AD-A054 482 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 9/2
HDM - COMMAND AND STAFF OVERVIEW.(U)
FEB 78 L ROBINSON N00123-76-C-0195
UNCLASSIFIED SRI/CSL-49 NL

| OF
AD
A054482

END
DATE
FILMED
6-78
DDC

HDM — COMMAND AND STAFF OVERVIEW

Technical Report CSL-49
Deliverable A007

February 1978

By: Lawrence Robinson

Prepared for:

Naval Ocean Systems Center
San Diego, California 92152

Attn: W. Linwood Sutton
Contract Monitor

Contract N00123-76-C-0195

SRI Project 4828

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

(14) SRI/CSL-49

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*

(6) HDM — COMMAND AND STAFF OVERVIEW.

5. TYPE OF REPORT & PERIOD COVERED

(9) Technical Report.

6. PERFORMING ORG. REPORT NUMBER

CSL-49 SRI Project 4828

7. AUTHOR(s)

(10) Lawrence Robinson

8. CONTRACT OR GRANT NUMBER(s)

(15) N00123-76-C-0195

9. PERFORMING ORGANIZATION NAME AND ADDRESS

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

Deliverable A007

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Ocean Systems Center
San Diego, California 92152

(11) 12. REPORT DATE

February 1978

13. NUMBER OF PAGES

72 (12) 62p.

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*

15. SECURITY CLASS *(of this report)*

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

n/a

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

n/a

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

abstraction, DSARC, embedded systems, formal specification, hierarchical structure, Hierarchical Development Methodology (HDM), methodology, project management, software-development process, software-development tools

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

HDM (the SRI Hierarchical Development Methodology) is an integrated set of concepts, procedures, languages, and on-line tools intended to assist in all stages of the software-development process. This document provides a description of HDM that is suitable for project managers or higher-level executives who are contemplating the use of HDM on software projects that they manage.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

410 646

JOB

CONTENTS

APPENDICES

ILLUSTRATIONS

TABLES

ACKNOWLEDGMENTS

The author wishes to acknowledge Lin Sutton, Neal Hampton, and other staff members of NOSC for conveying to me the problems of the Navy software environment and for their comments on the many drafts of this document. Jack Goldberg, Karl Levitt, and Mark Moriconi of SRI International provided many hours of thought-provoking discussion, as well as critical comments on this report.

I INTRODUCTION

This document provides an overview of the SRI Hierarchical Development Methodology (HDM) conceived at SRI International (formerly Stanford Research Institute). HDM is a set of concepts, procedures, languages, and tools that is intended to aid in the production of correct, reliable, and maintainable software. It has been developed at SRI under several government contracts, including this one. It relies heavily on the technique of formal specification to express both design and implementation decisions. Based on its use in several experimental software development efforts, HDM shows great promise for improving the quality of software.

This report describes several aspects of HDM:

* The motivation behind its development
* Its key features
* Its impact on project management
* Its relation to Department of Defense (DoD) procurement practices.

A glossary of terms related to HDM is contained in Appendix A.

II THE SOFTWARE PROBLEM

The seriousness of the software problem is well known to everyone associated with software development [1], [2].* The most important manifestations of the software problem are that software is of high cost and of low quality.

Software has become the dominant cost in computer system development (replacing hardware). As the hardware becomes cheaper and the software increases in complexity under growing user demand, its percentage of the total cost for system development is rapidly increasing. It is also very difficult to predict software costs: the predictions are usually exceeded no matter what they are. Software costs usually continue long after delivery: a great percentage of all "maintenance" is really the fixing of bugs discovered in the field. It is estimated that maintenance accounts for nearly 90 percent of the average life-cycle cost of software.

Furthermore, software often fails to work as was intended, causing problems for those who must use it. Including the disruptions caused by poorly designed software, the cost of faulty software in DoD is so high that it defies accurate estimation. Clearly this is a serious problem indeed. The fact is, nearly all software -- including that which "works" -- is defective, containing undetected errors, inconsistent interfaces, ignored conventions, and unmet standards.

Several questions can be asked about the software problem. What causes it? Why has it not yet been solved? What can be done about it? There are four properties of software that make it different from other technological domains that have been addressed rather successfully by engineering practices.

* Rapid Growth -- Software has grown more quickly as a problem area than any other field. As quickly as

---

* References cited in brackets are listed at the end of the text.

electronics has grown (over the past 75 years), software has grown even more quickly (over the past 25 years), being used for almost every application imaginable. For example, every company used to have its own ledger accounting system, with particular idiosyncrasies for each company. Now almost every company has a computerized system, together with specialized software (in many cases) for the company's individual needs. Because the development of good engineering practice lags behind the development of technology, it is only natural that software engineering has lagged behind. However, the need for good software has come so fast that, until fairly recently, there was almost nothing in the way of engineering practice to guide software developers. Ideas are beginning to evolve now, but there is still a long way to go. The attempt to "standardize" engineering practice at too early a stage in its development could be harmful.

* Complexity — The complexity of software is greater than that of any other technological pursuit. Complexity is determined by the number of degrees of freedom -- which, in a computer system is determined by the number of states, state transitions, and sequences of state transitions. A large digital machine has an extremely large number of possible states, or values of its internal memory. For example, consider a medium-sized machine, a 16-bit machine with 64k of memory. The possible number of states of the memory alone is (2 exp 1048576), a very large number (the exponent indicates the number of bits of storage). Each time a new bit of storage is added, the number of possible states increases by a factor of two. This does not include bulk storage devices, which have 1000 times as many bits as the main memory. When one considers not only the large numbers of states but the large number of ways of changing that state, the complexity of a machine then becomes literally incredible. Any solution to the software problem must find a way of managing the complexity inherent in software. As a result of this complexity, software systems are poorly understood. A large system is too complex to be understood by one person; even if parts of it are understood, the interconnections are usually so complex that the behavior of the whole system is not precisely known. In other branches of engineering, techniques have been developed for managing the complexity that exists. In software, where the need for such techniques is greater, adequate techniques are not in general use.

* Need for Completeness -- As stated above, software is complex because of the many possible states and sequences of states that occur in a program. To guarantee reliable software, every possible state must be accounted for. If a state is accounted for, it must either be demonstrated that

the state cannot occur or that it is a state that the user desires. A single unaccounted state can result in the catastrophic failure of a system; such failure can often endanger human life, as was the case in several software failures in the on-board computer of the Apollo missions. (The Apollo software was "exhaustively" tested at a cost of millions, yet some failures occurred.) Approaches such as testing account for a very small number of the states of a software system of reasonable size. It would take several lifetimes to test exhaustively all alternatives for such a system. Thus, a better method than testing must be used to assure reliable software.

* Need for Precision -- The precision required in software must be absolute; it is not a human (who can compensate for imprecision) that executes the instructions, but a machine that does only what it is told. Although other fields -- such as electronics -- also require precision in various places, software requires it everywhere in its complex domain: in the programs, in the interfaces between programs, and in the interfaces to users and other machines. However, very few current methods of software construction require precision, except in the implementation. A solution to the software problem will require precision throughout the software-development process.

Many of the measures proposed for the solution to the software problem are informal and/or address only one aspect of the software-development process (usually coding). These measures improve software somewhat; but in general, they do not result in software that is understood in every detail. When software is not understood, there is no absolute confidence in its reliability, and maintenance is usually difficult and time-consuming. Thus, these measures lessen, but do not solve, the software problem.

We believe that the solution lies in a set of formal (i.e., mathematically precise) techniques that produce a specification (or precise document) as the output of each stage of the software-development process. A specification for a system at a given stage of its development provides a precise and complete documentation of the decisions made up to that point, and becomes a requirement for the development that occurs at subsequent stages. Formal relationships can be defined among the specifications at the various stages. Thus,

specifications provide a basis for discussion, syntactic checking, and formal proof of the properties of a system, i.e., understanding the system. When the system is understood, its implementation, verification, and maintenance are straightforward.

A solution to the software problem should be based on

* A set of concepts that characterize desirable systems, i.e., the way they should be structured, described, and built.
* A set of procedures that divide the software-development process into stages.
* A set of languages to express the specifications at the various stages of software development.
* A set of on-line tools that check the syntactic consistency (and potentially the semantic consistency) of the specifications at each stage.

An integrated set of concepts, procedures, languages, and on-line tools is called a methodology. Various methodologies exist today, e.g., Chief Programmer Team [3], HDM ([4], [5], [6]), HOS [7], the Jackson Methodology [8], and Structured Design [9]. The remainder of this report describes HDM and its use in the development of software.

III A BRIEF DESCRIPTION OF HDM

III-A. Overview

As stated above, HDM is an integrated collection of concepts, procedures, languages, and on-line tools that is designed to serve as a medium for software development. This section gives a general overview of these components.

HDM provides a substantial amount of mechanism for the software-development process, primarily because it has been created for use with large production systems. It may not be cost-effective to use all of this mechanism for a small program (e.g., one that could be designed, implemented, and run in a single day). For large systems, however, the mechanism provides a complete record of the decisions made in the development of the system. Because a large number of decisions enter into in the development of a software system (even for a system of moderate size), there must be sufficient mechanism to record and to structure them. The following description of HDM should be read with this point in mind.

III-B. The Concepts of HDM

HDM unifies and formalizes the following widely accepted ideas initially proposed by Dijkstra ([10], [11]), Parnas ([12], [13], [14], [15]), and Floyd [16]:

* Hierarchical structure (Parnas, Dijkstra) is a particular kind of structure in which, if one part of the system depends on another for its implementation, then the reverse is not also true. Most software systems built with present techniques are either not hierarchically structured or, if there is a hierarchical structure, it is not an explicit part of the system design. The kind of hierarchical structure particular to HDM divides the system into levels of functionality, where a level is dependent only on the levels below it. Hierarchically structured systems have several advantages over conventionally structured software systems. The former are easier to understand and easier to

modify, and with hierarchically structured systems it is easier to attach responsibility to a single component if the system fails. Hierarchies are the standard type of structure in large organizations of people, such as corporations (see Figure 1) and have proved their utility there. For example, a line manager has complete responsibility for the set of people who work under him. To get better performance out of his suborganization, he can change it as he sees fit, but no other part of the organization need be changed. Similarly, a component of a hierarchically structured software system may have its implementation changed, without the need to change any other part of the system.

