HIGHER ORDER SOFTWARE TECHNIQUES

APPLIED TO A SPACE SHUTTLE PROTOTYPE PROGRAM

Margaret Hamilton and Saydean Zeldin

The Charles Stark Draper Laboratory

Cambridge, Massachusetts - U.S.A.

INTRODUCTION

It is inconceivable that mathematicians would solve or communicate mathematical problems without the formalization of mathematics or the language of mathematics. Yet, software has been developed without its own consistent set of formal laws or a software meta-language of its own. (The meta-language is not to be confused with currently existing higher order languages (HOL) used by programmers to code computer problems.) We define here a formal system so that within the framework of its axioms we can develop reliable software and yet not sacrifice flexibility needed to define and perform complex multi-programmed software.

We define Higher Order Software (HOS) as software expressed with meta-software and conforming to a formalized basic set of laws. HOS begins with problem formulation and ends with verified code. Performance formulation, logical formulation, verification, documentation, implementation, and management are all integral parts of HOS. HOS allows for a multi-programmed software system that 1) can automatically be changed during development or in real-time; 2) can interface with operational systems so that operational systems are an extension of the software; 3) provides error detection and recovery methods which protect both the software system and the operational system (man) from exceeding the limits of the environment within which they must operate; 4) can provide for the structuring of multi-programmed modules in real-time; and, 5) can be exhaustively tested for interface correctness without dynamic verification.

The concepts defined here combine proven APOLLO techniques[1], structured programming techniques[2, 3] and concepts obtained by theoretical considerations presented here. These concepts are now being developed and verified with an on-board prototype Space Shuttle Flight program. This program is a large-scale, real-time, multi-programmed, man-machine interface software system.

The ultimate aim is to be able to define any system with a software meta-language and a basic set of laws so that all the interfaces can be exhaustively tested. Further, change to a component in the system is not to affect non-related components. A parallel aim is to define an automatic system analyzer which will determine all inconsistencies in a given software definition.

RATIONALE FOR A FORMAL CONTROL SYSTEM

Multi-programmed software which interfaces in real-time with human and hardware systems has been most difficult to prove correct. The solution to this problem is not a trivial matter. The APOLLO on-board software had to contend with a system which had all of these characteristics. The executive system was asynchronous, but allowed for synchronous events or processes. It provided the astronaut with an asynchronous multi-level, multi-stacked, man-machine interface system. It further provided for sophisticated error detection and recovery for both hardware and software problems. These capabilities were crucial to the success of APOLLO software during both development and during flight. One mission proved this fact to the world in a dramatic way[4].

In the process of developing APOLLO and Skylab on-board software, reliable techniques for building large software systems were established. No known errors in the on-board software system occurred during all of the APOLLO and Skylab space flights[5]. The cost, however, of approaching this reliability was excessive. Analysis of software problems encountered during development suggested new and more formalized techniques to complement those proven and already established.

Further, during the development of the prototype Shuttle program, it became clear that specific rules governing the validity of functions and modules were necessary. Although specific tasks were assigned to separate groups of people, it was difficult to completely separate all dependencies between algorithms. For example, the navigation group determined that a history of mission phase was inherent to the navigation task of estimating the on-board vehicle state. It also became clear that if we were to attempt to prove the entire system to be correct, the use (control) of a module must be completely separated from the module itself.

For example, each task group assumed there was a necessity to interrogate system switches at their task level. This resulted in the development of unified modules. Even though within each individual task there was a structured sequence of operations, each unified module caused a proliferation of test cases system wide. Let us show a unified system (Figure 1). Mission phase, M_1, invokes navigation, N, with instructions to invoke N_1 and N_2. Mission phase, M_2, invokes N with instructions to invoke N_3 and N_2. Not only would each change to N imply re-testing of the M_1 interface, but would also imply re-testing the M_2 interface. Likewise, a change to M_1 or M_2 could require a reformulation of N. Here, one user would not require all sub-modules of N, yet N is unified because each user requires a particular combination of these sub-modules. Let us replace the unified concept with an example of HOS control (Figure 2). Here, M_1 invokes N_A with no instructions as to the manner in which N_A should invoke its functions. Likewise, M_2 invokes N_E consisting only of sub-modules N_2 and N_3. (Of course, N_2 can be a common computer function so that the formulation of the function appears only once within the computer.) A unified module can occur in a non-multi-programmed environment. However, in a multi-programmed environment, the interfaces of a unified module would be far more complex.

