       'h_process_device(process_uid) = device_cap;

END_MODULE
```



```
MODULE mail
      $ "This module permits users to send messages to one another.
         Messages are arrays of data.  Each user can send a message
         to any other user who has given him permission to do so.
         Users are indicated by symbolic name.  Each message received
         by a user is identified with the name of the sender."

    TYPES

slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
character_string: VECTOR_OF CHAR;
name: { character_string n | LENGTH(n) <= entry_name_length };
machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
message: VECTOR_OF machine_word;

    PARAMETERS

INTEGER max_queue_size $( the maximum number of words in a queue) ;

    DEFINITIONS

BOOLEAN invalid_target(name target_name; name sender_name)
   IS ~(sender_name INSET h_sender_list(target_name));
BOOLEAN message_too_long(INTEGER message_size;
                         VECTOR_OF STRUCT_OF(name sender_name;
                                             message data)
                           message_queue)
   IS  SUM(VECTOR(FOR i FROM 1 TO LENGTH(message_queue)
                  : LENGTH(message_queue[i].sender_name)
                   + LENGTH(message_queue[i].data)))
     + message_size
     > max_queue_size;
    $ "This exception is true if the message to be added will make
       the message queue longer than its maximum length."

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
VFUN get_slave(capability c) -> slave_capability u;
        $ "Returns the unique identifier of a capability."

    FROM directories:
INTEGER entry_name_length; $ "Length of user name"

    FROM user:
VFUN h_process_user_name(slave_capability process_uid)
                        -> name user_name;
    $ "Returns the name of the user on whose behalf the given process
       was created."
```

```
    FUNCTIONS

VFUN h_message_queue(name user_name)
                  -> VECTOR_OF STRUCT_OF(name sender_name;
                                         message data) message_queue;
    $ "This function returns the queue of messages for the
       given user."
    HIDDEN;
    INITIALLY
       message_queue = VECTOR();

VFUN h_sender_list(name user_name) -> SET_OF name name_list;
    $ "Returns the list of names of users who may send the given user
       a message."
    HIDDEN;
    INITIALLY
       name_list = { };

OFUN send_message(name target_name; message data)
                [capability process_cap];
    $ "The user indicated by target_name will be sent a message
       containing the given data.  The message will be marked
       as coming from the user on whose behalf the process with
       capability process_cap was created."
    DEFINITIONS
       name user_name
          IS h_process_user_name(get_slave(process_cap));
    EXCEPTIONS
       invalid_target(target_name, user_name);
       message_too_long(LENGTH(data) + LENGTH(user_name),
                        h_message_queue(target_name));
    EFFECTS
        'h_message_queue(target_name)
       = VECTOR(FOR i FROM 1
                TO LENGTH(h_message_queue(target_name)) + 1:
                IF i <= LENGTH(h_message_queue(target_name))
                  THEN h_message_queue(target_name)[i]
                  ELSE STRUCT(user_name, data));

OVFUN receive_message()[capability process_cap]
                  -> STRUCT_OF(name sender_name; message data) mail;
    $ "Returns the earliest message sent to and not already received
       by the user on whose behalf the executing process was
       created."
    DEFINITIONS
       name user_name
          IS h_process_user_name(get_slave(process_cap));
    EXCEPTIONS
       LENGTH(h_message_queue(user_name)) = 0; $ "No messages"
    EFFECTS
       mail = h_message_queue(user_name)[1];
        'h_message_queue(user_name)
       = VECTOR(FOR i FROM 1
                TO LENGTH(h_message_queue(user_name)) - 1:
                h_message_queue(user_name)[i + 1]);
```



```
OFUN accept_message(name sender_name)[capability process_cap];
    $ "The user on whose behalf the executing process was created
       will accept messages from the user with name sender_name."
    DEFINITIONS
       name user_name
          IS h_process_user_name(get_slave(process_cap));
    EFFECTS
        'h_sender_list(user_name)
       = h_sender_list(user_name) UNION { sender_name };

OFUN reject_message(name sender_name)[capability process_cap];
    $ "The user on whose behalf the executing process was created
       will not accept messages from the user with name sender_name."
    DEFINITIONS
       name user_name
          IS h_process_user_name(get_slave(process_cap));
    EFFECTS
        'h_sender_list(user_name)
       = h_sender_list(user_name) DIFF { sender_name };

VFUN sender_list()[capability process_cap]
                -> SET_OF name name_list;
    $ "Returns the list of names for users from whom the user
       on whose behalf the executing process was created will
       accept messages."
    DEFINITIONS
       name user_name
          IS h_process_user_name(get_slave(process_cap));
    DERIVATION
       h_sender_list(user_name);

END_MODULE
```

```
MODULE channel_io

    TYPES

machine_word: ONE_OF(capability, INTEGER, BOOLEAN, CHAR);
slave_capability:
{ capability u | EXISTS capability c: get_slave(c) = u };
access_string:
{ VECTOR_OF BOOLEAN as | LENGTH(as) = access_string_length };

    DECLARATIONS

BOOLEAN b, int;
slave_capability u;
capability c, cl;
capability d $( capability for an I/O device) ;
INTEGER stat, comm, i, o, n;
machine_word data;

    PARAMETERS

INTEGER control,
       device  $( each of these assumes a different value from 1 to
                  access_length);