* Abstraction (Dijkstra) is the process of isolating a few of the properties of an object to explain it or to understand it more easily. For many purposes the isolated properties (called an abstraction of an object) can be studied as if they were the object itself. In software systems, the object under study is the code that constitutes part of the system; the isolated properties are the effect of executing the code, expressed independently of how the desired effect is achieved. The concept of abstraction is widely used outside the technical arena. For example, the instructions for operating an automobile contain an abstract model of how an automobile runs; this model is an abstraction of what actually goes on in the engine and ignition system of an automobile. Clearly, it is unnecessary to understand the principles of the internal combustion engine to drive an automobile. Similarly, it is unnecessary to know about the inner workings of a program or an entire software system to use it. What is needed is a method for describing the properties of a program to the degree of abstraction needed for understanding any system that contains the program. In general, the use of abstraction can simplify the process of understanding a complex object, such as a program or a software system.

* Modularity (Parnas) is the property (of a system) of being composed of easily replaceable parts, or modules, having well-defined external interfaces. Modularity is in common use in hardware designs; for example, an operational amplifier is considered on the basis of its gain and impedance, rather than on its internal design. Another example of a modular system is a television set, any of whose individual components (e.g., circuit boards, picture tube, tuner) may easily be replaced if it fails. Hardware systems with the property of modularity are easier to maintain; the same is true for software systems, if reasonable criteria for modularity can be determined [14]. Simply dividing the system into subroutines (i.e., units of control) is not enough; the complexity of the shared data

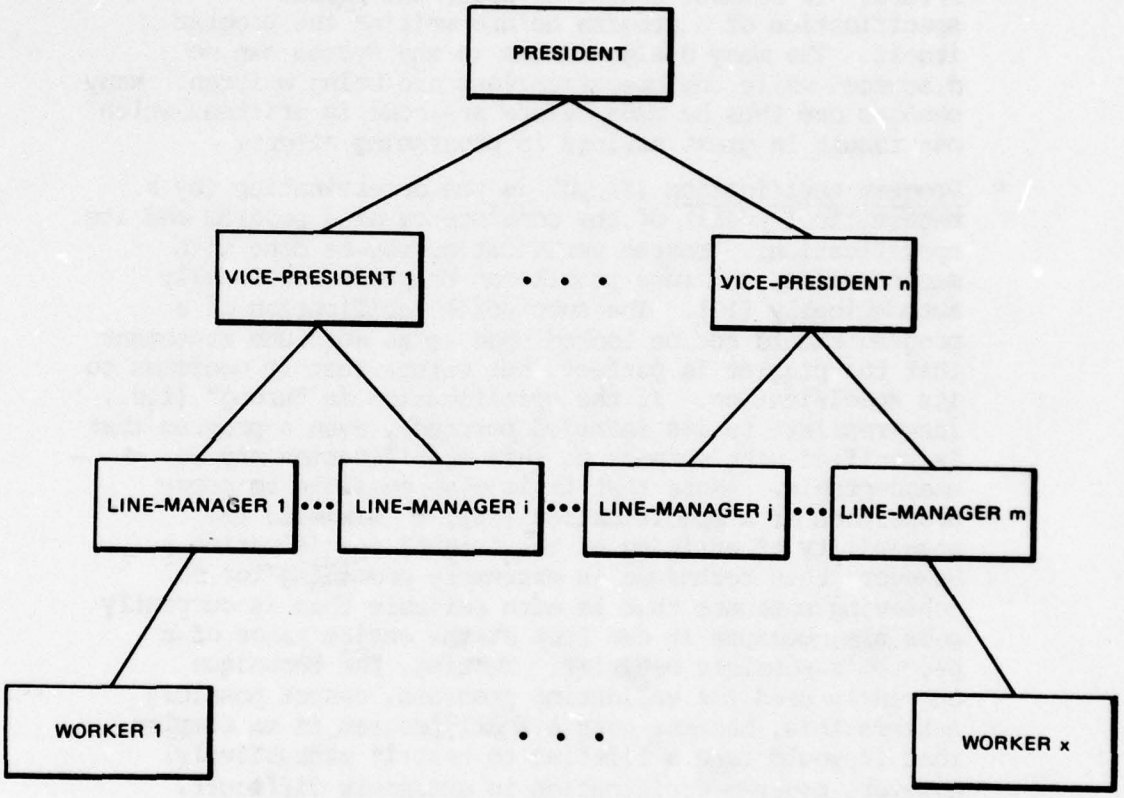

FIGURE 1 AN ORGANIZATIONAL HIERARCHY

often makes the interfaces difficult or impossible to define. Thus, a module in this context requires the isolation of data as well as control.

* Formal specification (Parnas) is the act of stating precisely, for all intended input data, the intended result of using a program. The statement itself is called a specification. To ensure precision, a specification language (as opposed to English) is used for this purpose. Unlike English, a good specification language is unambiguous (viz., all statements written in it have exactly one meaning), and the specifications written in it can be machine checked for grammatical and certain semantic errors. It is most useful to write the formal specification of a program before writing the program itself. The many design issues in the system can be discussed while the specifications are being written. Many changes can thus be made before any code is written, which can result in great savings in programing effort.

* Program verification (Floyd) is the determination (by a mathematical proof) of the consistency of a program and its specification. Program verification may be done with machine aids, and some proofs can be performed totally automatically [17]. The successful verification of a program should not be looked upon as an absolute statement that the program is perfect, but rather that it conforms to its specification. If the specification is "wrong" (i.e., inappropriate to its intended purpose), even a program that is verified with respect to this specification may be unacceptable. (Note that it is also possible to prove properties of a specification [18], to maximize the possibility of arriving at the "right" specification.) However, this technique is extremely promising for achieving software that is more reliable than is currently possible, because it can look at the entire range of a program's possible behavior. Testing, the technique currently used for validating programs, cannot possibly achieve this, because even a small program is so complex that it would take a lifetime to test it exhaustively. However, program verification is extremely difficult, having been successfully performed only on small systems (those containing less than 2000 lines of source code) of limited capability. We believe that verification of large systems can be achieved only by structuring them into small programs for which independent proof is feasible.

HDM uses all of the above techniques. For example, in HDM the structuring of software systems is based on abstraction, the units of specification are modules, and sets of modules (each set constituting a level) are arranged in a hierarchy. The specification languages of HDM

make it possible to use formal specifications as an aid in both development (including maintenance) and verification. The structuring of the design using HDM aids in decomposing the verification effort as well as the design effort.

Other methods for assisting software development may employ one or more of the above techniques, with some benefit. However, we believe that the ultimate solution to the software problem requires a comprehensive approach using at least all of these techniques and applying them to all stages of software development.

III-C. The Procedures of HDM

This subsection treats two issues of procedure concerning HDM: its breakdown into stages and its use in the development of system families.

III-C-1. The Stages of HDM

HDM divides the software-development process into stages, in which a particular activity is performed at each stage. However, one may not wish to adhere to this strict ordering in the development of a software system. Indeed, backtracking in the development process may be necessary. Alternatively, it may be desirable first to develop completely a part of the system, and to proceed with the rest of the system later. Thus, the following stages should be viewed as a categorization of the activities that take place in developing a software system using HDM, and a possible scenario for its use. The stages of HDM are shown in graphic form in Figure 2, and in tabular form in Table 1. They are listed as follows.

(1) Conceptualization Stage -- The designer analyzes the environment of the proposed software system and states precisely the problem to be solved. This environment contains constraints imposed by the user (manifested at the top level of the system) and constraints imposed by the hardware or programming language (manifested at the bottom level). Efficiency constraints, such as throughput, can also be expressed at this stage. At present, the output of the conceptualization stage is stated in terms of precise English, because HDM currently provides no formal language to support conceptualization.

