General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by NASA STI Compliance and Distribution Services

77-1446

Verification of an Axiomatic Requirements Specification

M. HAMILTON* and S. ZELDIN**
Higher Order Software, Inc.
843 Massachusetts Avenue
Cambridge, MA 02139

Abstract—**Verification of a system involves the identification of redundancies, incompleteness and inconsistencies of a system definition, description, implementation and execution. A system, defined formally, so that system properties can be understood in the same way by people and machines, can be analyzed automatically with respect to both syntax and semantics. With a system so defined, many aspects previously associated with verification can be eliminated. An example specification is used to demonstrate the properties of a system definition which supports elimination of unnecessary verification, maximum use of static verification, and minimum use of dynamic verification.**

I. INTRODUCTION

DESIGN AND VERIFICATION take place throughout a system development process. *Design* means to think. *Verify* means to think back. For each step of design there should be a counter-step of verification. At times, the process of design is one and the same as the process of verification. This occurs when certain design characteristics are included for the purpose of preventing unnecessary verification. In such a case, some types of verification requirements are designed out of the system. What is left is the second order verification which determines if unnecessary verification requirements with respect to a design have truly been eliminated, and then a need to verify only that which is truly part of the original intent of the design.

Many engineers talk about a desire to improve their own design techniques. These design techniques include those techniques for producing the design for a solution to a particular problem as well as the design of the process which will verify that solution. More often than not, these engineers appear to be talking about a different design process since they are involved in different types of systems or different phases of development within a given system. Actually, they are applying the same process (i.e., design) to a different "application."

From the point of view of a typical system development process, design could be the process of going from a concept to a set of requirements, a set of requirements to a set of specifications, a set of specifications to a set of code or a set of code to a computer. In every one of these processes each designer considers himself to have the task of preparing his design to eventually reside in a computer or machine environment. One of the problems with this approach is that a designer either worries unnecessarily about design considerations not relevant to his own process or he might leave out design considerations thinking someone has already taken care of them or that someone will take care of them later. The important consideration for each designer is to worry about designs which he should worry about and only those designs he should worry about. Each designer goes through the same process, but each designer should be applying that process to a different phase of the overall application. Thus the inputs and outputs of his design process should be both unique and self-contained.

Other than a good deal of insight, a successful designer has necessary and sufficient knowledge about his problem, an understanding of the nature of a design process, an understanding of the nature of the reverse of a design process (the verification process), and a means to effectively perform a reliable and workable implementation of his design.

It is desirable for a designer to have a methodology to support him in a design process. However, although a methodology can support a designer, it can never replace *him*. A tool can be developed to replace some of his functions with respect to himself, the designer, and even to replace all of his functions on a particular project. However, the designer still has the prerogative to create new designs and to design new uses for the same tool or new tools for different uses.

There are many methodologies today whose intent is to provide standards and techniques to assist the engineer in the design and verification process [1]. The developers of these methodologies are all proponents of reliable designs. And, most methodologies advocate similar techniques towards this aim. For example, it is a commonly accepted idea that it is beneficial to produce a hierarchical breakdown of a given design in order to provide more manageable pieces to work with. And, there are variations between methodologies. Some emphasize a concentration on data flow as opposed to functional flow [2] [3] [4] [5]; others emphasize just the opposite [6] [7] [8]; others indicate that both functional and data flow are of equal importance [9]; others emphasize documentation standards [10] [11]; others emphasize graphical notations [12]; and still others emphasize semantic representation [13].

There are certainly positive aspects in many of these methodologies and, in particular, in what they are trying to obtain. To be effective, however, a methodology should have techniques and rules for the purpose of defining systems which are consistent and complete. But these techniques and rules are useful only if they are within themselves consistent and complete, both with respect to each other and to the systems to which they are being applied. With a complete and consistent methodology, a system can be defined formally so that all systems which communicate with a given system can understand that system in the same way.

In order to define a complete system, a methodology must have the mechanisms to define all of the relationships which exist in a system environment [14]. This includes communication within and between systems and the resource allocation which provides for that communication. Thus not only must all data, data flow, function and functional flow be able to be explicitly defined, but the relationships (and control of the relationships) between data and data, between function and function, and between data and function must be able to be defined within any given system environment.

A methodology should have a mechanism to provide modularity in the formal sense. That is, any change should be able to be made locally and if a change is made, the result of that change should be able to be traced throughout both the system within which that change resides and throughout other systems communicating with that system.

A methodology should have the mechanisms to communicate and resource allocate a formally defined system in a manner which is transparent to the engineer, the user of these mechanisms. That is, even though the semantics of a system definition are formal, the syntax describing that system should be as flexible and as close to the "language" of the engineer as long as it contains the necessary information to be used in the system description.

*President, Higher Order Software, Inc.
**Vice President, Higher Order Software, Inc.

Copyright © American Institute of Aeronautics and Astronautics, Inc., 1977. All rights reserved.

We will discuss here the properties of the methodology of Higher Order Software (HOS) as they relate design and verification to the requirements we have set forth for methodologies in general. As an illustration of these concepts, we include a set of specifications for one of the development phases of a satellite navigation system, NAVPAK [15][16]. The problem is to define a set of specifications from a set of requirements which will be used to implement NAVPAK in a computer environment. The problem is a typical software system problem in that the requirements not only include the incorporation of navigation algorithms, but they also include constraints imposed by human users, sensors, available off-the-shelf software packages and an existing set of hardware.

II. Some Preliminaries on System Specification

An *object* is an existence of something. A *system* is an assemblage of objects united by some form of regular interaction or interdependence. The mechanism by which these objects are united can make the difference between a system which is reliable and one which is not. In HOS the objects of a system are united by a form of control where that form of control is determined by adherence to six axioms.

A system, itself, can be viewed as an object within another system. Objects of an HOS system can be described in terms of abstract control structures [17] that relate members of algebraically defined data types [18][5] or functionally defined data whose components are algebraically defined. When an object is viewed in terms of an abstract control structure, that object is in the state of *doing*. When an object is viewed as a member of a data type, that object is in a state of *being*.

HOS systems communicate in terms of data and functions. Functions can be in a being state or a doing state. Data can be in a being state or a doing state. Or, as Cushing puts it [19],

> ...anything that can *be*...a datum, can also *do*, by serving as input to a function, and anything that can *do*, i.e., a function, can also *be*, since functions themselves make up a data type.

System designers often have the problem of knowing when they are finished. For example, it is not clear when a system definition has been decomposed as far as it should be or if it has been decomposed to a state of unnecessary detail. In HOS, a design that is finished is one which has been hierarchically decomposed into a *complete system specification*. In this case, all terminal nodes of a specification tree represent primitive operations on data types (Appendix I).

Sometimes we need to talk about the relationships between one system and another system (e.g., NAVPAK and the User System). In order to talk about such relationships, it is necessary to understand the *structure* or organization of a system. In HOS the structure of a system can be made up of several systems where each system can be defined in terms of levels, layers, and instances of layers. A *layer* represents all performance passes of a system. An *instance of a layer* represents one performance pass of a system. A *level* represents a step of refinement within an instance of a layer of a system. A communication takes place between two systems if an instance of one system communicates with an instance of another system. Two systems are on the same layer if and only if an instance of one system always communicates with the same instance of the other system.

There are other relationships between systems other than that of communication between them as objects. A system, as an object, can communicate with other layers of a system such as the definition, description, name, or environment. That same system, as an object, can also communicate with those systems on which another system is implemented. All of these relationships between one system and another system are defined by still another system. This process (or system) which prepares one system to relate to another system is called *resource allocation*.

In the process of resource allocation, it is possible to assign a name to an object, an object to a name, or replace an object by an object or a name by a name. An engineer provides resource allocations to objects in his system when he is designing the system. This process can take place manually or automatically. He also designs resource allocation functions for the system itself to perform when it is being executed. Sometimes it is necessary to perform a resource allocation to a system before it can communicate with another system. Sometimes it is necessary to perform a communication between two systems before a new step of resource allocation is determined.

Thus, for example, one system may need to compute an output value in order that such a value can be assigned to an input variable in another system. Another system may need to compute an output value in order that such a value can be assigned as a function to a name of a node within a structure in another system (this type of assignment is necessary for a reconfiguration of functions in real time). Another system may need to assign a function from one system as a value to an input variable in its own system. And still another system may need to assign an instance of a function as a value to an input variable in its own system.

When we think about an object it is often too easy to think about it in too limited a sense. The problem is that everyone involved with that same object has a different viewpoint of that object. And, it is for this very reason that one designer will define the same object differently than another designer. Each designer designs a system with his own implicit assumptions or preconceived notions that are collectively different from anyone else. Unfortunately, many of these implicit assumptions are not usually made known until that object is finally executed on a computer. And sometimes, they are never made known. At least a system that is implemented on a computer has the advantage of the computer finding problems that designers of other non-computer problems would never find.

The only way to get around the problems of producing incomplete, ambiguous or redundant designs is to have the ability to define a system explicitly. In order to explicitly talk about an object, we need to understand *what* it is, its *type* (or characteristics), and its *definition*. (The definition of an object contains information about those objects controlled by that object as a controller. It does not include information about how that object itself is controlled.) To transfer an object from one system specification to another, we would include all of the above in addition to the object, itself, and a *description* of that object.

The controller of an object allocates its *name*, now it *exists* (i.e., being or doing), its *environment* (i.e., its residence), *when* it happens, *where* it comes from, its *representation*, its *implementation* (i.e., what makes it happen), and its *execution*, which includes the relationships of one system, as an object, *with* another system as an object (i.e., its layer and level communication relationships) and the relationships of one system as an object with another system in order to prepare that system for communication (i.e., its layer and level resource allocation relationships).

An object as a controller resource allocates. An object as a function communicates.

Each object in a system specification has a special meaning to an engineer as to *what* it is. For example, objects such as functions, input and output values, and input and output variables are commonly referred to by system designers. In HOS, we include these kinds of objects with very specific meanings, and we have other kinds of objects including those which are available as a convenient means of abstracting (or modularizing) system specifications. When we talk about an object, it is very helpful to say what it is, for in doing so we immediately know a lot more about its properties. For a given specification, an object in HOS could be, for example, a structure, an operation, a function, a data type, a variable, a value, a tree, a data structure, or a primitive operation. An example of objects that can be used in a system specification is shown in Figure 1. (A definition for these objects is given in Appendix I and Appendix II.)

We can tell the *type* of an object by referring it to a set of values

whose characteristics we have defined. A type is a set of values defined by a set of axioms. Each axiom is a true statement about control structures. Thus, if

y = A(x);

we can say

Where (y,x) *are* INTEGERS,
A *is a constant* FUNCTION:

In addition, if

z = C(b);

we might say

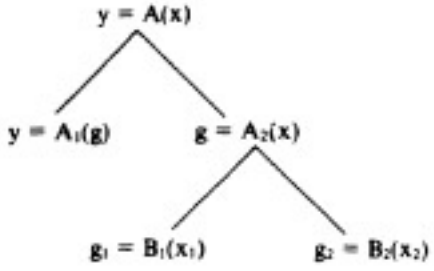
b *is an* A,
C *is a* FUNCTION,
z *is a* MATRIX;

To understand the meaning of a system, we *define* each of its objects. For example, one definition of an instance of a NAVPAK function, E, is

y = E(x);
Where x *is an* INTEGER,
y *is a* NATURAL:

One definition of E as data is

Where E *is a* FUNCTION;



STRUCTURE: the control relationships of the functions in System A

DATA TYPE: a set of values characterized by a set of primitive operations

VARIABLE: name of an object
"x" is an input variable of A
"y" is an output variable of A

VALUE: object being named
x is a value of data type integer (e.g., 1, 2, 3...)
y is a value of data type integer

FUNCTION: mapping between input values and output values where those values are represented by particular variables
A, A_1, A_2, B_1 and B_2 are functions of System A

TREE: geographical representation of a system (environment of System A)

NODE: location on tree. Function A resides at the root node of the control map for System of A.

DATA STRUCTURE: a set of variables whose values collectively represent the same value as a variable. In System A (x_1, x_2) is a data structure of x.

PRIMITIVE OPERATIONS: one of a set of operations which characterize a particular data type
A_1, B_1 and B_2 are primitive operations for System A

UNIVERSAL PRIMITIVE OPERATIONS: operations in which the operands are represented by variables whose values are of some type.

Figure 1
A Control Map of System A as a Function

We use a *description* to talk about the definition of a system. One description of an instance of function E is "y=E(x)". This description can be replaced by other descriptions. It could be replaced by ones described in an assembly language, specification language or a higher order language. In HOS we describe definitions with AXES or AXES based statements (Appendix II).

