U.S. Department of Commerce

National Bureau of Standards

Computer Science and Technology

NBS Special Publication 500-67

The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted;
mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

[illegible]
no. 500-67
1980

Computer Science
and Technology

NBS Special Publication 500-67

The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software

Karl N. Levitt
Peter Neumann
Lawrence Robinson*

Computer Science Laboratory
SRI International
Menlo Park, CA 94025

Sponsored by the
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

*Current address:
Ford Aerospace and Communications Corp.
Palo Alto, CA 94306

Special Publication

U.S. DEPARTMENT OF COMMERCE
Philip M. Klutznick, Secretary

Luther H. Hodges, Jr., Deputy Secretary

Jordan J. Baruch, Assistant Secretary for Productivity, Technology and Innovation

National Bureau of Standards
Ernest Ambler, Director

Issued October 1980

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-67
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-67, 54 pages (Oct. 1980)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 80-600157

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $3.75
(Add 25 percent for other than U.S. mailing)

FOREWORD

For nearly a decade, a goal of computer users and computer science researchers has been the development of computer programs with the assurance that the security of information being processed is maintained. The Institute for Computer Sciences and Technology of the National Bureau of Standards has developed a comprehensive program in computer security during this period which includes the investigation of technology that could satisfy this goal. The SRI Hierarchical Development Methodology is the result of a large research effort sponsored by NBS and several other Federal and private organizations. While a complete automated system for producing provably secure systems has not been produced and may not even be feasible in the foreseeable future, the results of the effort have provided a structure for developing secure software where little structure existed before. This report has been produced by the SRI technical staff based on the results of the large effort sponsored by and provided to several organizations. Although NBS cannot endorse the recommendations nor has it verified all of the results contained in the report, NBS is pleased to publish the report so that this approach to producing computer programs with improved security may be studied and applied where needed.

Table of Contents

List of Figures

THE SRI HIERARCHICAL DEVELOPMENT METHODOLOGY (HDM) AND ITS APPLICATION TO THE DEVELOPMENT OF SECURE SOFTWARE

Karl N. Levitt, Peter G. Neumann, and Lawrence Robinson *

Computer Science Laboratory
SRI International
Menlo Park, California 94025

ABSTRACT

This document provides an introduction to the SRI Hierarchical Development Methodology (HDM). The methodology employs a staged decomposition of the development process, which separates design, data representation, and implementation. For any given system development, HDM employs a hierarchical decomposition of the design and formal specifications of modules and their interconnections. Extensive tools are used throughout the development to check the appropriateness of the design and its implementation.

The role of HDM in developing secure systems is considered, and various current efforts using HDM to develop such systems are summarized. The use of the methodology is illustrated by a simple but complete example. A somewhat larger example of part of a secure data management system is also discussed.

Verification is not considered in this document, although HDM does facilitate verification. The consistency of formal specifications and their formal requirements can be formally shown, as can the consistency of programs with their specifications.

Key words: design methodology; formal specification; formal verification; hierarchical design; programming methodology; security.

----

* Larry Robinson is now with Ford Aerospace and Communications Corporation, Palo Alto, California 94306

1. Introduction

This report describes the SRI Hierarchical Development Methodology (HDM), a new approach for designing large software systems such as operating systems and data management systems. It is particularly appropriate for the development of system and application software that must meet stringent security requirements. It is being used in the development of several secure systems and an ultrareliable aircraft flight-control system. Its use provides significant aid in coping with many of the problems hitherto experienced in producing and maintaining such systems.

HDM embodies the following concepts:

- ☐ HDM structures the development process into a sequence of decisions, each of which is described precisely in an appropriate language.
- ☐ HDM separates development decisions into three phases -- design, representation, and implementation.
- ☐ HDM structures a system design as a hierarchy of abstract machines. Each machine is specified independently from other abstractions and independently of any ultimate implementation.
- ☐ HDM facilitates formal verification of designs as well as implementations.
- ☐ HDM is well suited to the design and implementation of secure systems. Selective use of verification (e.g., verification of the security of the design) can further enhance the value of HDM usage.
- ☐ HDM is supported by tools that aid in all stages of development.

In essence, HDM is an attempt to provide languages, guidelines, and tools to permit a designer to manage the complexity found in the development of modern systems.

The evolution of HDM has drawn heavily on many of the important ideas of computer science, most notably those of the following individuals:

- ☐ Dijkstra-- Complexity may be confronted by employing abstraction, in particular by realizing a system as a hierarchy of abstract machines.
- ☐ Parnas-- A system may be separated into modules, the understandability of each of which requires little (if any) knowledge of the inner details of the others. Modules may be specified formally without regard to their implementation.
- ☐ Wirth-- A very simple programming language may be effective, with most complex concepts being handled as abstractions.
- ☐ Hoare-- The data representation of an abstract machine may be formally specified.
- ☐ Floyd-- Program behavior may be formally specified, allowing proof of the consistency between a formal specification and its program code.

In developing HDM as a synthesis of these important notions, we developed a formal specification language (SPECIfication and Assertion Language-- SPECIAL) and included features in both HDM and SPECIAL as needed in the process of developing a number of real systems.

The remaining sections of this chapter summarize the major software advances that have led to HDM, define the concept of a "methodology", and justify the need for a suitable development methodology. The second chapter summarizes the most important features of HDM. The third chapter considers the relevance of HDM in the development of secure systems and applications. The fourth chapter provides a detailed self-contained illustration of the use of HDM by means of a simple example. The final chapter gives a more elaborate example, discussing the design and specifications of two levels of an illustrative secure data management system.

1.1 Advances in Software Technology

For the purposes of motivating the need for a new methodology, it is convenient to divide recent advances into two categories:

- □ New approaches to the design process.
- □ New approaches to the implementation process.

Many approaches to the implementation process have resulted in easily realized significant improvements in large systems. Among these are:

- □ Automatic flow-charters for high-level languages.
- □ Trace-and-interrupt packages in interactive execution.
- □ After-the-fact analysis packages that report on the various aspects of program response for test data, e.g., identifying statements that have not been executed.
- □ Programming without the "go to".

The edict -- thou shall not use the "go to" -- is not a panacea in itself since, as noted by Knuth [24], it is possible to write structured programs with the "go to", and unstructured ones employing only the "modern" control constructs. However, the edict derives from a more powerful notion, namely that by imposing restrictions on the design structure of the software system or on the rest of the development process, significant improvements can be realized. Thus, greater effort in the design process can pay off enormously by reducing the complexity of the implementation. This is the basis for most of the following design advances, all of which we view as fundamental.

Abstraction. The basic principle of abstraction in system development is that in order to solve an extremely difficult problem, it is useful to try to identify the details that are inessential in making a design decision and to hide them from the particular interface. In programming, these hidden details typically relate to what we call *data representation* or *implementation.* This approach is made more precise by some of the following concepts.

Abstract Machines and Hierarchical Decomposition. Dijkstra [11] suggested the following paradigm for "realizing" a program P that is to execute on a machine M. If it is a difficult task to write such a P, then define a new machine Mn and a program Pn whose execution on Mn satisfies the intent of P executing on M. The machine Mn will provide operations that can be invoked in the execution of Pn with some data in structures that will be modified and referenced. These data structures will be *abstract* in the sense that their actual representation in terms of the concrete data structures of M is not apparent to the program using the machine. Consequently, Dijkstra viewed Mn as an *abstract machine*, and Pn as an *abstract program* since it executes on an abstract machine. Now it remains to implement Mn, which is accomplished by viewing another abstract machine M(n-1) and a collection of abstract programs P(n-1), each of which implements an operation of Mn. This process continues until finally an abstract machine M1 is defined that is our target machine M. We denote M1 as the *primitive abstract machine*, "primitive" because it is the lowest-level machine under consideration, and "abstract" because it is not necessarily implemented in hardware. (The total number n of levels is the byproduct of the design process.)

The following important notions can be observed from this paradigm:

1. The system appears to be built as a hierarchy (or a sequence) of abstract machines. Parnas [35] shows that, in any structure said to be a hierarchy, it is necessary to identify the *components* of the hierarchy and the *relation* that binds them. In Dijkstra's view, the abstract machines are the components and the relation "realizes" is the binding relation. The collection of abstract programs executing on M(i-1) realizes the operations of Mi. (It has been suggested by Hamilton [19] and others that the hierarchy should take the form of a tree, rooted at the top. We feel that this notion produces no gain in generality, and thus is not incorporated into HDM; see Chapter 2.)

2. An abstract program executing on Mi can refer only to the operations provided by Mi. This exhibits the principle of *information hiding*, namely, it is easier to control the system development if only a restricted collection of operations is available to a program. The modularity principle discussed below reinforces this point.
3. From the perspective of the abstract program that executes on it, an abstract machine can be viewed as maintaining *abstract data structures* that are modified and queried, using only operations of that abstract machine. These data structures are abstract since they are meaningful only with respect to the operations of the machine, as compared with, for example, the "concrete" data structures of a real machine. This notion of *data abstraction* is extremely important in designing large systems, and offers significant advantages over the more conventional approach of *procedure abstraction.* In procedure abstraction, which is the basis of several contemporary methodologies, a large system is decomposed into procedures, but no attempt is made to form subsets of procedures that logically constitute a data abstraction.
4. The presentation describes the system as if it were realized *top down.* It is perhaps convenient to observe a system a posteriori in this manner. However, actual developments tend to undergo modifications at different levels in orders that are not strictly top-down. HDM recognizes the realities of evolutionary development, and helps to organize it. (Note that Dijkstra suggests that when a particular abstract machine is conceived, the designer usually has in mind certain lower-level machines that will eventually serve to implement it. To some extent, this phenomenon guides the design.)

Modules. All large systems exhibit some form of modularity. Parnas [37] has attempted to define a module more specifically and to show what can be gained by decomposing a system into modules. His view is that the internal details of a module should not affect the functioning of any other module. This property is vital both for understanding what service the module supplies and for limiting the effects of changes to a module. Parnas suggested that a module should be a collection of operations and abstract data structures--like an abstract machine, but with each module typically supporting one (or a few) abstract data concepts. Examples of modules might be file systems, memory managers, and message handlers, although these units can advantageously be substructured into several modules. In our view, an abstract machine is a collection of one or more modules. However, the reader may visualize each level as a single module.

Abstract Data Type. An *abstract data type* is a collection of entities called *objects* and a set of operations defined on objects of the type. Thus, the only access to the objects of an abstract data type is via the operations of the abstract data type. A system can be "realized" as an inverted tree of abstract data types (with the root at the top). A type i residing above type j, type k, ..., implies that the objects of the latter types collectively *represent* the objects of type i. Similarly, the operations of type i are implemented as abstract programs in terms of the operations of the latter types.

Program Specification. We have previously indicated that a formal specification for a program can be given to a user to describe what the program does, and to an implementor to specify the desired behavior of the program. There are several attributes of a good specification, including precision and clarity. Floyd [15] suggested the use of first-order predicate calculus as a specification language, while others-- notably McCarthy [29]--have advocated recursive function theory. In any event, it has become clear that a specification language should be based on a mathematical theory. A specification can also be associated with an abstract machine, module, or abstract data type to portray the effect of invoking operations. Parnas [36] has described a technique (although not a language) for specifying modules, which indicates the effect of invoking each operation on the values of abstract data structures. Our approach to specification is based on that of Parnas.

Another approach, due to Guttag [18] and Liskov and Zilles [28], and as applied to abstract data types, views the operations as mathematical functions. The specification consists of expressions in terms of these functions that describe the value of the functions for any sequence of function applications, i.e., any sequence of operations. A survey of several of the current specification techniques is given by Liskov and Berzins [27].