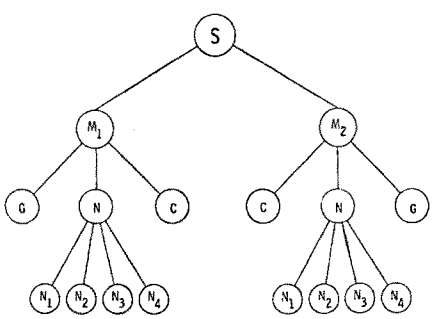

Figure 1: A Unified Example

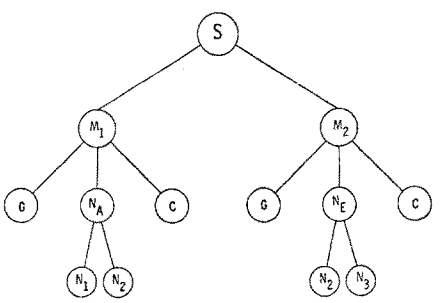

Figure 2: An Example Of HOS Control

The multi-programmed prototype program requires that a major task be used any number of ways depending on the environment of the mission phase. For example, the rate of convergence of a cyclic guidance scheme is dependent on the stability characteristics of the vehicle. Thus, the frequency of invocation of guidance is higher during atmospheric ascent than during orbital configurations. In order for the guidance scheme to control its own frequency, it must not only become acquainted with other system functions, but may then find it necessary to control tasks needed for functions which do not interact with guidance at all.

The concept that upper level modules control lower level modules has been most important to the development of rules governing modularity. Simplification resulting from enforcement of these rules has made it possible to completely separate system design from application algorithms. This is most important in the case of the actual Shuttle program, since the life expectancy of the program is at least twelve years. Mission dependent sequences must be obtained without modification to application modules. Application modules must be able to be "collected" from a library without retesting the mission module or the computer configuration.

Let us define a system in which every component has a unique definition as to both how it controls and how it is controlled. We can avoid a common programmer's problem: define a controller for a function which controls itself. This is a special case of Russell's well-known paradox: Suppose S to be the set or class of sets which are not elements of themselves; if $S \notin S$ then $S \in S$.

By separating the control of a function from the function itself, we can create functions free of hidden assumptions, directly apply mathematical reasoning to the function itself, and describe and discuss the formal control system as a separate process.

The formulation of the control system is developed by a basic set of six axioms. Each axiom assumes a basic mode of control.

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

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.

A module has the responsibility to perform a function. For that purpose, the module controls functions only on the immediate lower level. This is done by invocation (CALL* or SCHEDULE), by assignment of access rights to the input and output sets, and by the determination of the ordering of the functions on that level. Since every function receives input from and produces outputs for its

---

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

controller, the module and function are relative definitions. Thus, the upper level is a module with respect to its immediate lower level functions. The lowest level of any tree contains only functions. The highest level node of the entire tree structure is only a module.

The following symbols are used in the discussion below:

$\forall$, for every	o, controls
$\wedge$, logical 'and'	$\not o$, not controls
V, logical 'or'	$\overset{\uparrow}{o}$, interrupt
$\in$, element of	$\overset{\uparrow}{\not o}$, not interrupt
$\subset$, subset of	$\exists !$, there exists a unique
$\cup$, union of	$a \rightarrow b$, logical 'if a then b'
$\{\}$, set of	

Definition: The formal control system (Figure 3) is one in which each module, S, has a unique identification

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

$n_i m_i$ 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. At each level there are assumed N_i 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,

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

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[2]. 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 function, C, cannot invoke another function, D, which invoke function C for then C would be controlling itself.