We need to have a *name* for an object in order to mention that object [20]. A name itself can be an object with respect to its name. To talk about, for example, NAVPAK, we use the name of NAVPAK, "NAVPAK." If E is a function in NAVPAK, E can be controlled by mentioning the name of E. Thus "E" controls E in that the mentioning of E makes E happen, and "E" can be controlled by naming "E". Thus "'E'" controls "E". To give a name to E is to resource allocate "E" to E. In order to invoke E we need to have an object. E_c, which mentions the name of E. E, "E", "'E'" and E_c, which invokes E, all reside in some environment. In order to control E by E_c we must also resource allocate to E_c a name in order to talk about the controller E_c. E_c and its name must also reside in some environment. In Figure 2, A, A_1, and A_2 are nodes which locate positions on the control map; F, F_1 and F_2 are functions in System E. In order to obtain F on the control map, we locate F by F = Object ("F"). F, F_1 and F_2 are resource allocated by the very fact that they have names. F, as a controller, relates to the invocation of F_1 and F_2 by mentioning their names which is, in essence, also a process of resource allocating the names "F_1" and "F_2" to nodes A_1 and A_2. In the mentioning of "F_1" and "F_2", F as a function is replaced by functions F_1 and F_2. Similarly, in the case, for example, where we have an instance of the function F, defined as y = F(x), we are concerned with "y" and "x", in addition to "F", as well as where these names reside.

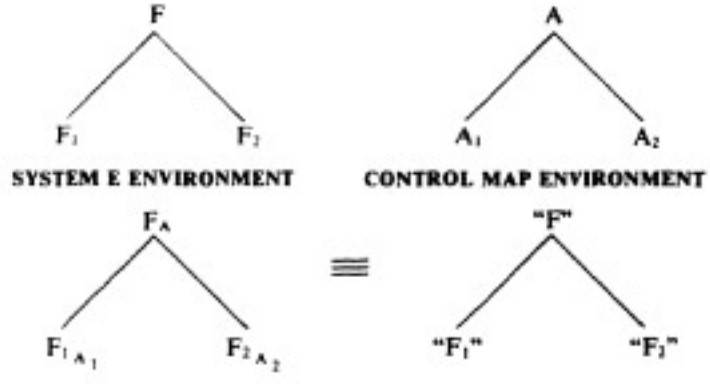


Figure 2
System E on the Control Map

When we relate to an object, it is important to know *how* it exists. In the definitions above, when E is viewed in terms of a control structure, E and x and y are viewed as doing. When E is viewed in terms of a type, E and x and y are viewed as being.

When one system relates to another system, it must know, among other things, the *environment* within which that system resides. This includes *where* that system is located and *when* that system exists.

In the case of Function E, we could say that the location of "y=E(x)" is *on* this line *on* this page *in* this collection of pages. For another Function A, we might say that its description, "y=A(x)" is located at node 12 on the control map. Or we might say that

"y=A(x)" is located at Register 10 on PDP-11/70 or at FLORIDA IMAGE on LCV file. Similarly with the definition for A, we could say y=A(x) is located at BLOCK 100 to 200 on Computer. With respect to function, B, and time, t, we could say y=A(x) *at* t or y=A(x) *after* B or y=A(x) *in* t or y=A(x) *before* t.

An example environment might be defined as follows:

y = A(z) at t at s
Where t *is a* TIME,
z *is a* LOCATION

or if the type information about A, t and z were contained on, for example, a file, we could simply say

A at t at z.

The *representation* of an object might vary depending on its environment. One representation of A, A's controller, their names, names of names, etc. could exist in a computer. In one computer, a rational number could be represented as a single-precision number, and on another computer, it could be represented as a double-precision number. In an HOS system specification, we often talk about these objects within the environment of a control map. In an HOS implementation, the nodes of a control map can be replaced by locations of a computer, the names of the nodes by names of instructions or names of locations in the computer. Thus if a specification is described within a control map environment, it can be transferred to the computer environment as part of the implementation process.

In order to *implement* a system (or make that system happen), we define that system, the environment of that system and the system that will execute that system. That system which executes another system is called a machine (Appendix II).

One of the most difficult problems facing system designers is that of resource allocating a design to execute on a particular machine. More subtle problems appear when we attempt to redesign our system to fit a machine. Once having done so, we can no longer understand our original design, since the resulting complexity hides the original intent with camouflages of implementation. We not only cannot trace input and output throughout our system design, but we are no longer even sure which input and output is relevant to our original problem. To try to change such a system with a new system requirement is a presumptuous notion.

Input and output can be traced throughout a given AXES system definition. Also data flow can be separated with respect to different layers of implementation. These features allow us to look at system interfaces which are relevant at the time they are relevant. We often hear the complaint from system designers about the problem of attempting to design a system "top-down" and then being forced to add details of implemetation on a lower level of the top-down design when resource allocation is addressed. This forces an iteration of the design process in order to incorporate the extra details of implementation back at the top and down through the top-down design to maintain consistency of the overall design. (Some designers merely add that extra detail without worrying about it because they don't know what to do with it, except to say it came from some other system; but then they lose track of its influence on their own system and the influence of their own system on other systems.)

In order to resource allocate a given system to a particular machine, it is necessary to define both the system and the machine in such a way that a change to one won't affect the other. It is also necessary to define the machine in such a way that the users of the same machine only affect each other when they are explicitly defined to affect each other.

Whenever we want to separate a system from the specification of its execution, we specify one layer for the system and another layer for the machine that runs that system. Likewise, that same machine can be looked upon as a system with respect to the machine which executes it. To implement System R, for example, a user might want to "run" R on the Os system. He could express such an implementation as

R *on* Os;

or he could express a more detailed implementation as

R at A on B *on* Os at m on P;
Where A *is a* RECORD,
B *is a* FILE,
m *is a* LOCATION,
P *is a* ROM;

In AXES these or similar statements can be used as long as the syntax and the semantics of the syntax are defined using AXES. The process of implementation continues until we arrive at the layer of a primitive machine. Consider potential resource allocation steps for a system within which NAVPAK might reside:

Determine system: (1)
S *is* System;
Determine I/O: (2)
Where S *on* (x_1,x_2);
Determine algorithms: (3)
(x_1,x_2)
Where (x_1,x_2) *on* Navpak;
Determine OS: (4)
x_2 = NAVPAK (x_1)
Where Navpak *on* Executive;
Determine NAVPAK computer: (5)
$Estate_n$ = Executive($Estate_1$);
Where Executive *on* PDP-11/70;
Computer: (6)
$Cstate_n$ = PDP-11/70($Cstate_1$);

Because each layer is defined as a control hierarchy, each machine that runs a system is able to use that system description as data and therefore has access to information about that system's ordering relationships as well as that system's data relationships. With respect to the six different layers for System S shown above, any one of these layers can be replaced by another implementation layer. If we want to move Navpak to another operating system, we change Step 4 to name a new Os for Navpak and replace Steps 5 and 6. If we wish to design a new algorithm for the same computer, we change Step 3 and replace Navpak with a new algorithm and then replace Steps 4, 5 and 6. If we want to move to a new computer and our operating system is independent of its computer (as it ideally would be), we can change Step 5 to refer to another computer and we then can replace Step 6.

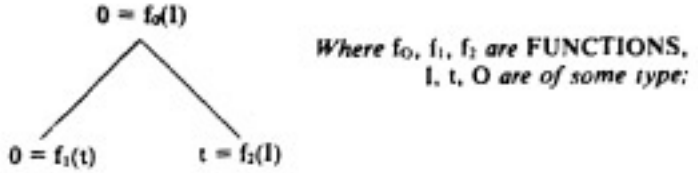
The *execution* of a system is a continuous process of resource allocating system objects for the purpose of communication and processing the communication between these objects, where that communication determines the next step of resource allocation. The start of an execution of an object is like the process of one object mentioning another object by first calling its name and then operating on the object itself.

By executing a system we realize the implementation of a system. An engineer can manually execute a system with the aid of a pencil and paper. One, however, more commonly thinks of computers as executing a system. In both cases, this execution process is a *dynamic* one. In the dynamic process we execute a system instance by instance. Each instance is a performance pass of the system.

Sometimes a system is viewed statically. In this case an engineer or a computer observes the description of a system by eyeballing statements or instructions. Examples of such static views are those performed by a compiler or an interpreter. An Os could view a system in both its static and dynamic states.

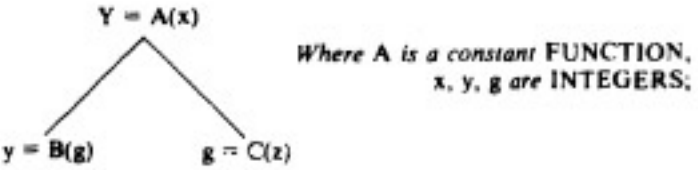
When we wish to define a set of systems to communicate with each other, we define a system whose purpose is to control their

communication. In order to define such a system, we have available a set of rules which will help us determine how to properly connect systems. As an example, consider two systems which are to be resource allocated to directly communicate with each other. A primitive control structure exists which provides the rules for properly defining two such systems, i.e., one of which is dependent on the other. Since we are in the specification stage we will choose to have the system reside on the control map, which in our case resides on a piece of paper. It could just as well reside on a graphics device. Here the primitive control structure, composition (Figure 3a), is used as a model for organizing our functions. That system which controls their communication is their controller.



Where f_0, f_1, f_2 *are* FUNCTIONS, I, t, O *are of some type;*

(A) Primitive Control Structure, Composition



Where A *is a constant* FUNCTION, x, y, g *are* INTEGERS;

(B) Use of Composition

Figure 3. An Example of the Resource Allocation of Systems Communicating on the Same Layer

We can tell from this control structure that any function can be "plugged into" f_0, f_1 and f_2 as long as it follows the rules of composition since f_0, f_1 and f_2 are defined as functions (Appendix II). A user of the composition control structure could plug System A into the structure as follows:

Where A *on* f_0, A_1 *on* f_1, A_2 *on* f_2;

Or, in Figure 3b, we have another example of the use of the structure in Figure 3a. Here, A, B, and C are values for f_0, f_1 and f_2 respectively.

In this use of the composition control structure, A controls B and C to communicate with each other on the same layer. A, as a controller, can be thought of as giving control commands where

C *when* x

B *when* g *from* C.

(Notice that B is viewed as a being and C is viewed as both being and doing by A.)

Since A is a constant function, it follows also that B and C are constant functions. Thus B and C are on the same layer of communication since both always relate to the same instance with respect to A.

The control relationship of A, B, and C are defined by composition, which is defined as an AXES based *STRUCTURE*, *Join* (Appendix II). Other control structures exist for the purpose of providing a controlled communication between systems. These include the other primitives, set partition (decision making) and class partition (parallel processing or independent functions). From a combination of primitives, we can form more abstract control structures (e.g., recursive functions). The rules for the control relationships of the primitive control structures are described in Appendix I. Similar rules exist for every defined AXES control structure.

The syntax that is associated with such *STRUCTURES* is used manually by a designer to define system specifications. The computer can automatically do the same thing with a *STRUCTURE* by writing the equivalent commands into a register instead of on a piece of paper. One way of specifying such a command is through the AXES *on* statement. These structures provide the mechanism for systems on different layers to communicate with each other in that the structure provides a means for an instance of one system to communicate with an instance of another system by simply adhering to the relationships of the structure in real time.

We use such a concept for the communication between functions in an asynchronous environment. For example, a control structure which can be used to define an interrupt is

STRUCTURE: $(y_1,y_2) = \text{Xch}(x_1,x_2)$;
Where y_1,y_2,x_1,x_2 *are of some type;*
$y_1 = \text{id}_2^2(x_1,x_2)$ *Coinclude* $y_2 = \text{id}_1^2(x_1,x_2)$;
SYNTAX: $(y_1,y_2) = \text{Xch}(x_1,x_2)$;
END Xch;

Here, the universal operations, id_2^2 and id_1^2, themselves defined in AXES to select a value from a set of values, as well as the *Coinclude*, which is one of the structures defined so that we can access the same value more than once, is used to define Xch*.

III. Automatic Verification

Once it is determined how to automate many manual processes, it is possible to streamline these processes without actually automating them. Manual verification of a system design using AXES-like syntax has uncovered design inconsistencies (e.g., preliminary analysis of a Navy message processor specification uncovered 29 errors, 27 of which would have been found by an automatic design analyzer [21].

Sometimes we can support a manual process with an automatic tool. The Structured Design Diagrammer† [17], an aid to manual verification of structured programs as well as a documentation tool, resulted from such an incremental approach.

Automated tools can be developed for both static and dynamic verification. The structuring executive is an example of an automated tool that can be developed for dynamic verification [17]. Automatic analysis of interface correctness can be provided on a static basis (without execution) by, for example, a *design analyzer* [17]. The manner in which a design analyzer can be used for static verification is discussed below.

