ing Techniques, J. N. Buxton and B. Randell, Ed. Brussels, Belgium: NATO Scientific Affairs Division, 1970, pp. 84–87.
[4] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, pp. 221–227, Apr. 1971.
[5] W. A. Wulf, "The GOTO controversy: A case against the GOTO," *SIGPLAN Notices*, vol. 7, pp. 63–69, Nov. 1972.
[6] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, pp. 549–557, Oct. 1974.
[7] E. W. Dijkstra, "On the axiomatic definition of semantics," *EWD 367*, privately circulated.
[8] D. L. Parnas, "On the criteria used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, Dec. 1972.
[9] ——, "A technique for software module specification with examples," *Commun. ACM* (Programming Techniques Dept.), pp. 330–336, May 1972.
[10] P. Naur, "Programming by action clusters," *BIT*, vol. 9, pp. 250–258, 1969.
[11] P. Henderson and R. Snowdon, "An experiment in structured programming," *BIT*, vol. 12, pp. 38–53, 1972.
[12] D. L. Parnas, "A course on software engineering techniques," in *Proc. ACM SIGCSE*, 2nd Tech. Symp., Mar. 24–25, 1972.
[13] E. W. Dijkstra, "Co-operating sequential processes," *Programming Languages*, F. Genuys, Ed. New York: Academic Press, 1968, pp. 43–112.
[14] W. R. Price, "Implications of a virtual memory mechanism for implementing protection in a family of operating systems," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1973.
[15] B. Randell and F. W. Zurcher, "Iterative multi-level modelling—A methodology for computer system design," in *Proc. IFIP Congr.*, 1968.
[16] D. L. Parnas, "On a 'buzzword' hierarchical structure," in *Proc. IFIP Congr.*, 1974, pp. 336–339.

David L. Parnas received the B.S. and M.S. degrees in electrical engineering, and the Ph.D. degree in systems and communications sciences, from the Carnegie Institute of Technology, Pittsburgh, PA, in 1961, 1964, and 1965, respectively.

He has held the position of Assistant Professor of Computer Science, University of Maryland, College Park, and was Assistant and Associate Professor of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. Since June of 1973 he has been Professor and Head of one of the two Research Groups on Operating Systems at the Technische Hochschule Darmstadt, Darmstadt, West Germany. He is also a consultant for the U.S. Naval Research Laboratory, Washington, D.C. His areas of research have been design methods for computer systems, process synchronization in operating systems, security mechanisms in operating systems, simulation techniques, and design automation.

Higher Order Software—A Methodology for Defining Software

MARGARET HAMILTON AND SAYDEAN ZELDIN

Abstract—The key to software reliability is to design, develop, and manage software with a formalized methodology which can be used by computer scientists and applications engineers to describe and communicate interfaces between systems. These interfaces include: software to software; software to other systems; software to management; as well as discipline to discipline within the complete software development process. The formal methodology of Higher Order Software (HOS), specifically aimed toward large-scale multiprogrammed/multiprocessor systems, is dedicated to systems reliability. With six axioms as the basis, a given system and all of its interfaces is defined as if it were one complete and consistent computable system. Some of the derived theorems provide for: reconfiguration of real-time multiprogrammed processes, communication between functions, and prevention of data and timing conflicts.

The first step in defining a system with a formal methodology is to apply a formalized set of rules. We have found that enforcing such rules, especially on a large project with many organizations, is very difficult. In fact, it is almost impossible without the aid of automated tools to describe the design process and its verification. We envision a scheme in which the definition of a given system can be described with an HOS specification language which, by its very nature, enforces the axioms with the use of each construct. A system defined in HOS can be analyzed automatically for axiomatic consistency by the Design Analyzer without program execution, and by the Structuring Executive Analyzer on a real-time basis. The result is that a software system can be developed efficiently with reliable interfaces. This is significant since interface testing in a large system accounts for approximately 75 percent of the verification effort.[1]

Manuscript received May 1, 1975; revised October 15, 1975. This paper was prepared under Contract NASA9-13809 with the Lyndon B. Johnson Space Center of the National Aeronautics and Space Administration, and under The Charles Stark Draper Laboratory, Inc., Internal Research and Development Funds. The publication of this paper does not constitute approval by the National Aeronautics and Space Administration of the findings or conclusions contained herein. It is published only for the exchange and stimulation of ideas. The Charles Stark Draper Laboratory, Inc., 68 Albany St., Cambridge, MA 02142 already holds the copyright to the contents contained in this manuscript.

The authors are with the Computer Science Division, The Charles Stark Draper Laboratory, Inc., Cambridge, MA 02142.

[1] Seventy-three percent of all problems found during the APOLLO integration effort were interface problems [2]; and verification accounts for 50 percent of the total software development effort [3]–[5].

Index Terms–**Axioms, formal methodology, functional decomposition, interface correctness, specification, static verification, structuring executive.**

I. Introduction

HIGHER Order Software (HOS) is a formal methodology for reliable systems specification and development [1]. In order to devise a methodology which is totally system oriented and not traditionally software oriented, it became apparent that the definition of a software system must be hardware independent, language implementation independent, and computer resident software independent. Thus, HOS is concerned only with computable functions and their relationships, e.g., hierarchical decomposition into subfunctions.

For this discussion we will consider the class of effectively computable functions to coincide exactly with the definition of the term, software. A function is said to be effectively computable if there is a mechanical and finite method of calculating the value of the function in all cases when its arguments are given [6]. A system function that is not computable can interface with a software system if we define a set of functional interfaces for that noncomputable system. Then it is possible to interface software with other systems, to model any system in the form of software, to show a noncomputable system within a total problem specification, or to integrate any combination of the above.

If a given system can be defined in such a way as to be "software," then it follows that a reliable methodology for defining general systems can be applied to software and vice versa. Secondly, if a given system is defined using software techniques, that system can be functionally tested on a computer. If we look at systems, in general, as software, it may help to more thoroughly understand a complicated system, or help make a seemingly complicated system less complicated.

It is possible within the framework of HOS to develop a new class of automatic tools. For example, the interfaces of an HOS system can be exhaustively tested by an automatic analyzer without program execution. We consider this tool to be important, since interface testing in a large system is known to be a very costly procedure.[2]

The HOS methodology can be used for the definition of software for multiprogrammed, multiprocessor, or multicomputer systems. For any of these systems, the approach would be to first use HOS to design one implementation independent system structure[3] and then, for implementation, to distribute the functions of that system among software and hardware resources while maintaining the original system structure.

II. Methodology

HOS is software expressed in its own metalanguage and conforming to a formalized set of laws. The basic components of HOS methodology are (Fig. 1): 1) the application of the formal set of laws to the design of a given problem; 2) a specification language adhering to these laws; 3) the automatic analysis of system interfaces by the Design Analyzer and the Structuring Executive Analyzer; 4) the architectural virtual layers produced from analyzer output in the form of software, firmware, or hardware[4]; and 5) the hardware that is transparent to the user.[4] In addition, support tools based on system consistency with the axioms could enhance a given development process in such areas as: performance analysis, simulation, design automation, definition of subsystem requirements, automatic documentation, and automatic management techniques.

A. Formulation

In HOS, any given software system can be represented by a single mathematical function where the input set defines the domain of the function and the output set defines the range of the function. The function representing the entire system is an assumption from which subfunctions, representing subsystems, are derived. Those functions that perform[5] the "system-function" describe subsystems, each (subsystem) of which may be represented by a single mathematical function of its own. Since we assume a hierarchical structure for a software system, we must define the elements and relations of that hierarchy [7]. We consider a software system to be a hierarchy in which the elements of a hierarchical system are the mathematical functions and the defining relation of that hierarchy is that of control (explained below).

The design for a particular software system is based on six axioms that describe control and are described by a metalanguage. These axioms explicitly define hierarchical software control, where control is a formally specified affect of one software object to another software object. Each affect is implemented by a mechanism (such as a language) which is effected at the next and only the next most immediate lower virtual layer.

A virtual layer is a system in which the input and output variables are completely defined by the axiomatic specification of any given system structure. For example, the decomposition of a software problem is formally defined level by level with respect to problem definition (i.e., the net effect of performing each subfunction at a given level is a functional redefinition of the most immediate higher node of the hierarchical tree), and layer by layer with respect to implementation (i.e., the net effect of translating a given control mechanism).

The formal definition of the control system, first described in [1], is reprinted in Appendix I (with minor modifications) as a guide to the following discussion. In addition, the definitions of the following aspects of control are provided below as an aid to the reader.

An access right provides for the ability to locate an element of a given set of variables, and once located, the ability to reference or replace a value of said element.
Invocation provides for the ability to perform a function.

[2]See footnote 1.
[3]A system framework which specifies the relationships for functions and their interfaces.

[4]Our work, to date, has concentrated on the first three areas of HOS methodology.
[5]Perform: provide one and only one element of the output set for a particular element of the input set.

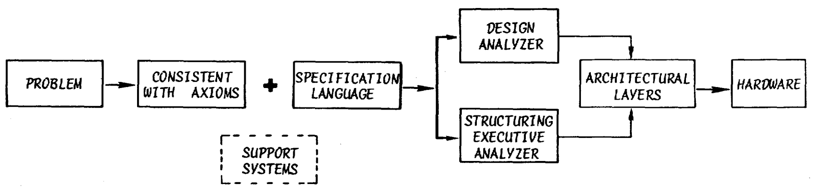


Fig. 1. Higher Order Software methodology.

Ordering provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements precedes the other said element.

Responsibility provides for the ability of a module to produce correct output values.

Rejection provides for the ability to recognize an improper input element in that if a given input element is not acceptable, null output is produced.

On the basis of the axioms (see Appendix I), many theorems have been derived. Theorems exist which minimize logical interfaces. For example, if a function from a given control level is removed and its controller module still maintains the same relationship between input and output, the function is extraneous. Violation of this theorem, in common practice, manifests itself in modules with many user options. With respect to the entire system, the use of extraneous functions proliferates test cases and complicates interfaces. Theorems exist for proper structuring of real-time multiprogrammed processes in which one or more of a controller's subfunctions is realized as an asynchronous process. In this way a function can be implemented as a process tree (a process and its dependents). For example, each module controls the priority relationships of its dependent processes. Not only are these priority relationships relative rather than absolute, but each controller at a given level has a higher relative priority than each process tree that implements its subfunctions. The order of process activation can be derived from the order of function invocation as defined by the axioms. In a multiprogramming environment there is a further constraint in that the order of process activation is preserved under interrupts occurring at any level in the hierarchy. This implies that if process A is of higher priority than process B, process A always interrupts all of B; i.e., the priorities of the dependent processes of A are all higher than the priorities of all dependent processes of B.

The axioms guarantee that data conflicts among functions are eliminated. In addition to the functional relationships between input and output variables, one can also attach a predicted time to the execution of a given function. Using the axioms, a particular system of functional relationships, and a particular interrupt structure, one can determine the maximum completion or delay time for a given set of subfunctions.

B. Interface Correctness

Interface correctness is a property of a system which results in the ability to perform a function without ambiguity. We believe that the interface correctness of any given software system can be proven if the system interfaces are shown to be consistent with the axioms of HOS. One aspect of an interface is that it relates the specification at one node to the specification at another node.

The specification of a node is the relationship of the element of the input set to the one and only element of the output set that corresponds to said input element. For example, if we can enumerate such a mapping, we could make a table for the specification such as:

x	y
1	2
3	6
.	.
.	.
.	.

The question: "do we want y to be 6 if x is 3?" is not relevant here, since the table is the original assumption, i.e., given $y = 3$ when $x = 6$.

If one and only one subfunction is required to assign a particular element of the output set of a controller, the interface between the elements (or table value) at the controller level to the elements at each subfunction level can be determined. That is, each element is accounted for in a unique way by the definition of the subfunction of the controller. The correct interface is validated by accounting for each element; it does not validate the specification, i.e., the table itself. The question: "does the table do the job?" is what we will call performance testing. Whenever a new "table" is defined, the question of valid assumption must be determined.[6] A new table is defined whenever the variables of the output set of one function are the variables of the input set of another function; both of which exist at the same level and are controlled by the same controller. Subsequent decomposition of each function must be consistent with the table at the higher level. Thus, with HOS, we can determine the domain and expected values when a performance test is required. In some cases, a performance test can be a formal proof of correctness. In other cases, the performance of a function can only be validated by demonstration (i.e., executing the function with sample values). When the specification itself is an approximation, we can validate the table by simulation.

Since an element is obtained from a value from each of the variables that determined an input or output set, the variables themselves bound the elements and are, therefore, considered

[6]Thirteen percent of all problems found during the APOLLO integration effort were due to performance specification errors [2].

to be an interface. The relationship between the variables at one node and the variables at another node are completely defined by HOS. The definition of that relationship makes it possible to provide a means to eliminate data conflicts.

Another aspect of an interface is timing, the property which considers the relative ordering of execution of subfunctions at a given level and the elapsed time taken to execute these functions. We do not explore the timing aspects in depth in this paper.

The designer is aided by the rule of node uniqueness and the theorems concerning self-control in determining the common boundary of function to function and variable to function. Similarly, the designer is aided by the ordering axiom in determining the absolute timing boundary by using the relative ordering per level, frequency of operation of a function, and the absolute time of execution of a function. The boundary here is the maximum delay time or completion time for a process. An operator (or function "name" without its variables) is not constrained by HOS; while operator to function boundaries are determined by the variables used to define the elements of the input set and the output set.

C. Specification Language Principles

The intent of an HOS specification language is to ensure that the properties of system performance can be completely separated from the properties of interface correctness. Such a language is not primarily concerned with computer execution, but rather with reliable system decomposition. We assume a computable system and, then, represent that system within the language. A system specification described by the language can be interpreted for computer execution. Whereas instructions written in lower level languages are statically checked by an assembler, and statements written in higher order languages (HOL's) are statically checked by a compiler; interface specifications written in an HOS language are statically checked by an analyzer.

The process of converting an HOS language from a specification level to the procedural[7] level can be performed by an automatic programmer. The automatic programmer is a definitional interpreter [8] that can apply the subfunctions of a node in the order that they are needed. This order is deterministic due to the constraints imposed by the axioms. The automatic programmer could be a part of the analyzer process for a given computer. If an intermediate language is desired for the implementation process, a translation from the automatic programmer to an existing compiler language can be developed.

Without an HOS language, the application of the HOS axioms is manual and requires considerable knowledge and experience for the system design process. The intent is to remove the manual process, and to incorporate into the axiom-dependent language constructs a means for expressing a given system. The constructs are chosen so that it is possible to check syntactically whether the application of a construct is consistent with the axioms. If each controller is represented by the application of a construct, it will be possible to ensure reliable interfaces of a given system by merely checking (either statically or at execution time) for proper application of that construct. This checking is done by a Design Analyzer statically and by a Structuring Executive Analyzer dynamically.