Program Verification. One approach to determining if a program, or the collection of programs corresponding to an abstract machine, performs as intended is to prove formally that it conforms to a formal specification. The proof process is similar to that employed by a mathematician in demonstrating that a theorem logically follows from a given set of definitions and axioms.

In Floyd's method [16], assertions are placed at strategic places in a program -- the input, the output, and selected intermediate points -- such that each loop contains an assertion. For each path in the program, where a path is defined by a program fragment bracketed by two assertions, a theorem is generated. If the corresponding theorems are true for all such paths (a finite number of theorems is thus assured), then the program is shown to be partially correct with respect to the input and output assertions. (A program is *partially correct* if, for all input data that satisfy the input assertions and that cause the program to halt, the output assertions are satisfied. An extension of the Floyd method can be used to prove that a given program halts for all input data that satisfy specific input assertions.) Other methods have been developed [5] that avoid the need for the intermediate assertions, but yield theorems that are more difficult to prove. Regardless of the method, computer aids are essential to generate the theorems (a relatively easy task) and to aid in proving the theorems (a relatively difficult task and at present a research area). Generalizations of Floyd's method have been developed to prove a hierarchical system of abstract machines ([40], [17], [21]). Since program verification is just emerging as a useful approach, it is not of major concern in this report. However, the techniques that have been used to structure and specify programs to enhance their verification usually also yield programs that have other desirable characteristics [34].

High-Level Programming Languages. The primary original benefit expected of high-level programming languages was the production of programs that could execute on many different machines. This is clearly of economic importance for application programs, and recently for system programs, as their prospects for portability have become enhanced. High-level programming languages also provide built-in powerful features (e.g., storage allocation), thus relieving the burden on the programmer. Recently, new features have been incorporated to aid the programmer in producing more error-free programs. Among these are (1) particular control constructs that often result in programs with cleaner structure, and (2) declarations of strongly typed variables that permit the detection of a large class of programming errors at compile-time. As the concept of data abstraction has become accepted, several recent languages ([10], [26], [47], [23]), have provided facilities for abstract data types. We are in favor of many of these augmentations to the concept of a high-level programming language, but reject the view that programming languages should continue to become more complex in order to provide those features. Many of these features do not aid in the implementation phase of development. Instead we advocate a methodology that provides several languages for system development, one of which is a programming language.

Tools. It has long been recognized that, in certain phases of the development of software, some software tools are extremely valuable in relieving the programmer of the burden of some tasks that are routine but tedious and error-prone. Common examples of such tools are: compilers, assemblers, and loaders. Recently, tools have developed to aid in other phases of system development. Such tools now allow the tracing of a program execution, the recording of program behavior under testing, and the documentation of subprogram interconnections. Although these new debugging tools have produced some benefit, the gain is not what was desired, primarily because these tools were not developed with a set of unified goals. Currently, there is interest in developing a unified collection of tools, sometimes denoted as a programming environment. For example, Teitelman [45] is actively developing the Interlisp environment as an extension of and as support for the language LISP. Interlisp includes special features to allow a programmer to back his program up to a previous execution point. These tools are collected into a single unified package. We intend to develop a similar environment to support HDM--a useful collection of tools already exists--and its extensions, including the incorporation of verification tools as they become available. Such tools are essential to relieve the designer of much tedious work and to ensure that the underlying rules of the methodology are followed.

1.2 Software Development Methodologies and Why They Are Needed

There have been numerous advances in software technology, many of which have been applied to real system development with some success. However, despite this progress, we believe that the practice is still inadequate, intensified by the need for larger and less error-laden systems.

It is of interest to ask why the wisdom of Dijkstra, Hoare, and Parnas, and others has not been widely accepted and creatively applied by software system designers. In our opinion, the reasons are as follows:

- ☐ The ideas represent an inherently new mode of thinking about systems that is not easy to understand or to apply routinely.
- ☐ The ideas have been illustrated only on particular, comparatively simple problems. Many system designers would experience difficulty in extrapolating to more complex problems, e.g., complex operating systems, message processing systems.
- ☐ No languages or formalisms have been provided to enable a designer to formulate decisions according to these ideas.
- ☐ There are gaps in the theory that prevent the application to complex systems.

The net result has been a misapplication of the basic ideas. Witness the intensity generated over "structured programming", a term coined by Dijkstra to denote the new approach to programming based on abstraction. The concept has been so trivialized by many of its current practitioners (some view it as just programming with single-input, single-output blocks or programming without the "go to") that Dijkstra has almost disavowed any connection with the term "structured programming."

A "methodology" for a technical discipline consists of notation, formalism, languages, procedures, and guidelines, all based on scientific principles. It is supported by on-line tools, and illustrated by worked-out examples. In addition, a methodology should be sufficiently robust to allow incremental extension to cover newly discovered problems and advances.

For the development of software, the current optimal choice for the "scientific principles" are the concepts of data abstraction and the mathematical basis of programming. The "procedures" should be of the form that precludes the writing of randomly structured programs, and that requires the statement of decisions in a particular order.

Several other methodologies for software development are being pursued elsewhere. These include (1) Higher Order Software (HOS) [19], (2) Chief Programmer Team [31], (3) various approaches involving structured design, and (4) an approach based on algebraic specifications [18]. We feel that (2) and (3) appear to be too informal and do not embody sufficient data abstraction. The others, although incorporating formalism and data abstraction, have yet to be tested on difficult real systems and do not yet have some of the important ancillary features of a methodology.

2. A Summary of HDM

This chapter provides a brief summary of the Hierarchical Development Methodology. HDM decomposes the design of a system into a hierarchy of abstract machines, linearly ordered with a different abstract machine at each *level* in the hierarchy. Each abstract machine in the hierarchy is dependent only on the functionality of lower-level machines. Each abstract machine provides all of the facilities (operations and abstract data structures) that are needed to realize (i.e., to implement operations of and to represent the data structures of) the machine at the next higher level. The facilities of the highest-level abstract machine, and only those of that machine, are visible to a user of the system. The lowest-level machine, denoted as the primitive machine, contains facilities that the designer deems as primitive, e.g., the hardware on which the system is running or a programming language. A machine is itself decomposed into *modules*, each module having operations and data structures which typically define a single abstract data concept. As in the Parnas module concept, the module is the *programming unit* of HDM; each of the modules may be independently implemented. The programs implementing a module can access the data structures of their own abstract machine, but not those of lower-level machines. Lower-level data structures may be modified only by the execution of lower-level operations. Thus the internal details of a module remain hidden from above the module.

In HDM there is a clear separation of the aspects of system realization into *stages*, as follows:

1. Conceptualization of the system.
2. Definition of the functions of the external interface and the structuring of those functions into a hierarchy of abstract machines, each consisting of one or more modules.
3. Adding further abstract machines to the structure of the entire system, including modules within the hierarchy that are not externally visible.
4. Formal specification of each module.
5. Formal representation of the data structures of each machine in terms of those of the modules at the next lower level.
6. Abstract implementation of the operations of each module, i.e., writing an abstract program for each abstract machine written in terms of the operations at the next lower level.
7. Coding, or transforming the abstract programs into efficient executable programs.

Parnas [38] has characterized software development as a sequence of decisions, where it is likely that decision di is dependent on earlier decisions d1, ..., d(i-1). What Parnas recognized as vital is that there is a proper order for decisions, namely the earlier decisions have the greater impact on the ultimate success of the system. Thus it is vital to identify the important decisions and to evaluate them critically. HDM has been designed to formalize this decision model.

Each of the stages of HDM involves the making of decisions, and HDM provides languages to express these decisions. Those decisions associated with stages (1) through (4) are generally considered as *design*. Those associated with stage (5) and with stages (6) and (7) involve *representation* and *implementation*, respectively. The decisions made from stage (1) to stage (7) are roughly in order of decreasing importance. For example, whether or not to use paging involves a design decision, and is clearly more important than how to store the page table --which is a representation decision. The algorithm for page replacement is an implementation decision. This approach contrasts with the current approach to software realization in which the program itself is used to capture all of the decisions of design, representation, and implementation. In a system designed according to HDM, the four stages would largely be pursued in order. Thus, all of the design decisions should be made before the representation or implementation is attempted. However, backtracking is normally expected. In addition, it is not implied that a designer first considers the highest abstract machine, then the next highest and so on, i.e., top-down design. We would expect that attention would be given to several abstract machines at a time, i.e. when a designer conceives of a particular abstract machine at a position in the hierarchy, he might also have in mind lower level abstract machines to implement that machine. It is also possible for the design to be accomplished top-down while the implementation proceeds bottom-up.

Module specification (stage 4) involves the expression of the intent of a module, independent of its implementation. The language SPECIAL (SPECIfication and Assertion Language) ([42], [39]) is used for this purpose and enables the concise and formal description of a module. SPECIAL is also used for writing intermodule representations (stage 5), which we call *mapping functions.* The intermodule implementation programs (stage 6) are called *abstract programs,* since each can be viewed as running on an abstract machine whose operations they invoke. Abstract programs are intended to be directly compiled into executable code (stage 7). The language used for writing abstract programs can be extremely simple since most of the complexity of the programs is embodied within the abstract machine operations invoked by the programs. We have developed a clean simple language (ILPL -- Intermediate Level Programming Language) to describe abstract programs. Alternatively, programs could be written directly in a modern programming language such as Ada, Euclid, or Modula.

The first three stages of HDM are fundamental to the development. The decisions precisely formulated for these stages provide an early documentation of a system, prior to implementation, and significantly more understandable than the implementation. They thus provide the basis for good implementation. The results of these stages also provide the assertions that define what correctness means for the system. Since each stage of HDM has an appropriate formal language for expressing the decisions made at that stage, machine checking is possible. Existing tools accomplish some types of machine checking for these stages.

The specifications for the highest-level abstract machine are a concise description of the system as seen by the user, but only in terms of those facilities that are relevant to the specifications, i.e., implementation details are omitted from the specification. In addition, the module specifications and mapping functions are used [40] to formulate assertions for the proof of the abstract programs. This report is concerned primarily with the design aspects of HDM, although references are included that discuss techniques for verification in HDM.

HDM is a new synthesis of several promising approaches to software design. It has been developed to address deficiencies in the current software practice. It has been clearly influenced by the concepts of hierarchical programming and its extensions, in particular the important principles of hierarchical design, of doing design prior to implementation, of decomposing a system into small manageable pieces, and of carrying out a proof of correctness simultaneous with design. Although these principles are well-known, they are difficult to apply to real systems. The key to the effectiveness of HDM is that it offers a practical means for constructing, manipulating, evolving, and maintaining formal program abstractions. This property is absent in current structured programming methodologies, and present in only primitive form in modern programming languages. Formal abstraction provides the mechanism for verification, separation of specifications and implementation, variations in the order of binding design decisions, family design, and other desiderata of modern system development.

At present, HDM is evolving and does not yet possess all of the on-line aids that would ease its routine use. For the immediate future (say the next two years), it will see its greatest use in systems where correctness is of extreme concern. We anticipate that in the future, HDM-like methodologies will be an important approach to the design of general software.

This document is intended to serve as an overview of HDM, describing in some detail most of the features needed to design and implement systems. Some attempt is made to justify particular features and to compare HDM with other approaches, but this report is not intended to be a complete survey on design methodologies. A more complete description of HDM can be found in the three-volume HDM Handbook [41, 44, 25].

Chapter 3 of this report discusses the uses of HDM in the development of secure systems and subsystems. Chapter 4 presents an example of the use of HDM, organized according to the stages of HDM outlined above. Chapter 5 presents part of the design of a secure data management system as an illustration of HDM's usefulness in designing a secure application system.

3. The Use of HDM for Attaining Security

This chapter presents some of the important aspects of HDM for developing software satisfying security requirements.