A. Design Analyzer

The design analyzer is intended to check the consistency of a specification written in AXES statements. As a visual aid in determining valid functional relationships, the design analyzer produces an automated version of a control map (a graphical representation of an HOS control hierarchy). The design analyzer also aids in the design verification process by means of heuristic algorithms to check a data type specification and proof-of-correctness algorithms to check the contents of a specification.

Interface Consistency

Analysis of a system specification for interface consistency requires a syntactic verification of each function decomposition. Decompositions can be analyzed directly, according to the considerations outlined for primitive control structures. If a structure of a given system is encountered in the static analysis, which cannot be formed from the primitives, then it is not a legal form for decomposition and will be declared to be in error. All relationships, which are found to be consistent with the rules of the primitive structures, are considered to be correct.

Data Type Analysis

Data-type specification techniques for AXES are based on the

*We named this structure after the XCH instruction, in the Apollo Guidance Computer (AGC), which was used to perform an interrupt.

†Now being specified in AXES and implemented in PASCAL/P for the MUST system [22] as a multi-language structured design diagrammer.

algebraic approach. Using this approach, it is possible to define abstract data types and the primitive operations performed on them without prescribing a manner of implementation.

The problem of guaranteeing that axioms are consistent is much simpler for the system designer than that of insuring that every necessary characteristic of the desired data object has been included. This fact is reflected in the formal theory of algebraic data type specification. It has been shown that it is impossible, in general, to verify a set of axioms mechanically [23]. The completeness and consistency of a data type specification can be mechanically guaranteed, however, if the designer systematically limits the complexity of the axiom definitions and requires that the operations be both primitive recursive and total. These limitations appear not to eliminate any axiomatizations that might be useful in specifying computational systems. The complexity of the axioms may also have to be limited to allow human comprehension of the overall data type specification. This may be necessary to guarantee that the defined data object is really the desired one.

This systematic procedure can be implemented as automatic design verification heuristics. These can interactively assist the system designer in generating a set of algebraic data type axioms that are suitable to mechanical verification of consistency and completeness. An abstract data type specification process would begin with the syntactic definition of the primitive operation interfaces [24]. The system designer will derive this information from what he knows about the desired operations, their domains, and their ranges. Once the data types have thus been specified, the design analyzer can then verify the axiomatic consistency of each instance of a primitive operation in a system control map.

Proof of Correctness

The interface consistency checks of the design analyzer will verify that a specification is complete and consistent, but it will not verify that the solution is the correct one to solve the initial problem. In AXES, we are free to define any operation we want on an already defined data type as long as the operation definition is consistent with the data axioms of the type. New operations can be characterized either as *OPERATIONS* or as *DERIVED OPERATIONS* [18]. An *OPERATION* is specified in AXES in the form of a control map. A *DERIVED OPERATION* is specified by means of assertions that describe the behavior of the *OPERATION* with respect to other already defined *OPERATIONS*. When we implement a *DERIVED OPERATION* in AXES with an *OPERATION*, that implementation must satisfy the assertions about that *DERIVED OPERATION*. It is envisioned that the design analyzer will have techniques for verifying that the *OPERATION* is correct with respect to the properties of a *DERIVED OPERATION*.

IV. The Navpak Structure

A. An Axiomatic Requirement Specification

The Navpak system is intended to update navigational parameters of earth-referenced satellites with imaging data transmitted to the ground from the satellite. The feasibility of an approach in which orbit and attitude estimates are obtained from landmark data extracted from earth images generated by an on-board radiometer has been investigated [16].

The imaging data is used to determine orbit and attitude state from landmark observations. In addition, when the orbit and attitude state is precisely known, the imaging data is used to (1) determine geographic coordinates of a landmark identified by pixel coordinates (i.e., line and element of a point on an image as seen by a satellite), (2) determine pixel coordinates of a landmark identified by geographic coordinates. This example is intended to provide a specification of user/machine relationships for the case in which the orbit and attitude state are not precisely known.

The process of determining orbit and attitude can be done with varying degrees of automation. The most manual approach is one in which the landmark observation is obtained from the user by displaying the image data (retrieved from data available on file) directly on graphics devices. In this case, the correlation function (i.e., correlating the geographic and pixel coordinates) is a human function. The function relating the observation to the state is performed by a computer. Here, the computer processing includes the computation of the landmark time from pixel coordinates, integrating the best known orbit/attitude information to the time of the geographic coordinates of the landmark (the time is geometrically computed from known geographic coordinates associated with the center of an image), computing the uncertainty of the observation (in terms of a correlation with an elliptical shaped "window"), and, upon user request, an orbit/attitude/covariance matrix update based on a classical "weighted least squares" statistical estimation algorithm [16].

A total automation of orbit and attitude determination involves automating the correlation function involved in the landmark registration in which preprocessed landmarks are input to the Navpak system and processed automatically one at a time (i.e., sequential state updates). When orbit/attitude information is very imprecise, total automation is not feasible. At these times, user interaction with the processing system is essential so that a user can make the ultimate decision as to whether a particular observation should be incorporated or not.

The interaction, or relationship between the user and a machine system is described by the AXES *STRUCTURE*, Navpak. An AXES *STRUCTURE* is used to define relationships among functions. The functions, themselves, may be variable in a *STRUCTURE*. The user chooses the values of the variable functions and data of the Navpak structure that will specify an instance of the Navpak *STRUCTURE* as a function. The mechanism available to the user to choose these values is the syntax of the Navpak *STRUCTURE*. This syntax is analogous to the user "menu" in that all of his available options are specified through the use of this syntax. The syntax can be specified in a manner familiar to a Navpak user as long as each variable of the *STRUCTURE*, i.e., function and data, are referenced within the syntax*. The interface between the user and Navpak, i.e., the syntax and the semantics, can be automatically checked by an AXES analyzer for consistency. The Navpak system described here is designed for automatic processing with the capability for manual override at crucial processing decisions. User options include the ability to continue processing data or function (i.e., reject an override option), reject data or function, or override data or function.

That is, the user may override the vehicle state data, the landmark measurement data, and the earth image data as desired. If the user chooses to override the default vehicle state, x, he may enter a new covariance matrix, c; a new satellite position, p; and velocity, v; and/or a new satellite attitude, α; all associated with a particular time, t. The user may also override the automatic correlation of a landmark with a known image and is ultimately in control of a measurement incorporation.

The machine system that interacts with a Navpak user will initially select a default scene (which contains an image) automatically. If the user is dissatisfied with the default selection, he may override this data and enter any scene he desires. When the user exercises this option, the Navpak machine checks the validity of the override and, if faulty, informs the user who may then enter a new scene name.

The Navpak system requires a large amount of data to be processed. Although much of the data (such as the images and preprocessed landmarks) are intended to be implemented by file representations, this description concentrates on the properties or characteristics of the data leaving unspecified a particular imple-

*The syntax for *STRUCTURE* Navpak has not been specified here, since more familiarity with the user "menu" is needed before doing so. Note that each substructure of Navpak would also have a syntax associated with it. That is, each substructure syntax would be associated with a portion of the user menu.

mentation. It is the intent of this specification to functionally represent the system so as to lead the way towards an efficient implementation. The data was so complex for this system that we chose to devise separate AXES *STRUCTURES* to describe many of the objects used within the Navpak structure (Figure 4). The structures in Figure 4 will be used for many substructures of Navpak.

In each structure of Figure 4, each variable is associated with a data type or a data structure whose components are of some data type. An example of a data-type specification used here can be found in Appendix III. Examples of the use of a data-type specification which indicates the possible values a variable can represent, e.g.,

covariance *is a* MATRIX;

is shown in Figure 4.

STRUCTURE: (covariance, position, velocity, attitude) = State (time);
  Where covariance *is a* MATRIX,
  (position, velocity, attitude) *are* VECTORS,
  time *is a* TIME;
END State;

(A)

STRUCTURE: (image, parameters, center) = Scene (identifier);
  Where identifier *is an* ADDRESS,
  center *is an* Observation,
  image *is a* MATRIX,
  (scan, solarDeclination, bodyrates) *replace* parameters,
  ($scan_1$, $scan_2$) *replace* scan,
  ($scan_1$, $scan_2$) *are* SCALARS,
  solarDeclination *is an* ANGLE,
  bodyrates *is a* MATRIX;
END Scene;

(B)

STRUCTURE: pixel = Observation (geographicCoordinates);
  Where (l,e) *replace* pixel,
  (latitude, longitude) *replace* geographicCoordinates,
  (l,e) *are* INTEGERS,
  (latitude,longitude) *are* ANGLES;
END Observation;

(C)

STRUCTURE: λ = Landmark (ϕ);
  Where ϕ *is an* Observation geographicCoordinates latitude,
  λ *is an* Observation geographicCoordinates longitude;
END Landmark;

(D)

STRUCTURE: I = Chip(n);
  Where n *is an* ADDRESS,
  I *is a* Landmark,
  image *is a* MATRIX;
END Chip;

(E)

STRUCTURE: (position,velocity,attitude) = VehicleState(time);
  Where time *is a* TIME,
    (position,velocity,attitude) *are* VECTORS;
END VehicleState;

(F)

Figure 4. Structures Used as Data for NAVPAK

We can also use the structure specifications, themselves, to indicate possible values a variable can represent, e.g.,

x *is a* State;

From the definition of State in Figure 4A, "x" represents an instance of a function in which a unique time identification is associated with the other components of "x" (e.g., covariance, position, etc.). Likewise, each Scene instance (Figure 4B) has a unique address (or name) associated with it.

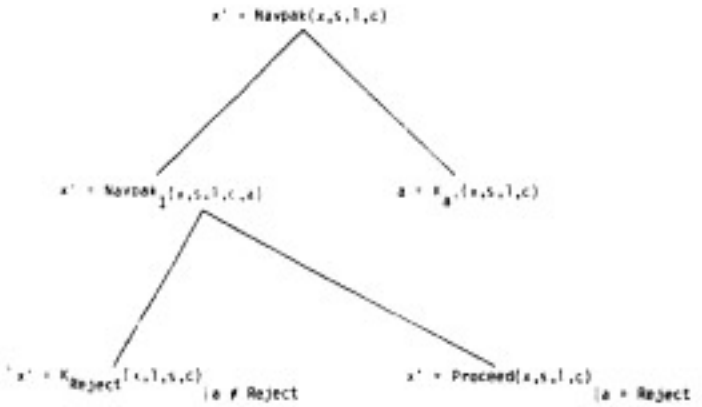
We can also use the structure specifications to indicate all instances of a function. In the example

I *is* Landmark;

"I" represents a set of Landmarks where Landmark is defined in Figure 1D. The implementation of I as a file representation may be a natural thing to do if I has many components. When I is to be used in an instance of Navpak, where I is retrieved (e.g., from a disc or file) can be determined by analyzing the access rights shown in the functional specification of the user system.

In some cases, structures are defined in terms of other structures. For example, Chip (Figure 4E), a set of landmarks and images where each Landmark image pair is identified by a unique name, is defined in terms of the Landmark structure. Landmark, itself, is defined in terms of the Observation structure (Figure 4C). In some cases, closely related structures are defined separately for modularity (c.f. Figure 4F and Figure 4A).

The relationship between the user option to terminate Navpak and further processing of Navpak is demonstrated by a control map and corresponding AXES syntax (Figure 5). The user option in Figure 5 is the selection of a value of "a'", which, in the use of Navpak, specifies a particular operation that produces a constant value for the output variable "a" (irrespective of any input value). That is, $K_{a'}$ defines a set of constant functions (see Appendix II for the universal primitive operations of AXES, one of which is $K_{a'}$). The leaves of a structure are either unspecified functions indicated by a unique identification related to the structure name [17].



(a)

```
Structure:  x' = Navpak(x,s,l,c);
    Where (x,x') are States,
          s is a Scene,
          l is Landmark,
          c is Chip,
          a is of some type;
x' = 1Reject                    Coucher  x' = Proceed 2(x,s,l,c)
                                Cojoin   a = Ka'(x,s,l,c);
    Where Q is (x,s,l,c,a,c');
    Partition of Q is 1Q|a ≠ Reject,
                      2Q|a = Reject;
End Navpak;
```

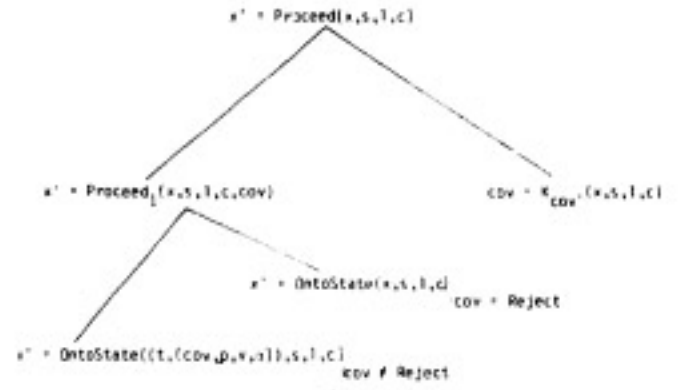
(b)

Figure 5. Definition of User Option to Process or Terminate Navpak

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

universal operations, or specified structures. In Navpak, since all other leaves of the structure are specified, only a' is a user option. In some other structures, if a leaf is unspecified, the user is required to specify a particular function. The primitive control structures of AXES are examples of structures whose bottom leaves are unspecified functions (Appendix I). Now suppose during the execution of Navpak the user ignores his option to terminate Navpak. That is, the value of "a'" is Reject (Reject is a special value of every data type in AXES that indicates an invalid value (Appendix I)). In this case, processing of the function represented by "Proceed" in Figure 5 is initiated. If a is valid, then the user has, in effect, accepted his option to terminate processing and the constant function, K_{Reject}, will be chosen to be executed. Note that a' is defined to be of an unspecified data type. This leaves the particular type specification to be left to the user.