We can define a basic unit within and HOS structure, the nodal family, as one which describes a particular node and its immediate, and only its immediate, lower level nodes. Within the context of a language, we describe the nodal family by means of a nodal set. The nodal set is a set of operations which collectively define all the relationships between the members of a particular nodal family. The nodal family, characterized by the method of decomposition of the "offspring" nodes with respect to the parent node, is defined within categories of the nodal set referred to as construct classes. A construct class represents a function decomposition, the subfunctions of which can only be regrouped recursively. Fig. 2 illustrates two possible regroupings of the same function. Note that the regrouped subfunctions do not change in any way.

A function is decomposed by partition or composition. The subfunctions of a decomposed function can be decomposed in the same way until the most primitive level of decomposition is reached.

Within the partition class, constructs indicate decomposition of the parent function by division of the variables of the parental output set or elements of the parental input set. Characteristics of a partition that select[8] elements of the parental input set [Fig. 3(a) and 3(b)] are 1) each subfunction produces a value for every variable of the output set; 2) each input set of each subfunction is a proper subset of the parental input set (cf. Section III). Fig. 3 shows the control map representation of such a partition. Here, a variable representing a subset of the domain of x is described by referring the subset (indicated by the subscript) to the total set (the input set defined by the values of the variable x). For example, a variable $x_{\{x|x>0\}}$ represents an input set by the values of x greater than zero.

Language constructs can be provided to reference subsets of the elements of the parental input set. Some constructs of this type, such as ***if A then B else C***, exist today in HOL's. In fact, the latter construct is also a basic construct of structured programming. Unfortunately, the enforced language construct found in existing languages provides only the branching mechanism, but does not require expression A to provide a partition; nor does it require subfunctions B and C to assign a value to the same output variables. It is interesting to note that a language construct such as ***if A then B*** is invalid for HOS, since the partition is incomplete and, therefore, the decomposition of the parental node is incorrect. An HOS selection, however, in the form of ***if then else*** would not only require a partition, but that partition is restricted in that neither subfunction set of input elements is to be equal, else the controller module would have no criteria for choosing a subfunction.

Automatic programming of a partition which selects elements

[7]Sequencing requirements.

[8]A selection divides the elements, a , where $a \in A$, into a partition, B, of A such that $b \in B$ and $A = \cup_{i,i \in I} b_i$ and for two sets $b_j \cap b_i = \varphi$.

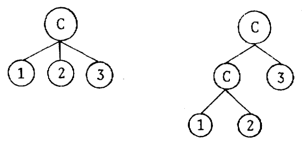


Fig. 2. Recursive regrouping of construct class, C.

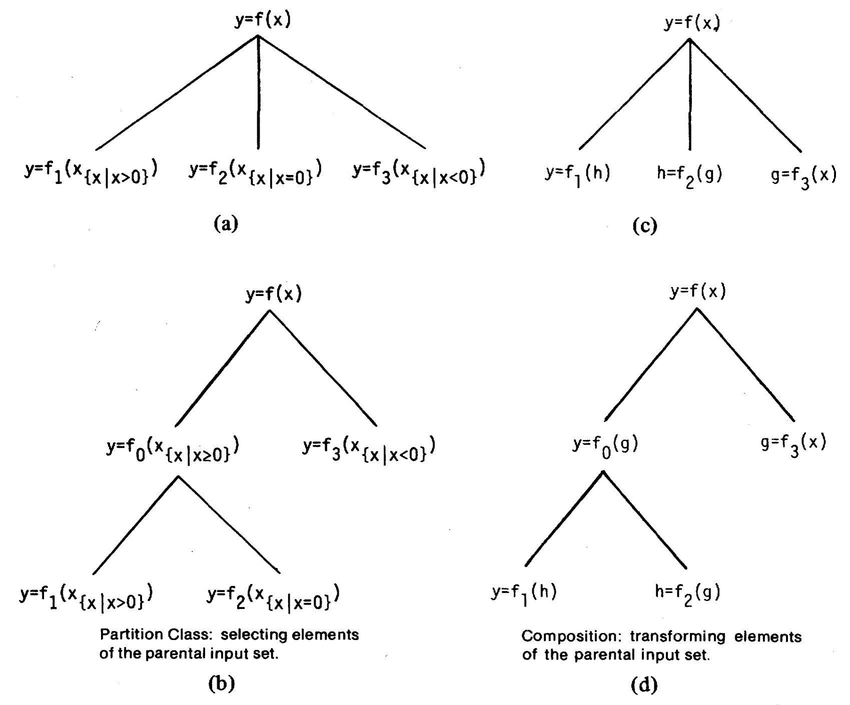


Fig. 3. Primitive control structures.

of the input set can optimize use of computer resources by using the fact that for each particular performance, only one subfunction will be executed. Since subfunctions of this type do not communicate with each other, these subfunctions could easily be dedicated, each to a separate computer. For a multiprocessor configuration, we might make use of the fact that once a subfunction is selected on a given level, other subfunctions on that level are no longer required.

A partition decomposition of another type is one in which the variables of the output set are divided (at least one for each subfunction). Here, all variables of the input set are distributed among the subfunctions. In this case, all subfunctions are performed for every element of the input set of the controller, and yet no two subfunctions assign the same output variable. Also, there is no communication between subfunctions, so that subfunctions of such a decomposition can be automatically sequenced for parallel processing. If performed in one computer, the flow of execution could proceed from one subfunction to the next, returning control to the controller when every subfunction has been performed. Some language constructs proposed for parallel processing, such as PARBEGIN,[9] could be used by the automatic programmer for procedural purposes if the simultaneous execution of subfunctions were analyzed as optimal for a particular nodal set of this type.

The decomposition class used to express communication on one level between one subfunction and one other is the composition class [Fig. 3(c) and 3(d)]. Here, variables of the input set of a function are all input to one subfunction. That subfunction transforms those variables to an intermediate set of variables. In turn, that intermediate set of output variables is referenced by one particular subfunction. This type of decomposition is best performed within one computer, since each subfunction communicates with another subfunction on the same level.

Composition of functions is enforced among HOS subfunctions by the requirement that variables of the input set of a function are not equal to variables of the output set of that same function. Also, since the controller explicitly specifies the subfunction relationships, the order of execution of the subfunctions can be determined independently from a particular sequence of language statements specifying these subfunctions. Many existing block structured languages have incorporated sequential branching mechanisms, such as the ***call***, to encourage use of composition techniques. Some languages prohibit the use of uncontrolled branching, i.e., no ***goto***'s to encourage composition techniques [9]. But, existing block structured languages encourage modules at a given level to invoke other modules at that same level, i.e., the architectural

[9] cf. Algol-68.

hierarchy of a program is not equivalent to the "control" hierarchy of a program.

It is important to note that certain traditional "software" functions are not computable from an interface correctness point of view. Such a noncomputable function (Fig. 4) shows an undetectable dynamic loop. In order to determine if a function is interface computable, an exercise is made to determine if an iterative mechanism can be transformed to a recursive formulation either manually or automatically. Fig. 4 represents a circular computation in that operator A, which directly invokes operator A, although seemingly recursive, is not, in that the function $y = A(x)$ is not computable. From these considerations the iterative construct, ***while A do B*** (as it exists in many programming languages) is invalid for HOS specification. The ***while*** is invalid for HOS specification because it can alter elements of its own input set and it can assign intermediate values to variables of its own output set. The property that a variable is accessed as an input variable and as an output variable of the same function is in direct violation of Theorem 3,4.1 [1]. Although loops are prohibited for HOS specification they will probably be used for implementation. The widely used form of ***while A do B*** could be constrained for system implementation by controlling such a construct with a ***for*** control mechanism and by use of a local variable protection mechanism for intermediate values. For an HOS specification we are restricted so that local initialization and consequent iteration is identified via a recursive formulation. Fig. 5 shows a recursive operator as a combination of primitive composition and partition control structures.

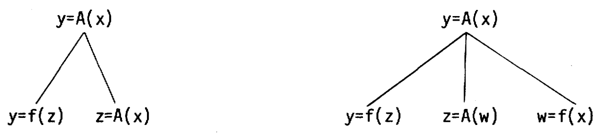


Fig. 4. Dynamic loop from an interface correctness viewpoint.

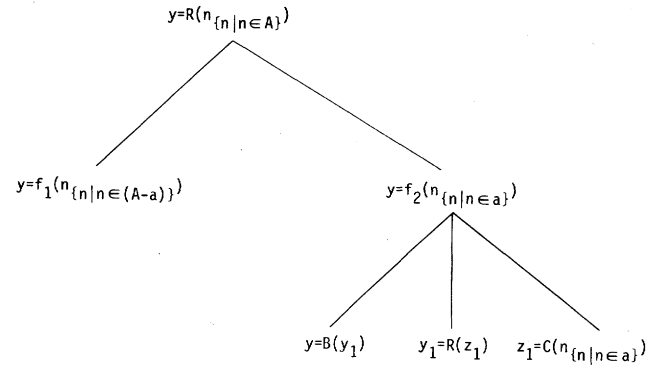

$y=R(n_{\{n|n\in A\}})$
$y=f_1(n_{\{n|n\in(A-a)\}})$
$y=f_2(n_{\{n|n\in a\}})$
$y=B(y_1)$
$y_1=R(z_1)$
$z_1=C(n_{\{n|n\in a\}})$

Fig. 5. Recursive operator (where $a \subset A$).

We consider here the use of the composition and partition control structures as applied to error detection and recovery specification in a real-time environment. A partition decomposition can be used for error detection and error recovery (Fig. 6). An error occurs when an output element is undefined, i.e., $y \mid y \in \varphi$ [Fig. 6(a)]. An error is detected when an input value is undefined, i.e., $x \mid x \in \varphi$. Error recovery implies that a subfunction is provided as an alternate function formulation for all values of the parental input set not acceptable to a primary formulation. For HOS, recovery from a detected error within a function is always provided for by its controller [Fig. 6(b)]. The nodal set for each nodal family can provide for: 1) alternate formulation of subfunctions in the event that a subfunction detects invalid input values, or 2) rejection of its own input set in the event that no alternate subfunction formulation exists. The nodal set for each nodal family always provides the ability to include a restart[10] mechanism for any subfunction because of the inherent structure of the system itself. This is true because a subfunction can never alter its inputs; every subfunction always has knowledge of its complete set of inputs; and inputs can be directly traced to one and only one other function. Thus, we may simplify hardware/software interface requirements in the case of a hardware failure: for in this case we could automatically, via the hardware, define a single node restart mechanism (i.e., HOS "single instruction" restart). With few exceptions [10], [11], most existing HOL's seem quite unstructured in their approach to such constructs, e.g., error scope is different than name scope, dynamic error scope allocation often exists with uncontrolled branch-on-error.

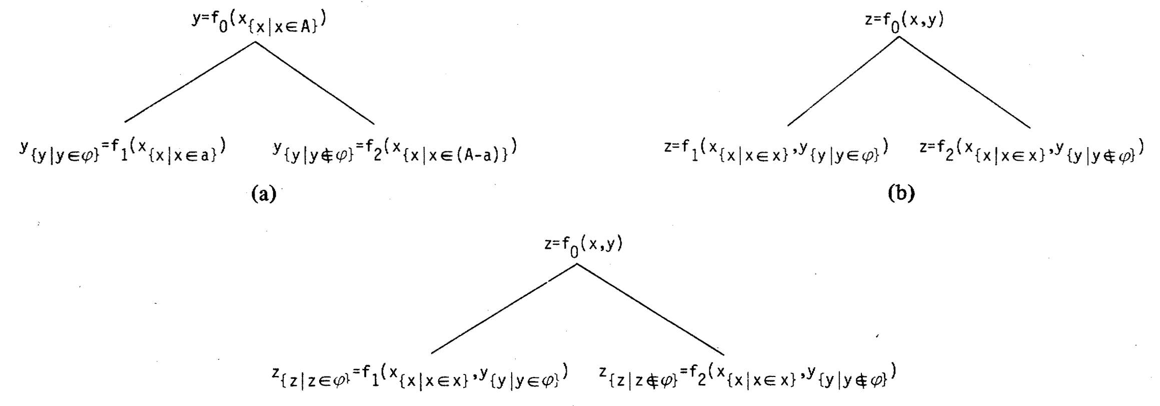
The integration (via composition) of error detection and recovery with error occurrence is demonstrated in Fig. 7. In Fig. 7(a) the structure of the system is completely specified in that each control relation and element appears on the formal control map.[11] The node represented by function f_2 provides error detection for variable y'. If y' is in error, i.e., $y' \mid y' \in \varphi$, then alternate A is invoked to provide a valid element for y. If y' is not in error, function f_6 assigns the valid y' element to y; where y is the true output variable for the controller node, represented by the function f_0.

Nodes f_3 and f_4 show the need for error handling procedures in that, here, only an indirect test on x (via testing w directly) is possible.

In such a manner, an abstract control structure (nonprimitive control structures) could be generated, e.g., ***P unless error then A*** [Fig. 7(b)]. We could look at the "implementation" of the abstract control structure to be the result of a transformation from the control structure to the structuring machine via a mechanism such as a compiler. For example, nodes f_1 and f_2 might be considered to be part of a compiler specification in that the compiler interface (e.g., necessary copies of "application" variables for restart purposes) could be extracted from the definition of f_1 and f_2. In Fig. 7(b), all control information is not seen on the representative control map; instead, the abstract control structure [whose formal specification is seen in Fig. 7(a)] and the representative control map are required to

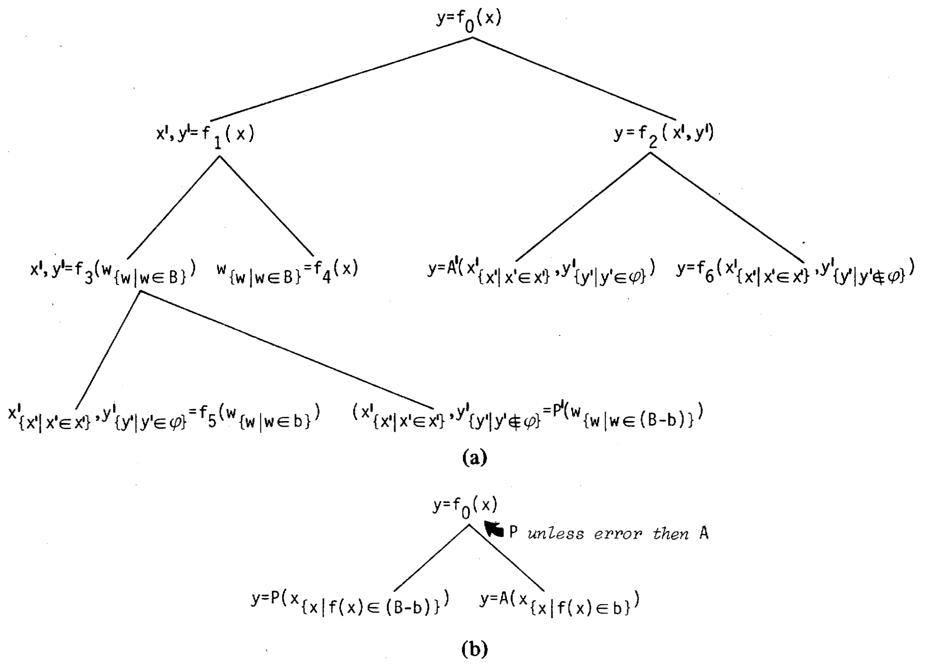
[10]Restart: when a process can be interrupted during its execution and can be arbitrarily reinitiated without any loss in the validity of its computation.

