

AIAA PAPER
75-593

HIGHER ORDER SOFTWARE - A METHODOLOGY FOR DEFINING SOFTWARE

by
M. HAMILTON and S. ZELDIN
The Charles Stark Draper Laboratory, Incorporated
Cambridge, Massachusetts

AIAA Digital Avionics System Conference

BOSTON, MASS. / APRIL 2-4, 1975

HIGHER ORDER SOFTWARE - A METHODOLOGY FOR DEFINING SOFTWARE

M. Hamilton and S. Zeldin
The Charles Stark Draper Laboratory, Inc.
75 Cambridge Parkway
Cambridge, Massachusetts 02142

75-5893

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: structuring 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. The definition of a given system can be described with an HOS meta-language which, by its very nature, enforces the axioms with the use of each construct. A system defined in HOS is 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 especially significant since interface testing in a large system accounts for approximately 75% of the verification effort.

I. Introduction

Higher Order Software (HOS) is a formal methodology for reliable systems specification and development¹. 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 for any given system.

A hierarchical decomposition of a function is possible if that function is computable. The class of effectively computable functions coincides exactly with the definition of the term, software. A numerical 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 given². A system 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.

For a typical avionics specification, we can define a total system comprised of a set of functions and functional interfaces. For example, to verify APOLLO on-board flight software³, we interfaced flight software in digital simulations with models of other systems including the APOLLO Guidance Computer (AGC), sensors, the astronaut and the universe. We also interfaced flight software in hybrid simulations with actual noncomputable systems such as the astronaut and with models such as the universe.

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. We have found that the use of HOS for software system design clarifies the intent of the designer and simplifies problems originally approached in a traditional manner; and, if we look at systems, in general, as software, it may help to either understand more thoroughly a complicated system, or help make a seemingly complicated system less complicated.

It is now possible within the framework of HOS to develop automatic tools to aid verification as well as design. For example, the interfaces of an HOS system can be exhaustively tested by an automatic analyzer without program execution. This is especially significant since interface testing in a large system is known to be a very costly procedure⁴.

The HOS methodology can be used for the definition of software for simplex, multiprogrammed, multiprocessor or multicomputer systems. For any of these systems, the approach would be to use HOS to design one implementation independent system structure and then, for implementation, to partition and distribute the software and hardware while maintaining HOS control structures.

¹73% of all problems found during the APOLLO integration effort were interface problems⁴; and verification accounts for 50% of the total software development effort^{5,6,7}.

II. Methodology

Higher Order Software is software expressed in its own meta-language and conforming to a formalized set of laws. The basic components of HOS methodology are (Figure 1): (1) the application of the formal set of laws to the design of a given problem; (2) a meta-language adhering to these laws; (3) the automatic analysis of design 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*; and, (5) the hardware that is transparent to the user*. In addition, support tools based on axiomatic consistency can enhance a given development process in such areas as: performance analysis, simulation, design automation, definition of subsystem requirements, automatic documentation and automatic management techniques.

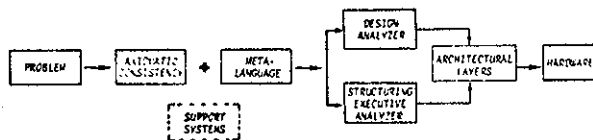


Figure 1: Higher Order Software Methodology

A. Formulation

In HOS, any given software system can be represented by a single mathematical function. The function representing the entire system is an assumption. Those functions that are invoked to perform**the "system-function" describe subsystems, each of which may be represented by a single mathematical function of its own.

Any given software system, as well as each of its subsystems, can be described by means of a tree structure that defines all logical possibilities of control. Each node (any point at which two or more branches intersect) of the tree represents a unique point of execution of a function.

Both the design for a particular software system and the foundations for the HOS meta-language are based on six axioms. These axioms explicitly define hierarchical software control, where control is an undefined affect of one software object to another software object. Each affect is described by a mechanism (such as a language) which is effected at the next and only the next most immediately lower virtual layer.

The six axioms¹ are:

- axiom 1: A given module controls the invocation of the set of valid functions on its immediate, and only on its immediate, lower level.*
- axiom 2: A given module is responsible for elements of only its own output space.*
- axiom 3: A given module controls the access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate, lower level function.*
- axiom 4: A given module controls the access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.*
- axiom 5: A given module can reject invalid elements of its own, and only its own, input set.*
- axiom 6: A given module controls the ordering of each tree for the immediate, and only the immediate, lower level.*

On the basis of these axioms, many theorems have already 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 mapping, 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 functions which communicate with each other. For example, the output set of one function can be the input set of another function only if said functions exist on the same control level and are controlled by the same immediate higher node. Theorems exist for proper structuring of real-time multiprogrammed processes. For example, each module controls the priority relationships of its dependent processes. Not only are these priority relationships relative rather than absolute, but each process at a given level has a higher relative priority than each process***tree (a process and its dependents). From the axiom concerning the ordering of functions, we can state with confidence that if a particular process must precede another, that order will be preserved. For we can prove that if functions are defined with HOS, the ordering between members of any given process tree cannot be altered by an interrupt. This

* Our work, to date, has concentrated on the first three areas of HOS methodology.