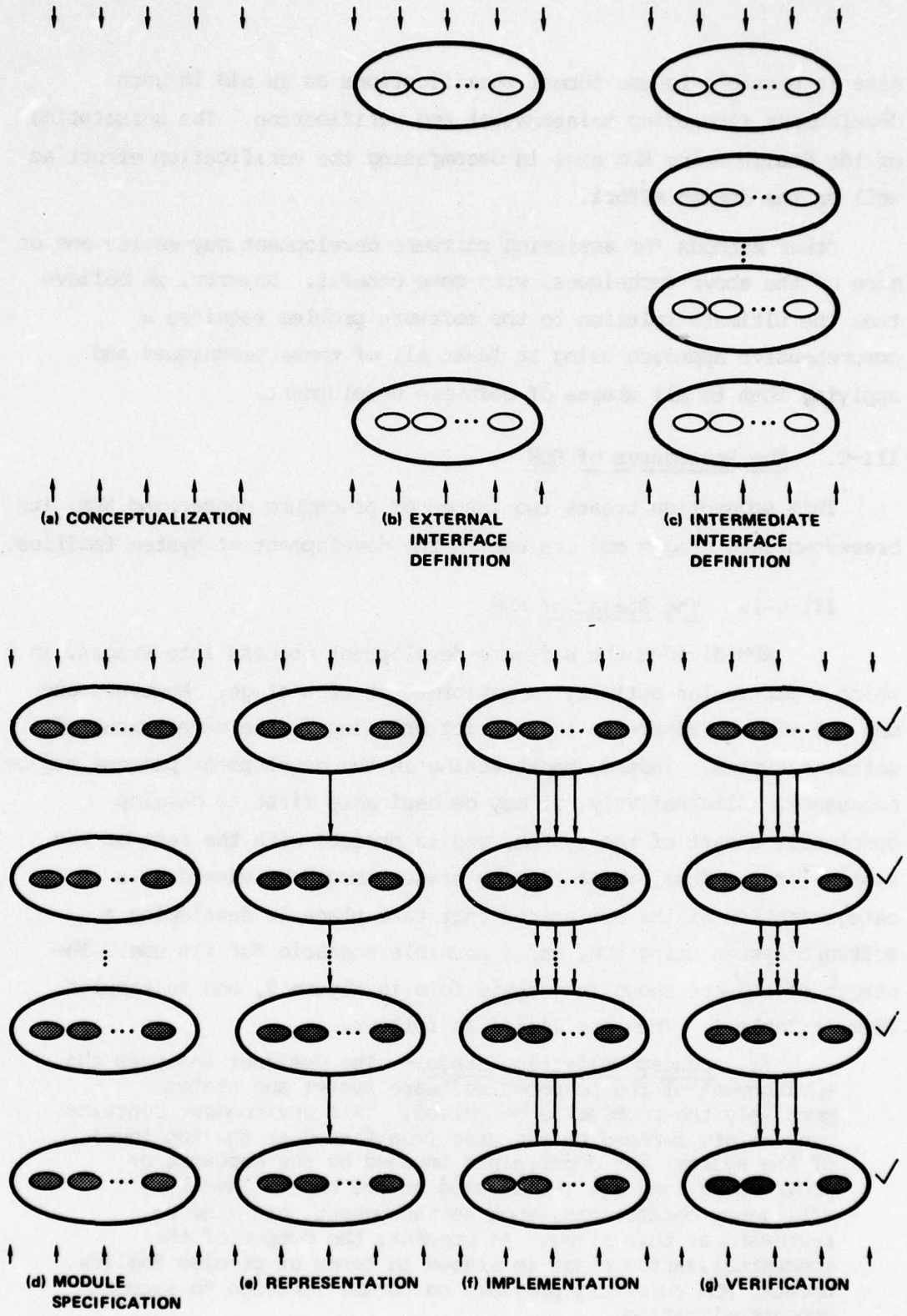

FIGURE 2 THE STAGES OF HDM

Table 1

THE STAGES OF HDM

Stage	Activity	Output	Language	Tool(s)
Conceptualization	Analysis of problem	Informal statement of the problem	--	--
External Interface Definition	Defining external behavior of system and decomposing external interfaces into modules and functions	List of modules at each interface and the functions of each module	HSL	Interface checker
Intermediate Interface Definition	Defining hierarchical structure and behavior of intermediate levels; decomposing intermediate levels into modules and functions	List of intermediate levels, the modules of each level, and the functions of each module	HSL	Interface checker, Hierarchy checker
Formal Specification	Formally specifying all modules of the system	Formal specifications for each module	SPECIAL	Module checker
Formal Representation	Describing data structures at one level in terms of data structures of lower level	Formal representations of the data structures of each level in terms of those of the lower level	SPECIAL	Representation checker
Abstract Implementation	Writing abstract programs to implement each function of the system	Abstract programs for each function of each module	ILPL	Implementation checker
Coding	Translation of abstract programs into concrete programs	Concrete programs for each function of each module	Conventional programming language	Preprocessors, compilers, assemblers, etc.
Verification	Mathematically proving that the system implementation meets its specifications	Transcript of proof	--	Verification system

(2) External Interface Definition Stage -- The external interfaces of the system (i.e., the top and bottom levels) are conceived. Each of these levels provides some functional capability in terms of a set of operations, and consists of one or more modules, or groups of related operations (note the discrepancy with the conventional usage of "module," meaning a single program or subroutine). A module is chosen on the basis of localizing certain decisions (concerning for example, representation or implementation) within the module, so that different modules can be implemented (or have their implementation changed) without regard to the implementation of any other module. While the conceptualization stage poses the problem to be solved, this stage describes the functional capability of a system that solves the problem. The output of this stage is a list of the modules of the top and bottom levels of the system, and a list of the operations of each module.

(3) Intermediate Interface Definition Stage -- The intermediate levels of the hierarchy are conceived. This process of defining intermediate levels can proceed in any manner: top-down, bottom-up, or randomly. For example, in conceiving levels top-down, one asks, "What level best realizes the level I already have?" In conceiving levels bottom-up, one asks, "What level best utilizes the level I already have?" This process continues until the entire hierarchy is conceived. Each intermediate level is also decomposed into modules. The number of intermediate levels depends on the complexity of the problem and on the taste of the designer. The output of this stage is a list of the modules of each of the intermediate levels, and a list of the operations of each module.

(4) Formal Specification Stage -- The formal specifications for each module are written. In addition to its operations, which are invoked by a user or an external program, each module contains a set of internal data structures that can be accessed or modified only via its operations; a specification of each operation is written in terms of the values of the internal data structures. The operations of a given module may also reference, in a restricted way, the operations and internal data structures of other modules at the same level; all such references must be explicitly stated. One goal in decomposing a system is to minimize intermodule references, making each module as self-contained as possible. The output of this stage is a set of formal specifications for each module of each level in the hierarchy.

(5) Formal Representation Stage -- Eventually each nonprimitive module (i.e., a module that does not appear at the bottom level) will be realized in terms of the next lower level. For each such module, its realization consists of the

representation of its internal data structures in terms of the internal data structures of the modules at the next lower level, and (at the next stage) the implementation of each of its operations as a program that invokes the operations of modules at the next lower level. The formal representation stage reflects the need to define the internal data structures of each nonprimitive module, which have no meaning by themselves, in terms of more primitive internal data structures. The formal representation of the modules at a level is written as a set of expressions (called mapping function expressions), one for each internal data structure of each module at the level; each expression defines its corresponding internal data structure in terms of the internal data structures of the modules at the next lower level. The output of this stage is a mapping (i.e., a set of mapping function expressions) for each level in terms of the next lower level.

(6) Abstract Implementation Stage -- Each operation of each module is implemented as an abstract program invoking the operations of the modules of the next lower level. It is in this stage that the implementation decisions are made. The programs are termed abstract because they describe only the sequence of calls to the operations of the next lower level, without regard to other details associated with the final code (see the next stage). The output of this stage is a set of abstract programs, one for each operation of each nonprimitive module.

(7) Coding Stage -- To produce a running system, a set of programs that run on the target machine must be generated. Here the target machine may be either a piece of hardware (in which case the programs are written in assembly language) or a high-level programming language (in which case the programs are written in that language). These programs can be generated by translating the abstract programs (either by hand or machine) into concrete programs that can actually be run. The concrete programs thus contain much detail that has been omitted from the abstract programs. The output of this stage is a set of concrete programs -- one for each abstract program -- that can be compiled, interpreted, or assembled with existing on-line tools.

(8) Verification. As stated above, verification is an extremely precise and complex consistency checking operation. The object of verification in the specific case of HDM is to see that the module specifications are correctly realized by the representations and abstract programs that have been supplied. Verification requires a formal mathematical proof, which can be performed either by hand or with the aid of an on-line program verification system. A verification technique especially suited to HDM has been developed [19]. Because verification is so difficult, it can be considered an optional

stage of HDM. In that case, other methods of validation, such as debugging and testing, would be used to ensure the correct operation of the developed system. The output of this stage is a transcript of the proof of the correctness of the system.

The stages of HDM describe a suggested time ordering, but by no means an inviolable directive, for developing a system. For example, there will be much iteration in the system development process. Iteration occurs when, at a given stage, it is realized that a mistake was made at an earlier stage of system development. HDM encourages the designer to go back and correct the mistake before proceeding. It is also possible to realize one level of the system in terms of another before some of the other levels have been designed. However, both upper and lower levels should be specified before any realization takes place.

In comparison to HDM, the conventional method of software development describes a system as a network of functional transducers (Figure 3), each performing a specific task. The network can have either of two interpretations:

* The Data Flow Model -- Here data enters a functional transducer at one of its possible entry points. The data is transformed by this transducer into the data that is seen at the exit points.
* The Control Flow Model -- Here control enters a functional transducer at one of its entry points. The transducer performs some task, usually including changes to some of the system's data structures. Control then passes through the transducer's exit points to some other parts of the system.

Both of these models are useful for understanding, in an intuitive way, what functions the system performs. However, they are deficient because they fail to describe the system's static structure and the properties of the data in the system. Systems described in this manner have only one level of abstraction, so either the data is considered in full detail or it is ignored. In addition, it is impossible to understand a component of the system in isolation, because many different parts of the system operate on the same data. Since the network itself may be extremely complex (possibly containing loops), a true understanding of the system with these methods is still nearly impossible. Conventional

