The Relationship Between Design and Verification

M. Hamilton and S. Zeldin
Higher Order Software, Inc., Cambridge, Massachusetts

The assumption is made here that a design process, in order to be effective, must include techniques that facilitate the effectiveness of the verification of the target design resulting from that process. The assumption is also made that these techniques can and should be universal in nature. That is, any system designer should be able to use these techniques to benefit his or her own design process and to check for the proper use of these techniques, both statically and automatically, with the aid of a common set of tools.

Once a set of universal techniques has been verified, there is no longer a need to verify such techniques each time a new system is designed. It follows, then, that there is no longer a need to verify or prevent those categories of problems that are known to exist no longer, given the correct use of those system design techniques that eliminates that class of problems.

Verification of a system design includes the identification of redundancies, logical incompleteness, and inconsistencies of a system definition, description, implementation, and execution. If a system design process inherently produces a design that no longer requires certain types of "after the fact" verification, many aspects previously associated with the verification process can be eliminated. We discuss our recent experiences in defining systems where we have attempted to show the relationship between design and verification. An example specification is used to demonstrate the properties of a system definition whose design supports elimination of unnecessary verification, maximum use of static verification, and minimum use of dynamic verification.

INTRODUCTION

A system development process is a system that develops another system. Such a system can be viewed as a process where each instance is continuously receiving requirements as inputs and producing specifications as outputs. In such a development system, *requirements* are those items that are desired or needed and *specifications* are the results that realize those requirements; one engineer's requirements could be another engineer's specifications [1].

There are several disciplines, or combinations thereof, that can occur as a development process. These disciplines include design, implementation, verification, management, and documentation. All of these disciplines also take place throughout a system development process and, depending on point of view, one engineer's design process could be viewed as another engineer's management, implementation, verification, or documentation process. Each of these disciplines is just as interchangeable with respect to each other, depending entirely on a given point of view.

A development process is viewed as a management process when it is considered with respect to its *control* of other disciplines.

A development process is viewed as a *documentation* process when it is considered with respect to its *description* of other disciplines.

A development process is viewed as a *design* process when it is considered with respect to its *definition* of other disciplines.

A development process is viewed as a *resource allocation* process if it is considered in terms of its *implementation* of other disciplines.

A development process is viewed as a *verification* process if it is viewed in terms of its *execution* of other disciplines.

A successful execution of a target system is directly dependent on a successful execution of a development process.

The design process is a focal point for all of the other disciplines. Not only does it determine if a

Address correspondence to M. Hamilton, President, or S. Zeldin, Vice-President, Higher Order Software, Inc., 806 Massachusetts Avenue, Cambridge, Massachusetts 02139.

system is going to work, but it also directly affects the effectiveness of the other disciplines. A design process, however, is not complete until the process itself or its results have been verified. It follows then that the verification process is a focal point for all the other disciplines as well.

For each step of design, there should be a "counterstep" of verification. This does not mean that for every new thought in the design process it is necessary to have a one-to-one corresponding "thought-back" for the entire verification process; quite the contrary—not only would such a method be time consuming, but it would also not be reliable. At times, in fact, the process of design could be interpreted as 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 requirement are designed out of the system. What is left is the second-order verification that guarantees that unnecessary verification requirements with respect to design have been eliminated, and then a need to verify only that which is truly part of the original intent of the design.

Many engineers desire to improve their own design techniques. These design techniques include techniques for producing the design for a solution to a particular problem as well as the design for the process that 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) in different ways. In the context of a typical system development process, design could be the process of developing concepts, requirements, specifications, code, or computers; likewise, design could be the process of going from a concept to a set of requirements, from requirements to a set of specifications, from specifications to a set of code, or from code to a set of computers. In each of these processes, a designer considers the task of preparing a design to reside eventually in a "machine" environment (e.g., a computer for a software system). One of the problems in a design approach is that the designer either worries unnecessarily about design considerations irrelevant to his own process or bypasses certain design considerations under the impression that they have already been, or will later be, handled by someone else.