** perform: provide one and only one element of the output set for a particular element of the input set.

*** a function that can be scheduled (invoked by a queuing mechanism)

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. From the axioms concerning data access, theorems exist which eliminate data conflicts between functions. In addition, there are theorems which relate the maximum completion or delay time for a process to a given interrupt structure.

By its very nature, the interface correctness of any given software system is proven if it adheres to the axioms of HOS. An interface is a common boundary between the specification at one node and 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 = 6 when x = 3.

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*. Thus, with HOS, we can determine where each performance test is required. In most cases, this question can only be validated by demonstration. When the assumption itself is an approximation, we 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 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 eliminate data conflicts. The boundary between two nodes is established by the unique nature of the node itself. It is from the unique ordering relationships between nodes that the relative timing between any two nodes can be established.

* 13% of all problems found during the APOLLO integration effort were due to performance specification errors*.

The rule of node uniqueness, and the theorems concerning self-control, determine the common boundary of function to function and variable to function. We use the relative ordering per level, frequency of operation of a function and the absolute time of execution of a function to determine the absolute timing boundary from one node to another. The boundary here is the maximum delay time or completion time for a process. Operator to operator boundaries are determined only by the software definition of a given software system; while operator to function boundaries are determined by the variables used to define the elements.

B. Meta-Language Principles

An HOS language is not primarily concerned with computer execution, but rather with reliable systems decomposition. If a system specification is computable, it can be represented within the HOS meta-language. If a system specification can be described by the HOS language, it 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) are statically checked by a compiler; interface specification 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** level can be performed by an automatic programmer. 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. If each controller is represented by a construct, and each construct follows all the rules; it will be possible to define reliable interfaces of a given system by merely using proper constructs. Since each construct is the node of an HOS structure, semantics checking for a system specification can be done by the language design analyzer by verification of axiomatic communication between nodes.

We can define a basic unit within an 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 meta 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

** sequencing requirements

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. Figure 2 illustrates two possible regroupings of the same function. Note that the regrouped subfunctions do not change in any way.

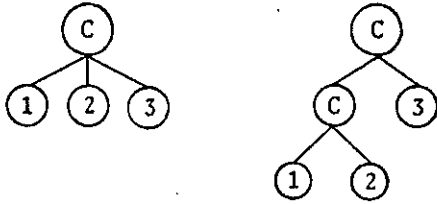


Figure 2: Recursive Regrouping of Construct Class, C

A function is decomposed by dividing the ordered pairs of that function into groups or not dividing the ordered pairs into groups. If we divide ordered pairs, we call that decomposition the partition class. If we do not divide, then we must transform. Either we can devise an explicit formula to transform the input set to an intermediate set and then to the output set, or we cannot. If we can devise such a formula, we call that decomposition class, composition. If we cannot, we must iterate the implicit formula until we find a solution to the intermediate set. If we can find such a solution, it can be represented by a combination of the partition and the composition classes. If we cannot find such a solution via a recursive formulation, we cannot decompose the original function.

The class, partition, is a construct class which performs a selection*. Within the partition class, constructs indicate decomposition of the parent function by division of the elements of the parental output set or elements of the parental input set. Characteristics of a partition that divide elements of the parental input set (Figure 3) are: (1) each subfunction produces a value for every variable of the output set; (2) a subfunction cannot alter the value of a variable of the input set; and (3) each input set of each subfunction is a proper subset of the parental input set (cf. System Design Section).

Figure 3 shows the control map representation of such a partition. Meta-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 higher order languages. 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 subfunctions B and C to pro-

vide a partition. 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.

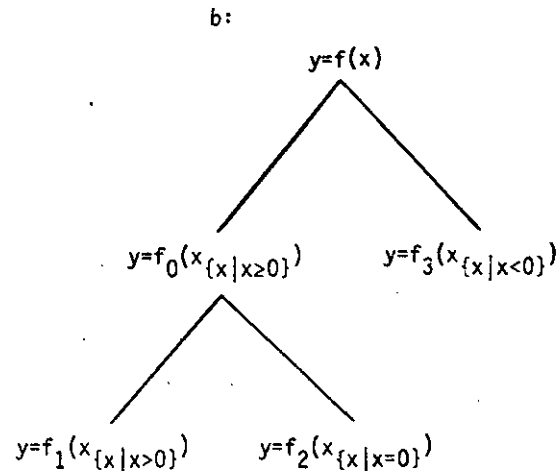
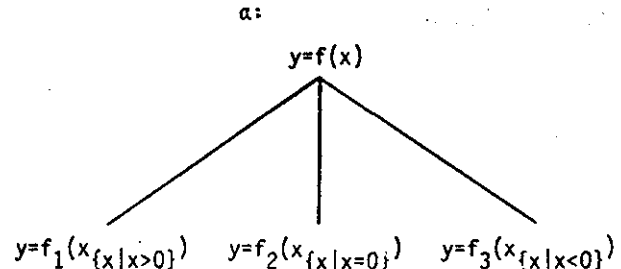


Figure 3: Partition Class: selecting members of the parental input set

Automatic programming of a partition which selects elements 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 similar nature can be used for error recovery (Figure 4), and error detection (Figure 5). Error recovery implies that a subfunction is provided as an alternate functional formulation for all values of the parental input set not acceptable to a primary formulation. In the event that no alternate formulation exists

* A selection divides the elements, a , where $a \in A$, into a partition, B , of A such that $b \in B$ and $A = \bigcup_{i,j=1} b_i$ and for two sets $b_j \cap b_i = \phi$.