The Navpak structure, as we will see shortly, is recursive. Each time a particular Landmark has successfully been updated or rejected, processing returns to the next recursive Navpak instance within the Navpak system. When the user has decided to continue Navpak, he may first choose to override the default covariance matrix by specifying a value of "cov'" (Figure 6). Whether he



(a)

```
STRUCTURE:  x' = Proceed(x,s,l,c);
    Where (x,x') are States,
          s is a Scene,
          l is Landmark,
          c is Chip,
          cov is a State covariance,
          p is x position,
          v is x velocity,
          a is x attitude,
          t is x time,

x' = Proceed1(x,s,l,c,cov)              conjoin   cov = Kcov'(x,s,l,c);

x' = OntoState1((t,(cov,p,v,a)),s,l,c)  Comither  x' = OntoState2(x,s,l,c);

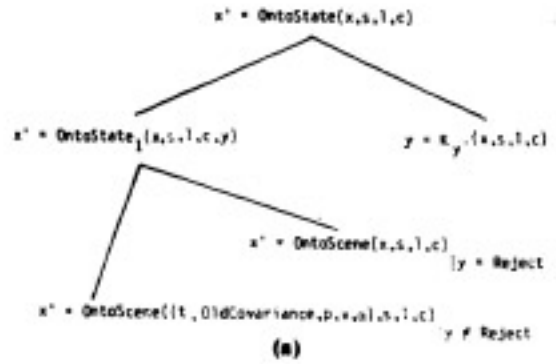
    Partition of (x,s,l,c,cov,x') is
                1(x,s,l,c,cov,x')|cov ≠ Reject,
                2(x,s,l,  cov,x')|cov = Reject;
END Proceed;
```

(b)

Figure 6. Definition of User Option to Override Default Covariance Matrix

overrides the default or not, processing will continue with OntoState located at both subnodes of "Proceed$_1$". If he chooses to accept the default value, OntoState will be selected using all default variables. If the user selects a new covariance matrix, OntoState is selected with the user-defined covariance matrix replacing the default value. Note that *STRUCTURE* Proceed does not check any constraints on the value of the default covariance matrix other than if it exists or not. If the specification were to be adjusted to detect more specific constraints, only the "OntoState" located at the left-most subnode of "Proceed$_1$" need to be replaced. The remaining Navpak structure need not be modified.

STRUCTURE OntoState provides the user with the option to override vehicle state data (Figure 7). At this point, the input variables associated with OntoState may represent a combination of user supplied and default values. Note that since y has been defined to be a VehicleState, the definition of the types associated with the components of "y" can be located by referring to the VehicleState definition. Again, further constraints on the user may be inserted at the left-most node below "OntoState$_1$", as desired, without impacting another part of the Navpak structure.



(a)

```
STRUCTURE:  x' = OntoState(x,s,l,c);
    Where (x,x') are States,
          s is a Scene,
          l is Landmark,
          c is Chip,
          y is a VehicleState,
          (t,p,v,a) replace y,
          OldCovariance is x covariance;

x' = OntoState1(x,s,l,c,y)                    Conjoin   y = Ky'(x,s,l,c);

x' = OntoScene 1((t,OldCovariance,p,v,a),s,l,c)  Comither
                                              x' = OntoScene 2(x,s,l,c);
    Where Q is (x,s,l,c,y,x');
      Partition of Q is
                1Q|y ≠ Reject,
                2Q|y = Reject;
END OntoState;
```

(b)

Figure 7. User Option to Override Vehicle State Default (where y is a Vehicle State)

STRUCTURE OntoScene (Figure 8) provides the user with the option to override the initial default scene. On subsequent Navpak instances, the user will select a new scene by exercising this option. If the option is not exercised, another landmark from the same scene can be obtained (the decomposition of OntoLandmark shown in Figure 9 gives the user the option to select a landmark).

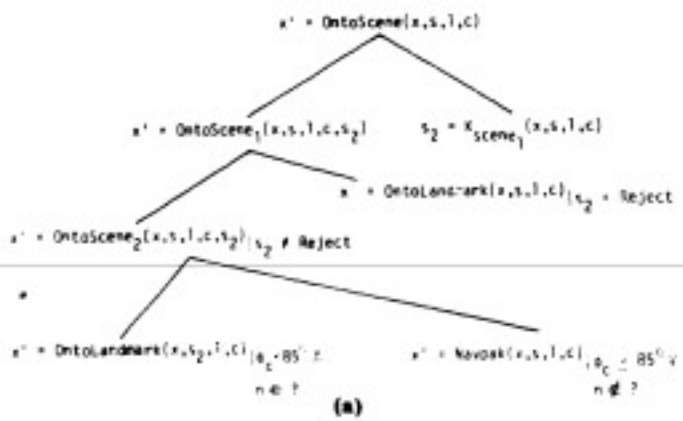
If a user has selected a new scene (i.e., $s_2 \neq$ Reject in Figure 8), a check is made to determine if any Landmarks can be seen from this scene, i.e., if the latitude of the center of the scene, ϕ_c is $\geq 85°$ the scene is rejected, and the next instance of Navpak is initiated with the original default scene value. In effect, we have specified here a "restart" of Navpak by specifying the recursive invocation of the Navpak structure. A check can automatically be made on the value of the scene identifier to assure the user he has made a valid choice. This check is indicated by the set of invalid addresses, ?, in Figure 8*.

If the new scene is acceptable or the default scene is to be used, *STRUCTURE* OntoLandmark is initiated. Here, the user may choose a new landmark or accept one of the default set of landmarks. If a new landmark is selected (i.e., $m \neq$ Reject in Figure 9), *STRUCTURE* E is initiated. *STRUCTURE* E defines the landmark extraction process and the manner in which a user can interact during the process. Here, the user can incorporate (or

*If desired, "?" can be unspecified in the structure and used in the syntax of the OntoScene structure so that a user may identify this set of addresses.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

not incorporate) a new landmark, and thus update the permanent Navpak state. In the event that a new landmark is not acceptable (i.e., if x" is Reject described at the right-most node under "Ontolandmark$_{1_1}$"), the AXES Failure *STRUCTURE* is used as an error recovery mechanism. That is, if E fails and a new landmark has been chosen, the next instance of Navpak uses the



(a)

```
STRUCTURE:  x' = OntoScene(x,s,l,c),
    where (x,x') are States,
          l is Landmark,
          c is Chip,
          (s,s2) are Scenes,
          θc is s2 center geographicCoordinates latitude,
          n is s2 identifier;

x' = OntoScene1(x,s,l,c,s2)        coupon    s2 = Kscene1(x,s,l,c),

x' = OntoScene2^l(x,s,l,c,s2)      coworker  x' = OntoLandmark^2(x,s,l,c),

    Partition of (x,s,l,c,s2,x') is

                1(x,s,l,c,s2,x')|s ≠ Reject,
                2(x,s,l,c,s2,x')|s = Reject,

x' = OntoLandMark^θ(x,s2,l,c)      coworker  x' = Navpak^θ(x,s,l,c),

    Partition of (x,s2,l,c,x') is

                4(x,s2,l,c,x')|θc < 85° & Element(n,?) = True,

                5(x,s2,l,c,x')|θc ≥ 85° or Element(n,?) = False,

    Where ? is a constant Set (of ADDRESSES).
END OntoScene,
```

(b)

Figure 8. The Selection of a New Scene

state, x, (which may be the original default value or the one obtained by the user override) the last chosen scene (possibly the original default one) and the default set of landmarks. If a permanent state update has been accomplished (i.e., a valid value of "x" has occurred), the next instance of Navpak is initiated using the updated state, x" the last chosen scene, and the default set of landmarks.

In Figure 9A,

m *is a* Landmark;

The design, in its present form, does not account for a possible user option to input a complete observation. A user may want this option if the state is very imprecise. To modify the Navpak structure to account for such a new user option, only Ontolandmark$_{1_1}$ and m need be changed. For example,

m *is an* Observation;

would replace

m *is a* Landmark;

and OntoLandmark$_{2_1}$ would be replaced with OntoLandmark'$_{1_1}$, as seen in Figure 10. OntoLandmark'$_{1_1}$ would check to see if the Observation pixel existed or not. If it did exist (i.e., m Observation pixel ≠ Reject in Figure 9), *STRUCTURE* Observe would be initiated. Here, the permanent state would be updated by calculating the time of the observation from geometric considerations, bypassing the automatic correlation function, and by proceeding to update the permanent state upon user request. If the m Observation pixel does not exist, OntoLandmark'$_{1_2}$ would be initiated. Note that OntoLandmark'$_{1_1}$ has the same decomposition as the original OntoLandmark$_{1_1}$ of Figure 9.

If a new landmark has not been selected (i.e., m = Reject in Figure 9), the user depends on the Navpak system to choose a landmark automatically. Note that the data structure of l is defined in AXES syntax as

(l$_1$, l$_2$) *replace* l

where

l *is* Landmark;

defines l to be a set of landmarks (refer to Landmark *STRUCTURE* in Figure 4). In Figure 9, l$_1$ is a Landmark and l$_2$ is a set of Landmarks. The set of Landmarks, l$_2$, exclude l$_1$ by using the infix operator "-", an operation which can be derived from the axioms on data type SET that have been defined in [17]. This describes l as an ordered set of Landmarks. The data structure of l can be described as the first landmark, l$_1$, and "all the rest", l$_2$. l$_2$, in turn can be represented by a data structure that describes it as an ordered pair of the first and all the rest. l can be implemented as a file where l$_1$ is the first record of the file and so on.

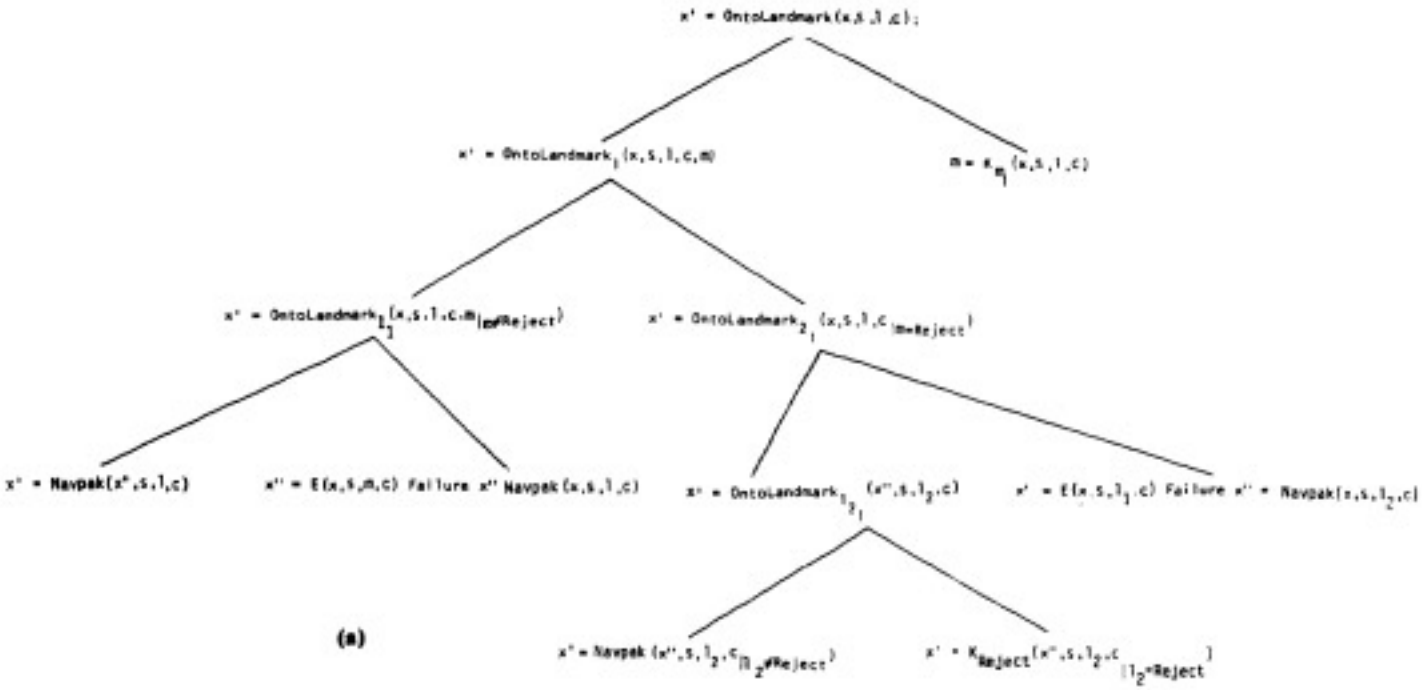
Here, defining l as Landmark leaves open the option of implementation (such as an array or file) at the structure layer. If the user desires a file to be used here, in the use of the structure (i.e., using the syntax of the structure), the user can specify the input variable that replaces l to be "on" a file.