$$C \not o (D\ o\ C)$$

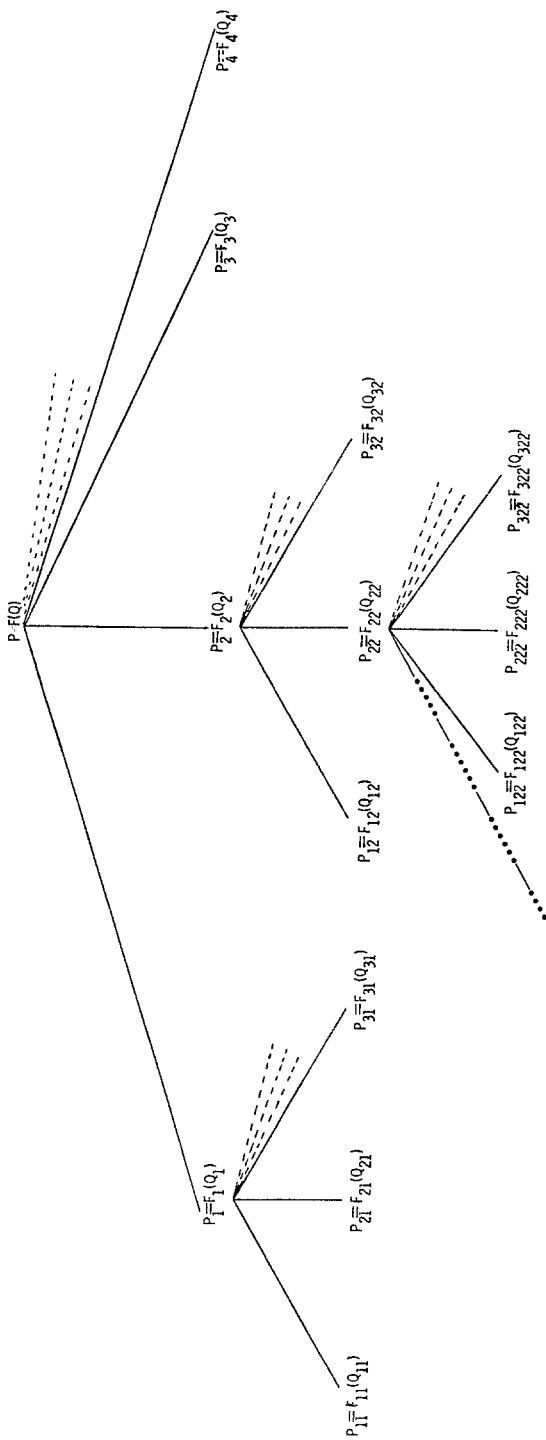

Figure 3: The Formal Control System: $S_{n_i;m_i} \equiv [P_{n_i;m_i} = F_{n_i;m_i}(Q_{n_i;m_i})]$

Corollary 1.1.1: A logical antecedent cannot be assigned by its consequent if a repetitive relationship controls the execution of the antecedent.

e.g., If function C is comprised of 'if G then D'
G cannot be assigned by D under the condition
stated above.

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 $level_{i+1}$, $F_{n_{i+1}n_im_i}$, is extraneous. The extraneous function is a direct violation of axiom 1, for if the function is not removed, $S_{n_im_i} \not\phi F_{n_{i+1}n_im_i}$.

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 (c.f. Figure 1).

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, $S_{n_im_i}$, is responsible for elements of the output space, of only $P_{n_im_i}$, such that the mapping $F_{n_im_i}$ ($Q_{n_im_i}$) is $P_{n_im_i}$. That is,

$$\forall j \forall n_im_i \; \exists! S_{n_im_i}, [(S_{n_im_i} \circ P_{n_im_i}) \wedge ((n_jm_j \neq n_im_i) \rightarrow S_{n_im_i} \not\phi P_{n_jm_j})] \quad (2)$$

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

$$\forall j \forall n_{i+1} \in N_{i+1} \; \exists! S_{n_im_i}, [(S_{n_im_i} \circ Y_{n_{i+1}n_im_i}) \wedge ((n_jm_j \neq n_{i+1}n_im_i) \rightarrow S_{n_im_i} \not\phi Y_{n_jm_j})] \quad (3)$$

NOTE: If any two modules, $S_{n_im_i}$ and $S_{n_jm_j}$, 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.

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

Thus, the module, $S_{n_i m_j}$, 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 output set of a function cannot be 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 output set of one function can be the input set of another function only if said functions 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. $y = f_i(x)$ 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.

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

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.

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

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

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

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 on the immediate, lower level.

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

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 $\{T_{n_{i+1} n_i m_i}\}$.

NOTE: The ordering of a set of functions determines the invocation so that a sequence of the said functions is established.

A process, ϱ, is a function scheduled to occur in real-time.