(Figure 5), the null set for x is detected and the subfunction performance is rejected for that input value. For HOS, recovery from a detected error within a function is always provided for by its controller. The nodal set for each nodal family provides for: (1) alternate formulation of subfunctions in the event that a subfunction detects invalid input values; (2) rejection of its own input set in the event that no alternative subfunction formulation exists; or (3) restart mechanisms for a subfunction. Errors can be found explicitly by HOS techniques. These include those caused by hardware failures or human operator errors. This is possible because a true output variable is never assigned a value unless the result is known to be valid. An error signal is sent by a subfunction only in the event of an invalid input value. Detection of an invalid value may be via a direct test on the value or an indirect test by an inner local function. Thus a subfunction, which can never alter its inputs nor assign invalid outputs, is always restart* protected. In addition, 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). This is possible since we can restart any subfunction without destroying a software process. With few exceptions²⁹, most existing higher order languages 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.

However, for HOS, it is required that both a subfunction and its alternate formulation be explicit subfunctions of the nodal set of the controller. Therefore, a language construct for error recovery such as *on error B else C* would imply that alternative 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 alternative 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 *on error reject else C*.

As illustrated in Figure 5, error detection is formulated by the function which returns to its controller without providing a valid element of the output set. The controller may itself "REJECT" if no alternative for its subfunction is available or it may provide an alternative functional formulation (Figure 4). 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 proceed to "REJECT" to the caller of the function.

* automatic selective snapshot/rollback capability.

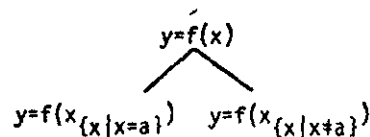


Figure 4: Error Recovery via Partition Decomposition

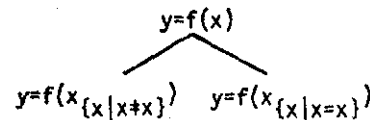


Figure 5: Error Detection via Partition Decomposition

Since an alternate to a function negates the intent of said function, in the event that more than one type of error can be detected within a subfunction, a reconfiguration of that subfunction is required for every distinct error type. This will be especially significant for multi-programmed structuring in that implicit errors in timing relationships, made invalid by a partial reconfiguration, are avoided.

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 (Figure 6) might be constructed as: *on error {B} case {B} else C*.

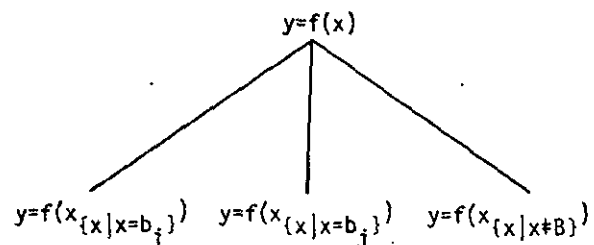


Figure 6
Multiple Error Recovery ($b \in B, b_i \cap b_j \cap \{x|x \neq B\} = \emptyset$)

In the case of some failures, the software can recover by restarting a process. A possible nodal set for such a situation might be *on error refresh A* where process A and all of its dependents would be terminated on every hardware "error" and then reinitiated (Figure 7). Here,

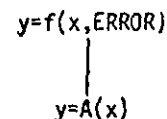


Figure 7: A Type of Restart Error Recovery

on event refresh A could be used to reinitiate any node of a system structure so that a human operator could dynamically interact at any specified level of a software system.

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, each subfunction is 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* 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. Here, variables of the input set are all input to one subfunction. That subfunction transforms those variables to an intermediate set of variables. In turn, many local transformations are possible until a value for all variables of the output set are assigned at one particular subfunction. This type of decomposition is best performed within one computer, since each subfunction is performed so that the flow of execution follows in linear sequence.

Composition of functions is enforced among HOS subfunctions by the requirement that variables of the input set are not equal to variables of the output set. Also, since the controller only specifies the subfunction relationships, the order of execution of the subfunctions is required to be independent from the sequence in which the subfunctions are referenced. Many existing block structured languages have incorporated sequential branching mechanisms, such as the CALL, to encourage use of composition techniques. Some languages even prohibit the use of uncontrolled branching, i.e., no GOTO's to encourage composition techniques¹⁰. But, existing block structured languages encourage modules at a given level to invoke other modules at that same level, i.e., the architecture 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. These noncomputable functions (Figure 8) are those for which there are undetectable dynamic loops. In order to determine if a function is truly computable, an exercise is made to determine if an iterative mechanism can be transformed to a recursive formulation either manually or automatically. Figure 8 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.