A designer should be concerned with the design that is to reside in that designer's development phase, and that design *only*. Each designer goes through the same generic process but should be applying that process to a different phase of the overall application. Thus the inputs and outputs of that 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 a particular 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 perform a set of effective implementations.

The verification process exists for the purpose of finding errors in the output of a design process. There are those errors that can always be found by automated means (provided that the design process incorporates proper procedures) and those that cannot always be found by automated means. We divide the former into two kinds. The first is determined by analyzing a system (or a set of subsystems) on a stand-alone basis. For example, if a specification has an inconsistency among its functions or if a computer program has a data conflict, such errors can be found by analyzing only the system in question. In this case, it is possible to design the system in such a way that checks can be made with respect to interface correctness (i.e., logical completeness, consistency, and nonredundancy). The second kind is that which is determined by checking one development layer with the development layer from which it evolved. An example of such a comparison is that of checking a computer program against its specification. Again, checks can be made with respect to interface correctness between layers.

Errors that cannot always be found by automated means are those that are determined by checking a development layer against the intent of the original designer. A small percent of large-system development errors fall into this category [2]. This problem is alleviated by providing both techniques that automatically eliminate other sources of errors and those that support the verification engineer in finding the remaining errors.

An ordering for a verification process then becomes apparent. One first concentrates on eliminating certain types of verification by following design principles that make this possible. This is the *conceptual phase* of the verification process. Then one concentrates on using these principles correctly. A check for correct use of principles can be performed both statically and automatically. This is the *static phase* of the verification process. Finally, one concentrates on verifying only that part of the design which is concerned solely with the performance of a particular algorithm. This is the *dynamic phase* of the verification process.

THE RATIONALE FOR USING A METHODOLOGY

An effective methodology can assist a designer with respect to both the design itself and its verification. There are concerns, however, on the part of some project managers with regard to introducing a new methodology into an organization especially when it affects an ongoing project. Often, unfortunately, methods are never improved because of insufficient time to introduce new methods when, in fact, part of the reason for not having enough time is the methods already being used.

How does one convince project managers of the benefits of introducing a methodology into their organization (excluding success stories about competitors)? Project managers, whose first priority is to deliver items that work, and work on time, are in the majority and must have some proof that the introduction of a methodology will serve their needs better. We have found that an effective way to demonstrate a methodology within a given project is to select a module within that project's environment and to show the differences in definition of that module using the new methodology as opposed to the methods already being used on that particular project. Invariably, the most significant results are those where the use of certain design techniques eliminates some traditional categories of errors. The power of new concepts is often realized when those errors are uncovered in an engineer's own system, especially when that module is thought to be already working!

Once a project manager sees that some methodology can be more effective than none (i.e., no common adopted methodology), an interest develops with respect to other methodologies. Which one is best? How do we choose between one and another? It then becomes apparent that there should be a common set of criteria by which to compare methodologies.

Some project managers are much harder to convince than others, with respect to using an effective methodology, because they are fortunate enough to have all "smart" people. It is true that the smartest person, by definition, would apply an effective methodology. An effective methodology, however, applied in common by several smart people, would far exceed the advantages of each smart person applying techniques in an ad hoc manner, since all the intricacies of a complex system are by nature beyond the grasp of any human being. The designs of all smart people must still be integrated. Thus a manager can be much more effective by defining a standard means to integrate the methods of these people before the fact rather than after.

Once a project manager decides to adopt a certain methodology, the immediate problem becomes how to implement it without impacting deliverable items of an ongoing project. In this case, a project manager can be assisted in using those aspects of the methodology that either make results more visible, find or prevent errors, or do any of these more quickly. Once these incremental techniques are introduced, engineers start feeling more at home with those aspects of the methodology and are much more prepared to start using other aspects when the proper time comes (e.g., the start of a new project).

Some progress has been made with respect to a particular project when the managers say "We are convinced, but how do we convince our engineers?" It often has been the case in such an environment that their engineers say "We are convinced, but how do we convince our management?"