    DEFINITIONS

BOOLEAN no_ability(capability c; INTEGER i)
   IS get_access(c)[i] = FALSE;
BOOLEAN not_device(slave_capability u)
   IS h_device_set(u) = FALSE;
BOOLEAN no_block(slave_capability u)
   IS block_length(u) = ?;
BOOLEAN address_bounds(slave_capability u; INTEGER i)
   IS i < 0 OR I >= block_length(u);
BOOLEAN bad_count(INTEGER n) IS n <= 0;
BOOLEAN no_output(slave_capability u) IS h_output_count(u) = 0;
BOOLEAN no_command(slave_capability u) IS h_command(u) = ?;
BOOLEAN too_much_input(slave_capability u)
   IS h_input_count(u) = 0;
BOOLEAN too_many_commands(slave_capability u)
   IS h_command(u) ~= ?;
BOOLEAN uninitialized_device(slave_capability u)
   IS h_status(u) = ?;
BOOLEAN no_interrupt(slave_capability u; BOOLEAN int)
   IS IF int THEN h_device_interrupt(u) = ? ELSE FALSE;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER read, write $( access rights) ;
INTEGER access_string_length;
slave_type: {interrupt_slave, processor_slave, inst_class_slave,
             memory_block_slave, device_slave, clock_slave,
             not_predefined} $( types of predefined slaves );
slave_type predefined_slave(slave_capability u) $( type of a
```



```
           predefined slave capability );
VFUN get_slave(c) -> u;
VFUN get_access(c) -> access_string as;

    FROM interrupts:
OFUN set_interrupt(c; cl);

    FROM memory:
INTEGER block_length(slave_capability u)
        $( number of words in a memory block) ;
VFUN block_read(c; i) -> data;
OFUN block_write(c; i; data);

    FUNCTIONS

VFUN h_device_set(u) -> b; $( true if uid u designates a device)
    HIDDEN;
    INITIALLY
       b =(predefined_slave(u) = device_slave);

VFUN h_device_interrupt(u) -> c;  $( interrupt capability for
                                     device with uid u)
    HIDDEN;
    INITIALLY
       predefined_slave(get_slave(c)) = interrupt_slave;

VFUN h_device_processor(u) -> c;  $( processor for handling
                                     interrupts from device u)
    HIDDEN;
    INITIALLY
       predefined_slave(get_slave(c)) = processor_slave;

VFUN h_input_block(u) -> c;  $( memory block for placing input data)
    HIDDEN;
    INITIALLY
       c = ?;

VFUN h_input_offset(u) -> o; $( offset for placing input data)
    HIDDEN;
    INITIALLY
       o = ?;

VFUN h_input_count(u) -> n;  $( number of cells remaining for input
                                data)
    HIDDEN;
    INITIALLY
       n = 0;

VFUN h_input_interrupt(u) -> b;  $( true if interrupt to be sent
                                    when input complete)
    HIDDEN;
    INITIALLY
       b = ?;
```

```
VFUN h_output_block(u) -> c;  $( memory block from which data is to
                               be output)
    HIDDEN;
    INITIALLY
       c = ?;

VFUN h_output_offset(u) -> o;  $( position in block where output
                                begins)
    HIDDEN;
    INITIALLY
       o = ?;

VFUN h_output_count(u) -> n; $( number of words to be output)
    HIDDEN;
    INITIALLY
       n = 0;

VFUN h_output_interrupt(u) -> b;  $( true if interrupt should be
                                   sent when output complete)
    HIDDEN;
    INITIALLY
       b = ?;

OFUN read_device(d; c; o; n; int);  $( tell device where to put
                                    next n words of data)
    DEFINITIONS
       slave_capability uc IS get_slave(c);
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       no_ability(c, write);
       not_device(u);
       no_block(uc);
       address_bounds(u, o);
       address_bounds(u, o + n - 1);
       bad_count(n);
    EFFECTS
       'h_input_block(u) = c;
       'h_input_offset(u) = o;
       'h_input_count(u) = n;
       'h_input_interrupt(u) = int;

OFUN write_device(d; c; o; n; int);  $( tell device where to get
                                     next n words of data)
    DEFINITIONS
       slave_capability uc IS get_slave(c);
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, write);
       no_ability(c, read);
       not_device(u);
       no_block(uc);
       address_bounds(u, o);
       address_bounds(u, o + n - 1);
       bad_count(n);
    EFFECTS
       'h_output_block(u) = c;
```



```
       'h_output_offset(u) = o;
       'h_output_count(u) = n;
       'h_output_interrupt(u) = int;

OFUN send_command(d; comm); $( send command to I/O device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, control);
       not_device(u);
       too_many_commands(u);
    EFFECTS
       'h_command(u) = comm;

VFUN h_status(u) -> stat; $( status of I/O device with uid u)
    HIDDEN;
    INITIALLY
       stat = ?;

VFUN receive_status(d) -> stat; $( visible form of h_status)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, read);
       not_device(u);
       uninitialized_device(u);
    DERIVATION
       h_status(u);

OFUN device_send(d; data); $( input data to system)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       too_much_input(u);
    EFFECTS
       EFFECTS_OF block_write(h_input_block(u), h_input_offset(u),
                              data);
       'h_input_offset(u) = h_input_offset(u) + 1;
       'h_input_count(u) = h_input_count(u) - 1;
        h_input_interrupt(u) AND h_input_count(u) - 1 = 0
       => EFFECTS_OF set_interrupt(h_device_processor(u),
                                   h_device_interrupt(u));

OVFUN device_receive(d) -> data; $( get data from system)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_output(u);
    EFFECTS
       data = block_read(h_output_block(u), h_output_offset(u));
       'h_output_offset(u) = h_output_offset(u) + 1;
       'h_output_count(u) = h_output_count(u) - 1;
        h_output_interrupt(u) AND h_output_count(u) - 1 = 0
```