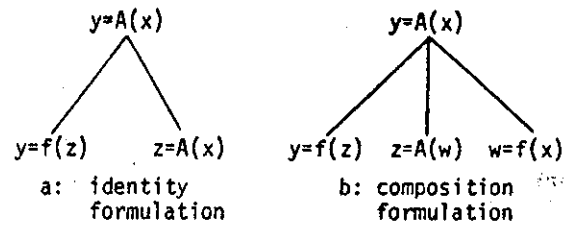


Figure 8: A Dynamic Loop

It is interesting to note that the iterative construct, the *while*, a basic construct of structured programming, is invalid for HOS. This construct exists in many programming languages today. The *while* is invalid because it can alter elements of its own input set and can assign intermediate values to variables of its own output set. For HOS we are restricted so that local initialization and consequent iteration is performed via a recursive formulation (Figure 9).

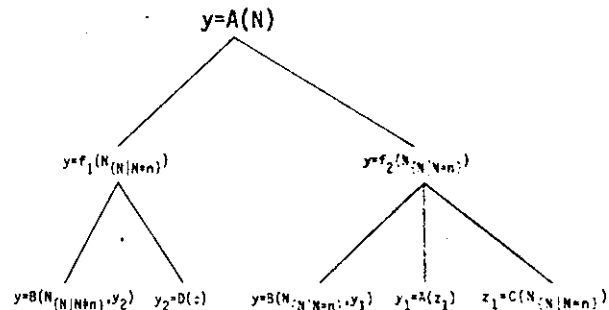


Figure 9: A Recursive Operator

Similarly, a real-time multiprogrammed iteration can be used to specify mutually dependent asynchronous processes. Here, the concept of assigning only valid elements of the output set prevents the interface error of another process interrupting the iterative process and using a shared variable which is being used to store intermediate values. The structure imposed on a system by the HOS axioms does not prevent functional decomposition. In fact, the unique node, the elimination of self-control, and the elimination of the dynamic loop assured by axiomatic decomposition makes it possible to decompose any computable system.

C. HOS Analyzers

The main function of an analyzer is to guarantee that a given HOS 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. Conventional real-time software systems cannot be exhaustively tested. With the design analyzer, any given software system, defined according to the rules of HOS,

* cf. ALGOL-68

can be exhaustively and statically verified for interface correctness. Interface errors in simplex, multiprogramming or multiprocessor systems are caused by data or timing conflicts. Given the HOS control system, it is now 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. Phase 1 will check a prototype software system design for violations of the axioms. In addition, a software system control map, showing all functional relationships of the prototype system, will be automatically produced. Phase 2 will both determine where performance testing is necessary, and show where it cannot be completed exhaustively. 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 to computer execution. The output from this phase could include information for the efficient use of hardware and output for the next most immediate lower architectural virtual layer.

If HOS methodology is maintained throughout a development process, a comparison of functions and their relationships can be made with analyzer results throughout the software phases of requirements, specifications, and implementation.

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. The structuring executive implements in real time, multiprogramming control constructs. The operating system requirements of the structuring executive handle aspects such as man/machine interface, hardware/software interface, error detection and recovery, real-time reconfiguration, dynamic allocation, self-analysis for timing and memory limitations, and an axiomatic analysis of the system for each new reconfiguration.

The real-time implementation of control constructs assumes: (1) an invocation mechanism, the *SCHEDULE*, to be used to invoke a module dependent on time, priority, event or cyclically dependent on time or event. In addition, the *SCHEDULE* assumes a relative priority indicating total order for each nodal family; (2) 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 option.

Figure 10 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 $e_i \in E$, 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 option. 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.

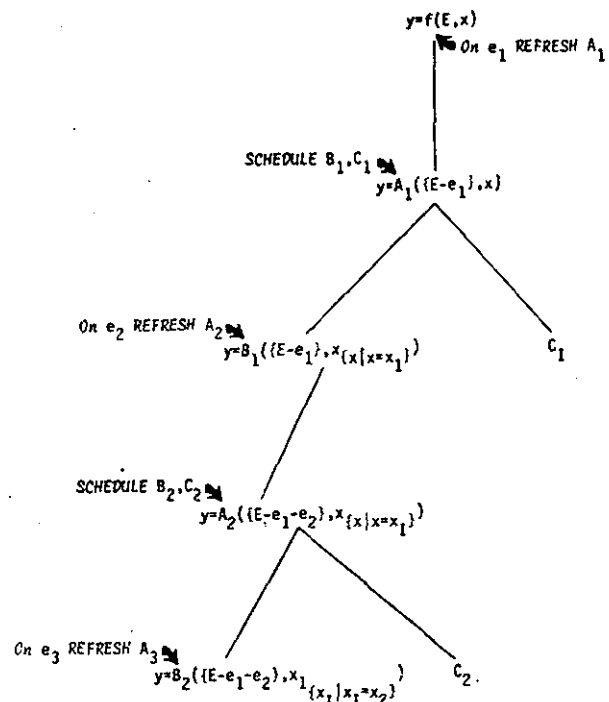


Figure 10: The Refresh Mechanism and Its Relationship to Man/Machine Interface

In addition, the human can select any reconfiguration of nodes in real time. Thus, the man/machine interfaces with each HOS system are multileveled, multistacked and can be multiconfigured. With such a system, an avionics pilot would not need to memorize a complicated crew selection list, since the software would provide automatic error detection and recovery.