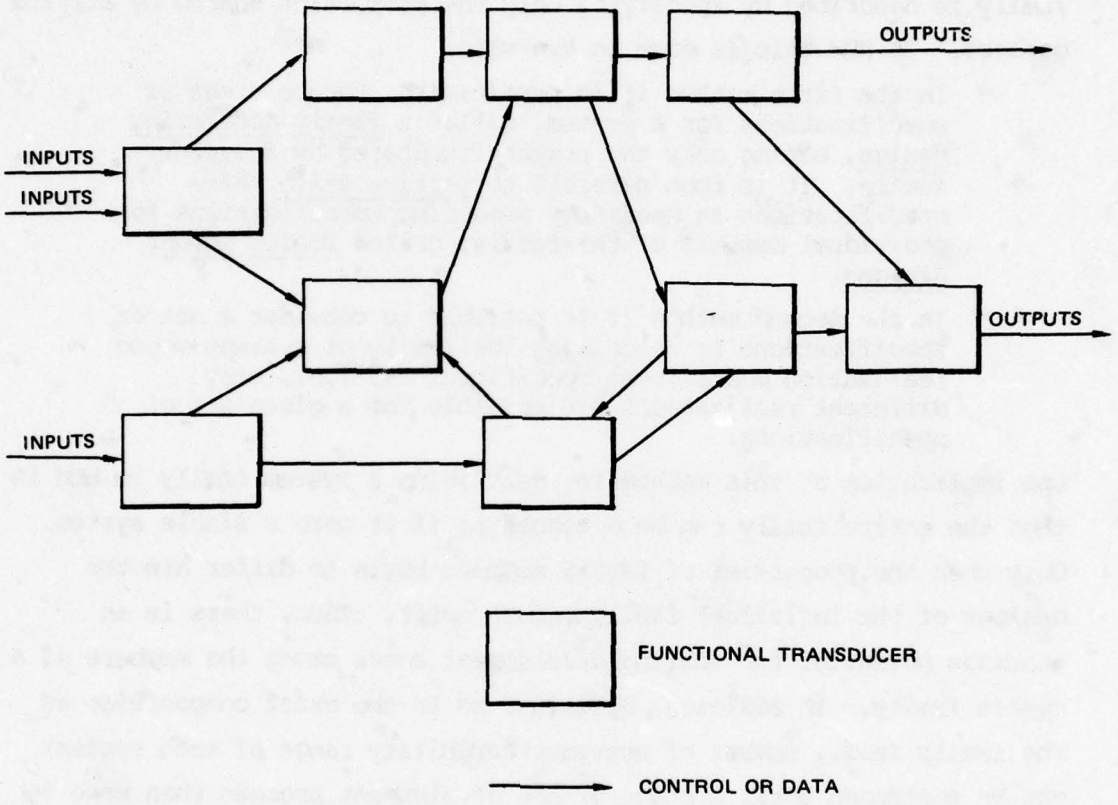

FIGURE 3 EXAMPLE OF THE FUNCTIONAL TRANSDUCER MODEL OF SYSTEM STRUCTURE

methods attempt to manage complexity by this form of structure, but the methods do not solve the entire problem because the structure does not truly separate the system's components.

III-C-2. Describing System Families in HDM

HDM enables the development of an entire family of systems, as well as a single system. A system family is a set of systems that share some common properties. In applying the system family concept to HDM, a family is described by specifying only the properties shared by all its members. In HDM this is done in two ways:

* In the first method it is possible to produce a set of specifications for a system, called a family generating design, having only the properties shared by a system family. It is then possible to particularize these specifications as needed by producing specifications for individual members of the family, called family member designs.
* In the second method it is possible to consider a set of specifications as describing the family of systems whose realization meets these specifications, i.e., many different realizations are possible for a given set of specifications.

One implication of this method for describing a system family in HDM is that the entire family can be designed as if it were a single system. Only when the properties of family members begin to differ are the designs of the individual family member split. Thus, there is an enormous potential for sharing development costs among the members of a system family. In addition, decisions as to the exact composition of the family (e.g., number of systems, capability range of each system) can be postponed until a stage of the development process when more is known about the trade-offs for these decisions. As a result of postponing decisions until a more appropriate time, the family ultimately developed will better fit the range of intended requirements, without gaps (requirements filled by no system member) or overlaps (where more than one system would be adequate for a particular set of requirements), resulting in further savings.

III-D. The Languages of HDM

HDM provides three specification languages: the first for writing formal specifications of modules and representations, the second for describing the arrangement of modules into levels and levels into a hierarchy, and the third for describing abstract implementations of modules. An example of the specifications of a complete system, using all three languages, is shown in Appendix B.

III-D-1. SPECIAL

The language for module specifications and representations is called SPECIAL (SPECIfication and Assertion Language) [20]. It allows the operations of a module to be specified independently of how they are implemented.

A SPECIAL description of the operations of a module is analogous to a description of the instruction set of a computer appearing in a programmer's manual. Both are given in terms of the effect of the operations on particular state variables. SPECIAL differs from the informal notation used in programmer's manuals in that it is mathematical, based on logic and set theory. It describes the effects of an operation as if the operation were indivisible, rather than as a sequence of smaller operations (the conventional approach). To describe the properties of real systems, SPECIAL also has some constructs from programming (e.g., types and exception conditions).

SPECIAL is also used to describe the representation of the data structures of a module specification in terms of the data structures of modules at the next lower level.

III-D-2. HSL

The language for describing levels and hierarchies of levels and representations is called HSL (Hierarchy Specification Language). HSL specifications are used to check the consistency of decisions shared by several modules or representations (whose specifications are expressed in SPECIAL). The syntax of HSL is extremely simple.

III-D-3. ILPL

The language for describing abstract programs (i.e., for specifying implementation decisions) is called ILPL (Intermediate Level Programming Language). It describes an implementation for each operation of each module as an abstract program.

It differs from programming languages in that it is extremely simple and has no built-in data structures. The simplicity is an aid to formal verification. The lack of built-in data structures allows the user explicitly to specify his own data structures as modules of the system appearing at the lowest level. This allows a single simple language to be used for many different applications, rather than using several languages or a single large "do-it-all" language. For example, a different set of modules would be used for systems programming than for either general high-level applications or for list processing. Thus, if ILPL is the abstract programming language, any existing programming language can be used for the final coding, with the data structures of that programming language incorporated into ILPL as a set of modules. Such versatility can contribute significantly to the solution of the portability problem.

Abstract programs written in ILPL can be checked for consistency with specifications written in SPECIAL and HSL, because all three languages are based on the same concepts.

III-E. The Tools of HDM

A complete set of tools for HDM would cover the stages of interface definition, formal specification, representation, abstract implementation, and verification. Prototype tools for the first three stages have already been developed. Those for abstract implementation will be developed (under Navy sponsorship) in the near future. Those for verification will be developed (under other sponsorship) over the next three years.

The currently implemented tools are as follows. The individual module specifications and representations (written in SPECIAL) are

checked by the module checker and the representation checker, respectively. The structure of the system, described in HSL and consisting of a description of the individual levels and the entire hierarchy, is checked by the interface checker and the hierarchy checker. The specification of the structure allows for checking of consistency among the specifications of the modules and representations that were checked individually by the first two tools described above. Each checking program outputs information needed by other checking programs, as well as diagnostic messages in case of an error. A diagram of the current tools is contained in Figure 4.

The unimplemented tools include an implementation checker for specifications in ILPL, and changes to the interface and hierarchy checkers so that they check for syntactic consistency of abstract implementations with the previously written specifications and representations. As stated above, the verification system will be available in not less than three years.

Other proposed tools, which are not connected with specific stages of HDM, are the module simulation system and the development data base. The module simulation system allows the user or designer of a system to test the functioning of a module (or of a complete level) after the module or level is specified, but before it is implemented. The module simulation system simulates the behavior of the module or level based on the specification alone. Although this simulation is very inefficient relative to the final implementation, it allows an accurate evaluation of a system's capability and utility to take place before the often heavy cost of implementation is incurred. The development data base would enable HDM to be better applied to the development of production software systems. The development data base keeps track of all changes that are made as the software is developed, and rechecks the system's consistency as need be, alerting the designer when a change in one part of the system affects another. Without such automated accounting measures, it would be extremely difficult to control the development of a large system.

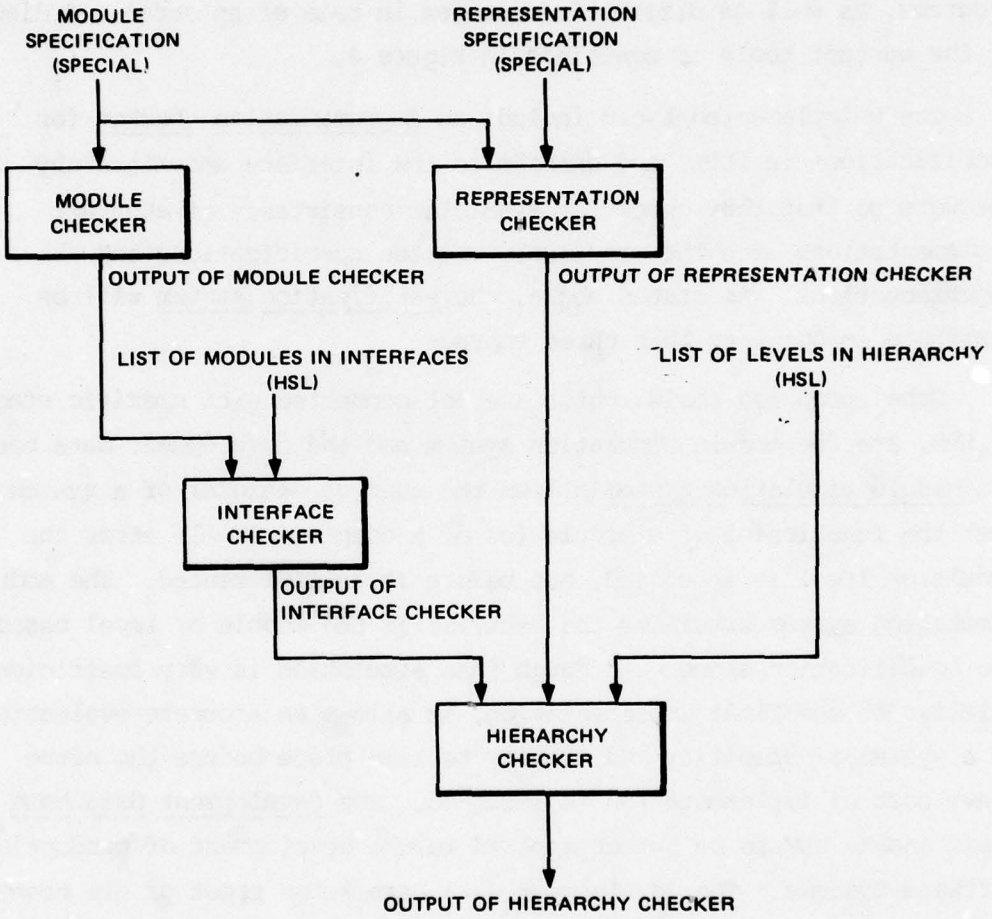

FIGURE 4 THE TOOLS OF HDM

IV MANAGEMENT IMPACT OF HDM

IV-A. Introduction

We believe that HDM can considerably assist the project manager in his task, especially for large projects. Project management encompasses many issues, and the following are discussed here: time requirements, personnel, estimation, project evaluation, documentation, maintenance, and scope of applicability.