```
      => EFFECTS_OF set_interrupt(h_device_processor(u),
                                  h_device_interrupt(u));

VFUN h_command(u) -> comm; $( command for I/O deevice with uid u)
    HIDDEN;
    INITIALLY
       comm = ?;

OVFUN device_command(d) -> comm; $( gets a command for a device)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_command(u);
    EFFECTS
       comm = h_command(u);
       'h_command(u) = ?;

OFUN change_status(d; stat; int);  $( change the status of a device
                                      and possibly send an interrupt)
    DEFINITIONS
       slave_capability u IS get_slave(d);
    EXCEPTIONS
       no_ability(d, device);
       not_device(u);
       no_interrupt(u, int);
    EFFECTS
       'h_status(u) = stat;
        int
       => EFFECTS_OF set_interrupt(h_device_processor(u),
                                   h_device_interrupt(u));

END_MODULE
```

APPENDIX C -- PROOFS OF IMPLEMENTATION

CONTENTS

GRAMMAR FOR ILPL

The nonterminals <symbol> and <number> are the standard ones allowed in INTERLISP. Those nonterminals beginning with the word SPECIAL are derived from the grammar of SPECIAL, to be found in Appendix A.

```
<program module> ::= PROGRAM MODULE <symbol>
                        [<SPECIAL types>]
                        [<SPECIAL declarations>]
                        [<SPECIAL parameters>]
                        [<SPECIAL definitions>]
                        <SPECIAL externalrefs>
                        <initialization>
                        <implementations>
                     END_MODULE

<parameters> ::= PARAMETERS { <declaration> ';' }+

<declaration> ::= <typespecification> <symbol> { ',' <symbol> }*

<typespecifcation> ::= <symbol>
                   ::= INTEGER
                   ::= BOOLEAN
                   ::= REAL
                   ::= CHAR
                   ::= STRUCT '(' [ <declaration>
                                    { ';' <declaration> }* ] ')'
                   ::= ONE_OF '(' <typespecification>
                                  { ',' <typespecification> }+ ')'
                   ::= VECTOR_OF <typespecification>

<initialization> ::= INITIALIZATION <program> ';'

<program> ::= <statement> | BEGIN <statement list> END

<implementations> ::= IMPLEMENTATIONS { <implementation> ';' }+

<implementation> ::= <fnprog>
                 ::= <subr>

<fnprog> ::= { OFUN_PROG | VFUN_PROG | OVFUN_PROG } <header>
                [<declarations>]
                <program>

<subr> ::= { OFUN_SUBR | OVFUN_SUBR | VFUN_SUBR } <header>
```

```
            [<SPECIAL definitions>]
            [<SPECIAL exceptions>]
            { <SPECIAL derivation> | <SPECIAL effects> }
            [<declarations>]
            <program>

<header> ::= <symbol>
                '(' [ <declaration> { ';' <declaration> }* ] ')'
              [ '->' <declaration> ] ';'

<declarations> ::= DECLARATIONS { <declaration> ';' }+

<statement list> ::= <statement> { ';' <statement> }*

<statement> ::= <simple statement>
            ::= DO <simple statement> WITH <exception prog list> OD
            ::= WHILE <expression> [ ASSERT <SPECIAL expression> ]
                  DO <statement list> OD
            ::= IF <expression> THEN <statement list>
                  [ ELSE <statement list> ] FI
            ::= FOR <symbol> FROM <expression> TO <expression>
                  [ BY <expression> ]
                  [ ASSERT <SPECIAL expression> ]
                  DO <statement list> OD
            ::= TYPECASE <symbol> OF <case prog list> END

<simple statement> ::= <call>
                   ::= <multiple assignment>

<call> ::= <symbol> '(' [<expression list>] ')'

<expression list> ::= <expression> { ',' <expression> }*

<multiple assignment> ::= <symbol> { ',' <symbol> }*
                              '<-' <expression>

<exception prog list> ::= <exception prog> { ';' <exception prog} }*

<exception prog> ::= <string constant> ':' <program>

<case prog list> ::= <case prog> { ';' <case prog> }*

<case prog> ::= <typespecification> ':' <statement list>

<expression> ::= <simple expression>
                   [ '{' <exception value list> '}' ]

<exception value list> ::= <exception value>
                             { ';' <exception value> }*

<exception value> ::= <string constant> ':' <expression>

<simple expression> ::= IF <expression> THEN <expression>
                            ELSE <expression>
```



```
                    ::= <expression> <binaryop> <expression>
                    ::= <unaryop> <expression>
                    ::= '(' <expression ')'
                    ::= <symbol>
                    ::= <number>
                    ::= <character constant>
                    ::= <string constant>
                    ::= TRUE | FALSE
                    ::= <expression> '[' <expression> ']'
                    ::= <expression> '.' <symbol>
                    ::= { INTPART | FRACTPART } <expression>
                    ::= <call>
                    ::= <structure constructor>
                    ::= <vector constructor>
                    ::= TYPECASE <symbol> OF
                          <case expression list> END
                    ::= RETURN '(' <string constant> ')'

<binaryop> ::= '*' | '/' | '+' | '-' | '=' | '~=' | '>' | '>=' | '<'
           ::= '<=' | AND | OR | '=>' | MOD

<unaryop> ::= NOT | '~' | '-'

<structure constructor> ::= '<' <expression list> '>'

<vector constructor> ::= VECTOR '(' [<expression list>] ')'
                     ::= VECTOR '(' FOR <symbol> FROM <expression> TO
                            <expression> ':' <expression> ')'

<case expression list> ::= <case expression>
                             { ';' <case expression> }*