[11]The formal control map may be looked at as a "structuring machine in which no additional information is required to establish complete control of any system node other than that represented on the control map. The primitive control structures of composition and partition are used to generate a formal control map.



$y=f_0(x_{\{x|x\in A\}})$
$y_{\{y|y\in\varphi\}}=f_1(x_{\{x|x\in a\}})$
$y_{\{y|y\notin\varphi\}}=f_2(x_{\{x|x\in(A-a)\}})$
(a)
$z=f_0(x,y)$
$z=f_1(x_{\{x|x\in x\}},y_{\{y|y\in\varphi\}})$
$z=f_2(x_{\{x|x\in x\}},y_{\{y|y\notin\varphi\}})$
(b)
$z=f_0(x,y)$
$z_{\{z|z\in\varphi\}}=f_1(x_{\{x|x\in x\}},y_{\{y|y\in\varphi\}})$
$z_{\{z|z\notin\varphi\}}=f_2(x_{\{x|x\in x\}},y_{\{y|y\notin\varphi\}})$
(c)

Fig. 6. Mechanism for error handling. (a) Error occurrence via partition decomposition ($a \subset A$). (b) Error detection and recovery via partition decomposition. (c) Error detection and rejection via partition decomposition.



$y=f_0(x)$
$x',y'=f_1(x)$
$y=f_2(x',y')$
$x',y'=f_3(w_{\{w|w\in B\}})$
$w_{\{w|w\in B\}}=f_4(x)$
$y=A'(x'_{\{x'|x'\in x'\}},y'_{\{y'|y'\in\varphi\}})$
$y=f_6(x'_{\{x'|x'\in x'\}},y'_{\{y'|y'\notin\varphi\}})$
$x'_{\{x'|x'\in x'\}},y'_{\{y'|y'\in\varphi\}}=f_5(w_{\{w|w\in b\}})$
$(x'_{\{x'|x'\in x'\}},y'_{\{y'|y'\notin\varphi\}}=P'(w_{\{w|w\in(B-b)\}})$
(a)
$y=f_0(x)$
P *unless error then* A
$y=P(x_{\{x|f(x)\in(B-b)\}})$
$y=A(x_{\{x|f(x)\in b\}})$
(b)

Fig. 7. The relationship between error detection and error recovery ($b \subset B$). (a) Formal control map (where P represents the primary formulation and A represents the alternate formulation, i.e., $PV(\bar{P}\wedge A)$). (b) Representative control map with abstract control structures where P and A are partial functions of P' and A', respectively.

completely define control. An abstract control structure and its subfunction interfaces [such as seen in Fig. 7(b)] can then be used as the nodal set of a controller.

An implementation language construct for error recovery such as ***C unless error then B*** would imply that alternate function B is invoked for any error signal detected from C. The nodal set of B or C may also include error recovery or detection operations. In the event that an alternate function is not available, an error detection operator must account for assignment of the null value to output variables. A nodal set for such a formulation might be ***C unless error then reject***.

Error recovery philosophy within HOS does not include arbitrary procedures for system errors to replace invalid values on an interrupt-like basis to continue a calculation. For HOS, a system error signifies return to the controller only. Thus, an error, caused by a negative value submitted to a square root function, would never return a substitute value (such as zero, as is sometimes done).

In the case of some failures, one might want to recover the software by restarting a process. A possible nodal set for such a situation might be ***P unless error then refresh P*** where P and all of its dependents would be terminated on every "***error***" and then reinitiated (Fig. 8). Here the error occurs within P and recovery permits P to be restarted until P finally has a

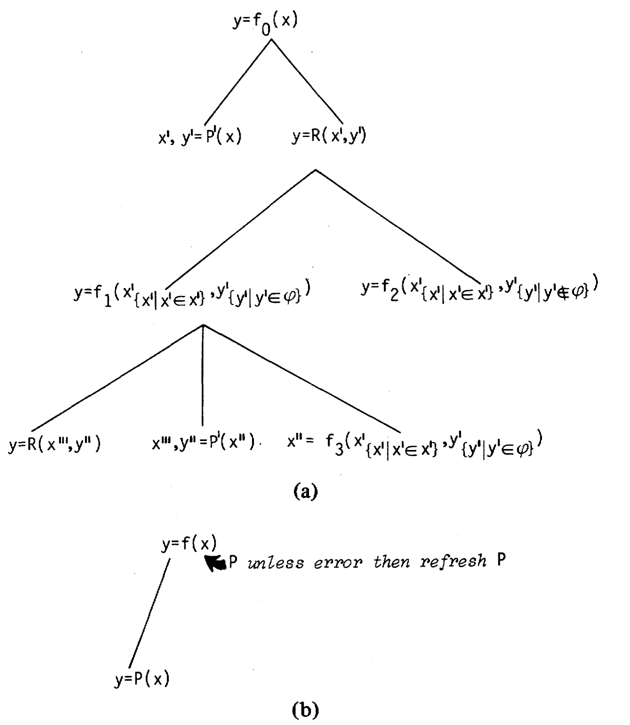

y=f0(x)
x', y'=P'(x)
y=R(x',y')
y=f1(x'{x'|x'∈x'}, y'{y'|y'∈φ})
y=f2(x'{x'|x'∈x'}, y'{y'|y'∉φ})
y=R(x''',y'')
x''',y''=P'(x'')
x''= f3(x'{x'|x'∈x'}, y'{y'|y'∈φ})
(a)
y=f(x)
P unless error then refresh P
y=P(x)
(b)

Fig. 8. Refresh: a recovery mechanism. (a) Formal control map (where P represents the reinitiated process). (b) Representative control map with abstract control structure [where $P(x)$ is a partial function of $P'(x)$].

proper value assigned to its output variables. Some errors occur outer to P, e.g., those caused by human interaction. In such a case, only the error detection and recovery, similar to Fig. 4(b), would be required.

In the event that more than one subset of the input set can be rejected, the nodal set for error recovery for such a subfunction might be constructed as: ***C unless error$_i$ then A$_i$***.

A reconfiguration (i.e., a partition in which the potential exists to execute more than one subfunction for a given performance of the controller) is required at the level at which the error is detected for every distinct error type. Thus, a new configuration would still be self-consistent with respect to interface correctness. This is especially significant for multiprogrammed structuring in that implicit errors in timing relationships, made invalid by a partial reconfiguration, are avoided.

A given system can be reconfigured in real time as a result of events. An event is a happenstance which if observed, is an indication to the system that a condition has occurred. On the other hand, an error is a special kind of an event in that, if it is observed, it is an indication to the system that something has gone wrong. Events (including errors) can occur from within a function or outer to a function. A keystroke indicating an action to be initiated would be an event outer to a function. If this keystroke were found to be incorrect, this "error condition" would indicate an action to be modified. In both cases, the event occurs outer to the function. Interprocess communication is an example of an event occurrence within a function. A parity in the hardware is an example of an error within a function.

Due to the functional elements of the HOS hierarchy, we are constrained to specify a termination condition beyond which the event is not valid and the function "***rejects***" to its controller. Here, we might use ***A on e reject after t***, where e is an event and t is the terminating condition. In such a way, human interaction can be formalized as events in a multiprogramming environment.

D. HOS Analyzers

The main function of an analyzer is to guarantee that a given hierarchical system is consistent with the axioms. Additional analyzer phases complement the main function so as to efficiently implement a given system for a computer. Automatic interface analysis is provided on a static basis (without source code execution) by the Design Analyzer and on a dynamic basis (during real time) by the Structuring Executive Analyzer.

Design Analyzer: Real-time software systems cannot be exhaustively tested. The intent of the Design Analyzer is to exhaustively and statically verify a given software system, defined according to the rules of HOS, for interface correctness. Interface errors in multiprogramming or multiprocessor systems are caused by data or timing conflicts. Given the HOS control system, it is possible to not only design a system with a known and small finite number of logical interfaces to verify, but to prevent both data and timing conflicts. Thus, with the Design Analyzer, the more expensive methods of simulation and/or dynamic verification can be limited to unit performance testing.

There are four phases to the Design Analyzer presently in development. Phase 1 will check a prototype software system design for interface correctness. In addition, a software system control map, showing all functional relationships of the prototype system, will be automatically produced. Phase 2 will determine where performance testing is necessary. This will distinguish between nodes that require proof of performance techniques for self-contained nodes versus those that require additional simulation techniques for nodes that interface with external systems. Phase 3 will provide functional timing and memory analysis for any given level of a system. This will either determine if a particular software system design is able to meet the constraints of "off-the-shelf" hardware or will determine hardware capability that is necessary to support a given software system. Phase 4 will supply an automatic programmer to either convert from the HOS language to an intermediate language or directly for computer execution. The output from this phase could include information for the efficient use of hardware and information for the next most immediate lower architectural virtual layer.

If HOS methodology is maintained throughout a development process, it is envisioned that functions and their relationships will be able to be compared from analyzer results at each software phase of development.

Structuring Executive Analyzer: The Structuring Executive Analyzer is a lower virtual layer module with respect to a given hierarchical HOS system in its dynamic state. We intend the structuring executive to implement, in real time, multiprogramming control constructs. The operating system requirements of the structuring executive are to handle aspects such as man/machine interface, hardware/software interface, error detection and recovery, real-time reconfiguration, dynamic allocation, analysis for timing and memory limitations, and an axiomatic analysis of the system in real time.

The real-time implementation of control constructs assumes: 1) an invocation mechanism, the ***schedule***, to be used to invoke a module dependent on time, relative priority, event or frequency; and 2) a restart mechanism, the ***refresh***, to cancel an active node and all of its dependents and then to reinitiate that node with or without a new element of the input set.

Fig. 9 shows a hierarchical HOS system example in which processes are invoked via a ***schedule***. Selected nodes can be reinitiated when an event e_i where $i \in I$ and $e_i \subset E$, occurs outer to process f. Event e_1 may occur dependent on a man/machine interface selection or a system error. If node A_2 is reselected, A_2 and all of its dependents are terminated and then reinstated, possibly with a new element of the input set available so that a new option could be selected. In this case A_3 would be reinitiated but A_1 would not be reconfigured. The man/machine interface aspects of the structuring executive allow for the human to interact with a given system at any node. If sequences selected are not compatible, the Structuring Executive Analyzer would detect an axiomatic error and automatically recover the system via a ***refresh*** mechanism.

In addition, the human can select any reconfiguration of nodes in real time without concern for introducing errors into the system. Thus, the man/machine interfaces with each HOS system are multileveled, multilayered, and can be multiconfigured. For example, an avionics pilot would not need to memorize the order of a complicated crew selection list, since the software would provide automatic error detection and recovery.

Analysis can be provided at required nodes by the executive, since the capability exists to provide timing and memory limit requirements for selected nodal families in advance. For example, if the throughput of a function is larger than a given limit within a specific length of time, a ***refresh*** could be used to restart or selectively restart a function with an option to include only the higher priority functions.

The structuring executive can provide reconfiguration in real time by a reordering of priorities based on instantaneous human or hardware inputs to a system. In the restructuring process, the structuring executive always maintains the relative timing relationships for each nodal family (and thus for a complete system) based on the fixed relative ordering relationships defined for each nodal set.

Both the Design Analyzer and the Structuring Executive Analyzer perform equivalent functions. 1) Whereas the Design Analyzer checks for interface errors statically, the Structuring Executive Analyzer ensures that the correct interfaces are maintained during real time. 2) Whereas the Design Analyzer determines where performance testing is necessary, the Structuring Executive Analyzer determines where human intervention is necessary. 3) Whereas the Design Analyzer predicts a potential resource problem, the Structuring Executive Analyzer detects and recovers from an actual resource problem. 4) Whereas the Design Analyzer provides an automatic programmer, the Structuring Executive Analyzer allows a reconfiguration of nodal families in real time.

III. System Design

Use of HOS implies that the designer no longer need be concerned with problems of interface correctness. Instead, more attention can be focused on the problem of correct performance specification.

Once a problem has been properly decomposed and automatically analyzed for interface correctness, a system implementation can be automatically optimized for various cost criteria where specification constraints can be minimized or determined. For example, we could minimize verification paths for performance analysis, minimize storage requirements, maximize the number of tasks performed per unit time, or determine cost-effective hardware requirements.

Although some specific constraints have been eliminated by use of existing language constructs [9], [12], the application of HOS implies that a restructuring of an entire system implementation can be done automatically.

Examples of automatic structuring requirements collectively provided for by HOS for system design are: 1) controlled branching (no undetected loops); 2) redefinition of variable domain; 3) order independent language statements; 4) controlled error recovery environment; and 5) enforcement of local initialization. Traditional software structures often prevent or discourage modification of a system for fear of misinterpreting hidden implicit specifications. For example, it is important not only that programs be written in a sequential manner [13], but that the order of execution of the language statements can be determined regardless of that sequence.

Example 1:

$$a\colon\ x = y \tag{1}$$

$$\textit{if}\ \ z > 10,\ \ x = y^2. \tag{2}$$

Here, the value of x is sequence dependent, yet the program can be written without branching (i.e., no ***goto***'s). This formulation would be rejected by a language analyzer. Potential modification to the above specification must consider the value of x to always be dependent on the value of z even though z does not appear in (1).

$$b\colon\ y = f(x) \tag{1}$$

$$x = f(z). \tag{2}$$

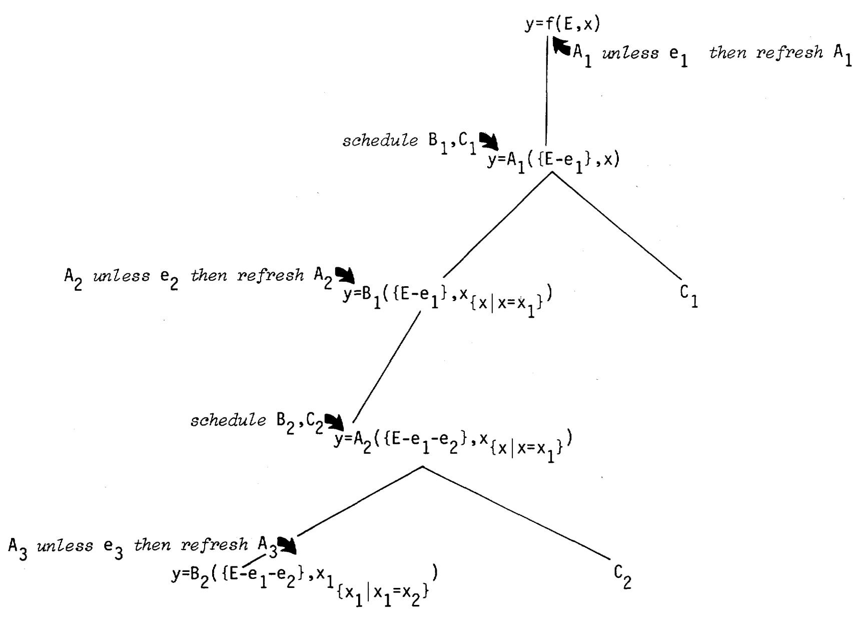
Here, it is readily determined that (2) must be executed before (1). Thus, for the HOS specification, the formulation is correct even if (2) appears before (1).