IV-B. Time Requirements

HDM differs vastly from conventional methods in the relative amount of time spent in its various stages. In HDM, much more time is required in the early stages (i.e., conceptualization, interface definition, specification, and representation) than in the later stages (i.e., abstract implementation, coding, testing, and maintenance).* With conventional methods, relatively more time is spent on the later stages.

The decisions made during the earlier stages are more important, because the decisions of the later stages are dependent on them. If the early decisions are wrong, the rest of the project is in trouble (as is often the case using conventional methods). In fact, the total time spent on a system using HDM should be much less, because implementation, testing, and maintenance are the major time expenditures in the development of software. Thus, HDM achieves a redistribution of the time spent to achieve a better software product in less total time.

---

* The optional HDM stage of verification is an extremely complex process; its time requirement will not be considered here, for lack of sufficient information.

IV-C. Personnel

Issues of personnel consist of skills and training, the project team, evaluation of personnel, and potential difficulties.

IV-C-1. Skills and Training

Learning HDM is not a trivial task, because it requires a substantially different way of visualizing software than conventional methods. Despite this difference, many individuals have by now learned to use HDM to think about programs and are applying it in practical development efforts. We believe that programmers, for both the Navy and for vendors, can be readily trained to use HDM; and, in the light of the seriousness of the software problem, we believe that the modest effort required will be well rewarded.

Of course, some specialized skills are needed to use HDM effectively. Besides the obvious skills such as knowledge of software (programming) and of the applications area, others -- which are typically developed from mathematical training -- include:

* The ability to think clearly — to be able to state a problem clearly and to divide it into component parts for solution.
* A good sense of aesthetics in design -- to arrive at elegant designs to solve the problem at hand.
* The ability to manipulate formal notations.

Naturally, most of the above skills are possessed to some extent by every good designer or programmer, but in a very specialized sense. Training in HDM requires the broadening of these skills toward the goal of writing specifications for hierarchically structured systems.

To use HDM, a designer or programmer must be taught the following:

* The model of computation on which HDM is based -- the hierarchy of abstract machines.
* "Abstraction," the technique by which one arrives at a hierarchy.
* The restrictions imposed by HDM, such as modularity.
* The languages of HDM — SPECIAL, HSL, and ILPL.

If program verification is attempted, it is desirable to have a general knowledge of mathematical proof, and program verification in particular. Any personnel performing formal verification would be trained in the particular verification techniques that are applicable to HDM.

Training an experienced person to start using HDM takes about a month. Preferably, such training should take place in the form of a course, with many graded problems. The learning rate is initially slow, because HDM may require that a programmer break some previous habits and to adopt new ones; however, after the initial hurdle is cleared, progress tends to be rapid.

IV-C-2. The Project Team

A project team that uses HDM tends to be small, usually not more than four to five designers or implementors working at a time. It is recommended that every team that designs or implements a system contain (or have as a consultant) at least one member who has actually designed or implemented a system using HDM. In the early stages of software development (encompassing the HDM stages of conceptualization, decomposition, and specification), the interaction among team members is usually very close. The interaction centers around the specifications, as soon as a draft of them has been written. During the later stages of development (encompassing the HDM stages of representation and implementation), the team members can work with more or less independence, because the formal specifications divide the implementation effort into independent programming units whose interfaces have been precisely defined. For best results, the project team for the implementation phase should contain or be supervised by a member of the design team. This is because in HDM, although specifications are written to be independent of a particular implementation, the designers have thought to some extent about implementation issues (to create a reasonable hierarchy). These thoughts should be communicated in some way to the implementation team, either by personal contact, by some written explanation of implementation issues, or by sample implementations accompanying the

specification. In addition, a designer may have an overall "concept" of a system, which he can convey to the implementation team.

IV-C-3. Personnel Evaluation

Using current methods, it is often difficult for a project manager to evaluate the quality of the designers and programmers working for him. One problem is that it is difficult and time-consuming to evaluate the designs and implementations produced by these people.

It is much easier to evaluate a design using SPECIAL and HSL, because these languages require more decisions to be explicitly written down. In addition, the restrictions provided by HDM -- and detected by the on-line tools -- are a means for screening a design or an implementation, because many poor designs or implementations -- not adhering to these restrictions -- will not pass through the tools. This contrasts with other methods, in which it is permissible to write any design, or any implementation that compiles; the rest is left up to the debugging skill of the designer or implementor. The structure of systems produced with HDM enables responsibility for a faulty piece of code to be placed on a specific person. In conventionally structured systems with imprecisely defined interfaces, this is more difficult to do.

The increased difficulty of using HDM may be looked upon as a blessing, because it provides a screening mechanism for software development personnel. The screening of designers and programmers is a significant problem; it was discovered [21] that skills among programmers with equivalent training and experience differed by as much as 26 to 1. Presumably a project manager would use the screening information provided by HDM in choosing a future team. Although there will now be fewer people from which to choose, these fewer people will produce better systems sooner that require less maintenance, thereby cutting the overall need for personnel. Because the design and implementation teams are smaller than in the conventional approach, fewer -- and more skilled -- personnel are needed. Note the similarity between HDM and the Chief Programmer Team [3] in this respect.

HDM imposes constraints upon system designers and implementors, but not in such a way as to minimize the need for creativity among them. For example, the ability to decompose a system properly into a hierarchy of abstract machines requires exceptional ingenuity and skill. However, HDM does not allow the kind of freedom that lets designers do everything they want, but no highly developed engineering practice does that. HDM specifically prohibits those techniques that produce systems of needless structural complexity, even if some designers have previously used the latter techniques to create functioning systems. Extensive use of HDM in the research environment of SRI has shown that it does not interfere with creativity, but rather presents more of a challenge to the designer to meet the constraints imposed by it: to build a working system that is also well-structured and elegant.

IV-C-4. Potential Difficulties

Managers should expect to encounter some initial resistance on the part of programmers and designers to the introduction of HDM. A similar experience has occurred with the introduction of structured coding techniques into programming shops. Although such techniques are generally recognized as leading to a better software product, there is still resistance to them. A manager must weigh the potential benefits of introducing something new against the resistance encountered by such an introduction. Particular difficulty may be encountered with SPECIAL, because few people have ever written precise specifications. The difference (and thus the expected resistance) is even greater than that encountered when changing from assembly language to a higher-level language for system implementation.

As stated above, the use of HDM requires more mathematical sophistication than is possessed by the average programmer. There are also levels of ability required in using the various stages of HDM. For example, it takes more ability to decompose a system into a hierarchy of abstract machines than it does to write the specifications for the modules of an abstract machine once the general function of the machine

is known. Similarly, it is more difficult for a designer to write specifications than it is for a programmer to read those specifications and to implement them. These differences correspond to the relationship between a systems analyst, a programmer, and a coder in the traditional approach to software development.

IV-D. Estimation

Estimation has always been difficult for project managers, because they are called upon to estimate effort and cost even before the problem has been formulated. Thus, many managers estimate the time needed to solve what looks like the problem, and then multiply by a suitable "fudge factor" (sometimes as great as 10).

Using HDM, it is still difficult to estimate the need for resources at the beginning of the project. However, after the specifications have been completed, it is straightforward to estimate the resources required for the remaining stages, because the major problems are now thoroughly understood. In addition, the specifications provide a good indication of the amount of code that will have to be written to implement the system. Using conventional methods, the problems are frequently misunderstood even after the design has been completed, so estimation is still difficult. Thus, by helping a manager to understand the problems at hand, HDM can improve estimation in the later stages of software development.

IV-E. Project Evaluation

A project manager must be able to evaluate the status of his project at all times, so that changes and adjustments can be made. HDM is a considerable aid in this process. The stages of HDM have built-in milestones, by which the manager can determine the status of the project. The outputs of each stage are structured so that a problem area can be localized and the situation remedied. System design specifications are particularly useful for this purpose. In the critical design review, for example, the reviewers could read the system specifications before the review and could be prepared to make extensive

comments during the review. Such a process would enable both managers and outside observers to judge the progress of a software development effort with greater accuracy than was previously possible.

IV-F. Documentation

Two kinds of documentation are useful after the system is in operation: that required for the users of a system, and that required for system maintenance. The user documentation should probably be written in natural language (i.e., English). The system maintenance documentation can be written in any suitable medium, but should probably include some natural language for explanation. HDM can be of utility in both aspects of documentation.

For user documentation, the HDM specifications of the user interface of the system can be used as the basis from which the natural language documentation is derived. The HDM specification is clear and unambiguous, allowing an excellent user document to be generated with ease.

The specifications, representations, and implementations produced by HDM form the basis of the maintenance documentation. The specifications and representations identify the parts of the total implementation, and tell what the latter are supposed to do. Even if no other documentation generated, it would still be possible to maintain the system with ease (thus, more easily than with conventional documentation). HDM provides a commenting facility in SPECIAL and ILPL so that additional insights can be written in-line with the specifications. Note that the HDM specifications convey mainly what decisions have been made; it is also desirable to have additional documentation in natural language to give the reasons for having made the decisions.

IV-G. Maintenance

The cost of maintaining a large production software system almost always outstrips the cost of developing the system in the first place. Maintenance usually has three forms:

* Removing previously undiscovered bugs
* Optimizing system performance
* Adapting to changing requirements.

In general, better systems require less maintenance, because the system contains fewer bugs, performs well, and is general enough to encompass many minor changes in requirements. We believe that HDM enables the construction of better systems, so the maintenance problem is somewhat lessened to begin with. When maintenance is needed for any of the three reasons listed above, HDM can also be of assistance.