<case expression> ::= <string constant> ':' <expression>
```

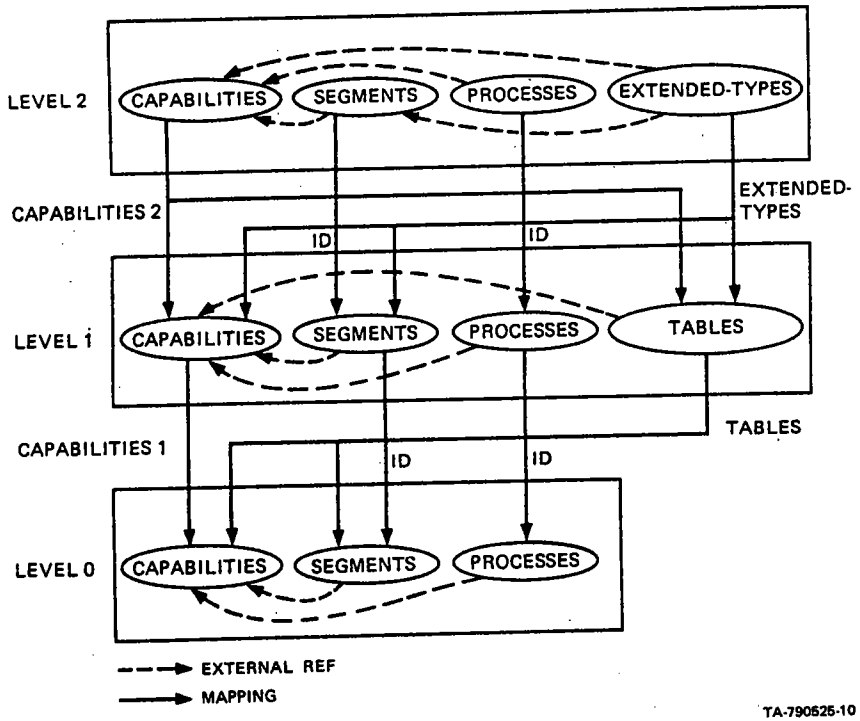


FIGURE C-1 STRUCTURE OF THE SAMPLE SYSTEM

INTERCONNECTIONS FOR EXAMPLE SYSTEM

These expressions are a formal description of the diagram on page C-II-1.1.
The interface specifications have a syntax as follows:
    (INTERFACE <interface name> <list of modules>).
The hierarchy specification has a syntax as follows:
    (HIERARCHY <hierarchy name> <list of implements relations>),
where the syntax for an implements relation is
    (<interface name> IMPLEMENTS <interface name>
                USING <list of mapping functions>).

```
(INTERFACE LEVEL2 CAPABILITIES SEGMENTS PROCESSES EXTENDED-TYPES)

(INTERFACE LEVEL1 CAPABILITIES SEGMENTS PROCESSES TABLES)

(INTERFACE LEVEL0 CAPABILITIES SEGMENTS PROCESSES)

(HIERARCHY SYSTEM (LEVEL0 IMPLEMENTS LEVEL1
                      USING TABLES CAPABILITIES1)
                  (LEVEL1 IMPLEMENTS LEVEL2
                      USING EXTENDED-TYPES CAPABILITIES2))
```

SPECIFICATIONS FOR THE EXTENDED-TYPE MODULE

```
MODULE extended_types

    DECLARATIONS

capability c, c1 $( arbitrary capabilities) , t,
         t1 $( type-manager's capabilities) ;
BOOLEAN b;
INTEGER i;

    PARAMETERS

capability ct $( type-creation capability) ,
         cs $( type capability for type SEGMENT) ;
INTEGER max_impl_caps;  $( maximum number of implementation
                           capabilities for any type)

    DEFINITIONS

BOOLEAN bad_type_manager(t) IS NOT is_type_manager(t);
BOOLEAN invalid_object(c; t) IS NOT is_type(c, t);
BOOLEAN invalid_impl_type(t)
   IS NOT(is_type_manager(t) OR t = cs);
BOOLEAN orig_impl_objs_present(c; t)
   IS EXISTS i: original(c, t, i);
BOOLEAN exc_original(c; t; i) IS original(c, t, i);
BOOLEAN no_impl_obj(c; t; i) IS impl_cap(c, t, i) = ?;
BOOLEAN bad_element(i; t) IS i < 1 OR i > impl_length(t);

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
OVFUN create_cap() -> c;

    FROM segments:
OVFUN create_segment(i) -> c;
OFUN delete_seg(c);
VFUN seg_exists(c) -> b;

    FUNCTIONS

VFUN is_type_manager(t) -> b; $( true if t is a type manager)
```



```
    INITIALLY
       b = FALSE;

VFUN is_type(c; t) -> b;  $( true iff the type of extended-type
                           object denoted by capability u is t)
    INITIALLY
       b = FALSE;

VFUN impl_cap(c; t; i) -> c1;  $( the ith implementation capability
                                  for the extended-type object
                                  denoted by c, with type manager t)
    EXCEPTIONS
       invalid_object(c, t);
       no_impl_obj(c, t, i);
    INITIALLY
       c1 = ?;

VFUN impl_length(t) -> i;  $( the length of the vector of
                              implementation capabilities for type t)
    EXCEPTIONS
       bad_type_manager(t);
    INITIALLY
       i = ?;

VFUN original(c; t; i) -> b;
         $( true if the object denoted by impl_cap (c, t, i) was
          created originally as an implementation object of the
          object denoted by c)
    EXCEPTIONS
       invalid_object(c, t);
       no_impl_obj(c, t, i);
    INITIALLY
       b = ?;