The attainment of security requires an overall perspective on the system needs. In general, it is very difficult (if not impossible) to enforce elaborate security policies in an application environment if the underlying system is insecure -- unless the application environment is extremely restrictive (e.g., has no sharing of resources, or hides all of the facilities of the underlying system). Thus it is necessary to consider the security provided by the operating system, not just the security provided by an application environment. Further, the attainment of security can be adversely affected by improper design, by poor choice of programming language, and by improper implementation. Any weak link could provide a critical flaw.

The ways in which HDM contributes to the attainment of secure software have been considered at length in [34]. These issues are only summarized here.

Suitability for verification is the major factor that differentiates HDM from the wealth of development methodologies. Since correctness is so critical for security, formal verification of security properties is considered mandatory for certain systems. HDM was developed to address the need for verifying large systems. HDM organizes the development into stages, the system into a hierarchy of abstract machines, and the machines into modules to produce units small enough and well structured enough to be amenable to verification. A verification methodology based upon this approach has been developed [40]. However, even if formal verification is not attempted, the precision and discipline imposed by HDM encourage sound design and implementation. The concentration on careful design and matching implementation, and the potential for analysis throughout make HDM an excellent choice when security is an issue.

3.1 Current Uses of HDM for Security

HDM is being applied to the design of several systems with critical requirements. These include secure systems designed at SRI, namely the Provably Secure Operating System (PSOS [33], [14]) and a secure real-time operating system (TACEXEC [13]). (HDM is also being used for NASA by SRI in the development of SIFT, an ultrareliable fault-tolerant computer system [46].)

HDM is being used outside of SRI as well. The Ford Aerospace and Communications Corporation is developing a system [KSOS] whose user interface is compatible with UNIX (Registered Trademark of Bell Laboratories) and which is based on a security kernel [30]. The security of the KSOS design is being subjected to formal proofs that the specifications are consistent with a formal model for multilevel security [2].

At the time of writing, all of the kernel specifications have been subjected to the proof process, and the proofs have pointed out the flaws remaining in the design. Honeywell is using HDM on its own version of KSOS [KSOS-6], and has used it in the past for a flight-control system [3] and for the design and proof of a secure kernel for a Multics-like system (together with the MITRE Corp.) [22]. In addition, there have been and are various other experimental uses of HDM.

Until now, most of the applications of HDM have been to operating systems or kernels in which there are extremely critical requirements. In many of these efforts, verification is an important consideration.

3.2 Requirements

A system should be designed with a clear understanding of what requirements it is to meet, particularly with regard to security. It is desirable to have a precise definition of what it means for a system to be "secure". For example, the PSOS design permits the implementation of highly sophisticated security policies; various properties of the basic PSOS protection mechanism have been formalized. The KSOS design has a security kernel which provides enforcement of a multilevel security policy (under which information at a given security level cannot filter down to a lower level). A formal requirement that the specification for each kernel function satisfies this model is being used for the proofs mentioned above. (An earlier version is given in [12].) It is also applicable to trusted processes that are authorized to selectively violate the security of the kernel.

3.3 Design

HDM enforces constraints on the way in which a design is defined, although it does not essentially constrain what the design can achieve or what functionality can be implemented. Use of hierarchical design structure and formal specifications for each module in the hierarchy contributes to the avoidance of many types of security flaws commonly found in the design and implementation of existing systems. For example, the notion of abstract machine specifications is particularly powerful in isolating a design from its implementation. Thus only the operations of an abstract machine are visible, and the data structures of the abstract machine can be encapsulated within the implementation. The specification language enforces strong "typing" of abstract objects, which also helps to avoid a large class of traditional security flaws. The systematic handling of exception conditions helps to avoid still another class of flaws. A detailed discussion of the use of HDM concepts in evaluating the designs of several existing systems is given in [34]. Proofs of consistency of the specifications with the requirements further help to eliminate design flaws.

3.4 Programming Languages and Implementation

A design specified in HDM may be implemented in a variety of ways. However, the desire for secure systems and for verified systems puts additional requirements on programming, and makes some programming languages much more desirable than others. For example, where secure systems are concerned, use of HDM leads to a design that is compartmented, e.g., to take advantage of the separation of policy and mechanism. It is desirable that these advantages be retained in the programming language.

Several recent programming languages have adopted features that make them desirable for use with HDM in the production of secure software. Such features include the creation and deletion of abstract data types and objects of those types, the strong enforcement of the typing implied by those data types, and the encapsulation of data abstractions and module implementations. These features are also seen to enhance verifiability. Language features for handling exceptions and synchronization of concurrent execution are also emerging. Newer languages that may eventually be useful for writing secure software are Euclid, Modula, Gypsy, and possibly a constrained version of the new DoD/1 language (Ada). It is intended that HDM support a variety of such languages (initially Modula [for KSOS], Pascal [for SIFT], and Ada). (Support for languages such as Fortran and Cobol would be possible, but these languages do not make full use of the power of the methodology.) In general, the use of a modern programming language aids in better software production. The use of program verification can further aid in this process.

3.5 The Role of HDM in Verification

Although verification is not a main thrust of this document, a few comments on verification are appropriate. In general, verifiability is greatly enhanced by the use of HDM [40]. The staged decomposition of the development process permits design proofs to be carried out before implementation is attempted (providing a formal means for early evaluation of the design), and then permits proofs of program correctness. The hierarchical decomposition of the design into levels of abstract machines is particularly valuable in simplifying both the design proofs and the program proofs. Use of formally based languages of HDM is vital to both types of proof. Design proofs demonstrate a formal consistency between the formal specifications (in SPECIAL) and a formal model (e.g., a model of the security requirements). These specifications also form a basis for the program proofs, verifying that the program implementing a module specification is consistent with its specification. As noted above, the choice of programming language can greatly influence the feasibility of verification.

4. A Simple Example of the Use of HDM

In this chapter, HDM is used to describe a complete -- although very simple -- system: a "stack" module implemented in terms of an "array" module. The discussion is organized into seven sections: a review of HDM, and one section for each stage outlined in Chapter 2.

In HDM, a system evolves from an initial concept to verified executable code as a sequence of "decisions". In each stage of the development process, the system developer makes a series of decisions. The stages are ordered so that improper decisions tend to be exposed early, and therefore can be corrected early.

The verification aspects of HDM are found in [40]. Some aspects of verification are discussed below in connection with the first six stages.

A primary concern is to illustrate the staged, decision-oriented development of a system using the three languages of HDM -- HSL (the Hierarchy Specification Language), SPECIAL, and ILPL. Brief introductions to these languages are given to produce a reasonably self-contained description. However, the simplicity of the example does not properly illustrate many of the advantages of HDM as applied to complex systems. More details on HDM and a more complex example appear in [41, 44, 25].

4.1 Review of the Mechanisms of HDM

In HDM, a system is realized as a linear hierarchy (a sequence) of *abstract machines*, sometimes called *levels*. The top level is called the *user-interface*, while the bottom level is called the *primitive machine*. These two machines together are called the *extreme* machines. The remaining levels are called *intermediate machines*. Each machine provides *operations*, each of which has a unique name and arguments. An operation is *invoked*, similar to a subroutine call in a conventional programming language, by associating values for the operation's arguments. The invocation of an operation can return a value and/or modify the *internal state* (abbreviated as *state*) of the machine, as reflected by the values of the machine's *abstract data structures*. As discussed later, the "return" of an operation can be either a value or an "exception", the latter corresponding to one of a number of conditions that are defined for the module.

The "user-interface" provides the operations that are available to the user of the system. The operations of the "primitive machine" are typically constructs of a programming language and possibly some of the hardware operations.

A machine *specification* characterizes the value returned and the new state for each possible machine operation and each possible state of the machine. The specification describes the *functional behavior* of a system (returned values for all input combinations), but not necessarily the performance of the system or the resources consumed by its execution.

The realization of a machine (not the primitive machine, hereafter noted as machine i) is a two step process. First, the abstract data structures of a machine i (i not 1) are *represented* by those of the next lower-level machine i-1. Second, each of the operations of a machine i (i not 1) is *implemented* as a program in terms of the operations of machine i-1. The collection of implementations for all machines excluding the primitive machine constitutes the *system implementation*.

A machine is sometimes decomposed into simpler units called *modules*. For the purposes of this discussion, a module may itself be viewed as a machine; however, in reality a module's specification need not be self-contained, unlike that of a machine.

Clearly, system implementation is the desired end-product of the system development process. However, its emergence takes place only at stage 6. In the five previous stages, important decisions are made that logically progress toward the end product.

4.2 Stage 1 -- Conceptualization

In stage 1, the problem to be solved is formulated in general terms. Typically, the statement is in terms of constraints imposed on the extreme machines, and of the performance expected from the system. Currently, English is employed as the description medium, although consideration is being given to a formal language for conceptualization. For our single example, we will utilize the Conceptualization stage to provide informal descriptions of the extreme machines.

The user interface provides a collection of individually accessible stacks, manipulatable by conventional stack operations. The primitive machine consists of a collection of individually accessible arrays, as provided by a conventional high-level programming language. This example is developed according to the stages of HDM. The completed example is presented in the following figures.

Figure 4-1: Specification of the STACKS Module

```
----------------------------------------------------------------
MODULE stacks  $( maintains a fixed number of stacks of integers,
               each of the same fixed maximum size)

   TYPES

stack_name: DESIGNATOR $( names for stacks) ;

   PARAMETERS

INTEGER max_stack_size $( maximum size for a given stack) ;


   FUNCTIONS

VFUN ptr(stack_name s) -> INTEGER i;  $( stack pointer, or
                         number of elements, of stack s)
   HIDDEN;
   INITIALLY
      i = 0;

VFUN stack_val(stack_name s; INTEGER i) -> INTEGER v;
  $( v is the ith value of stack s)
   HIDDEN;
   INITIALLY
      v = ?;

OFUN push(stack_name s; INTEGER v);
  $( puts the value v on top of stack s)
   EXCEPTIONS
      stack_overflow : ptr(s) = max_stack_size;
   EFFECTS
      'stack_val(s, 'ptr(s)) = v;
      'ptr(s) = ptr(s) + 1;

OVFUN pop(stack_name s) -> INTEGER v;
  $( pops the stack s and returns the old top)
   EXCEPTIONS
      stack_underflow : ptr(s) = 0;
   EFFECTS
      'stack_val(s, ptr(s)) = ?;
      'ptr(s) = ptr(s) - 1;
      v = stack_val(s, ptr(s));

END_MODULE
```

Figure 4-2: Specification of the ARRAYS Module

```
-----------------------------------------------------------

   MODULE arrays $( maintains a fixed number of fixed-size
                 integer arrays)


   TYPES

array_name: DESIGNATOR;


   PARAMETERS

INTEGER array_size $( the number of elements in an array);


   FUNCTIONS

VFUN access_array(array_name a; INTEGER i) -> INTEGER v;
   $( returns element i of array a)
   EXCEPTIONS
     array_bounds : i < 0 OR i > array_size - 1;
   INITIALLY
     v = 0;

OFUN change_array(array_name a; INTEGER i, v);
   $( changes the ith value of array a to v)
   EXCEPTIONS
     array_bounds: i < 0 OR i > array_size - 1;
   EFFECTS
     'access_array(a, i) = v;

END MODULE
```