Concerning the removal of bugs, a bug discovered in a system designed using HDM is often easily located and fixed, because precise specifications are available, the system is well-structured, and the system contains excellent error-handling facilities.

System performance can often be optimized easily in HDM, because most optimizations in HDM are changes in the implementation of a single module. Such a change can be made very easily, without regard to any other code in the system. Systems designed using conventional methods have many hidden dependencies, so even a small change such as an optimization can cause undesired behavior elsewhere in the system, which must be located and corrected.

Even the problem of changing requirements is handled more easily in HDM, because the structure of the system limits the effects of changes in the design resulting from changing requirements.

IV-H. Scope of Applicability

This subsection discusses the possible use of other methods in conjunction with HDM and the class of systems for which HDM is applicable.

Of particular importance to a project manager is the completeness of the set of methods that he has chosen -- in other words, whether one all-inclusive method is indicated, or whether several methods would be useful, to achieve the optimum result. Obviously it would be preferable to use one all-inclusive method, if a good one existed. HDM is complete in one sense and incomplete in another. It is a complete medium for stating design and implementation decisions once they are made, but does not -- in and of itself -- provide the criteria for making these decisions. However, no medium or methodology can be expected to do this completely. In other words, it serves no purpose to use another design methodology in addition to HDM. It might be useful to study the system -- for example, by simulation -- to determine how a given part of it might be designed.

HDM provides most of the criteria for judging the elegance of a design, but provides almost no criteria for judging its performance. At this time, performance has not been addressed in HDM, because

* Research in performance properties has lagged behind research in data properties. Thus, we felt that there was a good way to specify the data behavior of the system (embodied in SPECIAL), but no good way as yet to specify performance properties.
* Given the existing methods for performance prediction (such as simulation and analysis), we saw no good way of incorporating any of these into HDM, because abstraction and performance analysis are conflicting criteria from which to view a system.

However, it is possible that the near future will bring about a way of incorporating performance properties into HDM. Thus, it is recommended that HDM be used alone, with the possible exception for performance evaluation techniques, which may be of considerable help in deciding on design and implementation decisions.

HDM was originally created to specify operating systems. Some characteristics of operating systems that make them amenable to treatment by HDM are

* A "virtual machine" that is provided for the user.
* Many natural abstractions that are provided for users (such as processes, virtual memory, files) and that are used by parts of the system (such as pages and queues).

Special-purpose operating systems, such as message-processing systems, are also amenable to treatment by HDM. The same is true for database systems, which provide an abstract machine (perhaps a relational data base) for the user. On the other hand, compilers and systems for business data processing present problems for HDM. Compilers present difficulty because it is not easy to write in a precise way what it means for a compiler to function correctly: no one has yet done so for a nontrivial compiler. However, HDM is based on a model of computation in terms of which compilers can be described; future research in HDM will investigate the problem of precisely specifying a compiler. In any case, it is now possible to use HDM to specify the parts of a compiler; this exercise has proven useful in two applications of HDM: a program manipulation system ([23], [24]) and the syntactic processor for SPECIAL [20]. With regard to systems for business data processing, there is a minimal amount of abstraction: The data manipulated by the programs is the data seen by the user. The only possible abstract machine consists of a file and the programs that manipulate it (which typically read, write, and access the data in the current record). However, defining such an abstract machine is extremely useful, because the machine localizes all the code (usually distributed) that knows about the format of the file. Using this scheme, file formats can be changed with only local code modifications.

In considering the possible use of HDM for software development, the manager should not think that he is getting something for nothing. However, it is believed that many managers would be willing to pay some price for better and cheaper software.

V TRACK RECORD OF HDM

At this writing, HDM is largely experimental, having been used mostly to design and implement systems of a research nature. However, a production effort [25] is currently underway. Transfer to a production environment is still in the future, because training in the use of HDM takes some time to accomplish (see Section IV-C). However, the use of HDM is expanding, and it has been strongly considered for even greater use in both the government and private industry.

Besides trivial systems, HDM (or techniques that later evolved into HDM) has been used in fully implemented systems on several occasions. Running systems developed to date include:

* A KWIC index system [13]
* A virtual memory mechanism [26]
* A program-manipulation system ([23], [24])
* A system for checking the logical soundness of mathematical proofs
* A symbol table for a system that verifies COBOL programs [27].

The first three of the above systems were developed before HDM was originated, using the specification technique proposed by Parnas ([13], [14]). The language of these specifications, which has no formal syntax, is a rudimentary version of SPECIAL; the other rules and restrictions used in constructing these systems resemble those of HDM. HDM has formalized these rules and has provided a language and a framework for expressing them.

Another fairly large system has been designed and implemented using HDM, but has not yet been run. The system compiles a frequency table of words occurring on a file (see [28]). In addition, specifications are currently being written for the syntactic processor for SPECIAL, and implementations (in ILPL) will be written shortly.

HDM has also been used by SRI under government contract to specify several large systems, none of which has been implemented as yet. These include

* PSOS (Provably Secure Operating System) [29], whose major design goal is verifiable security (funded by DoD).
* SIFT (Software Implemented Fault Tolerance) [30], an ultra-reliable computer system for flight control on commercial aircraft (funded by NASA).
* KSOS (Kernelized Secure Operating System) [25], a secure version of UNIX intended for production use throughout the Department of Defense (funded by ARPA).
* RTOS (Real-Time Operating System), a real-time operating system for tactical applications (funded by the Army).
* A family of message processing systems [31] (funded under this contract to the Navy).

SIFT, RTOS, and KSOS will be implemented shortly, and a procurement for the implementation of PSOS is likely to be issued.

In the commercial realm, Honeywell has used HDM to specify a security kernel for Multics, and is currently using HDM to design the SCOMP (Secure COMPuting) System, which provides a secure kernel on a minicomputer to support many kinds of systems, and an aircraft navigation system [32]. In addition, Burroughs has used HDM for the design of a communication switching system [33].

With respect to verification, security properties of PSOS [18] and reliability properties of SIFT [30] have been proven using an abstract model as a definition of security or reliability, and specifications written in SPECIAL. Part of the implementation of PSOS has also been proved consistent with its specification [29]. Eventually, SIFT, KSOS, and PSOS (if implemented) will be completely verified, both for design properties and correctness of implementation.

HDM has been written into several proposals for systems whose properties must be satisfied. In the KSOS development mentioned above, precise specifications, "in a language other than English," are a required deliverable, and SPECIAL has been referenced in the RFP as a possible specification language. If the implementation of PSOS is

procured, the specifications of the system, written in SPECIAL, will be part of the RFP.

All of the limited experience with HDM has been positive. In the design phase of all these projects, the use of HDM has helped clarify the design issues, and -- although the design took longer -- the interfaces were clear and compact. The experience in implementation substantiates the conjecture that the implementation stage is straightforward, even if the design phase is difficult. After relatively little testing, at least two systems (the program-manipulation system and the symbol table for COBOL verification) have had bug-free operation over their use, which was somewhat limited. The program-manipulation system endured a substantial change in the implementation of its lowest level (for performance reasons) but required no modification of the code at any other level, because the interface defined by the specification of the lowest level was preserved. It is hoped that further experiences with HDM continue on this positive note.

VI A PRELIMINARY ASSESSMENT OF HDM

The following is a list of the most important aspects of HDM:

* Its complete approach to the expression of decisions required in software development.
* Its structuring of design and implementation.
* Its language for precisely expressing design specifications.
* Its flexible restrictions, which prohibit many bad design and programming practices while permitting creativity on the parts of designers and implementors.
* Its ability to check for completeness and consistency.

The above features of HDM promote the following positive attributes of software designed using HDM:

* Increased understandability
* Better management control
* More reliable systems
* Potential for cutting costs

These advantages of HDM have been adequately discussed in preceding sections of this report.

The following potential problems of HDM arise mostly from the difficulty to be encountered in transferring HDM to existing software production environments:

* Different personnel structure and requirements
* Different project management procedures
* More time to learn
* More difficult to use
* Applicability for all kinds of systems not yet proven.

The first two problems state that HDM is different, but a case has been presented that some departure from existing methods is necessary to solve the software problem. The second two problems state that HDM is difficult, reflecting the general difficulty inherent in the production

of software, as stated here in the presentation of the software problem. The difficulty of using HDM contrasts with methods that may make software production seem easy during the early stages, only to encounter difficulty during the later stages. These four problems may cause resistance to accepting HDM, even if it benefits the software-development process. This last problem is one of not having tested HDM on every conceivable kind of system; further research is necessary to arrive at a firm determination of the classes of systems for which HDM is applicable and perhaps a quantitative assessment of its applicability. It is our conjecture that HDM would be of considerable benefit for most classes of systems, but this has not been substantiated. Of course, since HDM has had only limited use in a production environment, this may not be a complete list.

VII HDM AND DOD PROCEDURES FOR SYSTEM DEVELOPMENT AND PROCUREMENT

VII-A. Relation of HDM to the DSARCs

There are three DSARCs (Defense Systems Acquisition Review Council), or review points, in the DoD software-development process. These are described in detail in [34]. More details concerning the relationship of HDM to the DoD software-development process are available in [35].

DSARC I deals primarily with the system requirements and a statement of the system concept. Insofar as requirements are concerned, HDM is not currently equipped for stating or evaluating system requirements. However, the system concept may be expressed informally in terms of abstract machines and programs, but having no structural decomposition as yet.

DSARC II evaluates the coarse design for the system (including both hardware and software), currently embodied in a Type A specification. At this point with regard to HDM, the functionality all system components should have been identified. These components can be described as abstract machines. Informal abstract programs (in the form of block diagrams) may be used to specify control within the system. The name and nature of each operation of each component will also have been specified.

DSARC IIB is the final design evaluation of the system before starting implementation of the prototype. This corresponds to the Type B specification for the hardware and the Program Performance Specification (PPS) for the software. The preliminary and critical design reviews are held before this DSARC. With regard to HDM, the specifications of both software (all levels) and hardware (in SPECIAL) will have been written and evaluated, and can be included in the PPS.