Finally, say, the engineers and their management are convinced of the practical advantages of using an effective methodology but become nostalgic for the old days, thinking that poorer methods left more room for creativity. It is true that an effective methodology provides more constraints for the designers but *only* in the area of preventing the production of errors. As a result, creative designers should be *less* constrained in producing better designs. Once the project manager recognizes this, selection of a methodology is imminent.

DESIRABLE PROPERTIES FOR A METHODOLOGY

A methodology can support but never replace a designer. A tool can be developed to replace some of the designer's functions in general or even all of them for a particular project; however, the designer still has the prerogative to create new designs and design new uses for the same tool or new tools for different uses.

Too often the same problems exist in the development of methodologies as do in the problems the methodologies are intended to address. That is, there are often inconsistencies within a methodology. In addition, improvements to a methodology are often ad hoc, and modifications to a methodology to fix or enhance it are made to already existing modifications. Likewise, in the attempt to select an existing methodology, there is always a risk of comparing (1) techniques addressing very different problems, (2) techniques intending to address a problem, but not effectively addressing it at all, (3) techniques with respect to nonexistent or ill-defined requirements, (4) the "syntax" of methodologies instead of the "semantics" of methodologies, (5) techniques based on unfamiliar paradigms with preconceived notions, (6) techniques addressing the wrong problems or those that are "in the noise," (7) techniques with respect to completion or amount of use rather than to the problems they are solving, and (8) techniques with respect to the algorithms they are being used to define.

There are many methodologies today whose intent is to provide standards and techniques to assist the engineer in the design and verification process [3]. The developers of these methodologies are all proponents of reliable designs, and most methodologies advocate some similar techniques towards this aim. For example, it is commonly accepted that it is beneficial to produce a hierarchical breakdown of a given design in order to provide more manageable pieces with which to work. However, there are variations among methodologies. Some emphasize a concentration of data flow as opposed to functional flow [4–7]; others, just the opposite [8–10], or both functional and data flow equally [11]. Still others emphasize documentation standards [12,13], graphical notation [14], or semantic representation [15].

There are certainly positive aspects in many of these methodologies and, in particular, in what they are trying to obtain. However, to make comparisons among them or to determine the effectiveness of individual ones, it is necessary to determine the properties by which to make those comparisons.

From our own experience in developing large systems, we have determined a checklist of properties by which to analyze the techniques or the methodology being used by a particular project. We believe that these properties are necessary if a methodology is to be effective in the design and verification processes of a large system development.

We make the assumption that a methodology should have techniques for defining systems that are consistent and logically complete; but these techniques are useful only if they are within themselves consistent and logically complete, both with respect to each other and to the system to which they are being applied.

A methodology should have a standard set of definitions that resides in a well-publicized and evolving glossary. In our experience, we discovered that a mere change in the definitions of such terms as *error* and *system* could have far-reaching practical implications on a large system development process [16].

A methodology should have the mechanisms to define *all* of the relationships that exist in a system environment [17]. This includes communication within and among systems and the resource allocation[1] that provides for such communication. Thus, not only must all data, data flow, functions, and functional flow be able to be defined explicitly, but the relationships (and control of the relationships) between data and data, function and function, and data and function must be able to be defined within any given system environment.

A methodology should have the mechanisms to define *all* of the relationships that exist between possible viewpoints (or development layers) of a system. If, for example, one is concerned with a definition of a system, it is viewed with respect to what it is supposed to do. If one is concerned with a description of a system, it is viewed with respect to whether or not the definition is effectively conveyed. If one is concerned with an implementation of a system, it is viewed with respect to whether or not the system is constructed to do what it is supposed to do. If one is concerned with an execution of a system, it is viewed with respect to whether or not the system does what it is supposed to do. Whereas the description and implementation layers of a system represent static views, the definition and execution layers of a system represent dynamic views.