Self-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 thrust 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) they both check for a violation of the axioms; (2) whereas the Design Analyzer determines where performance testing is necessary, the Structuring Executive Analyzer determines where human intervention is necessary; (3) they both provide for functional timing and memory analysis; and (4) whereas the Design Analyzer provides an automatic programmer, the Structuring Executive allows a reconfiguration of nodal families in real time.

III. System Design

Use of HOS methodology and its accompanying meta-language 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 specification.

Once a problem has been properly decomposed and automatically analyzed for interface correctness, system structure can automatically be 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^{10,11}, the application of HOS assures that a restructuring of an entire system 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 that system for fear of misinterpreting hidden implicit specifications. For example, it is important not only that programs be written in a sequential manner¹², but that the order of execution of the language statements can be determined regardless of that sequence.

Example 1:

a: $x = y$ (1)
if $z > 10$, $x = y^2$ (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 the meta-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: $y = f(x)$ (1)
 $x = f(z)$ (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 (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 B*
schedule A after B

The interpretation here is order independent. B will complete execution and then A will be initiated. Thus, this formulation is a valid 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:

if $x > 10$ then $y = h(x)$ else $y = g(x)$

The control structure for this partition implies that the controller is controlling its own function because the entire function is of the form $y=f(x)$ (Figure 11).

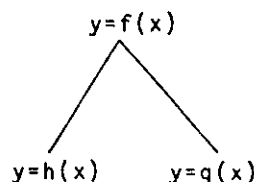


Figure 11: Immediate Self-Control

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.

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 that the controller function is controlling its own function.

proof of immediate self-control: 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.

Note that the partition of input elements (Figure 11) 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:

$z = x$
 if $z > 10$ then $y = h(x)$ else $y = g(x)$

Here, the control structure (Figure 12) shows, likewise, that this formulation of the entire function, $y = f(x)$, controls itself.

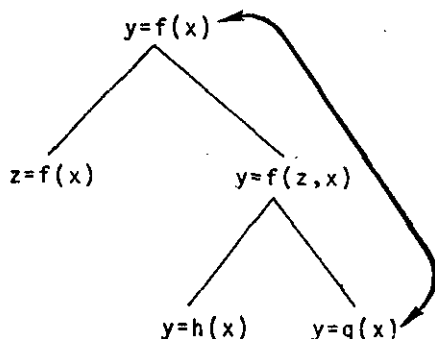


Figure 12: Indirect Self-Control

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 its own function.

proof of indirect self-control:



Figure 13: Proof of Indirect Self-Control

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, ref. 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 of 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, ref. 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.

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:

if $x > 10$ then $y=f(x_{\{x|>10\}})$ else $y=f(x_{\{x|\leq 10\}})$

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

The above discussion illustrates the types of problems that one can encounter by the use of a simple if then else statement common to many existing higher order languages. Imagine the potential difficulties that one can encounter with more

complicated statements if there is no formalized structure within which to work.

III. 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 performed⁴, and checklists for manual software disciplines³ 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^{13,14}. 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% of all the anomalies in the software were found by 'eyeballing'⁴, 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%! 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 process³; (2) the most sophisticated dynamic verification debugging process known at the time of APOLLO was used to verify software³; (3) there were no pure software problems found in the on-board flight software for all APOLLO flights¹⁵. In addition, we found that flexible software systems with fixed reliability modes are a key to managing software developments. A case in point is the APOLLO on-board asynchronous systems software. If the 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 disaster¹⁶.

The systems software of the on-board flight software for APOLLO consisted of a powerful set of asynchronous modules: the scheduler¹⁷, the display interface¹⁸ and the restart modules¹⁹. 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.

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 multistacked. That is, the astronaut was able to interface interactively with the on-board software at the top level of a mission phase as well as at the lowest level of a mission module in a hier-

archical 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 about a multiprogramming environment at any given instant in real time for any level in the hierarchical system for information consisting of many modules at one time. If an error occurred, all modules could be recovered or selectively recovered depending on job or time priority.

In addition, an AGC interpreter²⁰ 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^{21,22}; each module was designed and frozen early²³; no changes to the requirements of these modules were incorporated after they were frozen. That is, each of these systems software modules was truly on a lower virtual layer¹² 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* 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 software²³. 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 analyzer¹, there is no longer a need for a manual logical verification checklist²³. In addition, there will no longer be a need to dynamically verify software for interface problems. From these efforts evolved the formalized system, HOS, as it exists today. 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

HOS requires all controlled functions of a given controller to be on the next and only the next most immediate lower level. The controller establishes the communication network among its

* 73% of all problems found during the APOLLO integration effort were interface problems⁴.