Example 2:

a: ***schedule B*** (1)

schedule A on not B. (2)

Here, the intent as to the order of execution of A and B is ambiguous; therefore, the initialization of process A and B are sequence dependent. If statement (2) was executed before



y=f(E,x)
A1 unless e1 then refresh A1
schedule B1,C1
y=A1({E-e1},x)
A2 unless e2 then refresh A2
y=B1({E-e1},x{x|x=x1})
C1
schedule B2,C2
y=A2({E-e1-e2},x{x|x=x1})
A3 unless e3 then refresh A3
y=B2({E-e1-e2},x1{x1|x1=x2})
C2

Fig. 9. Refresh mechanism and its relationship to man/machine interface.

(1), process A would be initiated before process B. If process A was initiated before the completion of B, statement (2) is violated in real time. The analyzer would reject this formulation. Potential modification to the above formulation must consider the potential timing interface problem.

b: **schedule A** (1)

schedule A after B. (2)

The interpretation here is order independent. B will complete execution and then A will be initiated. Thus, this formulation is valid for an HOS specification.

An interesting observation of the effects of explicit requirements for HOS is illustrated by use of the partition class for problem specification. Suppose we wish to limit the values of a variable and specify a formulation to assign that limited subset of values to a given output variable. We shall show that it is not possible to camouflage this specification within HOS and how a traditional interpretation of a simple problem causes interface specification errors. Suppose we represent the specification by a simple partition of input elements:

$$y = \textit{if} \quad x > 10\, h(x) \quad \textit{or} \quad g(x).$$

The control structure for this partition implies that the controller is controlling its own function because the entire function is of the form $y = f(x)$ (Fig. 16). In HOS, the partition of input elements is affected by a controller restricting the domain of each of its subfunctions. This is done because the specification of a subfunction must have only the information it needs to constrain the implementation. This specification can be implemented by a simple ***if then else*** statement.

From Axiom 1, a module cannot invoke a function that performs the same function as the function at the node of said module. Two functions, each defined by the same sets of variables and producing the same mapping, are equal functions.

Immediate Self-Control Theorem:[12] *Two functions cannot exist such that each is defined by the same sets of variables and one of said functions is a subfunction of the other. For if this were the case, we can show the controller function is controlling itself.*

Note that the partition of input elements (Fig. 10) is not clearly specified since it appears as though each subfunction may use all elements of x. In fact, subsequent decomposition of $y = g(x)$ or $y = h(x)$ could lead to extraneous logic since not all values of x are valid; or more seriously, functions h and g could be inadvertently switched by an engineer unfamiliar with the original problem because the interfaces appear identical. In addition, more performance analysis than necessary would probably be attempted for verification of that subfunction, since it is entirely conceivable that an unknowing engineer may attempt to verify even those cases for x that are nonexistent to the subfunction. If input variables are not properly partitioned, it would be impossible to automatically restructure a system because it would be unclear as to which nodes could be interchanged. We try to reformulate the specification by assigning x to variable z as follows:

$$z = x$$

$$y = \textit{if} \quad z > 10\, h(x) \quad \textit{or} \quad g(x).$$

Here, the control structure (Fig. 11) shows, likewise, that this formulation of the entire function $y = f(x)$ controls itself.

Indirect Self-Control Theorem: *If a function is defined, a node defined within the tree of said function cannot be defined so that the sets of input variables are equal and the sets of output variables are equal. Again, we can show that if this*

[12] See Appendix II for proof of this theorem and subsequent theorems stated in this paper.

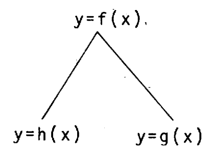


Fig. 10. Immediate self-control.

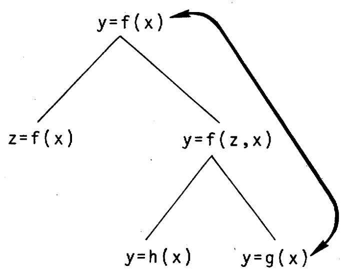


Fig. 11. Indirect self-control.

condition were to exist, the controller function is controlling itself.

In each case, when the subset of values is represented by all of x, we fail to follow the axioms. Thus, it is required to rename the subset of x to properly specify the problem. We avoid later ambiguities as to the proper values to be used within the subfunction:

$$y = h(x_{\{x \mid x > 10\}}) \;\; \textit{or} \;\; g(x_{\{x \mid x \leqslant 10\}}).$$

Further decomposition of $y = h(x_{\{x \mid x > 10\}})$ would never refer to all of x.

The above discussion illustrates the types of problems that one can encounter by the unconstrained use of a simple ***if then else*** statement common to many existing HOL's. Imagine the potential difficulties that one can encounter with more complicated statements if there is no formalized structure within which to work.

IV. Software Management

A. Background

In the past, we used two major methods to determine those tools necessary for future software development. Statistical analysis of software efforts was performed [2], and checklists for manual software disciplines [14] were determined from both APOLLO and SKYLAB on-board flight software efforts. We attempted to provide an ongoing statistical analysis of the Shuttle on-board guidance, navigation, and vehicle control software requirements integration effort [15], [16]. We discovered that in order for such an effort to be successful, the process of collecting anomaly and other statistics histories should be automated as much as possible. In addition, software management of all people and organizations involved must be convinced that such an exercise is a necessary one.

The APOLLO study was of great value for determining the direction we were to follow for future software efforts. For example, the fact that 44-percent of all the anomalies in the software were found by "eyeballing" [2], prompted us to draw a few subjective conclusions based on background and experience. Of course, the problems of analyzing statistics are well known. For example, if there were a system designed such that the only debugging aid was eyeballing, there would obviously be a huge percent of discovered problems found by eyeballing—to be exact . . . 100-percent! And, the number of problems found will always be less than the total number of existing problems in a given system. But, when we analyzed APOLLO on-board software data, we considered several significant factors: 1) the software management structure forced eyeballing to be a key part of the verification process [14]; 2) the most sophisticated dynamic verification debugging process known at the time of APOLLO was used to verify software [14][13]; 3) there were no pure software problems found in the on-board flight software for all APOLLO flights [17]. In addition, we found that flexible software systems are a key to managing software developments. A case in point is the APOLLO on-board asynchronous systems software. If the APOLLO Guidance Computer (AGC) systems software had not been asynchronous, the development process would have been much more expensive, much longer, and at least one of the APOLLO flights would have been a disaster [18].

The systems software of the on-board flight software for APOLLO consisted of a powerful set of asynchronous modules: the scheduler [19], the display interface [20], and the restart modules [21]. Although the development of these modules began in the early sixties, the first time they were exercised together on an actual flight was for APOLLO 7 (1966).

The scheduler was able to schedule modules based on priority or time. The display interface module was not only asynchronous but was also multileveled and multilayered. That is, the astronaut was able to interface interactively with the on-board software at the top level of a msision phase as well as at the lowest level of a mission module in a hierarchical layout of a mission sequence. In addition, the program was able to override a current display with one of higher priority. The restart module was able to store information for all the modules in a multiprogramming environment at any given instant in real time for any level in the hierarchical system. If an error occurred, all modules could be recovered or selectively recovered depending on job or time priority.

In addition, an AGC interpreter [22] was used to program applications algorithms with higher order operations such as matrix-vector arithmetic.

All of these system software modules had common attributes: even though all users could implement control via these modules, each system module was self-contained and its inner contents were hidden from the user [23], [24]; each module was designed and frozen early [25]; no changes to the re-

[13]The APOLLO all digital simulator simulated the AGC bit-by-bit and the environment (including the astronaut, universe, spacecraft, and sensors). In particular, the simulator had a very large set of debugging tools which provided prerun consistency checks; selective and repeatable snapshot/rollback capability; environment updates which could advance time during nonproductive mission sequences (thus saving hours of simulation time); monitors for dynamic runs (e.g., abort conditions for flight program simulations). In addition, the simulator included a test input language for a manual astronaut and test set-ups, and post editing tools for summarizing selective or complete mission sequences.

quirements of these modules were incorporated after they were frozen. That is, each of these systems software modules was truly on a lower virtual layer [14] than the applications modules.

When we analyzed the anomalies found during the integration effort, there were no problems discovered in, or communicating with, any of these systems modules. That is, all the interface problems found during the integration efforts[14] were a result of one applications module interfacing with another applications module.

Once the initial statistical analysis was performed, we further analyzed and categorized the anomalies and then determined rules which would have prevented each class of anomaly. We also refined the checklists for all disciplines of software, including the design, implementation, verification, documentation, and management phases of software [25]. We found out to our surprise that the refined manual checklists were identical for each discipline. We categorized these checklist items in order to decide those manual processes which could be automated for the software development process. As a result, we found that many of the manual processes, as well as the more conventional tools needed to assist these manual processes, could become obsolete. For example, now that logical testing can be provided statically and automatically by the analyzer, there is no longer a need for a manual logical verification checklist [25]. In addition, there will no longer be a need to dynamically exercise the software for interface correctness. From these efforts evolved the formalized system HOS. With a formalized system, it is possible to formally manage software systems as well as to automate many of the tasks of system management.

B. Modularity as Defined by HOS

Each controller in an HOS system establishes the communication[15] network among its subfunctions. This means that functions on the same level cannot *control* the communication with each other, but they can communicate with each other. In fact, all functions that are specified to communicate with each other must exist on the same level. Thus, it is possible for management to determine immediately, as well as automatically, from a management control map (which is equivalent to the software control map), those functions which are dependent on each other.

The invocation (Axiom 1), access (Axioms 3 and 4), element (Axioms 2 and 5), and ordering (Axiom 6) control relationships can be determined for any given module by knowing only the relationships among members of its own nodal family. From the control relationships of a module with respect to its nodal family, hardware memory requirements, absolute and relative timing requirements, throughput and transput can be predicted and, later, actually determined for every node at any level of a given system. If, for example, partitioning studies are necessary for multiprocessor configurations, a determination can be made on a level by level basis where it is feasible to partition for different processors, where it is feasible to perform parallel processing, or where timing constraints require functions to be dedicated to the same processor.

To understand a management scheme for HOS, it is necessary to clearly distinguish functions, modules, operators, and variables and to show the relationship of each of these units to a general library concept. A function is defined by the variables of the input set, the variables of the output set, and the set of operations of the nodal set. When the nodal set is shared by more than one controller, i.e., that nodal set exists as a member of a library, that nodal set is considered to be a module.

In this respect, we consider the library module to be equivalent to an operator: to function, we require the operands, i.e., variables of the input set and variables of the output set. As an operator, the module can reside only once in the computer and yet appear in many places on the system control map. As an operator, the module is self-contained in that it can effect a mapping and yet never affect the system: it can never control access to outer system data, can never effect the invocation of outer system modules, and can never effect the ordering of outer system functions.

Once a nodal family is verified as a module, the module, as an operator, is placed into a general library. These "frozen" operators can be collected by various controllers to form a new system [25]. When an operator is controlled, variables of the input set and variables of the output set are specified, and that newly formed function becomes a unique node of a system.

Uniqueness Theorem: *Each node of an HOS hierarchy is unique, i.e., two functions cannot exist within the same hierarchy such that the same relationship exists between input and output elements and the same sets of input variables and output variables are defined for each function.*

But $y = f(x)$ as a unique node does not imply that variable y, variable x, or operator f cannot appear individually at other nodes of the control system. For example, variable y must appear on the next most immediate lower level if $y = f(x)$ is decomposed at all.

Since a module controls the creation of its own mapping, an operator can appear at more than one node of a control system. For example, the process of recursive formulation of a function $y = R(n)$ (cf. Fig. 5) for HOS is essentially a combination of partition and composition in which the function, representing one subset of input elements, invokes the operator of the original module; the function representing the other subset of input elements is invoked after the last recursive step. Thus, operators can indirectly invoke themselves as in the case of operator R. Also, operators can be invoked from more than one path in a system as in the case of operator B.

Since each node is unique when it is controlled and each nodal set is a self-contained module when it is not controlled, we could form a library of nodal sets each describing a nodal family. The operators of each nodal set would appear as other library modules. The hierarchical relationships for a system where the nodal sets all appear in the same library could be determined automatically. Thus, with a library of this type, we could provide a mechanism to automatically restructure

[14] Seventy-three percent of all problems found during the APOLLO integration effort were interface problems [2].

[15] Communication: the specification for the transfer of elements between functions.

any HOS system. This means we can collect any set of nodal families to form a new system.

With this approach, many systems can be collected for different missions or different mission configurations. The same library can be used by all engineers in a group or by several organizations on a project. Throughout the development of a project, the functions and interfaces are maintained. With an automatic programmer or a translator, this process can be provided for automatically.

At the functional level, the modules are machine and implementation independent. At this stage, commonality and transferability among users can be maintained. Modules can be used over and over again. Changes can be made to the system on a module by module basis. The affect of the changes can be tracked automatically by means of the formalized definition of all the interface paths. With such knowledge, cost of changes as well as the life-cycle cost of a given system can be predicted. The complete knowledge of a module and its interfaces can also be used to measure programmer performance as well as to predict the size of future efforts more accurately.

C. Frozen Module Management

Given today's technology, it is recommended that each software module go through three phases of development [25]. The first phase defines the functions and their interfaces. The second phase defines the architectural characteristics of the system (hardware dependent, implementation language dependent, and resident software dependent). This phase maintains the functional relationships of the first phase. The third phase defines the actual execution code for the implementation of the previous phase.

Each "frozen" module is accompanied by automatic documentation. If a revision is made to a module, the documentation is also automatically updated. The categories of recommended documentation are: 1) Automatic structured design diagrams which show an ordered sequence of program flow. For the Space Shuttle requirements integration effort, this automated tool (designed [26] and developed [27] at The Charles Stark Draper Laboratory, Inc., Cambridge, MA) has replaced the manual production of flowcharts.[16] 2) Automatic control maps which show the functions and their relationships (this includes both the data flow and the operators for each function). 3) Automatic narratives in which variables and key words can be updated with program changes.

Each module evolves from one development phase to the next. The documentation for each module is also an evolvement from one development phase to the next. In this way, there is never a new development effort for a new phase, but always a continuation of the previous phase. The documentation is never obsolete, but is always automatically produced for each new revision of the module.

Someday, the only phase that should be required for software development will be Phase 1. Phase 2 will not be necessary, since implementation mechanisms such as hardware will be flexible enough to accommodate functional requirements. There will not be a need for an intermediate HOL, and we will no longer have to live with resident software which requires us to alter the requirements. Finally, Phase 3 will no longer be necessary because code will be produced automatically from Phase 1 requirements.

[16]Twenty out of 100 people were totally dedicated to the manual production of flowcharts in the Computer Science Division (formerly the Program Development and Verification Division) of The Charles Stark Draper Laboratory, Inc., during APOLLO.