OVFUN create_type(t; i) -> c;
         $( creates a new extended type (of type TYPE))
    EXCEPTIONS
       t ~= ct;
       i < 1 OR i > max_impl_caps;
       RESOURCE_ERROR;
    EFFECTS
       EFFECTS_OF create_cap() = c;
       'is_type_manager(c) = TRUE;
       'impl_length(c) = i;

OVFUN create_object(t) -> c;
         $( creates a new extended-type of object of type t,
          returns its capability c, and leaves the object
          uninitialized)
    EXCEPTIONS
       bad type_manager(t);
       RESOURCE_ERROR;
    EFFECTS
       EFFECTS_OF create_cap() = c;
```

```
      'is_type(c, t) = TRUE;

OFUN delete_object(c; t);
          $( deletes an extended type object denoted by capability
           c, with type manager t, assuming that all of its
           implementation objects (that are original) have been
           previously deleted)
    EXCEPTIONS
       invalid_object(c, t);
       orig_impl_objs_present(c, t);
    EFFECTS
       'is_type(c, t) = FALSE;
       FORALL i:
          'impl_cap(c, t, i) = ? AND 'original(c, t, i) = ?;

OVFUN create_impl_obj(c; t; i; tl) -> cl;
          $( creates an implementation object at position i of the
           implementation vector for object c. t is the type
           manager's capability for c. tl is the type-manager's
           capability for the type of the object being created)
    EXCEPTIONS
       invalid_object(c, t);
       bad_element(i, t);
       NOT no_impl_obj(c, t, i);
       invalid_impl_type(tl);
       RESOURCE_ERROR;
    EFFECTS
       IF tl = cs
         THEN EFFECTS_OF create_segment(0) = cl
         ELSE EFFECTS_OF create_object(tl) = cl;
       'impl_cap(c, t, i) = cl;
       'original(c, t, i) = TRUE;

OFUN insert_impl_cap(c; t; i; cl);
          $( inserts the capability cl to be get_impl_cap (c, t, i))
    EXCEPTIONS
       invalid_object(c, t);
       bad_element(i, t);
       NOT no_impl_obj(c, t, i);
    EFFECTS
       'impl_cap(c, t, i) = cl;
       'original(c, t, i) = FALSE;

OFUN delete_impl_obj(c; t; i; tl);
          $( deletes the object and entry for the ith
           implementation object of capability c. t is the type
           manager's capability for c. tl is the type manager's
           capability for the implementation capability. the
           implementation object must be original.)
    DEFINITIONS
       capability cl IS impl_cap(c, t, i);
    EXCEPTIONS
       invalid_object(c, t);
       no_impl_obj(c, t, i);
```



```
       NOT exc_original(c, t, i);
       invalid_object(cl, tl);
       orig_impl_objs present(cl, tl);
    EFFECTS
       IF seg_exists(cl)
         THEN EFFECTS_OF delete_seg(cl)
         ELSE EFFECTS_OF delete_object(cl, tl);
       'impl_cap(c, t, i) = ?;
       'original(c, t, i) = ?;

OFUN delete_impl_cap(c; t; i);
          $( deletes the ith element of the implementation vector
           of c, but only if the object is not original)
    EXCEPTIONS
       invalid_object(c, t);
       no_impl_obj(c, t, i);
       exc_original(c, t, i);
    EFFECTS
       'impl_cap(c, t, i) = ?;
       'original(c, t, i) = ?;

END_MODULE
```

SPECIFICATIONS FOR THE TABLE MODULE

```
MODULE tables
          $( The purpose of this module is to provide the
           synchronization required to implement the extended type
           module. It is necessary that extended type operations
           get exclusive access to the segements that implement the
           type (or types) in question. But we do not want these
           operations to tie up operations on other types.
           Therefore, we provide an intervening level in which we
           obtain indivisible access to the table that maps from
           type managers capabilities to type managing segments.)


    DECLARATIONS

capability c, cl;


    EXTERNALREFS


    FROM capabilities:
capability: DESIGNATOR;


    FUNCTIONS


VFUN table(c) -> cl;
    EXCEPTIONS
       table(c) = ?;
    INITIALLY
       cl = ?;

OFUN insert_table(c; cl);
    EXCEPTIONS
       table(c) ~= ?;
       RESOURCE_ERROR;
    EFFECTS
       'table(c) = cl;

OFUN delete_table(c);
    EXCEPTIONS
       table(c) = ?;
    EFFECTS
       'table(c) = ?;

END_MODULE
```


SPECIFICATIONS FOR THE SEGMENT MODULE

```
MODULE segments


    TYPES

bit: { 1, 0 };
bit_string: { VECTOR_OF bit bs | LENGTH(bs) = bit_string_length };
character_string:
{ VECTOR_OF CHAR cs | LENGTH(cs) = character_string_length };
machine_word:
ONE_OF(capability, INTEGER, bit_string, character_string, BOOLEAN);


    DECLARATIONS

capability s;
BOOLEAN b;
machine_word w;
INTEGER i, j;


    PARAMETERS

INTEGER maxsize $( maximum segment size) ;


    DEFINITIONS

BOOLEAN no_seg(capability s) IS seg_exists(s) = FALSE;
BOOLEAN address_bounds(capability s; INTEGER i)
   IS read(s, i) = ?;
BOOLEAN bad_size(INTEGER i) IS i < 0 OR i > maxsize;


    EXTERNALREFS


    FROM capabilities:
capability: DESIGNATOR;
OVFUN create_cap() -> capability s;
INTEGER bit_string_length, character_string_length;