DSARC III evaluates an operational system before production deployment. At this point the Program Design Specification (PDS) will have been produced for the software. With regard to HDM, the specification, representation, and implementation stages will have been completed by this time. All outputs of HDM can be included in the PDS. Last-minute changes can be made at this point, with the understanding that it is much more difficult to change a system after it has been deployed. The results of any validation or verification attempt should also be made known at this point.

HDM fits rather well into the current structure for the DoD software-development process. Outputs of HDM can add to general system understanding when added to the Type A specification, the PPS, and the PDS.

VII-B. Relation to Procurement

HDM has several uses in systems being procured by DoD (see also [35]). First an HDM specification may be required as a deliverable in the contract for a system design (as was done in the case of Secure UNIX, mentioned in Section V). Second, if the system has already been designed using HDM, satisfaction of the specification may be a requirement for the implementation. The specification itself may be written into the RFP (as is the case for the possible PSOS procurement, mentioned in Section V). These practices should help both DoD and the contractor, because DoD is demanding (or issuing) a precise statement of system properties, and because the contractor is clear as to what it should produce.

VIII CONCLUSIONS

HDM (the SRI Hierarchical Development Methodology) is an integrated set of concepts, procedures, languages, and on-line tools that encompasses all stages of the software-development process. The goal of HDM is the consistent production of software systems that are more correct, reliable, and maintainable than those resulting from the use of current techniques. The use of such an approach could lead to considerable cost savings over the development of software by conventional methods.

HDM leads to the creation of systems that are well-structured and precisely specified, for which it is often feasible to prove mathematically that the system meets its specifications -- even for large systems. HDM also provides for the development of an entire family of systems at once, resulting in additional cost savings.

HDM is fully operational for the design stages of software development. However, more work, both research and engineering, is required to fully develop the languages and tools for the other stages of HDM and to integrate them with the existing languages and tools.

Early experiences in the use of HDM have been favorable, but a full evaluation of its utility awaits more experience in the use of HDM to produce many classes of systems in a production environment.

REFERENCES

1. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, pp. 48-59 (May 1973).

2. Proceedings of a Symposium on the High Cost of Software, J. Goldberg, ed., (Stanford Research Institute, Menlo Park, California, 1973).

3. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. 11, No. 1, pp. 56-73 (1972).

4. L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "On Attaining Reliable Software for a Secure Operating System, " Proc. Intl Conf. on Reliable Software, 13-15 April 1975, Los Angeles, CA, in SIGPLAN Notices, Vol. 10, No 6, pp. 267-284 (June 1975).

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

6. K. N. Levitt and L. Robinson, "The HDM Handbook," Technical Report CSL-68, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park, California (April 1978).

7. M. Hamilton and S. Zeldin, "Higher Order Software -- a Methodology for Defining Software," IEEE Trans. Softw. Eng.," Vol. SE-2, No. 1, pp. 9-32 (March 1976).

8. M. Jackson, Principles of Program Design (Academic Press, London, 1975).

9. E. Yourdon and L. Constantine, Structured Design, (Yourdon, Inc., New York, 1975).

10. E. W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," in Report on a Conference on Software Engineering, Randell and Naur, eds. (NATO, 1968).

11. E. W. Dijkstra, "Notes on Structured Programming," in Structured Programming, C. A. R. Hoare, ed. (Academic Press, New York, 1972).

12. D. L. Parnas, "Information distribution Aspects of Design Methodology," Information Processing 71, pp. 339-344 (North-Holland Pub. Co., Amsterdam, 1972).

13. D. L. Parnas, "A Technique for Software Module Specification with Examples," Comm. ACM, Vol. 15, No. 5, pp. 330-336 (May 1972).

14. D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Comm. ACM, Vol. 15, No. 12, pp. 1053-1058 (December 1972).

15. D. L. Parnas, "On a 'Buzzword': Hierarchical Structure," Information Processing 74, pp. 336-339 (North-Holland Pub. Co., Amsterdam, 1974).

16. R. W. Floyd, "Assigning Meanings to Programs," Mathematical Aspects of Couputer Science 19, J. T. Schwartz, ed., pp. 19-32 (Amer. Math. Soc., Providence, Rhode Island, 1967).

17. R. S. Boyer and J S. Moore, "Proving Theorems about LISP Functions," J. ACM, Vol. 22, No. 1, pp. 129-144 (1975).

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

19. L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Comm. ACM, Vol. 20, No. 4, pp. 271-283 (April 1977).

20. O. M. Roubine and L. Robinson, "The SPECIAL Reference Manual," Technical Report CSL-45, SRI Project 4828, Contract N00123-76-C-1095, Stanford Research Institute, Menlo Park, California (January 1977).

21. H. Sackman, W. J. Erickson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," Comm. ACM, Vol. 11, No. 1, pp. 3-11 (January 1968).

22. H. D. Mills, "How to Write Correct Programs and Know It," Proc. Intl. Conf. on Reliable Software, 13-15 April 1975, Los Angeles, CA, in SIGPLAN Notices, Vol. 10, No. 6, pp. 363-370 (June 1975).

23. L. Robinson, "Design and Implementation of a Multi-Level System Using Software Modules," Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (June 1973).

24. L. Robinson and D. L. Parnas, "A Program Holder Module," Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (June 1973).

25. Secure Minicomputer Operating System (KSOS) Executive Summary — Phase I: Design of the Department of Defense Kernelized Secure

Operating System, Ford Aerospace and Communications Corporation, Palo Alto, California (March 1978).

26. W. R. Price, "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania (June 1973).

27. L. Robinson, M. W. Green, R. E. Shostak, and J. M. Spitzen, "The Verification of COBOL Programs," Final Report, SRI Project 3967, Contract DAHC-04-75-0011, Stanford Research Institute, Menlo Park, CA (March 1976).

28. K. N. Levitt and L. Robinson, "A Detailed Example of the Use of HDM," Technical Report CSL-72, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park, California (April 1978).

29. 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," Final Report, SRI Project 4332, Stanford Research Institute, Menlo Park, California (February 1977).

30. J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak, "The Design, Analysis, and Verification of the SIFT Fault Tolerant System," Proc. 2nd Intl. Conf. on Software Engineering, 13-15 October 1976, San Francisco, California, pp. 458-469 (October 1976).

31. L. Robinson and O. M. Roubine, "The Preliminary Design of a Family of Message Processing Systems Using HDM," Technical Report CSL-71, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park, California (April 1978).

32. W. E. Boebert, J. M. Kamrad, E. R. Rang, "Analytic Validation of Flight Hardware," Technical Report 77SRC63, Systems and Research Center, Honeywell, Minneapolis, Minnesota (September 1977).

33. K. S. Shankar and C. S. Chandersekaran, "Data Flow, Abstraction Levels and Specifications for Communications Switching Systems," Proc. 2nd Intl. Conf. on Software Engineering, 13-15 October 1976, San Francisco, CA, pp. 585-591 (October 1976).

34. Program Management Manual, NELCINST 5000.2 (Naval Electronics Laboratory Center, San Diego, California, January 1976).

35. L. Robinson, "The Relation of HDM to the Navy's Software-Development Process," Technical Report CSL-70, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park, California (April 1978).

Appendix A

GLOSSARY

ABSTRACTION — The process of isolating a few of the properties of an object to explain it or to understand it more easily. Also, the object containing only the properties that are isolated as part of the abstraction process.

CHECKER, HIERARCHY — The on-line tool of HDM that syntactically checks the specification of a hierarchy, written in HSL, against the specifications of the interfaces, modules, and representations in the hierarchy.

CHECKER, IMPLEMENTATION — The proposed on-line tool of HDM that syntactically checks specifications for abstract programs, written in ILPL, for self-consistency.

CHECKER, INTERFACE — The on-line tool of HDM that syntactically checks an interface specification, written in HSL, against the specifications of the modules contained in the interface.

CHECKER, MODULE — The on-line tool of HDM that syntactically checks a module specification, written in SPECIAL, for self-consistency.

CHECKER, REPRESENTATION — The on-line tool of HDM that syntactically checks the specification of a representation, written in SPECIAL, for self-consistency and for consistency with the higher- and lower-level modules that it references.

CODING — The seventh stage of HDM, in which the abstract programs of a system, written in ILPL, are translated into executable code that can be compiled, interpreted, or assembled using some existing tool.

CONCEPTS — The scientific results and global model that form the basis of a methodology.

CONCEPTUALIZATION — The first stage of HDM, in which the problem to be solved by a software system is formulated as independently of any solution as possible.

DATA BASE, DEVELOPMENT — The proposed on-line tool of HDM that keeps track of all specifications for a system and any changes made to them, rechecking consistency when necessary and alerting the user to any change that would make the system inconsistent.

DATA STRUCTURES, INTERNAL — The "memory" of a module, in terms of which the module's operations are defined.

FAMILY GENERATING DESIGN — A design, or formal specification, of a system, containing all the properties shared by a system family. Designs for the individual family members can be produced from the family generating design by the process of particularization.

FAMILY MEMBER DESIGN — The design, or formal specification, for a particular member of a system family, having all of the properties of its family and some additional properties not shared by the other family members.

FAMILY, SYSTEM — A set of software systems that share a set of common properties.

FORMAL — Precise, having a mathematical basis. HDM is formal in this sense. Note: This term has another meaning in the standard military terminology: having or appearing in a standard form.

HDM (Hierarchical Development Methodology) — A methodology for the development of large software systems, developed at SRI for the purpose of solving the "software problem." HDM is based on the concepts of hierarchical structure, abstraction, modularity, and program verification.

HIERARCHICAL STRUCTURE — A particular kind of structure in which, if one component of the system is dependent on another, the reverse is not also true. A system with this structure has advantages in regard to passing information and changing code within the system.