D. Assembly Control Supervisor

The organization of personnel involved in a software development process can correspond on a one-to-one basis with a given HOS control structure. For example (Fig. 12), the top level software system S corresponds to the Assembly Control Supervisor (ACS). The ACS manages the official assembly of software module S by monitoring the work of managers ACS_1 and ACS_2. The lower level software modules S_1 and S_2 must be given official approval before being allowed into the official assembly controlled by the ACS.[17]

Likewise, ACS_1 manages the official assembly by monitoring the activities of managers ACS_{11} and ACS_{21}. This management assignment process can be nested as deeply as desired.

It is very important to note that the software control structure is equivalent to the management control structure: managers at the same level do not determine interfaces with each other, but they do communicate via channels set up by the most immediate higher level manager.

This method can, in addition, complement other forms of software personnel management [28]-[30]. For example, if the chief programmer concept were to be used for personnel management, the ACS, ACS_1, ACS_2, and ACS_{21} would be the same person. However, the assembly control management system allows for additional managerial flexibility: 1) the ACS approach can be used for a small system or a very large system; 2) a project is never forced to be dependent on one individual; 3) modules can be developed off-line at any level and can be called into any official library providing an HOS structure is maintained; 4) a manager is always able to track modules developed under her/his supervision; 5) a manager can control all decisions related to internal module development without affecting other managers or modules; 6) changes can be made reliably and efficiently, since improper interfaces can be detected by the Design Analyzer; 7) system modification (personnel or software) can be tracked automatically during the entire development effort since every node in the system has a unique identification; 8) the management structure shows the relationship between groups and organizations within the system structure even though an organization acts as an "operator" and physically resides in another geographical location; 9) modules can be developed in parallel.

E. Reliability Versus Efficiency

There is almost general agreement that structured techniques are better than nonstructured ones. It is also generally agreed that a formalized method of structuring systems is necessary since every programmer or organization not only has different ideas of what a structured system is, but also has a need to measure the correctness of a given system structure.

[17]Top-down/bottom-up official building process (cf. [26]).

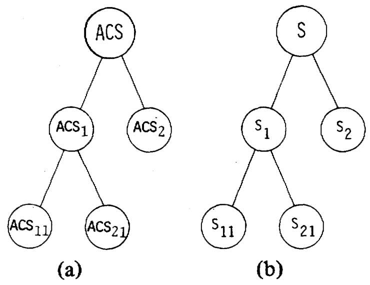


Fig. 12. Management structure and its one-to-one correspondence to the software structure. (a) Management structure. (b) Software structure.

Since structured programming concepts were first introduced [13], questions concerning timing and memory efficiency have been raised. The same questions are brought up again for HOS. Contrary to what we expected from initial studies, the process of structuring an algorithm with HOS would often be more efficient than with conventional methods. This was no doubt true because complete attention was given to the process of structuring.

It is also true, however, that one can look at a system manually in which the structuring is correct and more easily change the design to be more efficient. Of course, the reliability may be compromised if this process is necessary.

The recommended procedure is to always structure at the functional level using the axioms. The system should be designed as if it were to communicate with an asynchronous executive (one which allows hardware or software interrupts), especially if the system is intended for real-time application. This procedure allows for functions to be described in a natural state; whereas with a synchronous approach, the functions are divided into unnatural time slots that are implementation dependent and, of necessity, become interrelated to other system functions.

During the initial design process the functions and their interfaces are determined, which include timing and memory requirements for future implementation. At this time, it is possible, therefore, to either define hardware requirements for computation or to determine if the system can be built with "off-the-shelf" hardware [31].

If the existing hardware is not adequate, it is much easier and more reliable to delete capability and tune up for efficiency within a formalized structure than it is with a conventional system with no formalized structure. It is hoped that someday solutions to efficiency problems will be more powerful hardware, rather than a compromise of software reliability.

V. Results

HOS has been applied manually on two major projects. On the Shuttle Flight Software Requirements Integration effort at The Charles Stark Draper Laboratory, Inc., we partially applied the axioms to several algorithms at the functional level. On the DAIS avionics project the software requirements (an integrated avionics system) [32], [33], link editor (intermodule consistency checker), and the operating system (federated multiprocessor) [34] have been defined with HOS techniques. The Shuttle flight software is real time, and multiprogrammed. The DAIS software will be resident in a multiprocessor environment.

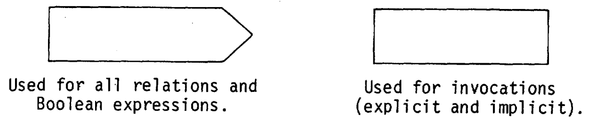

Used for all relations and Boolean expressions.
Used for invocations (explicit and implicit).

Fig. 13. Design diagram conventions.

We have found that the use of HOS for system design clarifies the intent of the designer and simplifies problems originally approached in a traditional manner. In particular, changes were much easier to make to those algorithms designed with the axiomatic approach, and these algorithms were much safer to integrate. Basically, the key differences between the HOS algorithms and those designed with conventional approaches are that the HOS modules never have knowledge of their users (i.e., ***callers, schedulers,*** etc.). They simply, as does an ***add*** instruction or a ***sum*** routine, perform a function for a user. Thus, we end up with a system of "instruction-like" modules. Examples of such instructions in an avionics system might be ***navigation, guidance,*** or ***vehicle control***.

Certain design aid tools in support of the basic HOS methodology are currently operational. For example, our automatic design diagrams (structured flowcharter and data intersection analysis tools) are now being used by several organizations involved in the Shuttle project [35]-[38]. Structured design diagrams are produced automatically from HAL source code. Programmers are able to obtain this output by simply inserting an extra job control command at the program compilation step. Prior to the automation of the structured design diagrams, programmers and engineers produced design diagrams manually for the purpose of structuring an algorithm before it was coded. Now, the automatically produced design diagram is used as a comparison with the original manually produced design and as a means of producing up-to-date and automatic documentation of the computer program.

Fig. 13 illustrates the two conventions used in generating design diagrams. Lines connect nested decision levels and linear execution flow. Execution flow is assumed to return in line at the completion of every object of every decision.

Table I shows simple examples of the design diagrams for equivalent source code. In addition, the equivalent control map is also shown.

We are presently involved in building a prototype HOS Analyzer.[18] For this prototype, we are translating HAL language [39] source code to an HOS control description. The control description is the language independent input to the analyzer (Fig. 14). From this research effort, we found that a specification language in which HOS functions could be formulated directly would have simplified the analyzer task. With an HOS specification language, we could have formalized a given problem and then translated that specification to HAL. Without such an aid, we have defined a particular grouping of statements to represent an HOS function. Each

[18]Internal Charles Stark Draper Laboratory Independent Research and Development Project "Automated Computable Systems."

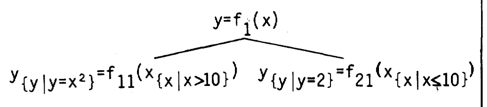
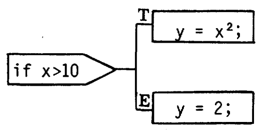
TABLE I
PROTOTYPE ANALYZER EXAMPLES

CONTROL MAP REPRESENTATION	STRUCTURED DESIGN DIAGRAM	HAL SOURCE CODE
$\vec{V}_{\vec{V}\,|\,\vec{V}=\overset{*}{M}^{-1}\vec{V}/S} = F(\overset{*}{M},\vec{V},S)$	$\boxed{\vec{V} = \overset{*}{M}^{-1}\vec{V}/S;}$	$\vec{V} = \overset{*}{M}^{-1}\vec{V}/S;$

COMMENTS

(1) An assignment statement.

(2) A matrix data type ($\overset{*}{M}$) vector data type ($\vec{V}$) and scalar data type (S) and the availability of the corresponding set of operations for these data types are assumed.

 $y=f_1(x)$
$y_{\{y\|y=x^2\}}=f_{11}(x_{\{x\|x>10\}})$
$y_{\{y\|y=2\}}=f_{21}(x_{\{x\|x\leq10\}})$
		if x>10 then y = x^2;
		else y = 2;

COMMENTS

(1) If f_{21} were modified so as to contain the relation x>20, the analyzer would detect an extraneous path.

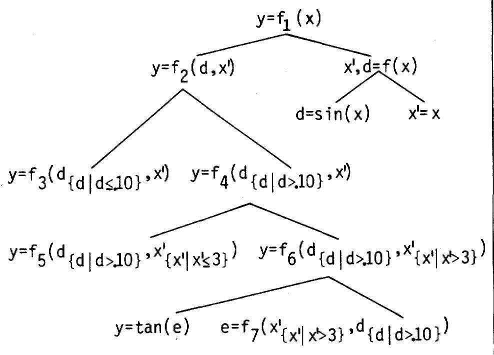
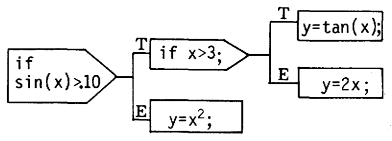
(2) Suppose the THEN branch assigns y as shown, but the ELSE branch were to assign a variable other than y. In this case, the analyzer would detect an error.

(3) If f_{21} were modified so as to assign a value to x, the analyzer would detect an error.

(4) If f_{11} were modified to CALL f_{21}, the analyzer would detect an error.

(5) A constant function is specified by restricting the elements of the output set (cf. node f_{21}).

CONTROL MAP REPRESENTATION	STRUCTURED DESIGN DIAGRAM	HAL SOURCE CODE

 $y=f_1(x)$
$y=f_2(d,x')$
$x',d=f(x)$
$d=\sin(x)$
$x'=x$
$y=f_3(d_{\{d\|d\leq10\}},x')$
$y=f_4(d_{\{d\|d>10\}},x')$
$y=f_5(d_{\{d\|d>10\}},x'_{\{x'\|x'\leq3\}})$
$y=f_6(d_{\{d\|d>10\}},x'_{\{x'\|x'>3\}})$
$y=\tan(e)$
$e=f_7(x'_{\{x'\|x'>3\}},d_{\{d\|d>10\}})$
	 if sin(x)>10
	T
	if x>3;
	T
	y=tan(x);
	E
	y=2x;
	E
	y=x²;
		if sin x>10
		then if x>3
		do;
		then y = tan(x);
		else y = 2x;
		end;
		else y = x^2;

COMMENTS

(1) Three nested levels of logic flow are shown on the design diagram.

(2) Five levels of data flow are shown on the control map.

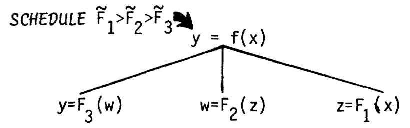
(3) Each function must appear as a node. A relation determines the partitioning of the elements of the input set. If an extraneous path exists due to an inconsistency in the specification itself, the analyzer will not detect such a path. For example, suppose sin(x)>10 were modified to be 2x<3. In this case the analyzer would not detect the fact that f_6 will never be executed. On the other hand, all logical inconsistencies among nested relations are checked for extraneous paths. For example, suppose sin(x)>10 were modified to be x<2. Here, f_6 would be shown to be logically inconsistent by the analyzer. It is not clear that the removal of all such paths is the best way to completely design a system. But, it is clear that if we know where these paths exist, we can explicitly make a design decision as to whether to remove such a path or not.

(4) The use of composition referred to by node f_7 is key to the use of library modules.

(5) e is a local variable.

TABLE I (CONTINUED)

CONTROL MAP REPRESENTATION	STRUCTURED DESIGN DIAGRAM	HAL SOURCE CODE

 SCHEDULE $\tilde{F}_1>\tilde{F}_2>\tilde{F}_3$
y = f(x)
$y=F_3(w)$
$w=F_2(z)$
$z=F_1(x)$
	CALL Designate_Priority(F) Assign P;
	Schedule F_1 at priority P_1;
	Schedule F_2 at priority P_2;
	Schedule F_3 at priority P_3;
		do;
		CALL Designate Priority (F) Assign P;
		Schedule F_1 at priority P_1;
		Schedule F_2 at priority P_2;
		Schedule F_3 at priority P_3;
		end;

COMMENTS

(1) This set of HAL statements implements the invocation of processes at a given level so that the priorities of the processes are ordered and are relatively lower than the controller process.

(2) F is an array of name variables. P is an array of numbers. The values assigned to P are ordered from highest to lowest. The assignment of values (rather than a relational form of priority) is necessary because of the restrictions of HAL.

(3) The analyzer assumes Designate Priority to be a lower virtual layer function, i.e., Designate Priority does not appear on the control map.

(4) In this case, the analyzer checks the syntax in that the number of scheduled processes must be consistent with the number of processes assigned a priority by Designate Priority.

(5) See Appendix I for a description of a real-time example using this concept.

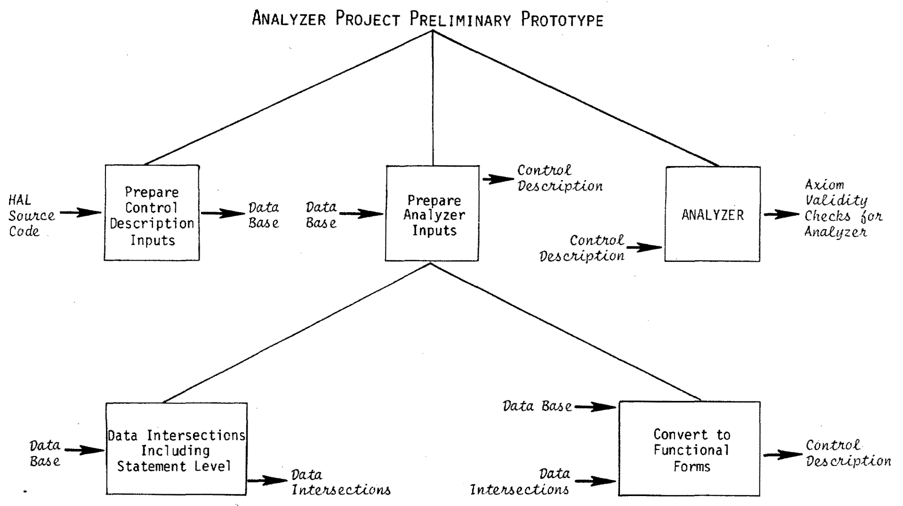

ANALYZER PROJECT PRELIMINARY PROTOTYPE
HAL Source Code
Prepare Control Description Inputs
Data Base
Data Base
Prepare Analyzer Inputs
Control Description
Control Description
ANALYZER
Axiom Validity Checks for Analyzer
Data Base
Data Intersections Including Statement Level
Data Intersections
Data Base
Data Intersections
Convert to Functional Forms
Control Description

Fig. 14. Analyzer prototype.

grouping is blocked by enclosing it within a ***do . . . end*** statement.

These groupings are limited to the basic forms of partition, composition. For partition, we use the HAL control statements ***if then else*** and ***case***, and restrict the variable usage. For composition, we use a ***do . . . end*** group and make use of the temporary variable facility in HAL for variables referenced and assigned within the ***do . . . end*** which, in effect, restricts the HAL scope rule. For recursion, we are devising a source macro because HAL does not allow recursive calls within the language. In addition, some statements (e.g., ***goto, while A do B***) are considered illegal for this prototype analyzer. A preliminary HOS control map can now be generated for the invocation of closed blocks [40]. A complete control map which details functional relationships to the statement level is now in development.