When the default landmark (m = Reject in Figure 9) is selected, *STRUCTURE* E uses the first default landmark. If E fails, the next instance of Navpak is initiated using the chosen state, the chosen scene, and the remainder of the default landmarks. Whenever a landmark is adequate to update the state, x (i.e., x" ≠ Reject described at the rightmost node under "OntoLandmark$_{1_1}$" in Figure 9), the next instance of Navpak is initiated using the updated state, the chosen scene, and the reminder of the default landmarks. Note that if there are no remaining default landmarks, Navpak processing is terminated automatically (c.f., "OntoLandmark$_{1_2}$" in Figure 9).

If the user has selected a new landmark, m, *STRUCTURE* E, at the right-most node under "OntoLandmark$_{1_1}$", attempts to use m to update the permanent state.

If this fails, the next instance of Navpak uses the complete default set of landmarks. If m is a good landmark and a valid value of "x"" occurs, the next instance of Navpak is initiated with the default set of landmarks. In this way, a user may change to an automatic mode at any time.

During the process of determining if the landmark measurement is suitable to be used to update the state, the user has control in that he may override the automatic correlation process at crucial decision points (Figure 11). The landmark extraction function, Lmkrgs, integrates the vehicle state and covariance matrix to the time of the landmark and provides the dimensions of the "window", ellipse, which describes the uncertainty in the measurement. The data structure of ellipse includes the scan-angle, ϵ, and out-of-plane angle, ψ, which geometrically correlates where the satellite is and where it is pointing with respect to the landmark observation. The landmark extraction function may reject the measurement automatically if it cannot find the landmark in the chosen scene. If the landmark extraction is not successful, *STRUCTURE* E is rejected. If the landmark extraction is successful, the measurement is automatically correlated with a preprocessed image of the landmark (found in chip) in an attempt to find a better measurement time within the "window." If



(a)

```
STRUCTURE: x' = OntoLandmark(x,s,l,c);
    Where (x,x') are States,
          s is a Scene,
          l is Landmark,
          c is a Chip,
          (l1,l2) replace l,
          l1 is  a Landmark,
          l2 is Set(of(Landmark-l1)),
          m is a Landmark;

x' = OntoLandmark1(x,s,l,c,m)      Cojoin    m = Km1(x,s,l,c);

x' = OntoLandmark 1 (x,s,l,c,m)    Conither  x' = OntoLandmark2 2(x,s,l,c).
                  l1

    Partition of (x,s,l,c,m,x') is
                 1(x,s,l,c,m,x')|m ≠ Reject,
                 2(x,s,l,c,m,x')|m = Reject;

x' = Navpak(x",1(s,l,c'))          Cojoin    x' = E1(x,s,m,c; Failure
                                             x" = Navpak1(x,s,l,c);

x' = OntoLandmark12(x",2(s,l2,c)) Cojoin     x" = E2(x,s,l1,c) Failure
                                             x" = Navpak2(x,s,l2,c).

x' = Navpak3(x",s,l2,c)            Conither  x' = bReject;

    Partition of (x",2(s,l2,c),x') is
                 a(x",s,l2,c,x')|l2 ≠ Reject,
                 b(x",s,l2,c,x')|l2 = Reject;

END OntoLandmark;
```

(b)

Figure 9. A User Option to Select a New Landmark

a negative correlation, ρ_1, is found, the measurement is rejected automatically. If a positive correlation is determined (see "Iterate, Figure 12), the user has the option to incorporate the measurement immediately, or if this option is not exercised, automatic processing continues until the ellipse is considered completely searched. If at any step of this process, the landmark is rejected, the error filters back up the E structure. Subsequently, if

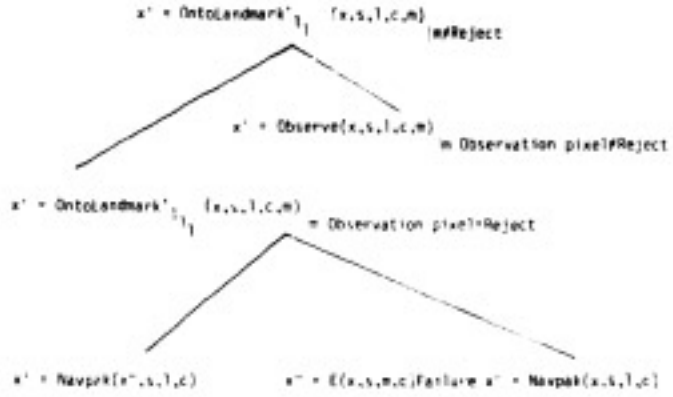
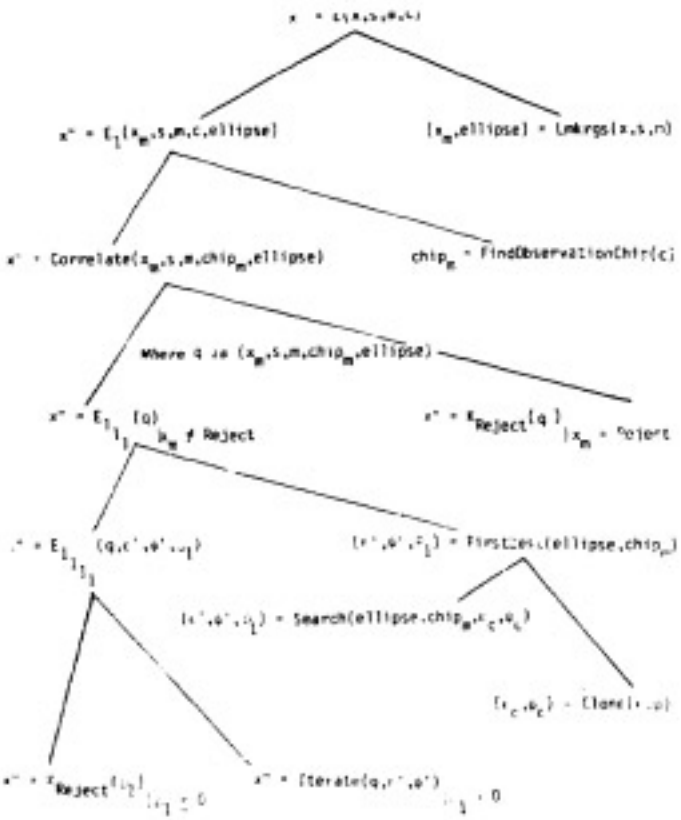


Figure 10. The Effect of a Modification on the Navpak Structure

E fails, error recovery is achieved, as seen in the higher level definitions for the Navpak structure.

At each "better" correlation, the user may decide to incorporate the measurement (see "Incorp", Figure 13). When the "best" correlation is determined, the measured scan, ϵ'', and out-of-plane, ψ'', angles are compared to a computed set of scan and out-of-plane angles at node F_1 in Figure 13*. The computed angles are used to construct a new ellipse and the measurement is tested to see if it can be found within the new ellipse (c.f., "QA" in Figure 13). If the measurement is not within the new ellipse, the measurement is rejected and the next instance of Navpak is initiated. If the measurement is computed to be successful, the user may exercise his option to reject the measurement if he is not satisfied with the results. If he does not exercise this option, the permanent state is updated, a successful instance of Navpak has been completed, and the next instance of Navpak uses the new state for its next measurement.

*The submodules of F_1 use many of the modules described in Imkrgs. These submodules, described as operations in AXES, can then be used to determine the structure of F_1.



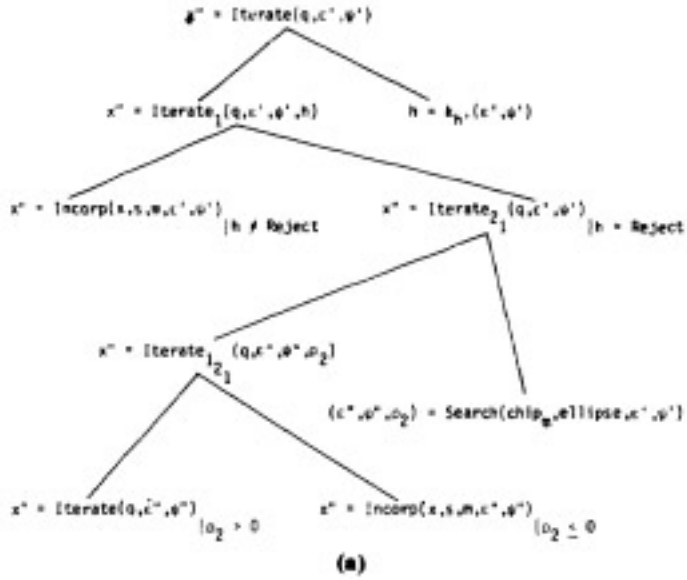
(a)

```
STRUCTURE:  x" = E(x,s,m,c);
    Where (x,x') are States,
          s is a Scene,
          m is a Landmark,
          c is Chip,
          (d,g,r,ϑ) replace ellipse,
          d is a SCALAR,
          g is a MATRIX,
          (r,ϑ,r_c,ϑ_c) are ANGLES,
          o_1 is a SCALAR,
          chip_m is a Chip;

x" = E_1(x_m,s,m,c,ellipse)          Comain   [x_m,ellipse] = Lmkrgs(x,s,m);
x" = Correlate(x_m,s,m,chip_m,ellipse)Cojoin   chip_m = FindObservationChip(c);
x" = E_1_1^1(q)                      Comither x" = ε_Reject^2(q);
    Where Q is (x_m,s,m,chip_m,ellipse);
    Partition of (q,x") is
                 1(q,x")|x_m ≠ Reject,
                 2(q,x")|x_m = Reject;
x" = E_1_1_1^1(q,r',ϑ',o_1)  Comain   (r',ϑ',o_1)= FirstBest^1(ellipse,chip_m);
(r',ϑ',o_1) = Search(ellipse,chip_m,r_c,ϑ_c)
                             Cojoin   (r_c,ϑ_c) = Clone(r,ϑ);
x" = ε_Reject^a(o_1)         Comither x" = Iterate^b(q,r',ϑ');
    Partition of (^2q,r',ϑ',o_1,x") is
                 a(new r',ϑ',o_1,x")|o_1 ≤ 0,
                 b(q,r',ϑ',o_1,x")|o_1 > 0;
END E;
```

(b)

Figure 11. Automatic Landmark Extraction



(a)

```
STRUCTURE:  x" = Iterate(q,r',ϑ');
    Where (x,x") are States,
          s is a Scene,
          m is a Landmark,
          c is a Chip,
          (x,s,m,c,ellipse) replace q,
          (d,g,r,ϑ) replace ellipse,
          d is a SCALAR,
          q is a MATRIX,
          h is of some type;

x" = Iterate_1(q,r',ϑ,h)          Cojoin   h = k_h(r',ϑ');
x" = Incorp^c(x,s,m,r',ϑ')        Comither x" = Iterate_2_1^d(q,r',ϑ');
    Where A is (q,r',ϑ',h,x");
    Partition of A is
                 c_A|h ≠ Reject,
                 d_A|h = Reject;
x" = Iterate_1_2_1^0(q,r",ϑ",o_2) Cojoin   (r",ϑ",o_2) =
                                     Search(chip_m,ellipse,r',ϑ');
x" = Iterate^e(q,r",ϑ")           Comither x" = Incorp^f(x,s,m,r",ϑ");
    Where J = (^Jq,r",ϑ",o_2,x");
    Partition of J is
                 e_J|o_2 > 0,
                 f_J|o_2 ≤ 0;
END Iterate;
```

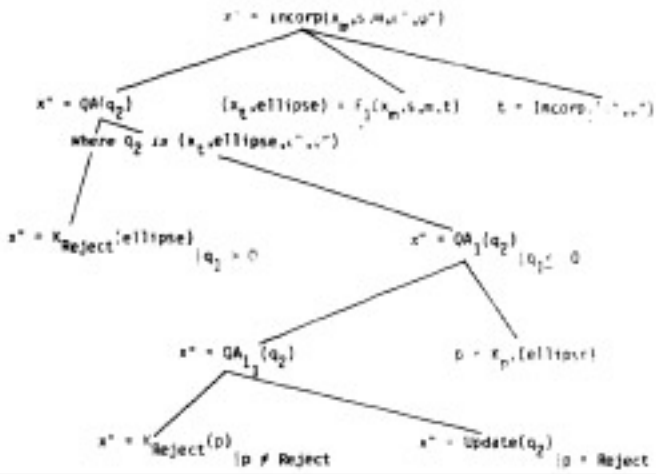
(b)

Figure 12. User Option to Intervene During Automatic Landmark Processing

B. Results

A structure of the system that relates a Navpak user to a machine system has been presented. The structure was devised from preliminary analysis [15] of available documentation from Goddard Spacecraft Center [25] and conversations with Martin Marietta staff. With this structure, we were able to demonstrate techniques which could be automated for manually verifying the specification, the manner in which user options can be inserted into a functional specification and the ease with which modifications could be made.