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 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 2) control relationships can be determined for any given function 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 transport 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 variable of the output set, and the set of operations of the nodal set. When the variables are not specified, that is, when the 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 system²⁴. 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.

Proof of Uniqueness: Suppose the function $y=f(x)$ existed within a system at two different nodes. The two nodes could exist at the same

level (Figure 14a). 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 1.2¹.

If $y=f(x)$ appeared as a node inner to its own tree (Figure 14b) the module would be controlling its own function and is invalid via the proof of immediate self-control (cf. System Design Section). Likewise, any node that can be traced up the tree or down the tree from $y=f(x)$ must be of the form $y*f(x)$ (Figure 14c) via the proof of indirect self-control (cf. System Design Section).

Consider $y=f(x)$ at one node. Variable y must communicate on its own level or perform an output element for its controller (Theorem 3.4.2¹). If y were a local variable (Figure 14d), 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 (Figure 14e), it would also be invalid because y could only communicate as an input variable, in this case, by Theorem 3.4.2¹.

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 3.1¹).

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 $y=f(x,z)$ (Figure 14f), 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 $y=f(z)$ (Figure 14g), 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 $y=f(x,z)$ (Figure 14h) or $y=f(z)$ (Figure 14i).

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 3.4.4¹). Thus, in this case $y=f(x)$ could not exist outer to $y=f(z)$.

* Communication: the transfer of elements between functions.

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.

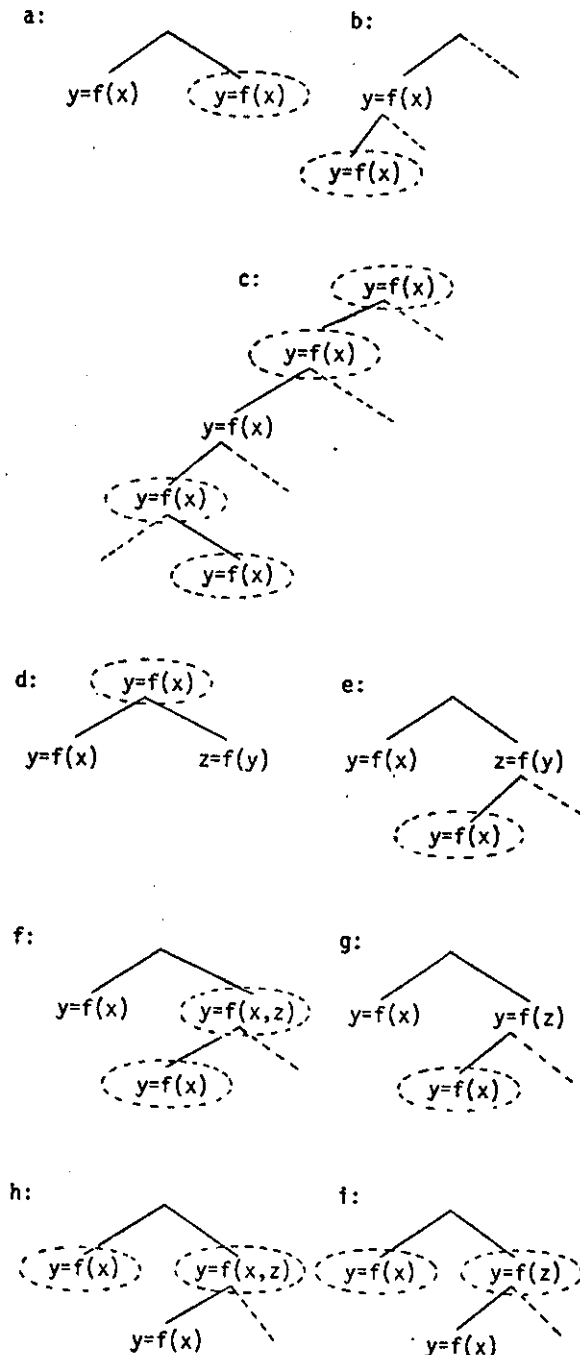


Figure 14: Proof of Uniqueness

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 $y=A(N)$ (cf. Figure 9) for HOS is essentially a partition 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, A. 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 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 functional modules remain fixed. At the specification and procedural level, the functions and interfaces are still maintained. With an automatic programmer or a translator, this process can be provided for automatically.

At the requirements or 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 predict the size of future efforts more accurately.

C. Frozen Module Management

Each software module goes through three phases of development²³: (1) The requirements phase defines the functions and their interfaces; (2) The specifications 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 requirements phase; (3) The final phase defines the actual execution code for the implementation of the specification 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 flowcharts which show an ordered sequence of program flow. For the Space Shuttle requirements integration effort, the automatic structured flowchart²⁴ (designed²⁴ and developed²⁵ at The Charles Stark Draper Laboratory, has replaced the manual production of flowcharts^{*}; (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 which are first introduced at the requirements phase in which variables and key words are updated with program changes.

Each module evolves from the requirements phase to the specifications phase to the code phase. The documentation for each module is also an evolution 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 hardware will be flexible enough to accommodate software 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.