HIERARCHY — A system or organization that has a hierarchical structure.

HSL (Hierarchy Specification Language) — In HDM, the language for specifying the modules that make up a level and the structuring of levels and representations that forms a hierarchy.

ILPL (Intermediate Level Programming Language) — In HDM, the language for expressing implementation decisions, i.e., for writing abstract programs.

IMPLEMENTATION, ABSTRACT — The sixth stage of HDM, in which abstract programs (in ILPL) are written, implementing each operation of each level.

INTERFACE DEFINITION, EXTERNAL — The second stage of HDM, in which the external interfaces of the system (that is, the top and bottom levels) are conceived.

INTERFACE DEFINITION, INTERMEDIATE — The third stage of HDM, in which the intermediate interfaces (that is, the intervening levels) are conceived.

LANGUAGES — In a methodology, the media for describing things according to the concepts on which the methodology is based.

LEVEL — A unit of a hierarchically structured system in HDM, providing a set of operations to the programs implementing the operations of the next higher level (or to the users, if it is the top level), and which — if it is not the lowest level — is implemented by a set of programs invoking the operations of the next lower level.

MACHINE, ABSTRACT — A level of a hierarchically structured system in HDM, providing a set of operations to the programs or users that use that machine.

MAPPING — A set of mapping function expressions, which defines the formal representation of one or more nonprimitive modules in terms of the modules at the next lower level.

MAPPING FUNCTION EXPRESSION — An expression, written in SPECIAL, that defines a particular internal data structure of a module of a given level in terms of the internal data structures of the modules at the next lower level.

METHODOLOGY — In software development, an integrated set of concepts, procedures, languages, and on-line tools for developing large systems.

MODULARITY — The property of consisting of easily replaceable parts called modules, having well-defined external interfaces.

MODULE — In general, a part of a system with a well-defined external interface, and which is easily replaceable by any component having the same interface. In HDM, a set of operations and internal data structures with the general properties of a module. Note: This definition is different from the conventional one, in which a module is a subroutine whose operations on shared data may not be well-defined.

OPERATION — A functional capability provided by a level or module that can be invoked by a user or a program that has access to that level or module. In a specification, an operation is defined in terms of the changes that its invocation makes to the module's internal data structures.

PARTICULARIZATION — The process of adding properties to a family generating design, to arrive at a family member design.

PROCEDURES — In a methodology, a description of the way in which a software system is developed, usually by dividing the software-development process into stages.

PROGRAM, ABSTRACT — In HDM, a program that describes a sequence of invocations to the operations of a given level of the system.

REPRESENTATION, FORMAL — A specification (written in SPECIAL) of the definition of the internal data structures of the modules at a given level, in terms of the internal data structures of the modules at the next lower level. Also, the fifth stage of HDM, at which the formal representations are written.

SIMULATION, MODULE — A duplication of the behavior of a module or a level of a software system, based only on the specification for that module or level.

SIMULATION SYSTEM, MODULE — An on-line tool that performs module simulation.

SOFTWARE PROBLEM, THE — The fact that large software systems often exceed their estimated costs of development, are unreliable, do not meet their requirements, and are difficult to maintain.

STAGE — A part of the software-development process, used primarily for the purposes of description. Different methods of software development divide this process into different sets of stages.

SPECIAL (SPECIfication and Assertion Language) — In HDM, the language used for specifying modules and representations. It specifies a program by stating what it does, independently of how it achieves that effect.

SPECIFICATION, FORMAL — A document explaining precisely what a given software system does, preferably without explaining how it is done. In HDM the formal specifications are written in SPECIAL. Also, the fourth stage of HDM, in which formal specifications are written for each module of each level of the system.

TOOLS, ON-LINE — In a methodology, a set of programs that process statements in the languages of the methodology and that enforce restrictions derived from the concepts of the methodology.

VERIFICATION, PROGRAM — The process of mathematically proving the consistency of a program with its specification, that is, that the program does what it is supposed to do.

VERIFICATION SYSTEM, PROGRAM — An on-line tool that performs program verification, either completely automatically or with user guidance.

Appendix B

EXAMPLE OF HDM

1. Introduction

This example illustrates the use of the three languages of HDM — HSL, SPECIAL, and ILPL — to describe a system completely.

2. Interfaces and Hierarchy of the Example

In this subsection each of the levels of the system is defined: the top level (LEVEL1), consisting only of the "stacks1" module; and the bottom level (LEVEL0), consisting only of the "arrays1" module. The hierarchy specification shows how the two levels are related using the "stacks1" representation.

```
(INTERFACE level1 stacks1)

(INTERFACE level0 arrays1)

(HIERARCHY stackexample (level0 IMPLEMENTS level1 USING stacks1))
```

3. Global Objects

In this subsection is provided a list of the global objects (primarily functions) of each module and representation.

STACKS1 MODULE

```
        stack: DESIGNATOR
        INTEGER max_stack_size
HP      VFUN ptr(stack s) -> INTEGER i
HP      VFUN stack_val(stack s; INTEGER i) -> INTEGER v
        OFUN push(stack s; INTEGER v)
        OVFUN pop(stack s) -> INTEGER v
```

ARRAYS1 MODULE

```
        array: DESIGNATOR
        INTEGER array_size
VP      VFUN access_array(array a; INTEGER i) -> INTEGER v
        OFUN change_array(array a; INTEGER i, v)
```

STACKS1 MAPPING

(no global objects)

4. Module Specifications

This subsection contains the formal specifications (written in SPECIAL) for the modules of the example system.

STACKS1 MODULE

```
MODULE stacks1  $( maintains a fixed number of stacks of integers,
                  each of the same fixed maximum size)


    TYPES

stack: DESIGNATOR $( names for stacks) ;


    PARAMETERS

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


    FUNCTIONS


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

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

OFUN push(stack 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 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
```

ARRAYS1 MODULE

```
MODULE arrays1 $( maintains a fixed number of fixed-size
                  integer arrays)


    TYPES

array: DESIGNATOR;


    PARAMETERS

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


    FUNCTIONS

VFUN access_array( array a; INTEGER i) -> INTEGER v;
    $( returns element i of array a)
    EXCEPTIONS
      array_bounds : NOT i INSET {0 .. array_size - 1};
    INITIALLY
      v = 0;

OFUN change_array(array a; INTEGER i, v);
    $( changes the ith value of array a to v)
    EXCEPTIONS
      array_bounds: NOT i INSET {0 .. array_size - 1};
    EFFECTS
      'access_array(a, i) = v;

END_MODULE
```

5. Formal Representation

The formal representation (in SPECIAL) of the internal data structures of the "stacks1" module, in terms of the internal data structures of the "arrays1" module, is provided in this subsection.

STACKS1 MAPPING

```
    MAP stacks1 TO arrays1;


    EXTERNALREFS

    FROM stacks1:
stack: DESIGNATOR;
INTEGER max_stack_size;
VFUN ptr( stack s) -> INTEGER i;
VFUN stack_val( stack s; INTEGER i) -> INTEGER v;

    FROM arrays1:
array: DESIGNATOR;
INTEGER array_size;
VFUN access_array( array a; INTEGER i) -> INTEGER v;


    INVARIANTS

FORALL array a: access_array(a, 0) INSET {0 .. array_size - 1};


    MAPPINGS

stack: array;

max_stack_size: array_size - 1;

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

stack_val( stack s; INTEGER i):
      IF i INSET {1 .. access_array(s, 0)} THEN access_array(s, i)
        ELSE ?;


    END_MAP
```

6. Abstract Implementation

The abstract implementation of the functions of the "stacksl" module in terms of the "arraysl" module, written in ILPL, is provided in this subsection.

STACKS1 IMPLEMENTATION

```
IMPLEMENTATION stacks1 IN TERMS OF arrays1;

    EXTERNALREFS

    FROM stacks1:
stack: DESIGNATOR;
INTEGER max_stack_size;
OFUN push(stack s; INTEGER v);
OVFUN pop(stack s) -> INTEGER v;

    FROM arrays1:
array: DESIGNATOR;
INTEGER array_size;
VFUN access_array(array a; INTEGER i) -> INTEGER v;
OFUN change_array(array a; INTEGER i, v);


    TYPE MAPPINGS

stack: array;


    INITIALIZATION
BEGIN
   max_stack_size <- array_size - 1;
END;


    IMPLEMENTATIONS

OPROG push(stack 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);
      END;
   change_array(s, 0, i);
END;

OVPROG pop(stack 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;
```

STACKS1 IMPLEMENTATION

END_IMPLEMENTATION

DISTRIBUTION LIST

Defense Documentation Center — 20 copies
Cameron Station
Alexandria VA 22314

Mr. Tony Allos — 1 copy
Code 6201
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92152
(with cover letter through Contracts)

Mr. Lin Sutton — 35 copies
Code 8223
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92152

Mr. William L. Carlson — 2 copies
Defense Advanced Research Projects Agency
Information Processing Technology Office
1400 Wilson Boulevard
Arlington, VA 22209

Mr. Neal Hampton — 15 copies
Code 8223
Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, CA 92152

Ms. Marlene Hazle — 1 copy
MITRE Corp.
Box 208
Bedford, MA 01730

Professor Stuart Madnick — 1 copy
MIT
Sloan School of Business
E53-330
Cambridge, MA 02139

Mr. John Machado — 5 copies
Naval Electronic Systems Command
National Center No. 1
Crystal City
Washington, DC 20360

Mr. William Rzepka — 5 copies
ISIM
Rome Air Development Center
Griffiss Air Force Base

Rome, NY 13441

Dr. Harold S. Schwenk, Jr. 1 copy
BGS Systems, Inc.
P.O. Box 128
Lincoln, MA 01773

Internal (Computer Science Laboratory of SRI International):

Jack Goldberg	1 copy
Karl Levitt	1 copy
Mark Moriconi	1 copy
Lawrence Robinson	5 copies
Brad Silveberg	1 copy

Total: 96 copies