Whereas the structured design diagram indicates execution flow of the actual algorithms for a given system and the data intersections between explicitly invoked modules, the control map shows functions and their relationships. For example, in a control map, the interfaces of the data are shown hierarchically. Thus, in the future the automatic data intersection feature in the design diagrams may no longer be necessary when the control map is completely automated. Ideally, the control map should be designed before the structured design diagram, since it is used as a guide in following the axioms. The control map can then be used as a guide in producing a structured design diagram. For, inherent in the proper control map is the resulting proper decomposition for both functional flow and data flow. (Note here that the use of basic structured programming rules could violate HOS, but HOS does not violate structured programming concepts.)

Specification language ground rules, structuring executive operating system design, and various other verification and documentation aids are in the preliminary design stage.

VI. SUMMARY

The key to software design, development, and management is reliability. The most important aspect of developing reliable software is that, of course, it *WORKS*. The second most important aspect of a reliable software system is that its development and maintenance is far less expensive than that of a conventional system. The costs of developing reliable software will be drastically cut, since by its very nature reliable software is flexible and it is *truly* modular. Modules for both applications systems and support tool systems can be used over again and collected for different configurations or for different applications. Modules can be developed independently of the hardware and, in fact, can be used with different and changing hardware systems. Applications modules can be separated from lower virtual ("systems software") modules. Modularity can be capitalized on to single out parts of systems which are secure from those which are available to common users. Modules provide a means for defining software units and milestones which can be measured. Thus, it is possible to more accurately monitor subsystem usage and predict life-cycle costs. Modularity allows for a system to be changed automatically during development or in real time where that change and its affect can be automatically monitored and recorded.

The key to software reliability is to design, develop, and manage software with a formalized methodology. The formal methodology of HOS is dedicated to systems reliability. We have found that the enforcement of formalized rules on a large project with many organizations is almost impossible without the aid of automated tools.[19] A given system, described with an HOS specification language, enforces the axioms with the use of each construct. A system defined in HOS is analyzed automatically for axiomatic consistency by the Design Analyzer on a static basis, and by the Structuring Executive Analyzer on a real-time basis. The ultimate aim is for the specification language to incorporate all features of the Design Analyzer into the static analysis of the language constructs, and all features of the Structuring Executive Analyzer into a real-time, dynamic analysis provided by the language.

The support tools of software systems can now be defined with HOS as well as communicate with formalized HOS systems. Since HOS is used throughout all phases of development and for all disciplines of development including management, design, implementation, verification, and documentation, the same software support tools can be applied for all phases and all disciplines of systems development.

The statistical analysis and creation of manual checklists of previous large software efforts provided us with reliable methods to be incorporated into future developments. It is clear that such efforts should be encouraged as ongoing projects for any large software effort. A formal methodology is now available as another method for understanding more about software itself, as well as about future requirements for software development. One of the aspects of this methodology is that so many of the design considerations of a conventional system that were ad hoc (such as asynchronous process handling, error detection and recovery, interface correctness and data management) are now formally related to the inherent structure of each HOS system.

With the axiomatic approach of HOS now available, we have already been able to determine, from theoretical considerations, many more tools and techniques which can be developed based solely on the fact that there is now a way to design and develop them. And the reason for this is that we are able to more clearly understand software systems, since there is a formal means to describe them. With a formal means to describe systems, there is an automatic way to communicate with them. With an automatic means to communicate with systems, there is now a means to exhaustively verify the interfaces of a given system automatically and statically. With a means to build reliable software systems, we are reminded that those same means can be applied to developing other systems—for software itself is, after all, a system.

APPENDIX I
FORMULATION

Let us describe a control system in which all logical possibilities of control can be represented as a tree structure. Each node (any point at which two or more branches intersect) of the tree represents a unique point of execution of a function. Each node and all its dependents represent the unique tree structure T.

A function, F: $Q \to P$ or $P = F(Q)$, is a mapping from the input set Q to the output set P. Each element of the input set is expressed as a unique element of the output set.

We define an A-dimensional input space by the values of the A variables $(x_1, x_2 \cdots x_A)$. And we define a B-dimensional output space by the values of the B variables $(y_1, y_2 \cdots y_B)$. An element of the input set $q \in Q$ is a particular point for $(x_1, x_2 \cdots x_A)$. An element of the output set $p \in P$ is a particular point for $(y_1, y_2 \cdots y_B)$.

In order to execute a function, we must define a controller, the *module*. The module exists at the node just immediately higher on the tree relative to the functions it controls.

[19]When the automatic structured flowcharter was first introduced, many programmers were converted to structured programmers over night, since this tool not only helped to enforce structured programming, but it also saved the programmer the work of manually producing a flowchart.

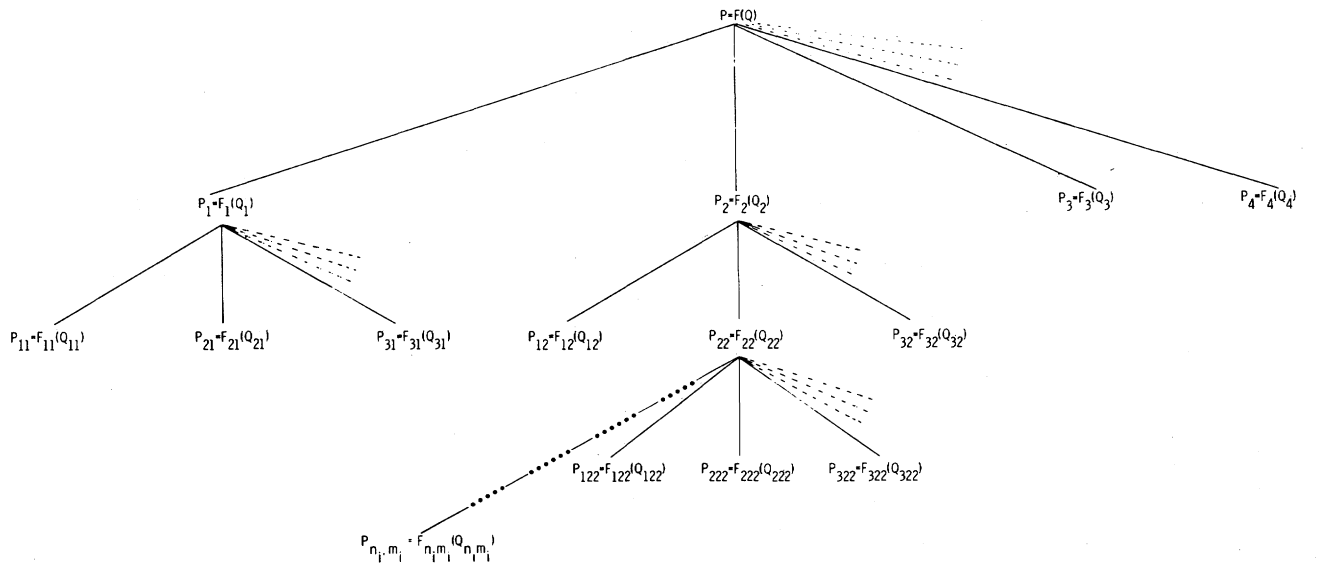


Fig. 15. Formal control system: $S_{n_i m_i} \equiv [P_{n_i m_i} = F_{n_i m_i}(Q_{n_i m_i})]$.

A module has the responsibility to perform a function. For that purpose, the module controls functions only on the immediate lower level. For implementation this is done by invocation (e.g., ***call***[20] or ***schedule***), by assignment of access rights (e.g., parameter passing or restricted use of global variables), and by the determination of the ordering of the functions on that level (e.g., priority assignments). Every function receives input from and produces outputs for its controller either directly or indirectly. Every node is a module with respect to its immediate lower level functions. With respect to control relationships, the elements of the lowest level of any tree are referred to only as functions; the highest node of the entire tree structure is referred to only as a module.

The following symbols are used in the discussion below:

$\forall$	for every
$\wedge$	logical "and"
$\vee$	logical "or"
$\in$	element of
$\subset$	subset of
$\cup$	union of
$\circ$	controls
$\not\circ$	does not control
$\overset{\uparrow}{\circ}$	interrupts
$\not\overset{\uparrow}{\circ}$	does not interrupt
$\exists!$	there exists a unique
$a \rightarrow b$	logical "***if a then b***"
$\{\ \}$	set of

Definition: The formal control system (Fig. 15) is one in which each module S has a unique identification

$$S_{n_i m_i} \equiv [P_{n_i m_i} = F_{n_i m_i}(Q_{n_i m_i})].$$

[20]A ***call*** can be explicit or implicit, e.g., $y = z^2 + 3$ is an implicit ***call***.

$n_i m_i$ defines a particular level of control in which i is the nested level of the module. $i = 1$ implies the level directly below the top level. n_i is the node position (from the left) relative to its most immediate higher node, m_i. At each level there is a set, N_i, of node positions, i.e., $n_i \in N_i$. m_i is the recursive relationship $m_i = n_{i-1} m_{i-1}$ defined for $i > 2$. If $i = 2$, $m_i = n_{i-1}$. If $i = 1$, $n_i m_i = n_i$.

Axiom 1: The module $S_{n_i m_i}$ controls the invocation of the set of valid functions on its immediate, and only its immediate, lower level, $\{F_{n_{i+1} n_i m_i}\}$. That is,

$$\forall j \, \forall n_{i+1} \in N_{i+1} \; \exists! \, S_{n_i m_i}, \; [(S_{n_i m_i} \circ F_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \longrightarrow S_{n_i m_i} \not\circ F_{n_j m_j})]. \quad (1)$$

Thus, the module $S_{n_i m_i}$ cannot control the invocation of functions on its own level.

It also follows that the module $S_{n_i m_i}$ cannot control the invocation of its own function.

In addition, the "no ***goto***" concept of structured programming is therefore consistent with the control system. For example,

> If "***C goto D***" exists, C loses control. e.g., C can control itself to terminate. In addition, if "***D goto C***" exists, D is controlling C and, in effect, is controlling itself.

Theorem 1.1: A module C cannot invoke function D, which, as a module, invokes function C, for then C would be controlling itself.

$$C \not\circ (D \circ C).$$

Corollary 1.1.1: A logical antecedent cannot be assigned by its consequent.

> e.g., If function C is comprised of "***if G then D,***" G cannot be assigned by D.

Theorem 1.2: If a function from level$_{i+1}$ is removed and the controller module at level$_i$ still maintains its same mapping, the function at level$_{i+1}$, $F_{n_{i+1} n_i m_i}$, is extraneous. The extraneous function is a direct violation of Axiom 1, for if the function is not removed, $S_{n_i m_i} \not\circ F_{n_{i+1} n_i m_i}$.

NOTE: Violation of Theorem 1.2, in common practice, manifests itself in modules with many user options. With respect to the entire system, the use of extraneous functions proliferates test cases and complicates interfaces.

Corollary 1.2.1: Consequents of a decision do not interrogate the antecedent for this would result in an extraneous function.

e.g., "*if G then D*" where *D* implies "*if G then E*" must be reduced to "*if G then E*."

Theorem 1.3: Assignment to a variable is restricted to one process when more than one process is concurrent. This is true because modules may only invoke valid functions, and a valid function has only one output value for a particular input value.

Axiom 2: The module $S_{n_i m_i}$ controls the responsibility for elements of the output space, of only $P_{n_i m_i}$, such that the mapping $F_{n_i m_i}(Q_{n_i m_i})$ is $P_{n_i m_i}$. That is,

$$\forall j \forall n_i m_i \exists!\, S_{n_i m_i}, [(S_{n_i m_i} \circ P_{n_i m_i}) \wedge ((n_j m_j \neq n_i m_i) \longrightarrow S_{n_i m_i} \not\circ P_{n_j m_j})]. \quad (2)$$

Thus, there must not exist any member of the input space for which no member of the output space is assigned. For, if this were not the case, we would have an invalid function.

Theorem 2.1: There may be more than one formulation for a particular function. It is only necessary that the mapping be identical. Equivalent computer functions may require a different formulation due to timing restrictions, etc.

Axiom 3: The module $S_{n_i m_i}$ controls the access rights to each set of variables $\{Y_{n_{i+1} n_i m_i}\}$ whose values define the elements of the output space for each immediate, and only each immediate, lower level function.

$$\forall j \forall n_{i+1} \in N_{i+1} \exists!\, S_{n_i m_i}, [(S_{n_i m_i} \circ Y_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \longrightarrow S_{n_i m_i} \not\circ Y_{n_j m_j})]. \quad (3)$$

NOTE: If any two modules, $S_{n_i m_i}$ and $S_{n_j m_j}$, require the same function formulation, the same set of computer residing instructions can be used for the functions as long as the access rights of the variables are controlled via Axiom 3.

Theorem 3.1: The variables whose values define the elements of the output space at level$_i$ are a subset of the variables whose values define the elements of the output space at level$_{i+1}$, that is,

$$Y_{n_i m_i} \subset \{Y_{n_{i+1} n_i m_i}\}.$$

Axiom 4: The module $S_{n_i m_i}$ controls the access rights to each set of variables $\{X_{n_{i+1} n_i m_i}\}$ whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

$$\forall j \forall n_{i+1} \in N_{i+1} \exists!\, S_{n_i m_i}, [(S_{n_i m_i} \circ X_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \longrightarrow S_{n_i m_i} \not\circ X_{n_j m_j})]. \quad (4)$$

Thus, the module $S_{n_i m_i}$ cannot alter the members of its own input set, i.e., the access to the elements of the input set of $S_{n_i m_i}$ cannot be controlled by $S_{n_i m_i}$.

Theorem 3,4.1: The variables of the output set of a function cannot be the variables of the input set of that same function. If $y = f(y, x)$ could exist, access to y would not be controlled by the next immediate higher level.

NOTE: Adherence to Theorem 3,4.1 simplifies error recovery techniques associated with parameter passing and functionally dependent iterative processes.

Theorem 3,4.2: The variables of the output set of one function can be the variables of the input set of another function only if the variables associated with the output set of the first function and the variables associated with the input set of the second function are variables of functions that exist on the same level and are controlled by the same immediate higher node. If $y = f_1(x)$ and $g = f_2(y)$, both functions exist at the same level. If $g = f_2(y)$ is at a lower level, access rights to the input set y imply y is determined before y exists. $y = f_1(x)$ at a lower level implies an alteration to the input set of $g = f_2(y)$.

Theorem 3,4.3: Each member of the set of variables whose values define the elements of the output space of a function is either a variable of the output space of the controller or is a variable of the input space for any of the functions on the same level excluding the variables of its own function.

$$\forall y_{n_i m_i} \in Y_{n_i m_i}, y_{n_i m_i} \in \{Y_{m_i} \cup \{\{X_{n_i m_i}\} - X_{n_i m_i}\}\}.$$

NOTE: Violation of Theorem 3,4.3 in common practice, manifests itself in modules that calculate by-product results for anticipated users: e.g., a Shuttle module that calculates the position vector of a vehicle might also calculate altitude, apogee, and perigee, instead of creating separate modules to perform the separate functions.