Figure 4-3: Mappings for STACKS and ARRAYS

```
---------------------------------------------------------

   MAP stacks TO arrays;


   EXTERNALREFS

   FROM stacks:
stack_name: DESIGNATOR;
INTEGER max_stack_size;
VFUN ptr( stack_name s) -> INTEGER i;
VFUN stack_val( stack_name s; INTEGER i) -> INTEGER v;

   FROM arrays:
array_name: DESIGNATOR;
INTEGER array_size;
VFUN access_array( array_name a; INTEGER i) -> INTEGER v;


   INVARIANTS

FORALL array_name a: access_array(a, 0) <= array_size - 1
                     AND
                  access_array(a, 0) >= 0;


   MAPPINGS

stack_name: array_name;

max_stack_size: array_size - 1;

ptr( stack_name s): access_array(s, 0);

stack_val( stack_name s; INTEGER i):
     IF 1 <= i AND i <= access_array(s,0)
       THEN access_array(s, i)
       ELSE ?;


   END_MAP
```

Figure 4-4: Abstract Implementation of the STACKS Module

```
IMPLEMENTATION stacks IN TERMS OF arrays;

   EXTERNALREFS

   FROM stacks:
stack_name: DESIGNATOR;
INTEGER max_stack_size;
OFUN push(stack_name s; INTEGER v);
OVFUN pop(stack_name s) -> INTEGER v;
   FROM arrays:
array_name: DESIGNATOR;
INTEGER array_size;
VFUN access_array(array_name a; INTEGER i) -> INTEGER v;
OFUN change_array(array_name a; INTEGER i, v);

   TYPE MAPPINGS
stack_name: array_name;

   INITIALIZATION
BEGIN
  max_stack_size <- array_size - 1;
END;

   IMPLEMENTATIONS

OPROG push(stack_name s; INTEGER v);
DECLARATIONS
  INTEGER i;
BEGIN
  i <- access_array(s, 0) + 1;
  EXECUTE change_array(s, i, v) THEN
    ON array_bounds : RAISE(stack_overflow);
    ON NORMAL: ;  END;
  change_array(s, 0, i);
END;

OVPROG pop(stack_name s) -> INTEGER v;
DECLARATIONS
  INTEGER i;
BEGIN
  i <- access_array(s, 0);
  IF i = 0 THEN RAISE(stack_underflow); FI;
  change_array(s, 0, i-1);
  v <- access_array(s, i);
  RETURN(v);
END;
END_IMPLEMENTATION
```

4.3 Stage 2 -- Extreme Machine Interface Design

In stage 2, more detail is developed for the two extreme machines, concerned primarily with the decomposition of these machines into modules and the selection of the operations of the constituent modules. An *interface description* is derived for each module, specifying the module's operations and providing supporting information. The interface description is sometimes [18] referred to as the "syntax" of a module, in contrast to the specification (stage 4) which is referred to as the "semantics".

For our example, each (extreme) machine is a single module: "stacks" for the "user-interface", and "arrays" for the "primitive machine". Hence we here refer to "stacks" and "arrays" both as machines and as modules.

4.3.1 Interface Description for "stacks"

```
 MODULE stacks

stack-name: DESIGNATOR

INTEGER max_stack_size

OFUN push(stack_name s; INTEGER v )
OVFUN pop(stack_name s) -> INTEGER v
```

Some brief remarks about the syntax of SPECIAL are appropriate. First, all reserved words are in caps. Second, SPECIAL is a "typed" language in that a type is associated with each item when declared, thus permitting subsequent appearances of the items in a specification (see stage 4) to be checked for consistency with their declared type. For present purposes, a *type* is a set of values. The type INTEGER (a primitive type of SPECIAL) has as values all of the integers -- positive and negative (including zero). The type BOOLEAN (also a primitive type of SPECIAL) has as values TRUE and FALSE. Although not needed for this example, there are additional primitive types. New types, e.g., sets, vectors, structures (records), subtypes, may also be constructed out of existing types.

One or more types noted as *designator types* can be associated with a module. The values of these types, called *designators*, serve as names for abstract objects of the module. The interface description of a module lists all of its designator types. For example, the "stacks" module interface description declares the designator type "stack_name" (an abbreviation for name-of-stack).

Following the designator types, the interface description lists the module's parameters. A parameter of a module is a symbolic constant that, upon initialization of the module, acquires a value which is not subsequently changed by any operation invocation. The parameter mechanism enables a module specification to have some generality. Often a module can appear in different machines in the hierarchy, with a different value for the parameters

Another reason for leaving the values of parameters unbound at specification time is that they are often dependent on the values of lower-level parameters, in a manner that is not decided until later stages.

The "stacks" module has the single integer-value parameter "max_stack_size", whose value is the maximum number of elements that can be in a stack. The reader should observe that we have made the decision for this example that all stacks of the module are of the same fixed size.

Finally, the interface description lists the operations of the module. Depending on whether its invocation returns a value and/or causes a state change, an operation is declared to be one of the following three kinds:

□ V-function (VFUN) -- returns a value, but causes no state change.[1]

[1]Consistent with Parnas' notation [36], we denote operations as "functions", even though they do not necessarily have the properties of mathematical functions.

- □ O-function (OFUN) -- causes a state change, but does not return a value.
- □ OV-function (OVFUN) -- returns a value and causes a state change.

The "stacks" module has two operations:

- □ "push" -- causes an integer v to be placed on top of stack s. [2]
- □ "pop" -- causes the integer value v on the top of the stack s to be removed and returned.

The reader should note that the decision to provide integer stacks is manifested by declaring the second argument of "push" and the returned value of "pop" to be of type INTEGER.

4.3.2 Interface Description for "arrays"

For "arrays", the interface description is as follows:

```
  MODULE  arrays

array_name: DESIGNATOR

INTEGER array_size

VFUN access_array(array_name a; INTEGER i) -> INTEGER V
OFUN change_array(array_name a; INTEGER i, v)
```

The designator type "array_name" has as values the names of arrays maintained by the module. All arrays are of a given fixed size, namely the value given to the parameter "array_size". Two operations are provided:

- □ "access_array" -- returns the integer v in the i-th location of array a.
- □ "change_array" -- causes the integer v to be stored in location i of array a.

As with "stacks", we have declared the values stored in arrays to be of type INTEGER.

4.4 Stage 3 -- Intermediate Machines and Interface Description

In stage 3 "intermediate machines" are selected to bridge the gap between the extreme machines. The choice of intermediate machines is one of the most creative aspects of the use of HDM. In general, relatively simple modules with relatively simple interdependencies are sought. As in stage 2, each intermediate machine is decomposed into modules, each of which is given an interface description. Also in stage 3, a *hierarchy description* of the system is produced in HSL (Hierarchy Specification Language), listing the modules assigned to each machine and the ordering of the machines in the hierarchy.

For the example, no intermediate machines are required. Thus the hierarchy description is

```
  STACKS EXAMPLE

(INTERFACE level1 stacks)
(INTERFACE level0 arrays)

(HIERARCHY stack_example (level0 implements level1)
```

[2]Hereafter we will refer to "stack s" as a shorthand for "the stack that corresponds to the designator s".

4.5 Stage 4 -- Module Specification

In this stage, a specification is written (in SPECIAL) for each of the modules identified in the two previous stages. The specification for the modules that constitute a machine provide a complete description of that machine's functional behavior. Thus the specifications of the "user-interface" modules completely describe the functional behavior of the system.

SPECIAL specifications have been designed to facilitate communication of design decisions and to be machine processable for automatic consistency checking. The semantics of SPECIAL can be stated precisely.

4.5.1 Expressions in SPECIAL

In presenting a program using a conventional programming language, one produces a sequence of *statements*. On the other hand, a specification in SPECIAL is a collection of *expressions*. Each expression is of a particular type, characterizing the type of the values returned by the expression. Expressions are constructed using constants, variables declared in the specification, built-in functions and connectives of the language, functions (O, OV, and V) of the module being specified, and additional functions declared to produce a more readable specification. The following are examples of types of expressions supported by SPECIAL.

1. **Arithmetic Expressions**

The value returned by an arithmetic expression is of type INTEGER or REAL. An arithmetic expression is a single constant, a variable or a user-defined function of type INTEGER or REAL, or is built out of existing arithmetic expressions using the operations "+", "*", "-", "/".

2. **Boolean Expressions**

The value returned by a boolean expression is of type BOOLEAN. The constants TRUE and FALSE are boolean expressions, as are variables and functions declared to be of type BOOLEAN. The operations AND, OR, "~" (NOT) and "=>" (IMPLIES) are used to build up boolean expressions from existing boolean expressions.

3. **Relational Expressions**

Using the infix relational operators (namely "=", "<", "<=", ">", ">=", "~="), boolean expressions are constructed from existing expressions. For "=" (or "~="), the resulting expression is of the form A = B (or A ~= B) where A and B are required to have the same type. For the other operators, each of the two component expressions is required to be of type INTEGER or REAL.

4. **Conditional Expressions**

A conditional expression is of the form IF P THEN Q ELSE R, where P is of type boolean, and Q and R are of the same arbitrary type. The type of the resulting expression is the type of Q (or R).

5. **Quantified Expressions**

To express properties relating to a large number of values, SPECIAL provides quantified expressions that are in the first-order predicate calculus. The universal quantified statement is written as

FORALL x | P(x): Q(x)

or

FORALL x: P(x)=>Q(x).

The meaning is "For all values of x such that P(x) is true, Q(x) is also true." Clearly, P(x) and Q(x) are of type BOOLEAN, as is the type of resulting expression. The variable x can be of any type, usually declared prior to its introduction in the specification.

The existentially quantified statement is written as

EXISTS x | P(x): Q(x),

which has the meaning "There exists a value x such that, if P(x) is true, then Q(x) is also true."

4.5.2 Role of "?" in SPECIAL

SPECIAL provides the particular value UNDEFINED (abbreviated as "?") to stand for "no value". It is used in a specification where the designer wishes to associate the absence of a meaningful value with a data structure. (UNDEFINED should not be confused with "don't care", which stands for some value.) UNDEFINED is only used in a specification, not in an implementation; no operation can return "?" as a value. For purposes of establishing type matching rules, however, "?" is assumed to be a value of every type.

4.5.3 Specification of "stacks"

Now we are ready to discuss the SPECIAL specification of the module "stacks". This specification consists of three *paragraphs*: TYPES, PARAMETERS, and FUNCTIONS. More complex modules would require additional paragraphs, omitted here for simplicity.

1. **TYPES paragraph**

Here the types referred to in the specification are declared. It is required that all designator types (e.g., "stacks" for this module) be declared, but the declaration of other types can be deferred until the first appearance of an item of that type. Note that comments -- $(This is a comment) -- can appear anywhere in a specification.

2. **PARAMETERS paragraph**

All of the parameters are listed as they appear in the interface description of the module.

3. **FUNCTIONS paragraph**

Most of the functionally interesting information in a module specification is embodied in the FUNCTIONS paragraph. Each of the operations of the module ("push" and "pop" for the module "stacks") is listed and individually specified. In addition, other functions, typically V-functions corresponding to data structures, are introduced to assist in the specification of the operations. It is emphasized that, except for the primitive machine, the data structures serve only for purposes of specification.

We separately consider V-functions and O- and OV-functions.

a. **Specification of V-functions**

For purpose of specification, a V-function returns a value and never causes a state change. A V-function is classified as [primitive or derived] and [visible or hidden]. Thus a V-function is one of four flavors, identified by the combination of reserved words that appear in its specification.

The *primitive* V-functions -- "ptr" and "stack_val" for the "stacks" module -- correspond to the module's data structures. Their specification requires the association of an initial value with each possible argument value. That is, all primitive functions are defined to be "total", although many argument values correspond to physically meaningless conditions. For such conditions, the value of the function is usually "?". The expression following INITIALLY specifies the initial value. The primitive v-function "stack_val" returns the INTEGER v corresponding to the i-th location in stacks. We have decided that the initial value v of "stack-val" for any stacks is to be "?" for all i. The expression

v = ?

which is understood to mean

FORALL s; i: stack_val(s, i) = ?

captures this decision. Note that in general the expression need not determine a unique initial value for a primitive V-function.