In addition to defining the specifications presented here, we investigated the landmark extraction process in detail in order



(a)

```
STRUCTURE:  x" = Incorp(x_m,s,m,ε",φ");
    Where (x_m,x",x_t) are States,
          s is a Scene,
          m is a Landmark,
          (ε",φ") are ANGLES,
          t is a TIME,
          p is of some type,
          (d,q,r,θ) replace ellipse,
          d is a SCALAR,
          q is a MATRIX,
          (r,θ) are ANGLES;

x" = QA(x_t,ellipse,ε",φ")      Cojoin      (x_t,ellipse) = F_1(x_m,s,m,t)
                                Cojoin      t = Incorp_1(ε",φ");

x" = K_Reject^1(ellipse)        Coeither    x" = QA_1(^2q_2),

    Where q_1 is q_22(r-r")^2+q_11(θ-θ")^2-2q_22(r-r")(θ-θ"),
          q_2 is (x_t,ellipse,ε",φ");
    Partition of (q_2,x") is
                1(q_2,x")|q_1 > 0,
                2(q_2,x")|q_1 ≤ 0;

x" = QA_1(^2q_2,p)              Cojoin      p = K_D(^2ellipse);

x" = K_Reject(^apl              Coeither    x" = Update(^bq_2);

    Partition of (^2q_2,p,^2x") is
                a(q_2,p,x")|p ≠ Reject,
                b(q_2,p,x")|p = Reject;

END Incorp;
```

(b)

Figure 13. Measurement Incorporation Based on User Acceptance of Quality Assurance (QA) Test and Subsequent State Update

to: establish the interface between it and the Navpak structure; determine modularity of subfunctions; determine data types that might be useful to a designer of Navpak; and determine potential control structures that would be useful to an engineer in specifying algorithms.

We were able to determine the interface between the landmark extraction process and the Navpak structure. The results are seen in Figure 11.

We were able to determine the modularity of the landmark extraction process and the results were used to further specify the measurement incorporation process for modules defined as AXES Operations.

We were able to identify data types and data structures some of which have been specified in [15].

An example of a potential control structure that we determined would be useful to an engineer in specifying navigational algorithms is one for integration. What is needed here, is a way to specify what one *wants* to integrate separately from *how* one wants to integrate. Also, such a structure would provide the flexibility to integrate as many equations as desired without modifying the integration. A control structure would be devised that would specify *what* to integrate. A user could then write, in a standard mathematical format, the equations desired to integrate. The definition of the control structure would specify how the integrator would work. Example syntax for such a structure might be

Differentiate from t_i *to* t_e *by* ΔT *on* RungeKutta:

$$Dx/Dt = f_1(t, \frac{\partial \dot{x}}{\partial x}, \frac{\partial \dot{x}}{\partial \dot{x}}, \dot{x}),$$

$$D\dot{x}/Dt = f_2(t, \frac{\partial \ddot{x}}{\partial x}, \frac{\partial \ddot{x}}{\partial \dot{x}}, \ddot{x}),$$

where variables x and $\dot{x}$, are to be integrated and RungeKutta is a user input in the example. A mechanism similar to this was available as part of the programming language the engineer used during Apollo [26] (although in the version used then, the user had to control both the loop that indicated the four passes through the fourth order RungeKutta (there was only one integrator available to the user) as well as the integration loop).

V. SUMMARY

We used a satellite navigation example as a demonstration of design and verification techniques which are intended to be used for real-time multiprogrammed or multiprocessed systems. The specification language, AXES, was used to specify this example to demonstrate methods for designing verification out of a system. We performed static analysis of these specifications manually by following a checklist that could have been performed by an automatic design analyzer. Such an analyzer could be developed to analyze a set of AXES statements defined for a family of avionics systems, such as Navpak.

In a system definition, we have certain facts we must explicitly state about objects and their relationships. We have structures, which are objects themselves, into which we can "plug" objects. We have rules to go with these structures and their respective objects to ensure that the interfaces between these objects are correct with respect to both communication and resource allocation. In the process of defining a structure or objects to go with a structure, the designer discovers missing, inconsistent, or redundant objects. In this way, the methodology supports the designer in doing things he might not have done as quickly, or, in fact, to do something he might have forgotten about completely. The methodology can also lead the way to defining a system which is modular. In doing so, the properties of modularity not only will provide a mechanism to make self-contained changes after a design is complete; but these same properties are even more useful along the way, and especially in the very initial stages of design. When the designer is accustomed to thinking about or defining new data types and structure definitions, he will abstract more quickly and thus be able to speed up his design process. (For example, it is easier to work with a matrix or a landmark rather than individual elements when it is possible to do so.) In defining objects, the designer will think of other useful system definitions that may not have occurred to him previously (for example, some new operations could be derived from a data-type definition or a combination of control structures).

Having available an effective methodology to support the designer in decomposing a system into manageable pieces is a

powerful design tool. And, in addition, knowing when he is through designing is crucial; for if he doesn't know, he either runs the risk of providing an incomplete design or of wasting time if he is done and yet still keeps designing.

Not only must a methodology be effective, but it must also be able to be used as well, and the results of that use should be made available to others. Having an extensible library of mechanisms such as a set of AXES statements can serve as management standards as well as save a lot of time for everyone involved in a project. Why should only one designer have the decimal system available to use in performing long division when all the rest are still trying to use Roman numerals [31]?

If you know what an object is that you want to specify, and if you know everything you need to know in order to talk about that object, and if you know how to relate that object to other objects in a formal, modular and understandable way, then the design process of specifying that object includes one of verification before the fact. This is the most important type of verification of all. The next type of verification, static verification, is used to check a specification to make sure it has followed a set of rules with respect to being formal and modular. In the future we envision that existing dynamic techniques and tools, such as simulators and operating systems, can be much simpler since certain kinds of verification that are necessary without the use of formal methods will no longer be necessary with the use of formal methods.

APPENDIX I*

Preliminaries of HOS

In HOS, the decomposition process for a system results in a tree structure. At the start of the decomposition process, the entire system is represented by the root of the tree which, hopefully, represents the requirements for the system. This definition, however, has many implicit (hidden) requirements. In order to arrive explicitly at the complete definition of the system, the root is decomposed by replacing it with a *nodal family* (a particular parent node and all of its offspring), which represents the decomposition of the root. This decomposition process, that of replacing a function by its nodal family, can be continued until the entire system has been specified. The resulting tree represents the complete system specification, wh[ere] the leaves represent primitive operations on the data types represented by the variables at those leaves. It may turn out that during the decomposition process a requirement is shown to be erroneous or missing. In such a case, an iteration of the system description is required.

The parent node of the nodal family controls its offspring. When referring to this control relationship, the parent node will be called a *module*, and its offspring will be called *functions*] The offspring of the nodal family are the functions required to perform the *module's corresponding function* (MCF) (i.e., the function that the nodal family replaces).

In the sections that follow, the variable that represents the domain elements of a function is referred to as the *input variable*, and the variable that represents the range elements of a function is referred to as the *output variable*. Individual domain and range elements may be called inputs and outputs, respectively.

A module, in performing its corresponding function (Figure AI-1), is responsible for determining if the inputs received are in the intended domain of the MCF. If an input is not in the intended domain of the MCF, it is in the unintended domain of the MCF and maps to a special value which is a value of every data type, the value Reject.

In a sense, the improper input element is not in the domain of the module's corresponding intended function (MIF), but is in the domain of the MCF, i.e., the module's corresponding unintended function (MUF).

Properties of the Primitive Control Structures

While a function can be decomposed in many ways, the HOS axioms [17] provide rules for the construction of nodal families (i.e., the decomposition of a function). From these axioms, three primitive control structures, which are used for functional decomposition, are derived [24].

These control structures are: *composition, set partition,* and *class partition.*

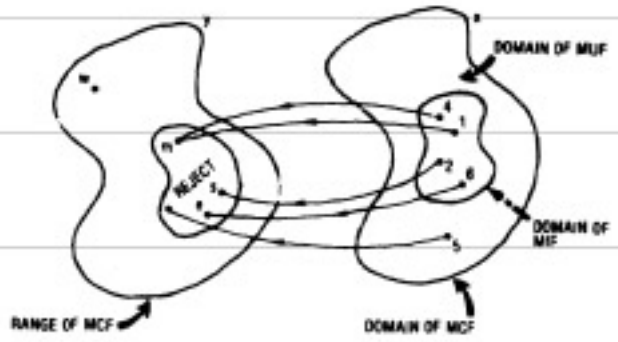


Figure AI-1. Illustration of a Function from X into Y



Figure AI-2.
An Example of Composition



Figure AI-3.
An Example of Set Partition

Composition is illustrated in Figure AI-2. In order to perform $f_3(x)$, the function f_2 must first be applied to x which results in output z. z then becomes an input to f_3 which produces the desired range element of the overall function.

It is important to observe the following characteristics of composition (characteristics are explained with respect to the example in Figure AI-2):

(1) One and only one offspring (specifically f_2 in this example) receives access rights to the input data, x, from module f_1.

(2) One and only one offspring (specifically f_3 in this example) has access rights to deliver the output data, y, for module f_1.

(3) All other input and output data that will be produced by offspring controlled by f_1 will reside in *local* variables (specifically "z" in this example). Local variable, "z", provides communication between the offspring f_2 and f_3.

(4) Every offspring is specified to be invoked once and only once in each process of performing its parent's MCF.

(5) Every local variable must exist both as an input variable for for one and only one function and as an output variable for one and only one different function on the same level.

Set partition, which involves partitioning of the domain, is illustrated in Figure AI-3. In the example, the set which comprises the domain is partitioned** into two subsets. For set partition, only one of the offspring will be invoked for each performance of the MCF at f_1 (the determination being based on the value of "x" received) and that offspring will produce the required range element for its parent module when it is performing.

The following characteristics with respect to set partition should be observed:

(1) Every offspring of the module at f_1 is granted permission to produce output values of "y".

(2) All offspring of the module at f_1 are granted permission to receive input values from the variable "x".

*Excerpted from [27].

**Partitioning implies the subdivision of the original set into non-overlapping (i.e., mutually exclusive subsets).

(3) Only one offspring is specified to be invoked per input value received for each process of performing its parent's MCF.

(4) The values represented by the input variables of an offspring's function comprise a proper subset of the domain of the function of the parent module.

(5) There is no communication between offspring.

Class partition is illustrated in Figure AI-4. While set partition involves partition of the domain into subsets, class partition involves partition of the domain variables into classes and the partition of the range variables into classes. In the example, it is assumed that the domain variable has an associated data structure comprised of two parts, "x_1" and "x_2". Likewise, the range variable has an associated data structure with the same number of classes as the domain's data structure.

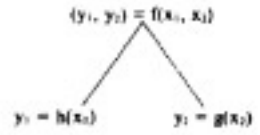


Figure AI-4. An Example of Class Partition

The following characteristics with respect to class partition should be observed:

(1) All offspring of the module at f are granted permission to receive input values taken from a partitioned variable in the set of the parent MCF domain variables, such that each offspring's set of input variables is non-overlapping and all the offspring's input variables collectively represent only its parent's MCF input variables.

(2) All offspring of the module at f are granted permission to produce output values for a partitioned variable in the set of the parent MCF range variables, such that each offspring's set of output variables is non-overlapping and all the offspring's set of output variables collectively represent the parent MCF variables.

(3) Each offspring is specified to be invoked per input value received for each process of performing its parent's MCF.

(4) There is no communication between offspring.

APPENDIX II*

Preliminaries of AXES

I. Axes Syntax Description

AXES is a formal notation for writing definitions of systems. These systems include systems which are mechanisms for defining other systems. Thus, for example, we could define a set of specification "macros" which collectively could form a language for defining a system or family of systems. Since each language statement would be a definition "macro" based on an integrated Higher Order Software (HOS) control hierarchy [17], the resource allocation to a particular machine could then be addressed independently from the definition of the system. Although it is not a programming language, AXES is a complete and well-defined language capable of being analyzed by a computer. AXES is intended to provide commonality between systems. Although users will have flexibility to choose different building blocks, these building blocks, when "compiled," will bring users to a common meeting ground with all other users of AXES.