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 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.

$$\forall j \forall k \forall n_{i+1}, w_{i+1} \in N_{i+1} \quad \exists! S_{n_i m_i}, [n_{i+1} \neq w_{i+1} \rightarrow (S_{n_i m_i} \circ (\varrho_{n_{i+1} n_i m_i} \overset{\wedge}{\delta} \varrho_{w_{i+1} n_i m_i}) \wedge ((n_j m_j \neq n_{i+1} n_i m_i) \rightarrow (S_{n_i m_i} \phi (\varrho_{n_j m_j} \overset{\wedge}{\delta} \varrho_{n_k m_k})))]$$

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 explicitly.

Corollary 6.4.2: If a process, $\varrho_{n_i m_j}$, 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 $\{\varrho_{n_{i+1} n_j m_i}\}$, 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 multi-program 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) → 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 n^{th} process has a frequency, f_n cycles per second, such that the period, t_n, for the n^{th} process is $1/f_n$, and the total maximum execution time for that process is Δt_n. The j^{th} 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.

REAL-TIME CONTROL SYSTEM

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 non-deterministic.

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 operator-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 non-deterministic 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 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 in 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 non-dependent process, and all of its dependents. Consider Figure 3 to be a subset of the Shuttle system, S.

S_1, S_2, S_3, S_4 - are the ascent mission phase, atmospheric entry mission phase, astronaut display, and abort mission phase respectively;

S_{11}, S_{21}, S_{31} - are the ascent guidance (G_A), ascent navigation (N_A), and ascent vehicle control (C_A) respectively;

S_{12}, S_{22}, S_{32} - are the entry guidance, G_E, entry navigation, N_E, and entry vehicle control, C_E, respectively;

S_{121}, S_{211} - are navigation state extrapolation, and measurement incorporation respectively.

Thus, for example, in Figure 3, 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} 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 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 is 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 that data is 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 non-operational 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 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 non-multi-programmed function can never be terminated by an outside controller. Thus, in this case, one would be advised to use a DO FOR WHILE instead.

Errors in traditional MP systems are either caused by data or timing conflicts.

The application of HOS prevents both types of conflicts. The axioms imply that the non-local variable is explicitly controlled. The restrictions imposed are consistent with reports of others[6, 7] who have found that implicit control of non-local variables tend to produce programs that are difficult to understand. The methods for proving each computer function to be valid is less cumbersome, for there is no longer a possibility of side effects (inputs cannot be altered) or redefinitions (outputs are explicitly subsets of the results of invoked functions). Axiom 6 prevents relative timing conflicts between processes and guarantees time-critical events. (Those events or cyclical processes which are time-critical and therefore synchronous in nature, are scheduled as the highest priority process in the system.) Finally, when memory or timing limits of the computer are reached, the dynamic scheduler resurrects the software system by terminating lower priority processes and leaving only the highest priority processes in the queue. With these concepts, it is immaterial whether breakpoints occur at every basic machine instruction, every HOL statement or via some other method. The key consideration is the absolute minimal time required to service highest priority processes and highest frequency interrupts.

Conventional real-time software systems can have infinite interfaces, or a very large finite number of interfaces to verify. This verification has been traditionally performed by exercising the software system on a dynamic basis. Such a system cannot be exhaustively tested. Given the HOS control system, it is now possible to design a system with a small finite number of logical interfaces to verify. These interfaces can be exhaustively tested by analyzing a given software system on a static basis. Thus, the more expensive methods of simulation and/or dynamic verification can be limited to unit performance testing.

The analyzer verifies the system for correctness by checking for all violations of the axioms. Second, the analyzer provides performance interface testing for absolute timing consistency. It is only necessary to provide timing analysis between functions on the same control level (c.f. theorem 6.10). Third, in a system which is not deterministic, the analyzer would predetermine the potential data conflicts that could happen in real-time by distinguishing between operationally assigned relative priorities and fixed assigned relative priorities. (Of course, in real-time, the system scheduling algorithm would automatically prevent the conflicting processes from executing in parallel.)

Included as input for the analyzer, then, is the information needed for each invocation of control: component definitions, processes scheduled, cyclic timing, absolute time, and fixed and operational priority relationships. In addition, for the analyzer performance interface testing, predicted (and eventually actual) time for each function are provided as input.

DEVELOPMENT OF THE SHUTTLE PROTOTYPE PROGRAM

The development of the prototype software system is both determined and facilitated by formalized definitions since they are the same for all the well-known disciplines of design, implementation, verification, management and documentation[8].