The other primitive V-function, "ptr" returns the value i of the stack pointer for stack s. The initial value of "ptr" is 0 for all stacks, reflecting the decision that all stacks are to be initially empty.

A *hidden* V-function cannot be called from outside the module, i.e., it is not an operation. The reserved word HIDDEN in the V-function specification declares the function to be hidden. Clearly, "stack_val" should be hidden since only the top element of the stack is to be accessible. However, some designs for a stack allow the pointer to be accessible.

The *visible* V-functions are operations that return a value, but do not cause a state change. They are identified by the absence of the word HIDDEN in the specification. As is the case for all operations, the specification can indicate a list of exception conditions. Since the "stacks" module has no visible V-functions, we defer discussion of exception conditions to the next section.

The value of a *derived* V-function is specified in terms of the values of the primitive V-functions. In the specification of a derived V-function, an expression that defines the returned value appears following the reserved word DERIVATION.

Because a V-function can serve multiple roles (say as an operation and a data structure), the length of a SPECIAL specification can be reduced, as compared with an alternative specification technique in which operations and data structures are separately specified.

b. **Specification of O- and OV-functions**

All O- and OV-functions are state-changing operations. An operation can return one of n exceptions ex1, ex2, ..., exn (we use the descriptive term "raise" in referring to exceptional returns), or can return "normally". No state change occurs when an operation invocation raises an exception. A value-returning operation (V- or OV-function) will return an actual value upon the NORMAL return; an O-function merely returns. Exception returns are a way of associating particular events with classes of states and values of the operation's arguments. In the specification of an operation, the specification of each exception condition consists of a name (typically a mnemonic for the condition) followed by a boolean expression that characterizes the condition. The list of exception conditions follows the reserved word EXCEPTIONS.

The behavior of an operation that has n exception conditions is determined as follows: if the expression corresponding to ex1 evaluates to true, then the first exception is raised; if the expression corresponding to ex1 evaluates to false and the expression corresponding to ex2 evaluates to true, then the second exception is raised; ...; finally, if the expressions corresponding to ex1, ..., exn evaluate to false, the operation returns normally.

For the O-function "push", there is the single exception condition, specified as:

```
stack_overflow: ptr(s) = max_stack_size
```

The expression evaluates to true when the number of elements in the stack is equal to the maximum size of a stack.

Following the reserved word EFFECTS, the state changes that can occur as associated with O- and OV-functions, together with the value corresponding to the NORMAL return of an OV-function, are specified. The specification consists of a collection of boolean expressions, each called an *effect* (in which the order of presentation is irrelevant). Semantically, the collection of effects should be read as a single expression which is the conjunction of the expressions corresponding to each of the effects. An effect can reference the following: arguments to the operation, values of primitive V-functions before the invocation ("old" values) of the operation, and value that primitive V-functions will obtain after the invocation ("new" values). In the specification, a single quote, "'" , preceding a primitive V-function indicates the value of the V-function after the invocation. The collection of effects defines the new value of each primitive V-function in terms of old values and argument values in the following way: the feasible new values for the primitive V- functions are those for which each of the effects evaluates to TRUE. Thus the specifications need not be *deterministic*, i.e., they need not define a unique new value for each primitive V-function argument list. However, the specifications for our simple example are deterministic.

When the new value of a primitive V-function for some argument is not constrained by the specification, it is assumed that the new value is identical to the old value.

For "push", the effects are:

```
'stack_val(s, 'ptr(s)) = v;
'ptr(s) = ptr(s) + 1;
```

They constrain the new value of "ptr(s)" to be the old value incremented by one, and the new value of the pointer for s to be the value v pushed onto the stack. Note that since the effects do not constrain the values of stack_val(s,i) for i ~= 'ptr(s), such values remain unchanged.

We will not burden the reader with a discussion of the effects for "pop", except for a few remarks. First, note that the returned value v is specified to be the INTEGER on the top of the stack in the old state. Second, the location at the top of the stack is the old state changed to be "?". It should be clear that this latter state change is apparent only in the specification. The implementation need not be concerned with this apparent storing of "?".

4.5.4 Specification of "arrays"

Since the specification of the module "arrays" is relatively straightforward, only a few clarifying remarks are necessary. The V-function "access_array" serves both as the principal data structure of the module and as an operation. Its invocation raises an exception if the actual argument i is out of bounds. Thus, although the function is defined to be total, its representation (for example, as a data structure in a programming language) need only account for values of i that are within bounds.

4.6 Stage 5 -- Data Representation

4.6.1 Overview of Module Representation

In this stage, the primary concern is with representing the data structures of each machine (other than the primitive machine) in terms of the data structure of the next lower-level machine. The description of the representation of a machine m in SPECIAL is denoted as the "m mapping". As with a module specification, a mapping can be checked for self-consistency, but also for consistency with the module specifications, interface description, and hierarchy description.

A mapping, similar to a module specification, does not act as executable code. Instead, a mapping is a formal description, serving as a record of the representation decisions and as an input to a verification system. Thus the representations are conveniently described using the SPECIAL expression mechanism.

Since the hierarchy for our example contains only two levels, only one mapping is required, for "stacks". The mapping contains three paragraphs: EXTERNALREFS, INVARIANTS, and MAPPINGS. (For more complicated systems, additional paragraphs would be required.) Before discussing the mapping in detail, it is appropriate to present informally the basic representation decisions.

4.6.2 Representation Decisions for "stacks"

Each stack of integers is represented as an integer array. The current value of the stack pointer for stacks is the value in the 0-th location of the array a corresponding to s. Each of the "defined" elements in stack s -- those in position 1, 2, ..., ptr(s) -- are in corresponding positions of array. Thus the locations of array a starting with location ptr(s) + 1 hold values that are inconsequential to the "stacks" module. Since all locations of the array except the 0-th are available to hold stack elements, the maximum stack size is the array size minus one.

4.6.3 EXTERNALREFS Paragraph

In the EXTERNALREFS paragraph are listed all the items of the upper level that are to be represented, and those of the lower level that are the targets of the representation. For both levels, the items of concern are primitive V-functions, parameters, and designator types. Clearly, the primitive V-functions and parameters are of concern here as they are the data structures of the respective machines. However, the mapping must also consider the designator types of the upper modules, since they embody a set of values that have meaning only at the upper module, and thus are part of the data of that module. The inclusion of type information here (although redundant with information in the module specification) permits the type checking of a mapping as a self-contained unit.

4.6.4 MAPPINGS Paragraph

In this paragraph the representation decisions that were informally presented above are precisely formulated. Each upper-level data item is separately represented, that is, associated with an expression in terms of lower-level data items. The expression associated with an upper-level data item can be viewed as a definition of that item in terms of the data items at the next lower level.

The first of the mappings

```
stack_name: array_name
```

captures the decision that the type "stack_name" is to be represented by the type "array_name". It is understood that each designator s of "stack_name" is to be represented by a unique designator a of "array_name", although at this point it is not necessary to define precisely the correspondence between values of the two types. In general, a designator type of the upper level can be represented by any type of the lower level. Thus, designators can be represented (for example) by integers; indeed, assuming that a primitive machine supporting designators is not available, the ultimate representation of designators is likely to be in terms of such primitive data types as integers, characters, or machine words.

The second of the mappings

```
max_stack_size: array_size-1
```

captures the decision that the maximum number of stack elements is one less than the size of an array.

The third of the mappings

```
ptr(stack_name s): access_array(s, 0)
```

captures the decision that the stack pointer is stored in the 0-th location of the corresponding array. Note that s, declared to be of type "stack_name", appears in the defining expression in a context in which "stack_name" has no meaning. Clearly, s in the defining expression refers to the unique "array_name" designator corresponding to s. In general, when an argument a of some type t associated with the upper level appears in the defining expression, it is assumed to be the unique element a' that is the representation of a. Thus the type of a in the defining expression is t', where t' is the type that represents t.

The fourth of the mappings

```
stack_val(stack_name s; INTEGER i) :
  IF i > 0 AND i <= access_array (s, 0)
    THEN access_array (s, i)
    ELSE ?
```

captures the decision that "defined" elements of the stack appear in corresponding elements of the array. For i corresponding to an undefined stack element, the expression must evaluate to "?"

4.6.5 INVARIANTS Paragraph

This paragraph contains boolean expressions (invariants) in terms of the lower level that are intended to be true after the execution of a program that implements an operation of the upper level machine. In effect, the invariants express constraints on the lower level state. It should be understood that the invariants are expected to be satisfied by any program referring to the operations of the lower-level machines. Generally, many invariants can be posed, but only those that assist in the verification, that are significant in the documentation of the system, or that simplify the implementation are included.

The single invariant of our example

```
FORALL array_name a:
  access_array(a, 0) <= array_size-1
      AND
  access_array(a, 0) >= 0
```

constrains the value in the 0-th location of all arrays to be bounded by 0 and array_size-1. Since the stack pointer is stored in the 0-th location of the corresponding array, this invariant indeed seems reasonable.

4.7 Stage 6 -- Abstract Implementation

In stage 6 each machine (other than the primitive machine) is implemented in terms of the machine at the next lower level. For machine i, the implementation consists of

- □ An *initialization* program whose execution causes the state of machine i-1 to become a state that maps (up) to the initial state of i; A program for each operation of i that satisfies its specifications.

All programs of the implementation of i reference operations of i-1.

For expressing the implementation programs, we have developed the language ILPL (Intermediate Level Programming Language). Although, in principle, almost any programming language could be used to express machine implementations, ILPL is particularly well-suited in that its syntax, type checking rules, and model of computation are compatible with the other languages of HDM.

We will not present a detailed description of ILPL, but instead illustrate some of its features in connection with the implementation of "stacks". First, we present a brief overview of the language.

4.7.1 Overview of ILPL

ILPL is an extremely simple imperative language, avoiding many of the complex features of high-order programming languages. The main purpose of ILPL is to describe a sequence of calls to operations. Some of the significant features of ILPL are the following:

- □ Simple argument passing discipline: In ILPL, all arguments are passed by "value". Of the conventional schemes for passing arguments -- by "value", by "reference", and by "name" -- "call by value" is conceptually the simplest. It has several advantages in implementing secure systems, including the avoidance of a wide class of security bugs referred to as "time-of-creation to time-of-use" modifications [34].
- □ Limited built-in data structures: In HDM, most of the data structures are provided by specified modules. Thus, ILPL need provide only a few simple kinds of data types, namely integers, characters, booleans, vectors, and structures (records).
- □ Controlled side-effects: Since an ILPL program consists mainly of calls to operations of a machine, the only side-effects are changes to V-functions as portrayed in the specifications of modules.
- □ Simple storage allocation: The only allocation carried out in the execution of an ILPL program is for local variables. Any dynamic allocation of objects is reserved for the modules that maintain such objects.
- □ No design aids in the language: Since HDM separates design and implementation decisions into distinct stages, all descriptions relating to design are expressed in SPECIAL or HSL.
- □ Structured exception handling: The program implementing an operation has multiple return points, one corresponding to the normal return and the remainder corresponding to the exceptional returns. A program referencing an operation "handles" any of the possible returns -- exceptional or normal -- for that operation.
- □ Type compatibility with SPECIAL: ILPL provides only a subset of the types of SPECIAL, essentially those that are easily implemented. Among those omitted is the "set". However, ILPL does support designator types, enforcing the same protection rules for designators as SPECIAL.

The implementation of "stacks" contains four paragraphs: EXTERNALREFS, TYPE MAPPINGS, INITIALIZATION, and IMPLEMENTATIONS, discussed next.

4.7.2 EXTERNALREFS Paragraph

All of the operations of both levels are listed. Also included are the parameters of both machines (since they can be referenced as operations) and the designator types. Complete type information is given here for all arguments and results, even though it duplicates information in the module specifications, allowing the implementations to be complete for purposes of type checking.