A methodology should have the mechanisms consistently and completely to define an object and its relationships *formally*. That is, every system in the environment of an object system (people, hardware, tools, software) should be able to understand a definition of an object and its relationships the same way.

A methodology should provide for *modularity*. That is, any change should be able to be made locally (with respect to levels and layers of development), 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, through-

[1]*Resource allocation* is the process (or system) that prepares one system to communicate with another system. (Such a process is, of course, a communication process with respect to the resource allocation of itself.) We define resource allocation in this way because of our finding that various engineers, including numerical analysts, programmers, hardware designers, and others, use "resource allocation" to mean very different things that reflect each of their specific interests. A definition was sought that described the process that focused on the fundamental feature of allocating resources, regardless of the characteristics of the specific resource. By having a definition that is concerned with the *general* property of resource allocation, ad hoc formulations are not required. In addition, one can then ask some fundamental questions about the ways in which systems can be constructed, regardless of the specific project. We have sought other definitions as well with the same aim in mind, i.e., to uncover fundamentals on which systems can be designed, constructed, and verified.

out other systems within that system's environment, and throughout all evolutions of the development of that system.

A methodology should provide a set of *primitive standard mechanisms* that are used both for defining and verifying a system in the form of a hierarchy.

A methodology should provide for a library of an *evolving* set of more powerful (with respect to simplicity and abstraction) mechanisms based on the standard set of primitive mechanisms. Having an extensible library of mechanisms 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 Arabic numerals available to use in performing long division when all the rest are still trying to use Roman numerals [18]?).

A methodology should allow system engineers to communicate in a language, with common semantic primitives and a *familiar dialect,* that is extensible, flexible, and serves as a "library" of common data and structure mechanisms.

A methodology should provide for a *development model,* including a set of definitions, tools and techniques, that supports a given system development process.

Finally, 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.

It is no coincidence that our own methodology has evolved with properties that correspond to those which we consider to be desirable ones for designing and verifying systems, since it was our direct experience with large systems that yielded the basis of the methodology of Higher Order Software (HOS).

SOME PROPERTIES OF HOS IN TERMS OF AXES

AXES, a specification language based on HOS, is a formal notation for writing definitions of systems. Although it is not a programming language, AXES is a complete and well-defined language capable of being analyzed by a computer. AXES [19] provides mechanisms to define data types (in order to identify objects), functions (in order to relate objects of types), and structures (in order to relate functions). AXES is the vehicle to define a system so that interface specifications can be automatically checked statically. The foundations of AXES are based on a set of control axioms derived from empirical data of large systems [2] and on the assumption of the existence of a universal set of objects. Figure 1 illustrates the evolvement of the primitive AXES mechanisms from the control axioms and the existence of the objects used to define systems. Each axiom describes a relation of immediate domination with respect to a functional system. We call the union of these relations control. From these axioms a set of three primitive control structures have been derived [20]. The primitive control structures identify control schemata on sets of objects. From the assumption that we can identify an object or a set of objects, a mechanism for defining an algebra for each distinct set of objects is provided in AXES. Each algebra takes the form of a set of axioms that relate operations applied to objects of a type. To form a system, new control structures are defined in terms of the primitive structures (Figure 2), or in terms of other nonprimitive control structures (Figure 3). Operations are defined implicitly by deriving them mathematically from the axioms on a type or explicitly in terms of control structures using already defined operations on a type. When an operation is defined both implicitly and explicitly, the intent of the specification can be cross-checked for correctness.