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, (Figure 15) 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 ACS₁ and ACS₂. The lower level software modules, S₁ and S₂, must be given official approval before being allowed into the official assembly controlled by the ACS^{*}.

* Twenty out of 100 people were totally dedicated to the manual production of flowcharts in the Program Development and Verification Group of The Charles Stark Draper Laboratory during APOLLO.

** Top-down/Bottom-up official building process (cf Reference 24).

Likewise, ACS₁ manages the official assembly by monitoring the activities of managers ACS₁₁ and ACS₂₁. 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^{26,27,28}. For example, if the chief programmer concept were to be used for personnel management; the ACS, ACS₁, ACS₂, ACS₁₁, and ACS₂₁ 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.

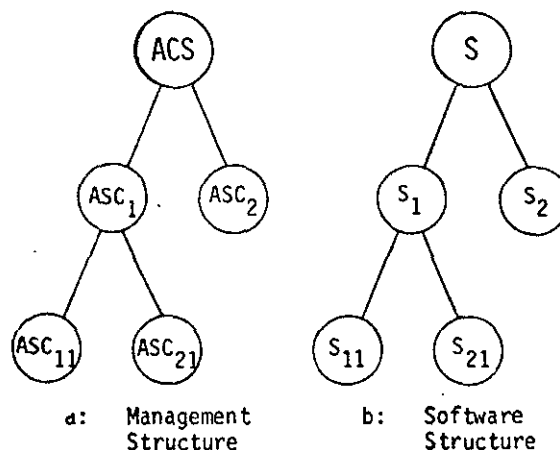


Figure 15

The Management Structure and Its One to One Correspondence to the Software Structure

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.

Since structured programming concepts were first introduced¹², questions concerning timing and memory efficiency have been raised. The same questions are brought up again for HOS, since HOS does not violate structured concepts*. 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 requirements/specifications level using the axioms. The system should be designed as if it were to communicate with an asynchronous executive, 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.

The major results determined during the requirements level design process²⁹ are: (1) the true functions and their interfaces; and, (2) the timing and memory requirements for future implementation. During the requirements design process it is possible, therefore, to either define hardware requirements for computation or to determine if the system can be built with available "off-the-shelf" hardware.

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

Certain design aid tools in support of the basic HOS methodology are currently operational. For example, our automatic structured flowchart and data intersection analysis tools are now being used by several organizations involved in the

Shuttle project^{30, 31, 32, 33}. Specification language ground rules, structuring executive operating system design, and various other verification and documentation aids are in the preliminary design stage.

We are presently involved in building a prototype HOS Analyzer**. As a temporary step, we are translating HAL language³⁴ source code to an HOS control description. The control description is the language independent input to the analyzer (Figure 16). 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 regardless of the number of HAL statements required to implement such a function. Without such an aid, we have defined a particular grouping of statements to represent an HOS function. Each grouping is blocked so that the control descriptor can recognize the pattern. These groupings are also limited to basic functional forms because our intent here is not to define the meta-language, but to build an analyzer. Each control description provides subfunction interfaces which include: function type, priority relationships, and input and output variables. A preliminary HOS control map can now be generated for the invocation of closed blocks³⁵. A complete control map which details functional relationships to the statement level is now in development.

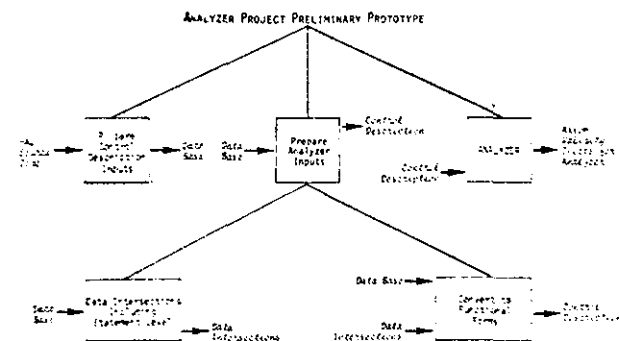


Figure 16: Analyzer Prototype

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

* General structured programming concepts can, however, violate HOS rules.

** Internal Draper Laboratory Independent Research and Development project "Software Control Analyzer."

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 effect 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 Higher Order Software 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*. A given system, described with an HOS meta-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 meta-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 on-going 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. With the axiomatic approach of Higher Order Software now available,

* 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.

we have been able to already 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. And, those "other" computable systems are, after all, software.

Acknowledgement

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 authors would like to express appreciation to Adele Volta for editing assistance.

The publication of this report 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.

References

1. Hamilton, M., and Zeldin, S., "Higher Order Software Techniques Applied to a Space Shuttle Prototype," In *Lecture Notes In Computer Science*, Vol. 19, ed. G. Goos and J. Hartmanis, pp. 17-31. New York:Springer-Verlag. As presented at: Programming Symposium, April 9-11, 1974, at Proceedings, Colloque sur la Programmation, Paris, France.
2. Turing, A.M., "On Computable Numbers with an Application to the Entscheidungs Problem," In *Proceedings of the London Mathematical Society Proceedings*, Ser. 2, Vol. 42, 1936.
3. Hamilton, M., "Management of APOLLO Programming and Its Application to the Shuttle," CSDL Software Shuttle Memo No. 29, May, 1971.
4. Hamilton, M., "Design of the GN&C Flight Software Specification," CSDL Document No. C-3899, February, 1973.
5. Millard, R., Internal Draper Laboratory Study of APOLLO (1969-1972).