4.7.3 TYPE MAPPINGS Paragraph

The mappings of the designator types of the upper machine are listed. Again, this information has already appeared (in the representation), but its appearance here means that the implementation is self-contained.

4.7.4 INITIALIZATION Paragraph

The "initialization" program is given which when executed will drive the lower-level machine to a state that maps to the desired initial state of the upper-level machine. For the example, the initialization program has only to establish a value for the "stacks" parameter "max_stack_size". Note that the image of the initial state of "arrays" is such that "ptr(s)" has the initial value 0.

The reader might wonder how the initial value of "?" for stack_val(s, i) is realized. Recall that the representation for "stack_val" is

```
stack_val(s; i) :
  IF i >= 1 AND i <= access_array(s, 0)
    THEN access_array(s, i)
    ELSE ?
```

But the initial value of access_array(s, 0) is 0, in which case the expression following IF is false for all i. Hence, for the initial state of "arrays" the representation of "stack_val" becomes

```
stack_val(s, i) = ?
```

which is the initial value desired.

4.7.5 IMPLEMENTATIONS Paragraph

Following are the programs that implement the operations of "stacks". The informal description of the program for "push" should serve as a documentation of the program, and assist a reader in grasping the syntax of ILPL.

```
   INFORMAL DESCRIPTION OF "push"
Retrieve the 0-th element of the array (p, the stack pointer);
If i=p+1 is beyond the array bounds (and thus exceeds the
  maximum stack size), raise the "stack_overflow" exception
  and exit;
Modify the i-th location in the array to be v (push v onto
  the stack);
Modify the 0-th location in the array to be i (increment the
  the stack pointer);
```

For the actual program, the first statement is

```
i <- access_array(s, 0) + 1
```

No exception is expected from the invocation of access_array(s, 0), since the second argument (0) is clearly in bounds. The second statement

```
EXECUTE change_array(s, i, v) THEN
  ON array_bounds : RAISE(stack_overflow);
  ON NORMAL : ;
  END;
```

illustrates the mechanism for exception handling in ILPL. Following EXECUTE is a reference to an operation, change_array(s, i, v), that can lead to an exception, in this case "array_bounds". The text following ON has the following meaning: If the "array_bounds" exception is raised as a result of the invocation of "change_array", then the "stack_overflow" exception is raised as the termination of the program for "push". (If the "change_array" operation had more exceptions than were expected, they would be accommodated by additional "ON" terms.) If the "array_bounds" exception is not triggered, then the invocation of "change_array" terminates "normally" by storing v in the i-th location of array s.

To complete the description of the program, no exception is expected for the statement

```
change_array(s, 0, i),
```

since it is seen that i is within bounds.

4.8 Stage 7 -- Coding

The abstract programs associated with stage 6 must ultimately be transformed into efficiently executable programs. That is the task of stage 7. In general, the task may be accomplished automatically or manually. The choice may rest on the actual hardware and on the tools available for compiling or assembling code. The development of automatic tools for accomplishing stage 7 is encouraged.

5. Illustrative Design of a Secure DMS

5.1 Introduction

This appendix illustrates the use of HDM in the design of secure applications subsystems by giving a skeletal design for a secure data management system. The fundamental emphasis is on the application of the methodology, and not on the detailed design of the data management system. Thus, the design is intentionally incomplete as it is intended to serve primarily as a vehicle for the illustration.

The goals of such a data management system are familiar: 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. The system should also be reliable and efficient in operation, and easy to use.

Consistent with the goal of demonstrating the suitability of HDM for developing an applications subsystem with appropriate security, the concept of a relational data base has been chosen as representative of suitable generality and scope. It is recognized that there is much controversy in the data management world over the various types of data management systems, and that there are applications for which the relational approach may be inappropriate. The choice of a relational model here should be construed not as an unqualified advocacy of that approach, but rather as illustrative.

The emphasis in the design given here is on the mechanisms of the intermediate levels of a 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. Similarly, lower-level issues such as operating system efficiency are also considered as secondary issues for present purposes.

5.2 Overview of the Design

The present design is fairly simple, but general. It can be easily embellished so as to increase efficiency or ease of use. However, whereas 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

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.

5.3 Relations

The literature on the relational model of data has been growing rapidly in the last few years, beginning with Codd [8]. For various developments, the reader is referred to [7], [32], [20], and [43], for example. An excellent survey is given by Chamberlin [6].

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 [7], 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.

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 Figure 5-1.) 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 Figure 5-1). (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" [6]. 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.)

5.4 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 tl_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.

5.5 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 Figure 5-2.

Specifications for the modules RELATIONS and PRIMITIVE-VIEWS are given in Figures 5-4 and 5-5, respectively. 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 tl_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 da'a 'rough a view. Similarly, operations exist to update, insert, and delete through a view. In the case of updating, .'e 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 Figure 5-1.

5.6 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).

5.7 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 [4], and [7].) 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.

5.8 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.

5.9 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.

In the present design, a view 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.

5.10 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. Figure 5-1 provides a summary of specific functions of each level.

The design is conceptually similar in concept to those of [32], [43], and [1].

5.11 The Detailed Design

The specifications for the RELATIONS module and for the PRIMITIVE VIEWS level are given in Figures 5-4 and 5-5, respectively. Figures 5-2 and 5-3 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".

5.12 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 corresponding directly to the designators for those abstractions (e.g., see [33]), 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 updating (e.g., batched updating) of data bases. For an interesting discussion of various implementation issues, see [9].

5.13 Conclusions

This appendix 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 systems. It is noted that the machine-independence of the relational interface makes it appropriate for implementation on a wide range of hardware and operating systems.

Figure 5-1: Summary of 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

Figure 5-2: Result of *extract*(rdb,v) on Tuple tw[i], if Selected

EXTRACT:		tw[i]			
twl[i]=		%	*	w	w'
vw[i]	%	%	%	%	%
	*	%	*	w	w'
	w	-	-	w	-

Note: "-" indicates no selection; w' ~= w.

Figure 5-3: Result of *and_views*(vw[i],vwl[i]); w' ~= w.

ANDVIEWS:		vw[i]			
vw2[i]=		%	*	w	w'
vwl[i]	%	%	%	%	%
	*	%	*	w	w'
	w	%	w	w	%

Figure 5-4: Specifications for the module RELATIONS

```
*****************************************************************
Relational data bases -- definitions

DOMAIN d: variable of a particular type whose range is a set of
 possible values

SCHEMA s: vector of domains

RELATION r for a schema s: subspace of the schema, representable
 as a set t_set of tuples (see below), plus
 authorization information.

TUPLE t of a relation r for a schema s: vector tw of values tw[i]
 (one per domain d_list[i] in s)

RELATIONAL DATA BASE rdb for a set of domains: set of relations
 for schemas on subsets of those domains.
*****************************************************************


MODULE  relations

   TYPES

domain : DESIGNATOR ;
schema : DESIGNATOR ;
tuple : DESIGNATOR ;
relation : DESIGNATOR ;
data_base : DESIGNATOR ;
character_string : VECTOR_OF CHAR ;

   DECLARATIONS

INTEGER i, j;
domain d;
schema s;
tuple t;
relation r;
data_base rdb;
character_string w $( domain value);
SET_OF character_string range $( range of value);
SET_OF domain d_set $( domains defined for a data base);
VECTOR_OF domain d_list $( schema definition);
VECTOR_OF character_string tw, tw1 $( tuple values);
SET_OF tuple t_set $( set of tuples in a relation);
SET_OF relation r_set $( set of relations in data base);
SET_OF data_base rdb_set;
VECTOR_OF BOOLEAN bv;
```

```
   PARAMETERS

INTEGER access_length $( length of access vector = 7);
INTEGER grant, revoke, insert, delete, update, destroy, read
        $( access codes for relations, distinct integers assumed
        among 1, 2, ..., access_length.);

   DEFINITIONS

BOOLEAN no_data_base(data_base rdb) IS
       NOT rdb INSET get_data_bases();

BOOLEAN no_relation(data_base rdb; relation r) IS
       NOT r INSET get_relations(rdb);

BOOLEAN no_domain(data_base rdb; domain d) IS
       NOT d INSET get_data_domains(rdb);

BOOLEAN no_schema(data_base rdb; schema s) IS
       get_domains(rdb,s) = ?;

BOOLEAN repeated_domains(data_base rdb;
                    VECTOR_OF domain  d_list) IS
       EXISTS  i : EXISTS  j :  i ~= j
                          AND  d_list[i]
                             = d_list[j];

BOOLEAN out_of_range(data_base rdb; domain d;
                character_string w) IS
       NOT w INSET get_range(rdb, d);

BOOLEAN some_out_of_range(data_base rdb; schema s;
                  VECTOR_OF character_string tw) IS
       EXISTS i : NOT tw[i] INSET get_range(rdb,
                  get_domains(rdb,s)[i]);

BOOLEAN domain_not_in_schema(data_base rdb; schema s; domain d) IS
       FORALL  i : d ~= get_domains(rdb, s)[i];

BOOLEAN no_tuple(data_base rdb; relation r; tuple t) IS
       NOT t INSET get_tuples(rdb, r);

BOOLEAN no_ability(data_base rdb; relation r; INTEGER i) IS
       get_access(rdb, r)[i] = FALSE;

   FUNCTIONS

VFUN get_data_bases() -> rdb_set;
    $( collection of relational data bases)
       INITIALLY
          rdb_set = {};
```

```
VFUN get_data_domains(rdb) -> d_set;
    $( set of domains for the data base)
       INITIALLY
          d_set = ?;

VFUN get_relations(rdb) -> r_set;
    $( relations in relational data base rdb)
       INITIALLY
          r_set = ?;

VFUN get_schema(rdb; r) -> s;
    $( the schema used for relation r)
       INITIALLY
          s = ?;

VFUN get_domains(rdb; s) -> d_list;
    $( list of domains in schema s)
       INITIALLY
          d_list = ?;

VFUN get_range(rdb; d) -> range;
    $( range of the domain variable)
       INITIALLY
          range = ?;

VFUN get_tuples(rdb; r) -> t_set;
    $( tuples in relation r)
       INITIALLY
          t_set = ?;

VFUN get_value(rdb; r; t; d) -> w;
    $( value for domain d in tuple t)
    EXCEPTIONS
       no_data_base(rdb);
       no_relation(rdb, r);
       no_tuple(rdb, r, t);
       no_domain(rdb, d);
       no_ability(rdb, r, read);
    INITIALLY
       w = ?;

VFUN get_values(rdb; r; t) -> tw;

    DEFINITIONS
       VECTOR_OF domain  d_list IS get_domains(rdb, s);
       schema s IS get_schema(rdb, r);
```

```
   EXCEPTIONS
      no_data_base(rdb);
      no_relation(rdb, r);
      no_tuple(rdb, r, t);
      no_ability(rdb, r, read);
   DERIVATION
      FORALL  i :  tw[i]
              = get_value(rdb, r, t, d_list[i]);

VFUN get_access(rdb; r) -> bv;
    $( the access authorization for relation r)
      INITIALLY
         bv = ?;

OVFUN create_data_base() -> rdb;
    $( creates a new data base)
      EXCEPTIONS
         no_data_base(rdb);
      EFFECTS
         rdb = NEW(data_base);
          'get_data_bases()
         = get_data_bases() UNION {rdb};

OVFUN create_domain(rdb; range) -> d;
    $( creates a new domain for the data base)
      EFFECTS
         d = NEW(domain);
         'get_range(rdb, d) = range;

OVFUN create_schema(rdb; d_list) -> s;
    $( creates a new schema with d_list as its domains.  Identically
      composed but differently designated domains may be included,
      while the identically designated domain may not appear twice.)
      EXCEPTIONS
        no_data_base(rdb);
        repeated_domains(rdb,d_list);
      EFFECTS
         s = NEW(schema);
         'get_domains(rdb, s) = d_list;

OVFUN create_relation(rdb; s) -> r;
    $( creates a new relation for the given schema)
      EXCEPTIONS
        no_data_base(rdb);
        no_schema(rdb,s);
   EFFECTS
      r = NEW(relation);
      'get_relations(rdb)
         = get_relations(rdb) UNION {r};
      'get_schema(rdb, r) = s;
      FORALL  i | 1 <= i AND i <= access_length :
        'get_access(rdb,r)[i] = TRUE;
```

```
OVFUN create_tuple(rdb; r) -> t;
    $( creates a new tuple t in relation r
    (with undefined values) in implementation, t is probably an
    integer, or else, identified by a domain value as a key.)
   EXCEPTIONS
      no_data_base(rdb);
      no_relation(rdb,r);
      no_ability(rdb, r, insert);
   EFFECTS
      t = NEW(tuple);
       'get_tuples(rdb, r)
      = get_tuples(rdb, r) UNION {t};

OFUN delete_tuple(rdb; r; t);
    $( deletes tuple t from relation r)
   DEFINITIONS
      VECTOR_OF domain  d_list IS get_domains(rdb, s);
      schema s IS get_schema(rdb, r);
   EXCEPTIONS
      no_data_base(rdb);
      no_relation(rdb,r);
      no_ability(rdb, r, delete);
      no_tuple(rdb,r,t);
   EFFECTS
       'get_tuples(rdb, r)
      = get_tuples(rdb, r) DIFF {t};
      FORALL  i : 'get_value(rdb, r, t, d_list[i]) = ?;

OFUN update_tuple_value(rdb; r; t; d; w);
    $( in tuple t of relation r, updates the value for domain d
    to w.)
   EXCEPTIONS
      no_data_base(rdb);
      no_relation(rdb,r);
      no_ability(rdb, r, update);
      no_tuple(rdb,r,t);
      domain_not_in_schema(rdb,s,d);
      out_of_range(rdb,d,w);
   EFFECTS
      'get_value(rdb, r, t, d) = w;
```

```
OFUN update_tuple(rdb; r; t; tw1);
    $( replaces the entire tuple)
   DEFINITIONS
      schema s IS get_schema(rdb,r);
      VECTOR_OF domain d_list IS get_domains(rdb,s);
       $( VECTOR_OF character_string  tw IS get_values(rdb, r, t))
   EXCEPTIONS
       no_data_base(rdb);
       no_relation(rdb,r);
       no_ability(rdb, r, update);
       no_tuple(rdb,r,t);
       some_out_of_range(rdb,s,tw1);
   EFFECTS
       FORALL  i :  'get_value(rdb, r, t, d_list[i]) = tw1[i];

OFUN destroy_relation(rdb; r);
    $( destroys relation r from data base rdb)
   DEFINITIONS
       SET_OF tuple  t_set IS get_tuples(rdb, r);
       VECTOR_OF domain  d_list IS get_domains(rdb, s);
       schema s IS get_schema(rdb, r);
   EXCEPTIONS
       no_data_base(rdb);
       no_relation(rdb,r);
       no_ability(rdb, r, destroy);
   EFFECTS
        'get_relations(rdb)
       = get_relations(rdb) DIFF {r};
       'get_tuples(rdb, r) = ?;
       FORALL  t |  t INSET t_set :
              FORALL  i :  'get_value(rdb, r, t, d_list[i]) = ?;
END_MODULE
```