AXES systems can be hierarchically decomposed into complete system specifications with the use of abstract control structures that relate members of algebraically defined data types. Three primitive control structures have been derived from six axioms that define completeness of control [24]. These primitive control structures provide rules for the definition of communication, parallel processing, and selection of functions. From a combination of primitives, we can form more abstract control structures (e.g., recursive functions).

The syntax of AXES [18] provides the mechanisms to specify control structures and data types. The purpose of AXES is to express a system specification which is equivalent to that same specification expressed graphically as a control map. Control structures can be described in AXES as structures, operations, and functions. Whereas a *structure* is a relation on a set of mappings, i.e., a set of tuples whose members are sets of ordered pairs, an *operation* is a set of mappings which stand in a particular relation. An operation results, mathematically, from taking particular mappings as the arguments (nodes) of a structure. By a *function*, we mean a set of mappings which stand in a particular relation for which particular variables have been chosen to represent their inputs and outputs. Whereas structures and operations can be described as purely mathematical constructs, a function is a hybrid, consisting of a mathematical construct and a linguistic construct, i.e., an assignment of particular names of inputs and outputs. Note that our use of "function" is slightly different from what is meant by "function" in mathematics. For the latter notion, we use the term "mapping" throughout this paper.

In AXES, a new data type can be defined simply in terms of the operations that are to be performed on the data [18]. That is, a data type is defined algebraically rather than operationally by making true statements (or axioms) about the equality of two control structures in which all the nodes are operations. Each such control structure is defined in terms of primitive operations of the data type of interest or of previously characterized primitive operations of another type (previously characterized primitive operations include universal primitive operations that have been defined, each of which is associated with any member of any data type).

The axioms associated with the definition of a data type are only those we *need* to characterize the data type. There are, of course, other operations that we find useful for other purposes. We are free to define any operation we want on an already-defined type as long as the operation definition is consistent with the axioms of the type. A new operation can be characterized either as an *OPERATION* or as a *DERIVED OPERATION*.

In AXES, we specify the behavior of an operation without specifying its decomposition by writing it as a derived operation, i.e., by means of true statements that describe the behavior of the operation with respect to other already-defined operations. Either kind of operation could be written as a control map, if desired. They differ in how they are specified, not in what they are. What distinguishes both of these kinds of operations from primitive operations on their data type is that their existence is provable mathematically from the existence of the primitive operations and the axioms of the type. In fact, if an *OPERATION* (which defines a function) and a *DERIVED OPERATION* (which defines the behavior) are both used to define the same function, the behavioral properties can be checked against the refinement properties to prove the correctness of a definition.

In describing AXES we will use variables and constants themselves to make statements about the values they name, and we will use the names of variables and constants to make statements about the variables and constants themselves.

To differentiate an object from its name, we introduced the "use-mention distinction" [20] in AXES. That is, we can talk about an object only by using a name of the object. (To talk about a man, for example, we have to use a sentence that contains the man's name, not the man himself.) The notation conventionally used for this is *enclosure within quotation marks*. To form the name of a given name (or written symbol of any kind), we include that name (or symbol) in quotation marks. (Successive embedding of quotation marks can be used if we want to talk about names, names of names, and names of names of names.)

In AXES, a *constant* symbol is the name of a particular value

*Excerpted from [28].

and corresponds to a proper noun like "John." A *variable* is the name of more than one possible value and corresponds to a common noun like "a man."

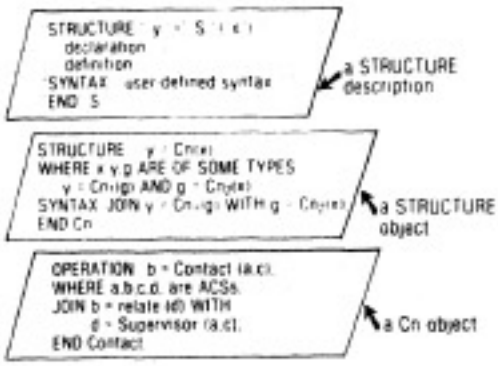


Figure AII-1. An Example of Abstract-Control-Structure Definition Layers with Respect to AXES

For example, in Figure AII-1, the top-most box is a description of part of AXES itself. The top-most box describes the AXES objects required to define a *STRUCTURE* in AXES. The sentence

"*STRUCTURE*:" y "=" S "("x");"

makes a statement about values by using the variable "y", "S", and "x" and about constant symbols by using the quotation-marked symbols such as "'*STRUCTURE*'", and "'='".

The middle box encloses an AXES object itself; that is, the middle box encloses the definition of a language statement derived from the definition of an AXES module. The Composition (Cn) *STRUCTURE*, defined in Figure AII-1, is one of the three HOS primitive control structures. Each primitive control structure has been defined as a *STRUCTURE* with AXES [18]*.

The middle box encloses an instance of the layer that the topmost box represents. If we could describe all of the structures that could possibly exist, then the complete set of structures would be the layer that the top-most box describes.

When a *STRUCTURE* in AXES is defined, the designer supplies the syntax (or description) so that a user of that structure can describe particular mappings that stand in the relation. For example, the bottom-most box in Figure AII-1 encloses an AXES object that is an instance of the Cn structure described in the middle box of Figure AII-1; that is, the bottom-most box is the definition of a system derived from the definition of a language statement, derived from the definition of an AXES module. In the bottom-most box, "b = Relate(d)" describes a particular function that "y = Cn_1(g)" represents in the middle box. Likewise, "d = Supervisor (a,c)" represents an instance of "g = Cn_2(x)". In the middle box, the objects that "y", "g" and "x" represent are described in the statement

Where x,y,g are of some types;

This statement means that x, y, and g are variables whose values are of an unspecified data type. In the bottom-most box the *WHERE* statement is used to specify a particular data type, and the operation, Contact, is a particular mapping.

Other control structures can be derived from already defined control structures and operations that operate on variables of any type. Operations that operate on variables of any type are called *universal operations*. Primitive universal operations are defined as

$$x = \text{Clone}_1(x) \quad (1)$$
$$(x,x) = \text{Clone}_2(x) \quad (2)$$
$$con = K_{con}(x) \quad (3)$$
$$x_1 = id_1^2(x_1,x_2) \quad (4)$$
$$x_2 = id_2^2(x_1,x_2) \quad (5)$$
$$(x_1,x_2) = St(x) \quad (6)$$
$$x = T(x_1,x_2) \quad (7)$$

(1) and (2) are used to specify more than one variable with the same value. (3) is used to choose a constant symbol. (4) and (5) are used to select the value of one of a set of variables. (6) and (7) are related by T(St(x)) = x. These are used to create a value of a data structure from a value of a data type (i.e., St) or to create a value of a data type from a value of data structure (i.e., T).

Universal operations have, as their bottom nodes, universal primitive operations. Universal operations are defined as *STRUCTURE* in AXES because they operate on values which are variables.

We can now define a structure whose syntax can be used to define more than one system having access to the same value. Here, we use the universal operation

$y = id_*^b(x)$

which is used to select particular variables out of a set of variables as well as the universal primitive operation

$(x,x) = \text{Clone}_2(x)$

to determine the meaning of the relationship among the unspecified functions that appear as bottom nodes of the structure definition.

STRUCTURE: y = J(x);
Where y,g,w,h *are of some type*;
Where b *is a* NATURAL;
Where a *is a* SET *(of* NATURALS);
$y = J_1(g,w)$ *Join* $(g,w) = J_2(x)$;
$(g,w) = J_{12}(x,x)$ *Join* $(x,x) = \text{Clone}_2(x)$;
$g = J_{112}(x)$ *Include* w $id_*^b(x)$;
$g = J_{1112}(h)$ *Join* $h = id_1^b(x)$;
SYNTAX: $y = J_1(g,w)$ *Cojoin* $g = J_{1112}(h)$;
END J;

In using the syntax of a structure, an instance of the layer of the structure definition can be obtained. In the *Cojoin* structure, there are actually four unspecified mappings besides the top node: J_1, J_{1112}, id_*^b, id_c^b. But in the use of the *Cojoin*, the value of "w" and "x" uniquely determines the particular id_*^b function. Likewise, the value of "h" and "x" uniquely determines the particular id_c^b. Thus, only J_1, J_{1112} need appear in the syntax of the *Cojoin* definition. The collective set of values that replaces the variables described in the syntax can be traced to each node of the structure definition. For example, if

$(a,b) = F(r,t,s)$

is defined as

$(a,b) = A(p,q,r)$ *Cojoin* $(p,q) = B(r,t)$

The first and second statement collectively form an instance of the *Cojoin* structure.

In this example,

"x" has the value "(r,t,s)"
"J" has the value "A"
"y" has the value "(a,b)"
"g" has the value "(p,q)"

*The syntax for set partition is

$^2y = f_1(^1x)$ *Otherwise* $^2y = f_2(^2x)$;
Partition of (x,y) *is any Partition*;

Here, the left superscript indicates a member of a member of a partition of "x".
The syntax for class partition is

$y_1 = f_1(x_1)$ *Include* $y_2 = f_2(x_2)$;

"w" has the value "r"
"J_{111}" has the value "B"

and, since the input to F has three components, "b" in the structure definition has the value "3", since "w" has the value "r", which has one component, "a" in the structure definition has the value "l", and so on. The structure syntax names the objects necessary so that an instance of the structure definition is obtained. Any instance of a structure must itself be an HOS system.

In the *Cojoin* structure, systems that communicate with each other can access the same value. Likewise, we define other structures; one so that independent subfunctions can access the same value (the *Coinclude*), and one so that subfunctions whose invocation depends on the value of the controller's input set need not access the entire set of variables of the input set (the *Coeither*) [32].

If the controller's function is $y = F(x)$ and $y = (y_1, y_2)$;

$g = id^{b}_{a}(x)$;

$h = id^{b}_{c}(x)$;

then

SYNTAX: $y_1 = A(g)$ *Coinclude* $y_2 = B(h)$;
SYNTAX: $y = A(g)$ *Coeither* $y = B(h)$;

Structures, in addition to the primitive control structures, e.g., *Cojoin*; can be used to define other structures. For example, the *Whereby*, defined with the *Cojoin*, gives us the facility to use constant symbols as operands of a function.

SYNTAX: Whereby y = W(h,CON);
The *Whereby* is used as in

y = x + 1

Here, the constant symbol "1" defines the particular K_{con} operation that makes the instance of the *Whereby* structure a function. Note also that the operator "+" (an instance of W) is used as an infix operator. In AXES we are free to use either prefix or infix notation, as desired.

We can visualize the instance of a structure as being either written down on a piece of paper by a human being, or to a register by a software or hardware process. To check an instance in an HOS system, the use of a *STRUCTURE* is compared with the *STRUCTURE* definition itself by an analyzer. All instances of a structure can be viewed as being supplied to the structure dynamically. The *STRUCTURE* for an asynchronous system, such as an operating system or the Higher Order Machine (HOM) [33], is a recursive relation relating each state of a machine to a previous state of the same machine within an instance of a machine system. To check the instances of an asynchronous machine in a real-time environment, an analyzer is used to check not only the use of the *STRUCTURE* with the *STRUCTURE* definition, itself, also to check to see that all the users of that *STRUCTURE* are consistent.

We indicate the potential happening of each machine instance by specifying a user system to be *"On"* the machine system, e.g., the syntax *Where* A *on* HOM specifies the initial nodal family of system A to be used by the first machine instance and the nodal family for each next recursive instance of the HOM function K_A to be determined by the ordering relationships of the nodal families within system A. A nodal family is a 3-tuple whose members are functions which stand in a particular relation (c.f. Appendix I). By indicating only "HOM" in the syntax of this structure, rather than, for example "y = HOM(x)," the state of the HOM remains hidden from the user.

II. Levels and Layers in AXES

In AXES we emphasize in our notation the separation of the layers within one system or between systems. The layer relationships are defined with the *where* statement in AXES.

In particular, we distinguish between in *instance of a layer* (representing one performance pass of a system)* and a *layer* (representing all performance passes of a system). We distinguish between *communication within one layer*, which always represents the same instance, and *communication between layers*, which takes place when an instance of one layer communicates with an instance of another layer (e.g., real-time asynchronous processes). We distinguish between (1) the system and the definition of that system, (2) the system and the description of that system, (3) the system and the implementation of that system, and (4) the system and the execution of that system.