The Shuttle prototype program is consistent with the HOS phases of development. The first phase defines a general problem. This includes those parts which are performed by and interface with a computer. The second phase defines the

algorithms for the functions performed by a computer. These first two phases are hardware and HOL independent. The third phase evolves the algorithms to include architectural aspects (hardware, HOL, etc).

In a real-time system, an asynchronous executive, which handles interrupts with unique priorities, is able to maintain the natural state of the real-time functions and their relationships for all phases of software development. It is for this reason, the real-time prototype program is a multi-programmed system.

Each completed module in the system is entered into an official library as soon as it is verified and approved by designated experts of the appropriate area of expertise. The "assembly" control supervisor approves each module for inclusion into the official library. These official modules can be collected, top-down, to form a given defined system from each HOS component definition. (Conceivably, a subset of the development collector in a flight computer could rearrange components to provide for real-time changes to the software system.)

The HOL, HAL[9], has provided us with the basic constructs needed (i.e., IFTHENELSE, DOCASE, DOFORWHILE, CALL, and SCHEDULE*) to aid in coding HOS software. The structured flowcharts[10] (Figure 4) help the programmer to arrange code in a linear sequence. The functional control map (Figure 3) helps the programmer to arrange functions according to control levels.

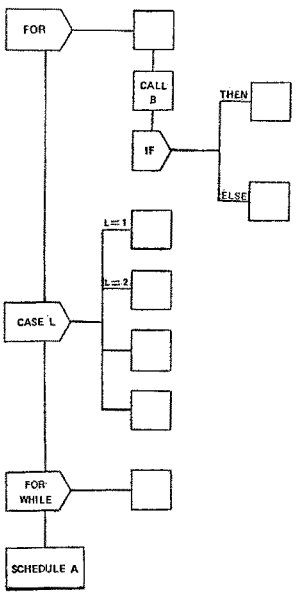

Figure 4: Functional Structured Flowcharts

(Flow is assumed to return in line at the completion of every object of each decision.)

*HAL does not presently have the capability to provide priority relationships as discussed above.

The modules are presently verified by a statement level closed loop simulator with environment modules for the flight computer, vehicle, astronaut, universe, etc. It is our aim to minimize the necessity for dynamic simulation by use of the formalized HOS language, HOS axioms and the tools which include the collector, the analyzer, and the structuring scheduler algorithm.

CONCLUSION

HOS concepts are now being applied to a prototype Shuttle flight software system. By providing software with its own meta-software and its own universal system, not only can we produce reliable systems, but we can also communicate these systems to others. Development and real-time flexibility are not sacrificed. The only limitations applied are those which prevent a potential error from occuring, i.e., the only flexibility missing is that which allows for flexibility of errors.

ACKNOWLEDGEMENT

The authors would like to express appreciation to Donald DeVorkin for his critical review of this paper, and to Adele Volta for editing assistance.

This paper was prepared under Contract NAS9-4065 with the Lyndon B. Johnson Space Center of the National Aeronautics and Space Administration.

The publication of this paper does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained herein. It is published only for the exchange and stimulation of ideas.

REFERENCES

1. Hamilton, M., "Management of Apollo Programming and its Application to the Shuttle," CSDL Software Shuttle Memo No. 29, May 27, 1971.

2. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., Structured Programming, Academic Press, London and New York, 1972.

3. Mills, Harlen, "Top-down Programming in Large Systems," Courant Computer Science Symposium, June 29 - July 1, 1970.

4. Hamilton, M.,"Computer Got Loaded," Datamation, March, 1971.

5. Hamilton, M., "First Draft of a Report on the Analysis of Apollo System Problems During Flight," CSDL Shuttle Management Note No. 14, October 23, 1972.

6. Wulf, W.A., and Shaw, M., "Global Variable Considered Harmful," Sigplan notices, February, 1973.

7. Knuth, D.E., "The Remaining Trouble Spots in Algol 60," CACM 10, October 10, 1967.

8. Hamilton, M., Zeldin, S., "Higher Order Software Requirements," CSDL E-2793, August, 1973.

9. Intermetrics, Inc., "HAL/S Language Specification," January, 1974.

10. Hamilton, M., Zeldin, S., "Top-down, Bottom-up, Structured Programming and Program Structuring," Rev. 1, CSDL E-2728, December, 1972.