Figure 5-5: Specifications for the module PRIMITIVE-VIEWS

```
*****************************************************************
PRIMITIVE_VIEW v for a relation r: a vector vw of values vw[i]
(one per domain d_list[i] in the schema of the relation), plus
authorization. Each vw[i] is either a value w, or the universal
value "*", or the null value "%".

NONPRIMITIVE VIEW: a set of two or more primitive views
for the same schema.

VIEW view for a schema: any set of primitive views.

VIEW_DIR view_dir for a data base: the set of primitive views
maintained by the VIEWS module.
*****************************************************************


MODULE  primitive_views

   TYPES

primitive_view : DESIGNATOR ;
character_string : VECTOR_OF CHAR ;
view_vector : VECTOR_OF character_string ;

   DECLARATIONS

BOOLEAN b;
INTEGER i, j;
domain d;
tuple t, t1;
relation r;
data_base rdb;
character_string w, w1, w2;
SET_OF character_string range;
primitive_view v, v2, vj;
view_vector vw, vw1;
SET_OF primitive_view view_set;
SET_OF tuple t1_set $( extracted relation);
VECTOR_OF BOOLEAN bv, bv1;
SET_OF data_base rdb_set;
SET_OF relation r_set;
SET_OF domain d_set;
SET_OF tuple t_set;
VECTOR_OF character_string tw;

   PARAMETERS

INTEGER grant, revoke, insert, delete, update, destroy, read;
```

```
   DEFINITIONS

BOOLEAN no_view(data_base rdb; primitive_view v) IS
      NOT  v
         INSET get_views(rdb, get_view_relation(rdb, v));

BOOLEAN is_revocable(data_base rdb; primitive_view v) IS
      revocable(rdb, v) = TRUE;

BOOLEAN no_data_base(data_base rdb) IS
      NOT rdb INSET get_data_bases();

BOOLEAN no_relation(data_base rdb; relation r) IS
      NOT r INSET get_relations(rdb);

BOOLEAN no_domain(data_base rdb; domain d) IS
      NOT d INSET get_data_domains(rdb);

BOOLEAN out_of_range(data_base rdb; domain d;
                character_string w) IS
      NOT w INSET get_range(rdb, d);

BOOLEAN no_ability(data_base rdb; relation r; INTEGER i) IS
      get_access(rdb,r)[i] = FALSE;

BOOLEAN no_view_ability(data_base rdb; primitive_view v; INTEGER i) IS
      get_view_access(rdb, v)[i] = FALSE;

   EXTERNALREFS

FROM relations :
   DESIGNATOR domain, schema, tuple, relation, data_base;
   VFUN get_data_bases() -> rdb_set;
   VFUN get_relations(rdb) -> r_set;
   VFUN get_data_domains(rdb) -> d_set;
   VFUN get_range (rdb; d) -> range;
   VFUN get_tuples(rdb; r) -> t_set;
   VFUN get_values(rdb; r; t) -> tw;
   VFUN get_access(rdb; r) -> bv;
   OVFUN create_tuple(rdb;r) -> t;
   OFUN update_tuple_value(rdb;r;t;d;w);
   OFUN update_tuple(rdb;t;tw);
   OFUN delete_tuple(rdb;r;t);

   FUNCTIONS

VFUN get_views(rdb; r) -> view_set;
    $( views for the relation)
   EXCEPTIONS
      no_data_base(rdb);
   INITIALLY
      view_set = {};
```

```
VFUN get_view_relation(rdb; v) -> r;
    $( schema for the view)
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
   INITIALLY
      r = ?;
VFUN get_view_vector(rdb; v) -> vw;
    $( the tuple of %, *, and values defining the view.)
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
   INITIALLY
      vw = ?;

VFUN get_view_access(rdb; v) -> bv;
    $( the access code for the view)
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
   INITIALLY
      bv = ?;

VFUN extractable(w; w1) -> b;
    $( TRUE IF value w can be extracted by the selector value
    w1.  Note that the specification handler requires "%%", not "%".)
   HIDDEN;
   DERIVATION
      IF w1 = "*" OR w1 = "%" OR w1 = w
        THEN TRUE
        ELSE FALSE;

VFUN selects(rdb; t; v) -> b;
    $( TRUE IF tuple t conforms to the primitive view v)
   HIDDEN;
   DEFINITIONS
     relation r IS get_view_relation(rdb,v);
     view_vector vw IS get_view_vector (rdb, v);
     VECTOR_OF character_string tw IS get_values (rdb, r, t);
   DERIVATION
      IF (FORALL  i :  extractable(tw[i], vw[i])
                     = TRUE)
        THEN TRUE
        ELSE FALSE;

VFUN extract(rdb; v) -> t1_set;
    $( extracts those tuples conforming to the view, filtered by %)
   DEFINITIONS
      relation r IS get_view_relation(rdb, v);
      SET_OF tuple  t_set IS get_tuples(rdb, r);
      view_vector vw IS get_view_vector(rdb, v);
    EXCEPTIONS
        no_data_base(rdb);
        no_view_ability(rdb, v, read);
        no_view(rdb, v);
        no_relation(rdb, r);
    DERIVATION
       t1_set
        = { t1 | (FORALL  t |  t INSET t_set:
           (IF selects(rdb, t, v) = TRUE
              THEN (FORALL  i : IF  vw[i] = "*"
               THEN get_values(rdb,r,t)[i] = get_values(rdb,r,t)[i]
                ELSE get_values(rdb,r,t)[i] = vw[i])
                    ELSE TRUE)) };
VFUN and_views(w; w1) -> w2;
     $( ANDs two views together)
    HIDDEN;
    DERIVATION
        w2
        = (IF w1 = "%"
              THEN "%"
              ELSE (IF w1 = "*"
                      THEN w
                      ELSE (IF w = w1 OR w = "*"
                              THEN w
                              ELSE "%")));

VFUN revocable (rdb; v) -> b;
    $( TRUE if v revocable)
    HIDDEN;
    INITIALLY b = ?;

VFUN get_revocable_views (rdb; v) -> view_set;
    $( the set of revocable views for the given view)
    HIDDEN;
    INITIALLY view_set = {};

OVFUN create_view(rdb; r; bv) -> v;
     $( creates a primitive view for a relation)
    DEFINITIONS
        view_vector vw IS get_view_vector(rdb, v);
    EXCEPTIONS
        no_data_base(rdb);
        no_ability(rdb, r, grant);
    EFFECTS
        v = NEW(primitive_view);
         'get_views(rdb, r)
        = get_views(rdb, r) UNION {v};
        'get_view_relation(rdb, v) = r;
        'get_view_access(rdb, v) = bv;
        revocable(rdb, v) = FALSE;
        FORALL  i : vw[i] = "*";
```

```
OVFUN create_restricted_view(rdb; v; vwl; bvl) -> v2;
    $( creates a revocable view, with desired authorization)
   DEFINITIONS
      view_vector vw2 IS get_view_vector(rdb, v2);
      view_vector vw IS get_view_vector(rdb, v);
      VECTOR_OF BOOLEAN bv IS get_view_access(rdb, v);
      VECTOR_OF BOOLEAN bv2 IS 'get_view_access(rdb, v2);
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
      no_view_ability(rdb, v, grant);
      is_revocable(rdb, v);
   EFFECTS
      v2 = NEW(primitive_view);
       'get_revocable_views(rdb, v)
      = get_revocable_views(rdb, v) UNION {v2};
      revocable(rdb, v2) = TRUE;
      FORALL  i : vw2[i]
      FORALL j : bv2[j] = (bv[j] AND bvl[j]);
OFUN destroy_view(rdb; v);
    $( destroys view)
   DEFINITIONS
      view_vector vw IS get_view_vector(rdb, v);
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
      no_view_ability(rdb, v, destroy);
   EFFECTS
      'get_view_relation(rdb, v) = ?;
      FORALL  i : vw[i] = ?;
      'get_view_access(rdb, v) = ?;

OFUN revoke_restricted_views(rdb; v);
    $( revokes all views created by create_restricted_view
    (rdb,v,vwj,bv). Note this is a strong revocation, rather than a
     selective revocation -- which could be achieved with
     slightly more mechanism.)
   EXCEPTIONS
      no_data_base(rdb);
      no_view(rdb, v);
      no_view_ability(rdb, v, revoke);
   EFFECTS
      get_revocable_views(rdb, v) = {};
      FORALL  vj | vj INSET get_revocable_views(rdb, v) :
              'get_view_relation(rdb, vj) = ?
               AND 'get_view_access(rdb, vj) = ?
               AND 'revocable(rdb,vj) = ?
              AND (FORALL  i : get_view_vector(rdb, vj)[i] = ?);
```

```
OFUN v_update(rdb;v;d;w);
   $( updates the values of a relation through a view --
     for all tuples selected by the view.)
   DEFINITIONS
     relation r IS get_view_relation(rdb,v);
   EXCEPTIONS
     no_view_ability(rdb,v,update)
     $(EXCEPTIONS_OF update_tuple(rdb,r) redundant:
        no_ability(rdb,r,update) OK by construction.);
     no_data_base(rdb);
     no_view(rdb,v);
     no_relation(rdb,r);
     no_domain(rdb,d);
     out_of_range(rdb,d,w);
   EFFECTS
     FORALL t | (selects(rdb,t,v) = TRUE):
        EFFECTS_OF update_tuple_value(rdb,r,t,d,w);

OFUN v_insert(rdb;v;tw);
   $( inserts a new tuple into the relation referred to by
     the given view v: view (%, %, ...) with insert access code
     is sufficient.)
   DEFINITIONS
     relation r IS get_view_relation(rdb,v);
   EXCEPTIONS
     no_view_ability(rdb,v,insert);
     no_view_ability(rdb,v,update);
   EFFECTS
     LET t | EFFECTS_OF create_tuple(rdb,r) = t IN
       EFFECTS_OF update_tuple(rdb,t,tw);

OFUN v_delete(rdb;v);
   $( deletes all tuples selected by the view:
     requires delete access)
   DEFINITIONS
     relation r IS get_view_relation(rdb,v);
   EXCEPTIONS
     no_view_ability(rdb,v,delete);
                $( no_ability(rdb,r,delete) redundant)
   EFFECTS
     FORALL t |selects(rdb,t,v) = TRUE:
        EFFECTS_OF delete_tuple(rdb,r,t);

END_MODULE
```