A machine is a system which executes another system. There are dedicated machines, asynchronous machines, and asynchronous machines. A dedicated machine always performs the same function. Thus the "mapping" of an AXES *FUNCTION* could be viewed as a dedicated machine. A synchronous machine must only execute one system to completion before another system uses that machine. An AXES *OPERATION* could be viewed as a synchronous machine. An asynchronous machine may execute instances of more than one system before either system reaches execution completion. Thus, an AXES *STRUCTURE* can be viewed as an asynchronous machine.

The environment of a machine must be secure in that (1) a user should not have to be concerned with any of the details that have to do with its execution, and (2) a user should not be allowed to have visibility into another user's environment.

In an asynchronous machine there are several types of data, all of which must be maintained as secure data throughout all instances of the machine. These types of data include (1) temporary values which exist for one or more users, (2) values from another machine with respect to the machine itself as a user, (3) values with respect to the variables of the machine itself, (4) values which are functionally related to a previous instanceof the machine system, (5) values which are functionally related to a previous instance of the machine for a given instance of a user, and (6) values which are functionally related to a previous instance of the machine for a given instance of another machine.

The *definition* (dynamic state) of a system is equivalent to the formal semantics of a system. The *description* (static state) of a system is equivalent to the syntax of a system. The *implementation* (static state which includes a system, a machine to run that system, and the mechanisms necessary to relate that system to the machine) of a system is equivalent to that same system ready to be exercised. The execution (dynamic state which includes a system, a machine running that system, and mechanisms which relate that system to the machine) of a system is that system being exercised by a machine.

Not only must we be aware of the types of system layers, but we also must be aware of how many different definitions, descriptions, implementations, and executions are possible or potentially possible for one system. Most important, we must determine those states which are necessary and those which are not only unnecessary but which are causing serious difficulties in the development of a system. Towards this end it is necessary to have available a means for determining both the types and the nature and number of states within each type [32].

Sometimes the need for the definition of the layers of a system depends on how the system is to be developed and executed. On one project the users might wish to compile source code before a target system is ready to be executed. On another project users might wish to interpret the code in real time. Thus, not only must the layers of a system be determined, but also it must be determined when, how, and where transformations from one layer to another layer take place.

---

*as opposed to a *level* which is a step of refinement (or more explicit definition) within a given instance of a layer.

In the resource allocation process, a name is assigned to a value or a value to a name. Resource allocation also includes the ability to replace a name by an equivalent name or a value by an equivalent value. Sometimes one layer is produced from another layer by a third layer. Sometimes the description of a layer, as opposed to the layer itself, becomes the object of communication. Sometimes the same layer is resource allocated as a function to one layer and as data to another layer.

The general concepts of layer communication can be related to familiar examples. When two asynchronous processes are in the execution mode, a value from a given process is assigned to a name (or variable) associated with another process. Conversely, that other process assigns a value to the name associated with the first process. When an integer is implemented, a specific representation (or value) for a specific machine is assigned to the name representing the integer. When a compiler compiles an HOL program, it assigns names (or registers) to values in order to translate from one description to another. It also fulfills an implementation function in that operations and data are replaced with specific machine-dependent values. A compiler, in order to operate on an HOL program, must be able to read the description layer of the program as input for the translation process. In addition, it requires further input of its own in order to provide the implementation layer of the program. An OS system has a more complex job in that it is usually required to communicate with both the description of a system and the system itself. When a function is refined to a lower level of more detailed functions, the control integrity of the values and names from the parent layer must be maintained at the layer of the offspring.

Ideally, we want to be able to define a system, as abstractly as possible; describe that system in a syntax we can all relate to; verify that description; implement that system for a machine that will talk directly to that system; and collect the machine mechanisms, and only those mechanisms, that are necessary to execute a given system. In such a way we can eliminate dependencies on a particular primitive machine until the very end of a development process. For when we go from one definition to another, from one description to another, from one implementation to another, or from one execution to another, we are really resource allocating to another machine level. Thus, there should be no need to resource allocate for more than one given layer at a time.

APPENDIX III*

Some Specifications Relating TIME

In "AXES Syntax Description" [18] we provided algebraic specifications for six intrinsic abstract data types: booleans, properties, sets, naturals, integers, and rationals. We have now developed algebraic specifications for a number of other abstract data types that are of particular usefulness in the specification of operating systems and other machine-dependent software.

Time is an essential data type in any information processing system. It makes possible the scheduling of internal processes and provides an interface between the computer and external events. The essential characteristics of times are that they can be distinguished from each other (this is a universal characteristic of all data types, as far as we can tell); that they are linearly ordered, so that, given two different times, one always precedes the other; and that they support a notion of time flow.

The data type TIME can be specified in AXES as in Figure AIII-1. In the specification, Advance is the process of beginning at the time indicated by the first argument and advancing by the amount of time indicated by the second argument. Notafter is the relation that holds between two times if the first is earlier or simultaneous with the second. Equals is the relation of equality. Using Notafter, rather than something like Precedes, simplifies the axioms considerably. Precedes is defined later as an *OPERTION*, as seen in Figure AIII-2.

*Excerpted from [29] and [30].

DATA TYPE: TIME;
PRIMITIVE OPERATIONS:
time$_3$ = Advance(time$_1$, time$_2$);
boolean = Notafter (time$_1$,time$_2$);

AXIOMS:
Where t, t_1,t_2,t_3 *are* TIMES,
Where Notime *is a constant* TIME;
1. Equal (t,t) = True;
2. Equal (t_1,t_2) = Equal (t_2,t_1);
3. Entails (Equal (t_1,t_2) & Equal (t_2,t_3), Equal (t_1,t_3) = True;
4. Notafter (t,t) = True;
5. Entails (Notafter (t_1,t_2) & Notafter (t_2,t_3), Notafter (t_1,t_3)) = True;
6. Entails (Notafter (t_1,t_2) & Notafter (t_2,t_1), Equal (t_1,t_2)) = True;
7. Notafter (t_1,t_2)! Notafter (t_2,t_1) = True;
8. Advance (t,Notime) = t;
9. Advance (t_1,t_2) = Advance (t_2,t_1);
10. Advance (t_1, Advance (t_2,t_3)) = Advance (Advance(t_1,t_2),t_3);
11. Notafter (Advance(t_1,t_2),t_1) = Notafter(t_2,Notime);

END TIME;

Figure AIII—1. Data Type TIME

OPERATION: b = Preceeds?(t_1,t_2);
Where b,b_1,b_2,b_3 *are* BOOLEANS;
Where t_1,t_2 *are* TIMES;
b = And (b_1,b_2) *Join* (b_1,b_2) = F_0(t_1,t_2);
b_1 = Notafter(t_1,t_2) *Coinclude* b_2 = F(t_1,t_2);
b_2 = Not(b_3) *Join* b_3 = Equal(t_1,t_2);
END Preceeds?;
DERIVED OPERATION: time$_2$ = Reverse(time$_1$);
Where t *is a* TIME;
Advance (Reverse(t),t) = Notime;
END Reverse;
OPERATION: t_3 = Regress(t_2,t_3);
Where t_1,t_2,t_3,t_4 *are* TIMES;
t_1 = Advance(t_1,t_4) *Cojoin* t_4 = Reverse(t_2);
END Regress;

Figure AIII-2
Operations Using Data Type TIME

Axioms 1-3 characterize equality simply as an equivalence relation, because the conceptual structure of the time notion is not rich enough to support a specific equality relation, such as the one on naturals. Axioms 4-6 impose a partial ordering on times and Axiom 7 makes the ordering total (linear). Axiom 8 characterizes Notime as the time for which Advance has no effect. Axiom 9 says that Advance is commutative, and Axiom 10 says it is associative. Axioms 8 and 9 together guarantee that Notime behaves as a time origin and as a null time increment. Axiom 11 relates Advance to the linear time ordering.

REFERENCES

[1] Ramamoorthy, C.V. and So, H.H. "Appendix A to Requirements Engineering Research Recommendations", *Software Requirements and Specifications: Status and Perspectives*, August, 1977.

[2] Jackson, M.A. *Principles of Program Design*. Academic Press, N.Y., 1975.

[3] Bridge, R.F. and Thompson, E.W. "A Module Interface Specification Language," Information Systems Research Laboratory, University of Texas at Austin, Technical Report No. 163, December 1974.

[4] Guttag, J., Horowitz, E., and Musser, D. "The Design of Data Structure Specifications," *Proc. 2nd International Conference on Software Engineering*, October 1976.

[5] Robinson, L and Holt, R.C. "Formal Specifications for Solutions to Synchronization Problems," Computer Science Group, Stanford Research Institute, 1975.

[6] Computer Sciences Corporation, "A Users Guide to the Threads Management System," November 1973.

[7] Alford, M.W. "R-Nets: A Graph Model for Real-Time Software Requirements," *Proc. MRI Symposium on Computer Software Engineering*, April 1976.

[8] Davis, C.G. and Vick, C.R. "The Software Development System," *Proc. 2nd International Conference on Software Engineering*, October 1976.

[9] Hughes Aircraft Company. "1975 IR&D Structured Design Methodology, Vol.II: Structured Design," FR 76-17-289, 1975.

[10] Teichroew, D. and Hershey, E.A. III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, January 1977.

[11] IBM. "HIPO: Design Aid and Documentation Tool," IBM, SR20-9413-0, 1973.

[12] Ross, D. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, January 1977.

[13] Wilson, M.L. "The Information Automat Approach to Design and Implementation of Computer-Based Systems," IBM, Report IBM-FSD, June 1975.

[14] Hamilton, M. and Zeldin, S. "Some Preliminaries on System Specification," BMD Memo No. 2, Higher Order Software, Inc., Cambridge, MA, Sept. 16, 1977.

[15] "A Demonstration of AXES for NAVPAK," Technical Report No. 9, Higher Order Software, Inc., Cambridge, MA, September 1977.

[16] Fuchs, A.F., Velez, C.E., and Goad, C.C. "Orbit and Attitude State Recoveries from Landmark Data," *The Journal of Astronautical Sciences*, Vol. XXIII, No. 4, October - December 1975.

[17] Hamilton, M. and Zeldin, S. "Higher Order Software - A Methodology for Defining Software," *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 1, March 1976.

[18] Hamilton, M. and Zeldin, S. "AXES Syntax Description", Technical Report No. 4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

[19] Cushing, S. "The Software Security Problem and How to Solve It", Technical Report No. 6, Revision 1. Higher Order Software, Inc., Cambridge, MA, July 1977.

[20] Searle, J.R. "Review of J.M. Sadock, *Toward a Linguistic Theory of Speech Acts*." *Language 52*, 1976.

[21] Pankiewicz, R. "Specification Language Memo No. 8." The Charles Stark Draper Laboratory, Inc., Cambridge, MA, December 11, 1975.

[22] Straeter, T. "MUST (Multipurpose User-Oriented Technology Program Plan Preliminary." NASA/Langley Research Center, Hampton, VA, October 21, 1976.

[23] *Fundamentals of Mathematics*, "Analysis", Vol. III, ed. H. Behnke, et al, trans. S.H. Gould. The MIT Press, Cambridge, MA, 1977, pp. 471-475.

[24] Hamilton, M. and Zeldin, S. "The Foundations for AXES: A Specification Language Based on Completeness of Control." Doc. R-964. Draper Laboratory, Cambridge, MA, March 1976.

[25] O'Neill, E.M. and Stull, H.E. "Navpak Design for Landsat and Kalman Filter Applications," CSC/TM-77/6012, Computer Sciences Corporation, 1977.

[26] "Users Guide to MAC-360, Revised August 1973." The Charles Stark Draper Laboratory, Inc., Cambridge, MA.

[27] Hamilton, M. and Zeldin, S. "Integrated Software Development System/Higher Order Software Conceptual Description." Higher Order Software, Inc., Cambridge, MA, Nov. 1976.

[28] Hamilton, M. and Zeldin, S. "The Manager as an Abstract Systems Engineer." IEEE Catalog No. 77CH1258-3C, Sept. 1977.

[29] Heath, W. "Some Specifications for the Operating System of the Apollo Guidance Computer (AGC)" in *Techniques for Operating System Machines*, Higher Order Software, Inc., Cambridge, MA, July 1977.

[30] Cushing, S. and Heath, W. "Some Data Types for Operating Systems" in *Techniques for Operating System Machines*, Higher Order Software, Inc., July 1977.

[31] Cashman, M.W. "An Interview with Prof. Edsger W. Dijkstra," *Datamation*, Vol. 23, No. 5, May 1977.

[32] Hamilton, M. and Zeldin, S. "Operating System Machine with Respect to Resource Allocation" in *Techniques for Operating System Machines*, Higher Order Software, Inc., Cambridge, MA, July 1977.

[33] Heath, W. "A Higher Order Machine (HOM) for Higher Order Software (HOS)" in *Techniques for Operating System Machines*, Higher Order Software, Inc., Cambridge, MA, July 1977.

ACKNOWLEDGEMENTS

We would like to express appreciation to Andrea Davis, Gail Lopes and Roger Stoltz for help in preparation of this paper.