    FUNCTIONS


VFUN seg_exists(s) -> b; $( true for currently existing segments)
    INITIALLY
       b = FALSE;
```

```
VFUN read(s; i) -> w;  $( returns the ith machine work in segment s)
    EXCEPTIONS
       no_seg(s);
       address_bounds(s, i);
    INITIALLY
       w = ?;

VFUN seg_size(s) -> i; $( size of segment s)
    EXCEPTIONS
       no_seg(s);
    DERIVATION
       CARDINALITY({ j | read(s, j) ~= ? });

OVFUN create_segment(j) -> s;  $( creates a new segment s of size j)
    EXCEPTIONS
       bad_size(j);
       RESOURCE_ERROR;
    EFFECTS
       EFFECTS_OF create_cap() = s;
       'seg_exists(s) = TRUE;
       FORALL i | i >= 0 AND i < j: 'read(s, i) = 0;

OFUN delete_seg(s); $( deletes segment s)
    EXCEPTIONS
       no_seg(s);
    EFFECTS
       'seg_exists(s) = FALSE;
       FORALL i: 'read(s, i) = ?;

OFUN write(s; i; w);  $( writes machine_word w into the ith
                        location of segment s)
    EXCEPTIONS
       no_seg(s);
       address_bounds(s, i);
    EFFECTS
       'read(s, i) = w;

OFUN change_seg_size(s; j); $( changes the size of segment s to j)
    EXCEPTIONS
       no_seg(s);
       bad_size(j);
       RESOURCE_ERROR;
    EFFECTS
       FORALL i:
           'read(s, i)
          =(IF 0 <= i AND i < j
              THEN IF read(s, i) = ? THEN 0 ELSE read(s, i)
              ELSE ?);

END_MODULE
```



```
MODULE processes


    DECLARATIONS

INTEGER i, j;
capability p $( process capability) ;
VECTOR_OF capability vc;
VECTOR_OF VECTOR_OF capability vvc;


    PARAMETERS

SET_OF capability process_set $( set of all processes) ;


    DEFINITIONS

INTEGER grabstacksize(p) $( size of the grabstack for process p)
   IS CARDINALITY({ i | grabstack(p, i) ~= ? });
capability grabbed_by(vc)  $( capability for the process that
                              grabbed capability tuple vc)
   IS SOME p |(EXISTS i; j: vc = grabstack(p, i)[j]);


    EXTERNALREFS


    FROM capabilities:
capability: DESIGNATOR;


    FUNCTIONS


VFUN grabstack(p; i) -> vvc;  $( stack element i of vectors of
                               capability tuples grabbed by process
                               p)
    HIDDEN;
    INITIALLY
       vvc = ?;

OFUN grab(vvc)[p];  $( grabs all capability tuples in vvc; blocks
                     if any of them have already been grabbed)
    EXCEPTIONS
       NOT p INSET process_set;
       RESOURCE_ERROR;
    DELAY UNTIL FORALL vc | EXISTS i: vvc[i] = vc:
                  grabbed_by(vc) = ?;
```

```
    EFFECTS
       'grabstack(p, grabstacksize(p) + 1) = vvc;

OFUN release()[p];  $( releases the last capability tuple grabbed
                       by p)
    EXCEPTIONS
       NOT p INSET process_set;
    EFFECTS
       'grabstack(p, grabstacksize(p)) = ?;

END_MODULE
```


SPECIFICATIONS FOR THE CAPABILITY MODULE

```
MODULE capabilities


    TYPES

capability: DESIGNATOR;
bit: { 1, 0 };
bit_string: { VECTOR_OF bit bs | LENGTH(bs) = bit_string_length };
character_string:
{ VECTOR_OF CHAR cs | LENGTH(cs) = character_string_length };
machine_word:
ONE_OF(capability, INTEGER, bit_string, character_string, BOOLEAN);


    PARAMETERS

INTEGER bit_string_length, character_string_length;

    ASSERTIONS

character_string_length > 0;
bit_string_length > 0;


    FUNCTIONS


OVFUN create_cap() -> capability c;
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       c = NEW(capability);

END_MODULE
```

MAPPING FUNCTIONS FOR EXTENDED-TYPES

```
MAP extended_types TO segments, tables, capabilities;
          $( The representation for the extended-type manager
           consists of a type segment for each type maintained by
           it, and a central table that maps type managers'
           capabilities into type segments. The central table is
           maintained by the table module. Each type segment
           consists of an integer specifying the size of the entry
           (impl_length + 2) in location 0, followed by a sequence
           of entries. An entry consists of an extended-type
           capability (a 0 designates a null entry), followed by a
           sequence of impl_length implementation capabilities
           (a 0 designates an undefined impl_cap), followed by a
           single bit string whose ith element is 1 if the ith
           impl_cap is original. If the ith impl_cap is undefined,
           the bit MUST be 0. If the entry is null, all bits MUST
           be 0 and all impl_caps must be 0. The invariants state
           the above, plus the facts that all type segments exist
           and have posistive sizes that are 1 MODULO the entry
           size. The first word of the type segment is always an
           integer, and for each entry of size n, the first n-1
           posistion are either capabilities or 0 and posistion n
           is a bit_string.)
```

TYPES

```
bit: { 1, 0 };
bit_string: { VECTOR_OF bit bs | LENGTH(bs) = bit_string_length };
character_string:
{ VECTOR_OF CHAR cs | LENGTH(cs) = character_string_length };
machine_word:
ONE_OF(capability, INTEGER, bit_string, character_string, BOOLEAN);
machine_word_type:
{ integer_type, capability_type, boolean_type, bit_string_type,
 character_string_type };
```

DECLARATIONS