References

1. 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, San Jose (25356), February, 1976.

2. T. Berson and J. Barksdale. KSOS: Development Methodology for a Secure Operating System. NCC '79, NY NY, AFIPS, June, 1979.

3. W. E. Boebert, J. M. Kamrad, E. R. Rang. Analytic Validation of Flight Hardware. Honeywell, 77SRC63, Systems and Research Center, Minneapolis, Minnesota, September, 1977.

4. R. F. Boyce and D. D. Chamberlin. A Structured English Query Language. SIGFIDET, Ann Arbor, Michigan, ACM, May, 1974.

5. R. S. Boyer and J S. Moore. Proving Theorems about LISP Functions. *J. ACM 22*, 1 (1975), 129-144.

6. D. D. Chamberlin. Relational Data-Base Management Systems. *ACM Computing Surveys 8*, 1 (March 1976), 43-66.

7. D. D. Chamberlin, J. N. Gray, and I. L. Traiger. Views, Authorization, and Locking in a Relational Data Base System. National Computing Conference, AFIPS, 1975, pp. 425-430.

8. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM 13*, 6 (June 1970), 377-387.

9. E. F. Codd. Implementation of Relational Data Base Management Systems, (Panel Discussion from the 1975 NCC). *Bulletin of ACM SIGMOD 7*, 3-4 (1975), 3-22.

10. O.J. Dahl. The SIMULA 67 Common Base Language. Norwegian Computing Center, Oslo, 1968.

11. E. W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, Academic Press, New York, 1972. C. A. R. Hoare, ed.

12. R. J. Feiertag, K. N. Levitt, L. Robinson. Proving Multilevel Security of a System Design. Sixth Symp. on Operating System Principles, 16-18 November 1977, ACM, November, 1977, pp. 57-67. In Operating Systems Review, Vol. 11, No. 5

13. R.J.Feiertag. TACEXEC. SRI International, Menlo Park CA, Final Report, April, 1979.

14. R.J. Feiertag and P.G. Neumann. The Foundations of a Provably Secure Operating System (PSOS). National Computer Conference 1978, Vol. 48, AFIPS, 1979, pp. 115-120.

15. R. W. Floyd. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science, American Mathematical Society, Providence, Rhode Island 19* (1967), 19-32. J. T. Schwartz, ed.

16. R. W. Floyd. Toward Automatic Synthesis of Programs. Congress 71, North-Holland Publ., IFIP, 1971.

17. D. I. Good, R. M. Cohen, C. G. Hoch, L. W.Hunter, D. F. Hare. Report on the Language Gypsy, Version 2.0. Institute for Computing Science and Computer, The University of Texas at Austin, May, 1978. Applications Report ICSCA-CMP-10

18. J. V. Guttag. *The specification and application to programming of abstract data types.* Ph.D. Th., Department of Computer Science, University of Toronto, 1975. Computer Science Research Group Tech. Report CSRG-59

19. M. Hamilton and S. Zeldin. Higher Order Software -- a Methodology for Defining Software. *IEEE Trans. Softw. Eng. SE-2*, 1 (March 1976), 9-32.

20. G. Held, M. Stonebraker, and E. Wong. INGRES-- A Relational Data Base System. National Computing Conference, AFIPS, 1975, pp. 409-416.

21. C. A. R. Hoare. Notes on Data Structuring. In *Structured Programming, C. A. R. Hoare, ed.*, Academic Press, New York, 1972.

22. Honeywell Corp. Project Guardian Final Report. Honeywell Information Systems, Inc., Federal Systems Division, McLean VA, September, 1977. ESD-TR-78-115

23. J. D. Ichbiah, J. P. Rissen, J. C. Heliard. The Two-Level Approach to Data Independent Programming in the LIS System Implementation Language. In *Machine Oriented Higher Level Languages, B. vander Poel and H. Maarsen, Eds.*, North-Holland Pub. Co., Amsterdam, 1974.

24. D. E. Knuth. Structured Programming with Go To Statements. *Comp. Surveys 6*, 4 (December 1974), 261-301.

25. K. N. Levitt, L. Robinson, B. A. Silverberg. The HDM Handbook, Vol III: A Detailed Example in the Use of HDM. SRI International, June, 1979.

26. B. H. Liskov, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. Abstraction Mechanisms in CLU. MIT Laboratory for Computer Science Computation Structures Group, July, 1978. Memo 161

27. B. H. Liskov and V. Berzins. An Appraisal of Program Specifications. In *Research Directions in Software Technology, P. Wegner (ed)*, MIT Press, 1979.

28. B. H. Liskov and S. Zilles. Specification Techniques for Data Abstraction. *IEEE Trans. Softw. Eng. SE-1*, 1 (March 1975), 7-19.

29. J. McCarthy. Towards a Mathematical Science of Computation. Congress 1962, Amsterdam, The Netherlands: North-Holland, IFIP, 1961, pp. 21-28.

30. E.J. McCauley and P. Drongowski. KSOS: Design of a Secure Operating System. NCC '79, NY NY, AFIPS, June, 1979.

31. H. D. Mills. How to Write Correct Programs and Know It. Intl. Conf. on Reliable Software, 13-15 April 1975, Los Angeles, CA, in SIGPLAN Notices, , June, 1975, pp. 363-370.

32. J. Mylopoulos, S. Schuster and D. Tsichritzis. A Multilevel Relational System. National Computing Conference, AFIPS, 1975, pp. 403-408.

33. P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A Provably Secure Operating System: the System, Its Applications, and Proofs. SRI International, May, 1980. This is the Second Edition, replacing the 1977 edition

34. P.G. Neumann. Computer Security Evaluation. National Computer Conference 1978, Vol. 47, AFIPS, 1978, pp. 1087-1095.

35. D. L. Parnas. Information Distribution Aspects of Design Methodology. Information Processing 71, IFIP, 1972, pp. 339-344.

36. D. L. Parnas. A Technique for Software Module Specification with Examples. *Comm. ACM 15*, 5 (May 1972), 330-336.

37. D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Comm. ACM 15*, 12 (December 1972), 1053-1058.

38. D. L. Parnas. On a 'Buzzword': Hierarchical Structure. Information Processing 74, IFIP, 1974, pp. 336-339.

39. L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena. A Formal Methodology for the Design of Operating System Software. In *Current Trends in Programming Methodology, Vol. I, R. T. Yeh, ed.*, Prentice-Hall, New York, 1977.

40. L. Robinson and K. N. Levitt. Proof Techniques for Hierarchically Structured Programs. *Comm. ACM 20*, 4 (April 1977), 271-283.

41. L. Robinson. The HDM Handbook, Vol I: The Foundations of HDM. SRIInternational", June, 1979.

42. O. M. Roubine and L. Robinson. The SPECIAL Reference Manual. SRI International, CSL-45, January, 1977.

43. H. A. Schmid and P. A. Bernstein. A Multilevel Architecture for Relational Data Base Systems. University of Toronto, 1975.

44. B. A. Silverberg, L. Robinson, K. N. Levitt. The HDM Handbook, Vol II: The Languages and Tools of HDM. SRI International, June, 1979.

45. W. Teitelman. INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto CA, 1978.

46. J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar Smith, R. E. Shostak, and C. B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proc. IEEE 66*, 10 (October 1978), 1240-1255.

47. W.A.Wulf, R.L. London, and M. Shaw. An Introduction to the Construction and Verification of ALPHARD Programs. *IEEE Trans. Soft. Eng., Vol. 2* (December 1976), 253-265.

NBS-114A (REV. 9-78)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS SP 500-67	2. Gov't. Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE Computer Science and Technology - The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software	5. Publication Date October 1980
	6. Performing Organization Code
7. AUTHOR(S) Karl N. Levitt, Peter G. Neumann, and Lawrence Robinson	8. Performing Organ. Report No.
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Laboratory SRI International Menlo Park, California 94025	10. Project/Task/Work Unit No.
	11. Contract/Grant No. NBS 5-35932
12. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)* Institute for Computer Sciences and Technology National Bureau of Standards Department of Commerce Washington, D.C. 20234	13. Type of Report & Period Covered Final
	14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 80-600157

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

16. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.)*

This document provides an introduction to the SRI Hierarchical Development Methodology (HDM). The methodology employs a staged decomposition of the development process, which separates design, data representation, and implementation. For any given system development, HDM employs a hierarchical decomposition of the design and formal specifications of modules and their interconnections. Extensive tools are used throughout the development to check the appropriateness of the design and its implementation.

The role of HDM in developing secure systems is considered, and various current efforts using HDM to develop such systems are summarized. The use of the methodology is illustrated by a simple but complete example. A somewhat larger example of part of a secure data management system is also discussed.

Verification is not considered in this document, although HDM does facilitate verification. The consistency of formal specifications and their formal requirements can be formally shown, as can the consistency of programs with their specifications.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Design methodology; formal specification; formal verification; hierarchical design; programming methodology; security.

18. AVAILABILITY [✓ X] Unlimited	19. SECURITY CLASS (THIS REPORT) UNCLASSIFIED	21. NO. OF PRINTED PAGES 54
☐ For Official Distribution. Do Not Release to NTIS		
[✓ X] Order From Sup. of Doc., U.S. Government Printing Office, Washington, DC 20402	20. SECURITY CLASS (THIS PAGE) UNCLASSIFIED	22. Price $3.75
☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161		

USCOMM-DC

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, D. C. 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ____

Company ____

Address ____

City ____________________ State ______________ Zip Code __________

(Notification key N-503)

☆ U. S. GOVERNMENT PRINTING OFFICE : 1980 340-997/279