6. Fair, L., et al., SDC Technical Memorandum TM-2314/000/00A, p.57, *Planning Guide for Computer Program Development*, May, 1965.
7. Boehm, B.W., Rand Memorandum RM-6213-PR, "Some Information Processing Implications of Air Force Space Missions: 1970-1980, January, 1970.
8. Randell, B., "Research on Computing System Reliability at the University of Newcastle Upon Tyne, 1972-73," Computing Laboratory. Technical Report Series No. 57, January, 1974. Pub. University of Newcastle Upon Tyne, Claremont Tower, Claremont Road, Newcastle Upon Tyne, Scotland.
9. Miller, J., et al., *CS-4 Language Reference Manual*, Intermetrics, Inc., Cambridge, Mass.
10. Wulf, W.A., et al, "Bliss: A Language for Systems Programming," Comm. ACM 14,12, December, 1971.
11. Schwartz, J., *On Programming, An Interim Report on the SETL Project*, Computer Sci. Dept., Courant Inst. Math. Sci., New York University. Installment 1. Generalities (1973), Installment 2. The SETL Language and Examples of its Use (1973).
12. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., *Structured Programming*, Gen.Ed. C.A.R. Hoare, New York:Academic Press, 1972.
13. Hamilton, M., and Zeldin, S., "MERCURY Statistical Analysis," Draper Lab. MERCURY Memo No. 24, April, 1973.
14. Brodeur, N., "MERCURY Statistical Analysis Report Revisited," Draper Lab. MERCURY Memo No. 34, November, 1973.
15. Hamilton, M., "First Draft of a Report on the Analysis of APOLLO System Problems During Flight," Draper Lab. Shuttle Management Note No. 14, October, 1972.
16. Hamilton, M., "The AGC Executive and Its Influence on Software Management," Draper Lab. Shuttle Management Note No. 2, February, 1972.
17. Laning, H., AGC Program Sundisk, Rev. 267, Draper Lab., NASA 2021108-011, November, 1967.
18. Hamilton, M., AGC Program Sundisk, Rev. 267, Draper Lab., NASA 2021108-011, November, 1967.
19. Lickly, D., AGC Program Sundisk, Rev. 267, Draper Lab., NASA 2021108-011, November, 1967.
20. Muntz, C., "Users Guide to the Block II AGC/LGC Interpreter, Draper/MIT Document No. R-489, April, 1965.
21. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, 12: 1053-1058, December, 1972.
22. Parnas, D.L., "A Technique for Software Module Specification with Examples," Communications of the ACM, Vol. 15, 5:330-336, May, 1972.
23. Hamilton, M., and Zeldin, S., "Higher Order Software Requirements," Draper Lab. Document No. E-2793, August, 1973.
24. Hamilton, M., and Zeldin, S., "Top-down/Bottom-up, Structured Programming and Program Structuring," Rev. 1, Draper Lab. Document No. E-2728, December, 1972.
25. Daly, W., "Automatic Flowcharts," Draper Lab. MERCURY Memo No.53, March, 1974.
26. IBM, "Chief Programmer Teams Principles and Procedures," Federal Systems Division Report FSC-71-5108, June, 1971.
27. Weinberg, G.M., *Psychology of Computer Programming*, London:Van Nostrand Reinhold, 1971.
28. White, J.R., and Presser, L, "A Tool for Enforcing System Structure," In *Proceedings of the ACM*, 1973.
29. Hamilton, M., "A Discussion of Higher Order Software Concepts as They Apply to Functional Requirements and Specifications," Draper Lab. Document No. P-019, Dec., 1973.
30. The Charles Stark Draper Laboratory, Inc., "Submittal of Structured Flowcharts of the Approach and Landing Test (ALT) GN&C Design for the Space Shuttle Orbiter GN&C Integration Task," January, 1975, Cambridge, Mass.
31. Rockwell International Corporation, (1) "Space Shuttle Orbiter Approach and Landing Test, Level C, Functional Subsystem Software Requirements Document Guidance, Navigation and Control, Part A, Guidance," Appendix A, November, 1974. (2) As above, Part B, Flight Control, Downey, California.
32. Honeywell, Inc., "Horizontal Flight Digital Autopilot Requirement Definition," ED 21532, PRD No. UA17, Vol. II, Sections 5-7, July, 1974, Sarasota, Florida.
33. 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, Federal Systems Division, Houston, Texas, February, 1975.
34. Lickly, D.J., et al., *HAL/S Language Specification*, Intermetrics, Inc., Cambridge, Mass.
35. Goddard, G., "Control Map Development," Draper Lab. Software Control System Analyzer Memo No. 4, January, 1975.