```
capability c, cl, $( arbitrary capabilities)
          t, $( type manager's capability)
          s; $( segment capability)
BOOLEAN b;
INTEGER i, j, k, kl, nobjs;
machine_word w;
```

PARAMETERS


```
capability ctl,
          csl,  $( representations for type-creation and
                   segment-creation capabilities, respectively)
          ex_secret;  $( a secret capability for grabbing)
```

DEFINITIONS

```
machine_word_type type_of(w)
   IS TYPECASE w OF
       INTEGER: integer_type;
       capability: capability_type;
       BOOLEAN: boolean_type;
       bit_string: bit_string_type;
       character_string: character_string_type;
     END;
bit_string allzeros
   IS VECTOR(FOR i FROM 1 TO bit_string_length: 0);
```

EXTERNALREFS

```
    FROM extended_types:
capability ct, cs;
INTEGER max impl_caps;
VFUN is_type_manager(t) -> b;
VFUN impl_length(t) -> i;
VFUN is_type(c; t) -> b;
VFUN impl_cap(c; t; i) -> cl;
VFUN original(c; t; i) -> b;

    FROM tables:
VFUN table(c) -> cl;

    FROM segments:
INTEGER maxsize;
VFUN seg_exists(s) -> b;
VFUN read(s; i) -> w;
VFUN seg_size(s) -> i;

    FROM capabilities:
capability: DESIGNATOR;
INTEGER bit_string_length, character_string_length;
```

INVARIANTS

```
FORALL c; cl: table(c) = table(cl) => cl = c OR table(cl) = ?;
FORALL c: LET s = table(c) IN
  seg_exists(s) AND
  seg_size(s) > 0 AND
  (LET nobjs = read(s, 0) IN
    type_of(nobjs) = integer_type AND
```

```
    nobjs > 1 AND
    seg_size(s) MOD nobjs = 1 AND
    (FORALL k | k > -1 AND k * nobjs < seg_size(s):
      LET i = k * nobjs + 1 IN
         $( first position contains capability for extended_type
         object or 0 for ?.  if ? then all original positions and
         all impl-caps must be 0)
        (type_of( read(s, i)) = capability_type OR
          (read(s, i) = 0 AND read(s, i+nobjs-1) = allzeros AND
            (FORALL j INSET { 1 .. nobjs - 2 }: read(s, i+j) = 0)))
        AND
         $( the next nobjs-2 positions contain implementation
         capabilities. a 0 in that position means ? and its original
         bit must be 0)
        (FORALL j | j > 0 AND j < nobjs - 1:
          type_of( read(s, i+j)) = capability_type OR
            (read(s, i+j) = 0 AND read(s, i+nobjs-1)[j] = 0) AND
       (FORALL k1 ~= k : read(s, i) ~= read(s, k1*nobjs+1)))));
```

MAPPINGS

```
ct: ct1;

cs: cs1;

max_impl_caps: MIN({ bit_string_length, maxsize - 3 });

is_type_manager(t): table(t) ~= ?;

is_type(c; t):
    LET s = table(t)
     IN(IF s = ?
          THEN FALSE
          ELSE(EXISTS k:
                  k MOD read(s, 0) = 1 AND read(s, k) = c));

impl_length(t): read(table(t), 0) + 2;

impl_cap(c; t; i):
    LET s = table(t)
     IN(IF s = ?
          THEN ?
          ELSE IF i < 1 OR i > read(s, 0) - 2
                THEN ?
                ELSE LET k | k MOD read(s, 0) = 1
                               AND read(s, k) = c
                     IN(IF read(s, k + i) = 0
                          THEN ?
                          ELSE read(s, k + i)));

original(c; t; i):
    LET s | s = table(t)
```



```
    IN(IF s = ?
         THEN ?
         ELSE IF i < 1 OR i > read(s, 0) - 2
               THEN ?
               ELSE LET k | k MOD read(s, 0) = 1
                              AND read(s, k) = c
                    IN(IF read(s, k + i) = 0
                         THEN ?
                         ELSE read(s, k + read(s, 0) - 1)[i]
                              = 1));

END_MAP
```

HIGHER-LEVEL MAPPING FOR CAPABILITIES

```
MAP capabilities TO tables, capabilities;
          $( The hidden capabilities are either capabilities for
           type segments or ex_secret, used for grabbing to ensure
           that no higher level program grabs the same capability
           tuple)


    TYPES

capabilityl:
{ capability c | c ~= ex_secret
               AND(FORALL capability cl: table(cl) ~= c) };


    PARAMETERS

capability ex_secret;


    EXTERNALREFS


    FROM capabilities:
capability: DESIGNATOR;
INTEGER bit_string_length;
INTEGER character_string_length;

    FROM tables:
VFUN table(capability c) -> capability cl;


    MAPPINGS


capability: capabilityl;

bit_string_length: bit_string_length;

character_string_length: character_string_length;

END_MAP
```


MAPPING FUNCTIONS FOR TABLES

```
MAP tables TO segments, capabilities;
          $( The table module is represented in segment st
           (a parameter of this mapping) , with the keys
           (i.e., the arguments to table) stored in even segment
           locations and the corresopnding value stored in the
           stored in the following location. A null entry has key
           st, which is secret from the higher level
           (as specified in the mapping capabilitiesl) %. The
           invariants state that segment st exists, the size of st
           is even, and that each key (except st) occurs at most
           once in st)