Theorem 3,4.4: Each member of the set of variables whose values define the elements of the input space of a function is either a variable of the input space of the controller or is a variable of the output space for any of the functions on the same level excluding those variables of its own function.

$$\forall x_{n_i m_i} \in X_{n_i m_i}, x_{n_i m_i} \in \{X_{m_i} \cup \{\{Y_{n_i m_i}\} - Y_{n_i m_i}\}\}.$$

Axiom 5: The module $S_{n_i m_i}$ controls the rejection of invalid elements of its own, and only its own, input set $Q_{n_i m_i}$, that is,

$$\forall j \forall n_i m_i \exists!\, S_{n_i m_i}, [(S_{n_i m_i} \circ Q_{n_i m_i}) \wedge ((n_j m_j \neq n_i m_i) \longrightarrow S_{n_i m_i} \not\circ Q_{n_j m_j})]. \quad (5)$$

Axiom 6: The module $S_{n_i m_i}$ controls the ordering of each tree $\{T_{n_{i+1} n_i m_i}\}$ for the immediate, and only the immediate lower level.

$$\forall j \forall n_{i+1} \in N_{i+1} \exists!\, S_{n_i m_i}, [(S_{n_i m_i} \circ T_{n_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \longrightarrow S_{n_i m_i} \not\circ T_{n_j m_j})]. \quad (6)$$

Thus, the module $S_{n_i m_i}$ controls the ordering of the functions, the input set, and the output set for each node of $\{T_{n_{i+1} n_i m_i}\}$. A process $\mathcal{P}$ is a function scheduled to occur in real time.

NOTE: If two processes are scheduled to execute concurrently, the priority of a process, $\tilde{\varrho}$ determines precedence at the time of execution.

If one process has a higher priority than another process, the higher priority process can interrupt the lower priority process. The lower priority process is resumed when the higher priority process is either terminated or is in its wait state.

Theorem 6.1: The priority of a process is higher than the priority of any process on its most immediate lower level.

Theorem 6.2: If two processes have the same controller such that the first has a lower priority than the second, then all processes in the control tree of the first are of lower priority than the processes in the control tree of the second.

Theorem 6.3: The ordering between members of any given process tree cannot be altered by an interrupt. For, if process A is of higher priority than process B, process A interrupts all of B, i.e., the priorities of the dependent processes of B are less than the priority of B.

Theorem 6.4: A module $S_{n_i m_i}$ controls the priority relationships of those processes on the immediate, and only the immediate, lower level.

$$\forall j \, \forall k \, \forall n_{i+1}, w_{i+1} \in N_{i+1} \, \exists! \, S_{n_i m_i}, [n_{i+1} \neq w_{i+1} \longrightarrow (S_{n_i m_i} \circ (\varrho_{n_{i+1} n_i m_i} \overset{\uparrow}{\circ} \varrho_{w_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \longrightarrow (S_{n_i m_i} \not\circ (\varrho_{n_j m_j} \overset{\uparrow}{\circ} \varrho_{n_k m_k}))].$$

Corollary 6.4.1: An explicit priority relationship exists among each pair of processes at the same level which branch from the same most immediate higher level node.

NOTE: If the priority of process $\varrho_{a_i m_i}$ is greater than the priority of $\varrho_{b_i m_i}$, and the priority of $\varrho_{b_i m_i}$ is less than the priority of $\varrho_{c_i m_i}$, the controller at node m_i does not control the priority relationships of $\varrho_{a_i m_i}$ and $\varrho_{c_i m_i}$. To avoid a violation of Theorem 6.4, the priority relationship of $\varrho_{a_i m_i}$ and $\varrho_{c_i m_i}$ must also be stated explicity.

Corollary 6.4.2: If a process $\varrho_{n_i m_i}$ can interrupt another process $\varrho_{w_i m_i}$, the control tree of $\varrho_{n_i m_i}$ can interrupt $\varrho_{w_i m_i}$ and each member of the control tree of $\varrho_{w_i m_i}$.

Corollary 6.4.3: Since $S_{n_i m_i}$ controls the priority relationships of the set of processes $\{\varrho_{n_{i+1} n_i m_i}\}$, then it itself cannot interrupt any member of the processes of the control tree $\{T_{n_{i+1} n_i m_i}\}$.

Corollary 6.4.4: A process cannot interrupt itself.

Lemma 6.4.4.1: The multiprogram statement ***wait*** is an implicit decision of a process to interrupt itself at a future time, and therefore is not consistent with the control system.

Corollary 6.4.5: A process cannot interrupt its controller.

Corollary 6.4.6: A process $\varrho_{n_i m_i}$ that can interrupt another process $\varrho_{n_j m_j}$ affects the absolute time loss for $\varrho_{n_j m_j}$.

Theorem 6.5: If the antecedent of a decision within a module is a time relationship, the relationship is extraneous. For, only if the relationship is removed, could the ordering be controlled by the next higher level.

e.g., "(time $= t_1) \rightarrow A$" can be replaced by A where the controller schedules A at time t_1.

Theorem 6.6: If a function is invoked by a ***call*** or ***schedule***, the corresponding module of said function can invoke another function by a ***call***.

Theorem 6.7: A ***schedule*** of two processes may be commutative, but a ***call*** of two functions is not commutative.

Theorem 6.8: If a function $F_{n_i m_i}$ is invoked by a ***call***, the module corresponding to the function $S_{n_i m_i}$ cannot invoke a process. If the module schedules a process and the set of operations of the function $F_{n_i m_i}$ is performed before the process is executed, the module does not control the scheduled process.

Theorem 6.9: A ***schedule*** must always cause the processes invoked to be dependent so that the higher level maintains control at all times.

Theorem 6.10: The maximum time for a cycle of a process to be completed or delayed can be determined. Consider a particular level which has N processes. The nth process has a frequency, f_n cycles per second, such that the period t_n for the nth process is $1/f_n$, and the total maximum execution time for that process is Δt_n. The jth process has a maximum number of cycles $t_n f_j$ during time t_n. Then, $t_n f_j \Delta t_j$ is the maximum time a process could consume during t_n.

From Corollary 6.4.1, there exists priority relationships for N processes such that

$$\tilde{\varrho}_{1 m_i} > \tilde{\varrho}_{2 m_i} \ldots > \tilde{\varrho}_{n_i m_i} \ldots > \tilde{\varrho}_{N_i m_i}.$$

Therefore, process n can be completed within time t_n if its maximum completion time t_c is less than t_n where

$$t_c = \sum_{j=1}^{J} t_n f_j \Delta t_j + \Delta t_n,$$

and J equals the total number of processes of higher priority than process n on the control level of n. Also, the maximum delay time for process n is the maximum completion time for the process of the nearest higher priority on its own level.

THEOREM: It is possible to control all functions of any given software problem such that one controller controls all functions.

If we have only one level of control, every function can be performed, all access rights can be allocated, and the ordering between functions can be controlled.

Thus, it is always possible to perform every required function of any software system by adhering to the six axioms of the control system.

A Real-Time Control Problem

The design of any system which does not have the potential to assign the same variables concurrently is deterministic. In this type of system, functions can either be functionally independent, functionally dependent, or mutually dependent.

An independent function is one in which the output set is not the input set of another function. A dependent function is one in which the output set is an input set of another function. Two functions are mutually dependent if the output set of the first is the input set of the second, and the output set of the second is the input set of the first.

The design of any system which has the potential to assign the same variables concurrently is nondeterministic.

A system is able to provide for mutually dependent functions to be executed concurrently if the HOS axioms are applied in real time. With such a system, the operator need not memorize permutations of proper operational sequences. In addition, the software system is able to handle error detection and recovery for all operater-selected processes by simple algorithms rather than by special algorithms for each operator selection or by complicated table look-up algorithm schemes.

Consider a typical Shuttle example of a nondeterministic system where the modules M_1 and M_2 are mission phases. M_1 and M_2, if executed in parallel, could both perform the same function of ***guidance, navigation,*** and ***vehicle control***. In this case, a mechanism is needed to prevent the M_1 and M_2 processes from conflicting with each other.

Let us now consider a dynamic scheduling algorithm (or structuring executive). The scheduler: 1) controls the ordering of those modules which can vary in real time dependent on operator selection; 2) assigns priorities to processes based on the relative priority relationships, according to Axiom 6, for each control level; 3) prevents a violation of the HOS axioms so that no two processes can conflict with each other; and 4) determines when the total resources of the computer are approached. Such a dynamic scheduler would assume the following tools: process locks, data locks, and a scheduling algorithm which provides relative and variable priorities. The process lock for each process locks out lower priority processes other than those on its own tree from assigning data for as long as the process is active (i.e., executing or in the wait state). Data locks within a process temporarily lock out all other processes from reading elements of a data block when that block is being updated. The controller for a process is a real-time scheduler. The scheduler invokes a process, via the schedule statement, to automatically set a process lock, assign a priority, and set up data locks for the process invoked.

Each scheduled process is dynamically assigned a unique priority set which bounds unique priority sets of its dependent processes. The priority of a process is determined by its 1) level of control; 2) order of selection by the operator; and 3) predetermined priority relationships. The highest priority process is, by definition, the highest level controller. Each controller has a higher priority than the processes it controls. In order to compare the priorities of two processes, a process chain up the tree for each process is made until one level before the branches intersect. The priorities of the parent processes at that level determine the priority of the two processes in question, i.e., the process with the highest priority parent has the higher priority. Thus, we have a system where a process and all its dependent processes are either all higher or all lower than another nondependent process, and all of its dependents. Consider Fig. 1 to be a subset of the Shuttle system S.

S_1, S_2, S_3, S_4	Ascent mission phase, atmospheric entry mission phase, astronaut display, and abort mission phase, respectively.
S_{11}, S_{21}, S_{31}	Ascent guidance (G_A), ascent navigation (N_A), and ascent vehicle control (C_A), respectively.
S_{12}, S_{22}, S_{32}	Entry guidance G_E, entry navigation N_E, and entry vehicle control C_E, respectively.
S_{121}, S_{211}	Navigation state extrapolation and measurement incorporation, respectively.

Thus, for example, in Fig. 15 the relative priorities of ascent guidance S_{11}, and measurement incorporation S_{22}, are determined by comparing the priorities of the parent mission phases S_1 and S_2.

Each module defines priority relationships for each function it controls. For example, S controlling $\{S_i\}$ might have the priority relational information: $((\tilde{S}_1, \tilde{S}_2) < (\tilde{S}_4))$. S_3 is a function invoked by a ***call*** and, therefore, has the same priority as the scheduler S. The priorities S_1 and S_2 are initially equal, but their priorities (and thus, priority sets) are decided by the ordering of schedule invocation. Yet, S_1 and S_2 are always of lower priority than S_4. A typical Shuttle relationship is $(\tilde{C}_E > \tilde{G}_E > \tilde{N}_E)$, where the dependent relationships between ***vehicle control***, ***guidance***, and ***navigation*** are maintained on a fixed relative priority basis. In the latter example, C_E can interrupt N_E. C_E is functionally dependent on G_E, i.e., C_E uses the output set of G_E as its own input set. In addition, a mission phase schedules N_E at a higher frequency than G_E. At all times the priority relationships of C_E, G_E, and N_E remain fixed.

If S_1 is selected first, S_2 cannot interrupt S_1 as long as S_1 or any of its dependent processes are being executed. When S_1 is in a wait state, S_2 can execute, but only if S_1 is not ready to execute. When S_1 is ready, process set S_1 interrupts process set S_2. If the S_2 set attempts to assign data process locked in the S_1 set, S_2 and its dependents are terminated by the scheduler. At this point the last display is regenerated by the scheduler of the terminated process, thus giving the astronaut complete visibility. If however, S_1, when it becomes active, attempts to assign data which are process locked by S_2, S_2 is terminated since S_1 has a higher priority lock than S_2. If S_2 attempts to read data process locked by S_1, and these data are presently being assigned by the other set, the S_2 process waits for the S_1 block of data to be updated. Likewise, S_1 must wait to read data presently being assigned by S_2.

The operational levels of a system are, by definition, those levels where the operator has at least one option. Each level has the potential to be an operational level. Consider the Shuttle example. The operational level S allows the astronaut to select, reselect, proceed from, or terminate S_1, S_2, or S_4 via S_3. Operator errors at each operational level are prevented via the process lock mechanism. Due to HOS axioms, at a nonoperational level (i.e., one where a conflicting process is not initiated by the operator) an error of this type would not occur. Without a static analyzer, however, the process lock mechanism of the scheduler would discover the error dynamically. Of course, the analyzer avoids an expensive way to find a software error.

Alternatives for reselecting or terminating an existing process depend, to a large extent, on desired operational procedures. The scheduler could display a "select" or "terminate" option for each operational process. Or, the operator could request the highest level scheduler which has the highest priority to terminate a specific process. If S_1 is selected when S_1 is in the queue, either the first S_1, or the second, is terminated.

Consider two processes S_1 and S_2, where S_1 has a higher priority than S_2. Scheduler S_{11} schedules S_{111}. S_{111} could have a very low or a very high priority relative to but less than S_1: but relative to S_2, S_1, and all its dependent processes have higher priorities. Thus, if S_{111} is controlled by a ***do while*** instead of a cyclic schedule, S_2 and its dependent processes are locked out for the duration of the ***do while***. However, if S_{111} is scheduled cyclically, S_2 can be processed when S_1 and its dependents are in the wait state. In conventional priority schemes, this would not be the case, since S_2 could be arbitrarily assigned a priority less than S_1 but greater than S_{111}. Thus, the use of a ***do while*** construct as a substitute for a cyclic process is discouraged. It is interesting to note that a ***do while*** within a nonmultiprogrammed function can never be terminated by an outside controller. Thus, in this case, one would be advised to use a ***do for while*** instead.

Appendix II

Theorems directly stated in the text are presented here, with corresponding proofs.

Immediate Self-Control Theorem: Two functions cannot exist such that each is defined by the same sets of variables and one of said functions is a subfunction of the other. For if this were the case, we can show the controller function is controlling itself.

Proof: If the sets of the input variables of two functions are equal and the sets of the output variables of the same two functions are equal, the mappings are either the same or different. If the mappings are different, then the specification must be incorrect, for the controlled function must produce the result for its controller. If the mappings are equal, the two functions are equal.

Indirect Self-Control Theorem: If a function is defined, a node defined within the tree of said function cannot be defined so that the sets of input variables are equal and the sets of output variables are equal. Again, we can show that if this condition were to exist, the controller function is controlling itself.

Proof: Consider Fig. 16. The lower level function must perform for its controller since y is not local. Each subsequent higher node must likewise perform for its controller since y is not local to any node below the highest node in which it is a member of the set of output variables (cf. Theorem 3,4.4, [1]). The lower level must be able to perform for each value of its input set. But the set of input variables is equal to the highest set of input variables. Therefore, the mapping must be the same or there would exist at least one value for y which would be incorrect. Thus, function f must be identical to function g. If we assign a value for y for each element of x, then we have performed the total function for the immediate controller.