    TYPES

bit: { 1, 0 };
bit_string: { VECTOR_OF bit bs | LENGTH(bs) = bit_string_length };
character_string:
{ VECTOR_OF CHAR cs | LENGTH(cs) = character_string_length };
machine_word:
ONE_OF(capability, INTEGER, bit_string, character_string, BOOLEAN);


    DECLARATIONS

INTEGER i, k;
BOOLEAN b;
capability c, cl, s;
machine_word w;


    PARAMETERS

capability st;


    EXTERNALREFS


    FROM tables:
VFUN table(c) -> cl;

    FROM segments:
VFUN seg_exists(s) -> b;
VFUN read(s; i) -> w;
VFUN seg_size(s) -> i;

    FROM capabilities:
capability: DESIGNATOR;
```

```
INTEGER bit_string_length, character_string_length;

    INVARIANTS

seg_exists(st);
seg_size(st) MOD 2 = 0;
FORALL i | i MOD 2 = 0 ; k | k MOD 2 = 0 :
         read(st, i) = read(st, k) =>
            i = k OR read(st, i) = st OR read(st, k) = ?;

    MAPPINGS

table(c):
    LET k | read(st, k) = c AND k MOD 2 = 0
     IN read(st, k + 1);

END_MAP
```


LOWER-LEVEL MAPPING FOR CAPABILITIES

```
MAP capabilities TO capabilities;
          $( This module makes the capability st secret from the
           higher levels, by the appropriate designator mapping)

    TYPES

capabilityl: { capability c | c ~= st };

    PARAMETERS

capability st;

    EXTERNALREFS

    FROM capabilities:
capability: DESIGNATOR;
INTEGER bit_string_length;
INTEGER character_string_length;

    MAPPINGS

capability: capabilityl;

bit_string_length: bit_string_length;

character_string_length: character_string_length;

END_MAP
```

IMPLEMENTATION FOR EXTENDED-TYPE MODULE

```
PROGRAM MODULE extended-types

$( The implementation of extended-types first checks to see
whether the type capability t submitted to it is valid, by looking
it up in the central table (i.e., calling table(t)).  If there is
no entry, an exception is returned.  The result is the type segment
capability, tab.  Any search for an extended-type capability
c involves a linear search over segment locations equal to 1
modulo the entry size (read( tab, 0) = impl_length(t) + 2).
The first location of the entry contains 0 if the entry is null.
The next impl_length slots contain the sequence of implementation
capabilities for c. A 0 designates an undefined implementation
capability.  All slots must be 0 if the entry is null.  The last
location contains a bit vector designating whether the ith
implementation capability is original (i.e., created specifically
for that extended-type object), by a 1 in the ith position of the
bit vector.  The bit vector must contain a 0 in the ith position
if the ith implementation capability is undefined, and a 0 in all
positions if the entry is null.

The creation of a new type involves the creation of a new type
segment and a new type capability, and insertion of the
correspondence between them into the central table.  The creation
of a new object involves the insertion of a new capability (for
the object) in the type segment as either the first location of
a previously null entry or the first location of a new entry
created by lengthening the segment.  Object deletion involves
making the corresponding entry in the type segment null.  The
implementation capabilities and the original bits MUST be zeroed out
by the deletion programs.  Creation and deletion of impl_objects
and impl_caps is straightforward.

"The partitioning for this module is by types:

      <is_type_manager(t), impl_length(t),
        <is_type_manager(c, t), <impl_cap(c, t, i),
          original(c, t, i), for all i>, for all c>>.

The lower level is partitioned as follows:

      <table(t), seg_exists(table(t)),
         <read(table(t), i), for all i>>.

The grab arguments are

      VECTOR( VECTOR(t, ex_secret))

or

      VECTOR( VECTOR(t, ex_secret), VECTOR(t1, ex_secret))
```



```
depending on whether one or two types are involved. ex_secret is a
secret capability for grabbing.")

    TYPES

bit: { 1, 0 };
bit_string: { VECTOR_OF bit bs | LENGTH(bs) = bit_string_length };
character_string: { VECTOR_OF CHAR cs |
                        LENGTH(cs) = character_string_length };
machine_word: ONE_OF( capability, INTEGER, bit_string,
       character_string, BOOLEAN );
not_cap: ONE_OF( INTEGER, bit_string, character_string, BOOLEAN);

    DECLARATIONS

capability c, cl, $( arbitrary capabilities)
           t, $( type manager's capability)
           s, $( segment capability)
           p; $( process capability)
BOOLEAN b;
INTEGER i, j, k;
machine_word w;
VECTOR_OF VECTOR_OF capability vvc;

    PARAMETERS

INTEGER bit_string_length, $( length of a bit string in a machine
                              word)
        character_string_length; $( length of a character string in
                                    a machine word)
capability ctl, csl, $( representations for type-creation and
                        segment-creation capabilities, respectively)
           ex_secret, $( a secret capability for grabbing)
           this_process; $( a capability for this process)

    DEFINITIONS

bit_string allzeros IS
       VECTOR( FOR i FROM 1 TO bit_string_length: 0);

BOOLEAN not_there(c; s) IS FORALL  k | k >= 0:
           LET i | read(s, 0)*k + 1 = i IN read(s, i) ~= c;

INTEGER max_impl_caps IS MIN({ maxsize-3, bit_string_length });

BOOLEAN bad_type_manager(t) IS table(t) = ?;

BOOLEAN invalid_object(c; t) IS LET s = table(t) IN
      IF s = ? THEN TRUE
        ELSE not_there(c, s);

BOOLEAN orig_impl_objs_present(c; t) IS LET s = table(t) IN
       read(s, search_seg(c, s)+read(s, 0)-1) ~= allzeros;

    EXTERNALREFS
```