Any other subfunction at that lower level is extraneous (cf. Theorem 1.2, [1]), because every value of y can be obtained via that one subfunction. Thus, the most immediate controller and its subfunction are equal. We can either eliminate the lowest subfunction as redundant or recognize that a formulation of this type has not decomposed the controller and redesign. If we eliminate the lowest subfunction, we again continue to apply the same argument until we reach the highest node at which the equal function exists. At this point there is no recourse other than to admit the function is controlling an equal function.

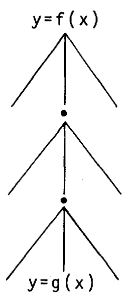


Fig. 16. Proof of indirect self-control.

Uniqueness Theorem: Each node of an HOS hierarchy is unique, i.e., two functions cannot exist within the same hierarchy such that the same relationship exists between input and otuput elements and the same sets of input variables and output variables are defined for each function.

Proof: Suppose the function $y = f(x)$ existed within a system at two different nodes. The two nodes could exist at the same level [Fig. 17(a)]. In this case, one of the nodes could be removed and, since the functions are equal, the controller could still perform its same mapping. Thus, all but one of the equal functions on the same level are extraneous and invalid via Theorem 1.2 [1].

If $y = f(x)$ appeared as a node inner to its own tree [Fig. 17(b)] the module would be controlling its own function and is invalid via the proof of immediate self-control (cf. Section III). Likewise, any node that can be traced up the tree or down the tree from $y = f(x)$ must be of the form $y = f(x)$ [Fig. 17(c)], via the proof of indirect self-control (cf. Section III).

Consider $y = f(x)$ at one node. Variable y must communicate on its own level or perform an output element for its controller (Theorem 3,4.3 [1]). If y were a local variable [Fig. 17(d)], y could not communicate to any higher level because the controller of $y = f(x)$ could not perform an output element for y. Thus, $y = f(x)$ could not exist outer to the level at which y were a local variable. Suppose, on the other hand, that $y = f(x)$ was invoked by its controller to perform an output element for the controller. If $y = f(x)$ appeared as a node inner to any tree on the same level in which it is local [Fig. 17(e)], it would also be invalid because y could only communicate as an input variable, in this case, by Theorem 3,4.2 [1].

Suppose y does not communicate on its own level. The controller of $y = f(x)$ must be a selector in which case y could be assigned by another node at the level of $y = f(x)$ or y is not known to another node at the level of $y = f(x)$. In the case where y is not known to a node which exists at the same level as $y = f(x)$, $y = f(x)$ could never exist inner to that node (Theorem 3.1 [1]).

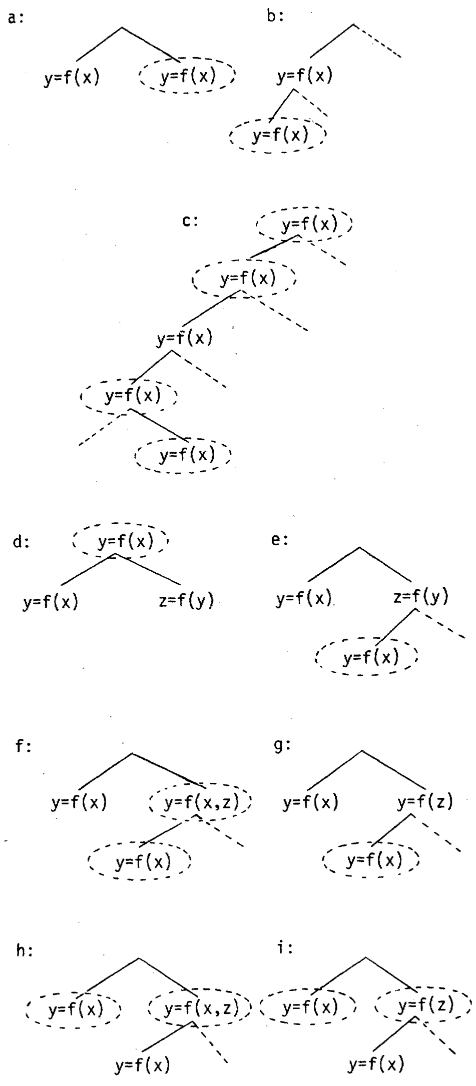


Fig. 17. Proof of uniqueness.

In order for $y = f(x)$ to be considered as a node inner to any tree on that same level at which another $y = f(x)$ node is known to exist, the node at the same level as $y = f(x)$ can be of the form $y = f(x, z)$ or $y = f(z)$. If that function were of the form $y = f(x, z)$ [Fig. 17(f)], then the controller of $y = f(x, z)$ and $y = f(x)$ is an invalid function because it is possible that for a given element of x, more than one value of y can exist. This will be the case if we assume a selection mechanism (i.e., we have an invalid partition) or if we try to perform both functions (i.e., concurrent assignment or invalid composition). If that function were of the form $y = f(z)$ [Fig. 17(g)], the function $y = f(x)$ could not exist at any node inner to the node $y = f(z)$ because the variable is not known inner to $y = f(z)$. Thus, if $y = f(x)$ exists, another node of this form, $y = f(x)$, cannot exist at the same level or at an inner node to any other node at said level.

If y performs a direct output value for its controller, the form of the controller must be $y = f(x, z)$ [Fig. 17(h)] or $y = f(z)$ [Fig. 17(i)].

If the controller is of the form $y = f(z)$, x is internal to $y = f(z)$ and can never exist outer to $y = f(z)$ (Theorem 3,4.4 [1]). Thus, in this case $y = f(x)$ could not exist outer to $y = f(z)$.

If the controller is of the form $y = f(x, z)$, it is not possible to have a node of the form $y = f(x)$ at the same level as $y = f(x, z)$, for the same reasons presented above (i.e., invalid function formulation of the controller). In order for $y = f(x)$ to exist at any higher level, we must perform a value for y by attempting a higher controller of the form $y = f(x, z)$. Each time, the higher controller becomes an invalid function. Thus, we have shown that $y = f(x)$ must be a unique node of a system.

ACKNOWLEDGMENT

The authors would like to express appreciation to L. Robinson of Stanford Research Institute for a most helpful and critical review of this paper. The authors would also like to thank A. Volta for technical editing assistance, and G. Lopes for the preparation of the manuscript.

REFERENCES

[1] M. Hamilton and S. Zeldin, "Higher order software techniques applied to a space shuttle prototype program," in *Lecture Notes in Computer Science*, vol. 19, G. Goos and J. Hartmanis, Ed. New York: Springer-Verlag, pp. 17-31, presented at Program. Symp. Proc., Colloque sur la Programmation, Paris, France, April 9-11, 1974.

[2] M. Hamilton, "Design of the GN&C flight software specification," Charles Stark Draper Lab., Cambridge, MA, Doc. C-3899, Feb. 1973.

[3] R. Millard, internal Charles Stark Draper Lab. study of APOLLO, Cambridge, MA, 1969-1972.

[4] L. Fair *et al.*, *Planning Guide for Computer Program Development*, System Development Corp., Santa Monica, CA, Tech. Memo. TM-2314/000/00A, May, 1965, p. 57.

[5] B. W. Boehm, "Some information processing implications of air force space missions: 1970-1980," Rand Corp., Santa Monica, CA, Memo. RM-1213-PR, Jan. 1970.

[6] A. M. Turing, "On computable numbers with an application to the entscheidungs problem," in *Proc. London Math. Soc.*, ser. 2, vol. 42, 1936.

[7] D. Parnas, "On a 'buzzword': Hierarchical structure," in *1971 Fall Joint Comput. Conf., IFIPS Conf. Proc.*, vol. 39. Montvale, NJ: AFIPS Press, 1971.

[8] D. D. Chamberlin, "The single-assignment approach to parallel processing," in *AFIPS Proc.* New York: Elsevier, 1974.

[9] W. A. Wulf *et al.*, "Bliss: A language for systems programming," *Commun. Ass. Comput. Mach.*, vol. 12, Dec. 1973.

[10] B. Randell, "Research on computing system reliability at the University of Newcastle upon Tyne, 1972-1973," Comput. Lab., Univ. Newcastle upon Tyne, Newcastle upon Tyne, Scotland, Tech. Rep. Ser. 57, Jan. 1974.

[11] J. Miller *et al.*, *CS-4 Language Reference Manual*, Intermetrics, Inc., Cambridge, MA.

[12] J. Schwartz, *On Programming, An Interim Report on the SETL Project*, Dept. Comput. Sci., Courant Inst. Math. Sci., New York University, New York, "Installment 1: Generalities," 1973; "Installment 2: The SETL Language and Examples of its Use," 1973.

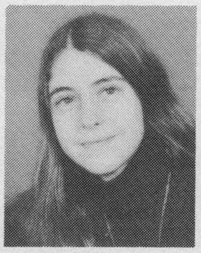
[13] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, C. A. R. Hoare, Gen. Ed. New York: Academic, 1972.

[14] M. Hamilton, "Management of APOLLO programming and its application to the shuttle," Charles Stark Draper Lab., Cambridge, MA, Software Shuttle Memo. 29, May 1971.

[15] M. Hamilton and S. Zeldin, "MERCURY statistical analysis," Charles Stark Draper Lab., Cambridge, MA, MERCURY Memo. 34, Nov. 1973.

[16] N. Brodeur, "MERCURY statistical analysis report revisited," Charles Stark Draper Lab., Cambridge, MA, MERCURY Memo. 34, Nov. 1973.

[17] M. Hamilton, "First draft of a report on the analysis of APOLLO system problems during flight," Charles Stark Draper Lab., Cambridge, MA, Shuttle Management Note 14, Oct. 1972.
[18] —, "The AGC executive and its influence on software management," Charles Stark Draper Lab., Cambridge, MA, Shuttle Management Note 2, Feb. 1972.
[19] H. Laning, AGC Program Sundisk, Charles Stark Draper Lab., Cambridge, MA, Executive Rev. 267, NASA 2021108-011, Nov. 1967.
[20] M. Hamilton, AGC Program Sundisk, Charles Stark Draper Lab., Cambridge, MA, Display Interface Rev. 267, NASA 2021108-011, Nov. 1967.
[21] D. Lickly, AGC Program Sundisk, Charles Stark Draper Lab., Cambridge, MA, Restarts Rev. 267, NASA 2021108-011, Nov. 1967.
[22] C. Muntz, "Users guide to the block II AGC/LGC interpreter," Draper/M.I.T. Doc. R-489, Apr. 1965.
[23] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, Dec. 1972, pp. 1053–1058.
[24] —, "A technique for software module specification with examples," *Commun. Ass. Comput. Mach.*, vol. 15, May 1972, pp. 330–336.
[25] M. Hamilton and S. Zeldin, "Higher order software requirements," Charles Stark Draper Lab., Cambridge, MA, Doc. E-2793, Aug. 1973.
[26] —, "Top-down/bottom-up, structured programming and program structuring," Charles Stark Draper Lab., Cambridge, MA, Rev. 1, Doc. E-2728, Dec. 1972.
[27] W. Daly, "Automatic flowcharts," Charles Stark Draper Lab., Cambridge, MA, MERCURY Memo. 53, Mar. 1974.
[28] IBM, "Chief programmer teams principles and procedures," IBM Fed. Syst. Div., Gaithersburg, MD, Rep. FSC-71-5108, June 1971.
[29] G. M. Weinberg, *Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.
[30] J. R. White and L. Presser, "A tool for enforcing system structure," in *Proc. Ass. Comput. Mach.*, 1973.
[31] M. Hamilton, "A discussion of higher order software concepts as they apply to functional requirements and specifications," Charles Stark Draper Lab., Cambridge, MA, Doc. P-019, Dec. 1973.
[32] B. McCoy, "DAIS avionic software development techniques," presented at the Amer. Inst. Aeronaut. and Astronaut. Digital Avionics Syst. Conf., Boston, MA, Apr. 2–4, 1975.
[33] G. Boetje, "Managing software development: A new approach," presented at the IEEE Nat. Aerospace Electron. Conf. (NAECON), Dayton, OH, June 10–12, 1975; Charles Stark Draper Lab., Cambridge, MA, Doc. P-165.
[34] D. DeVorkin, "The DAIS executive system," to be published.
[35] The Charles Stark Draper Lab., Inc., "Submittal of structured flowcharts of the approach and landing test (ALT) GN&C design for the space shuttle orbiter GN&C integration task," Charles Stark Draper Lab., Cambridge, MA, Jan. 1975.
[36] Rockwell Int. Corp., "Space shuttle orbiter approach and landing test, level C, functional subsystem software requirements document guidance, navigation and control, part A, guidance," append. A, Nov. 1974; also as above, "Part B, flight control," Downey, CA.
[37] Honeywell, Inc., "Horizontal flight digital autopilot requirement definition," Honeywell, Sarasota, FL, ED 21532, PRD UA17, July 1974, vol. II, sect, 5-7.
[38] IBM, "Space shuttle orbiter avionics software," vol. III, "Applications software, part 1-GN&C," "Approach and landing test (ALT) functional design specification," 75-SS-0473, Fed. Syst. Division, Houston, TX, Feb. 1975.
[39] D. J. Lickly *et al.*, *HAL/S Language Specification*, Intermetrics, Inc., Cambridge, MA.
[40] G. Goddard, "Control map development," Charles Stark Draper Lab., Cambridge, MA, Software Control System Analyzer Memo. 4, Jan. 1975.



Margaret Hamilton received the A.B. degree in mathematics from Earlham College, Richmond, IN, in 1958.

Since 1965 she has been with The Charles Stark Draper Laboratory, Cambridge, MA, where she is currently Division Leader of the Computer Science Division. She heads a group of software engineers, the members of which are engaged in both research and application in the areas of systems flight software for: Space Shuttle, C4, DAIS and F8. In addition, her group has been involved in software design and implementation for automatic aircraft control, production control systems, computer graphics, biomedical research, transportation, consumer durables, data management, helicopter navigation and control, and instrumentation redundancy studies. She is presently engaged in designing software techniques covering subjects of software reliability and management. As head of a 100 plus member programming group, she was responsible for the design, development, verification, and documentation of all command and lunar module on-board software for the Apollo and Skylab programs. She has personally contributed to such software activities as command module flight software design and programming, quality assurance, test engineering, systems analysis, real-time software systems integration, multiprogramming, task management, error detection and recovery, and man/machine interface design.

Saydean Zeldin received the A.B. degree in physics from Temple University, Philadelphia, PA, in 1961.

Since 1966 she has been with the Charles Stark Draper Laboratory, Cambridge, MA, where she is currently Section Chief of the Computer Science Division. She leads a group of software engineers presently engaged in the design of software techniques which include such areas as software reliability, management, design techniques, verification methods, and real-time multiprogrammed systems. Her interest in the reliability of large-scale software systems stems from her experience in the design, development, and verification of the Apollo on-board software. These software activities included: command and lunar module flight software design and programming, test engineering, systems analysis, and real-time software systems integration. As a Principal Engineer at The Charles Stark Draper Laboratory she personally contributed to Apollo programs involving orbital maneuvers, atmospheric entry, and mission sequences that interfaced with the astronaut and various hardware systems.