Once we have a library of control structures, data types, operations, and derived operations, we are ready to form a particular AXES definition using these mechanisms (Figure 4); Figure 5, a graphical illustration of an AXES definition, demonstrates the integration of the concepts illustrated in Figures 1–4. Here the top node of the system is concerned only with the top-level function of defining all predictable systems, whereas the second-level nodes are concerned with

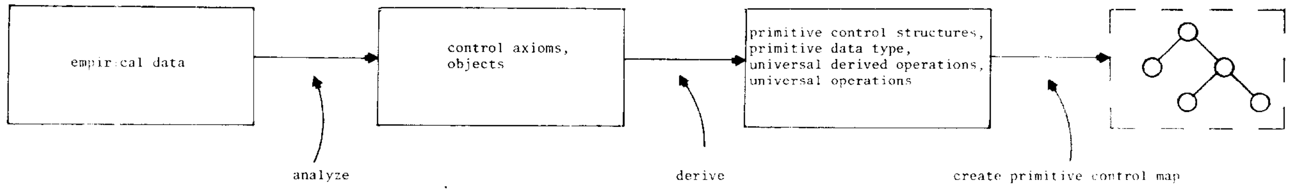

empirical data
control axioms, objects
primitive control structures, primitive data type, universal derived operations, universal operations
analyze
derive
create primitive control map

Figure 1. Define primitive AXES mechanisms.

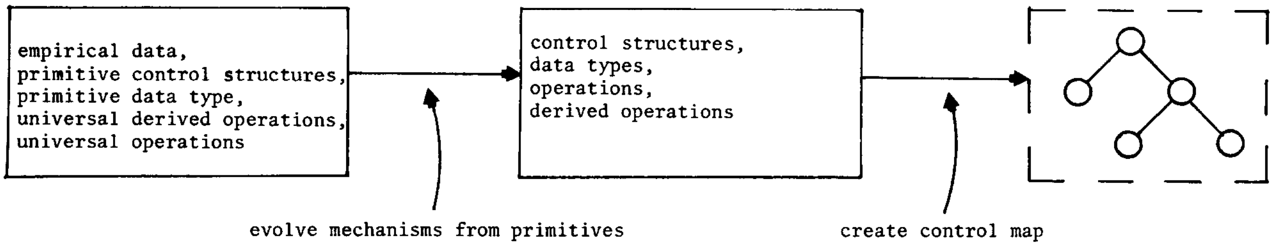


Figure 2. Evolve new AXES derived mechanisms from primitives.

functions such as analyzing the empirical data to produce axioms and objects. These axioms and objects in turn are used as input for deriving primitive control structures, a primitive data type, universal derived operations, and universal operations. These objects are then used as inputs for defining systems. The third level represents the decision "go" or "no go." That is, if there are no empirical data left, all systems in the world are defined. If there are empirical data left, there are more systems in the world to define. The fourth level represents a recursive pass of "define systems" (on the second level) and the whole process of evolvement starts again.

AXES systems are those systems that are defined directly with AXES or with mechanisms defined with AXES. AXES was designed to have a capability for defining both the relationships within a given system environment and between development layers of that system's development process.

Since AXES systems are HOS based and HOS is based on a consistent set of rules or axioms, all AXES systems have a *formal* set of properties. HOS emphasizes completeness of control, where control is defined by axioms that establish the relationships for invocation of functions, input and output, input access rights, output access rights, error detection and recovery, and ordering of functions. Control affects an object, the relationships of an object, and the relationships of the development of an object. Everyone defining a module using AXES must follow the same rules in constructing the structure of that module. For example, not only is every object in a system controlled, but every object has a unique controller. The intent is to eliminate ambiguity in understanding either the behavior of an object or the behavior of that object's relationships.

There are many aspects of *modularity* inherent in AXES systems. For example, the definition of the behavior of an object is completely separated from the definition that uses the object; the definition of a development layer is independent from those layers that evolve from it (for example, the specification of a system is independent of its implementation); and AXES provides a way of defining control mechanisms that are functional, as opposed to procedural. (The definition of a control mechanism specifies total ordering among functions, which implies that the description of that definition is order independent. This does not rule out the possibility, however, of describing a procedural process as a functional mechanism.)

AXES systems also display other distinctive properties of modularity:

Both the mechanisms defined with AXES and the systems defined with these mechanisms behave as if they are "instructions"; e.g., a given control structure has no knowledge about a higher-level control structure.

Control, or the chain of command, can be traced directly on a control map. As a result function flow (including both input and output) can be traced di-

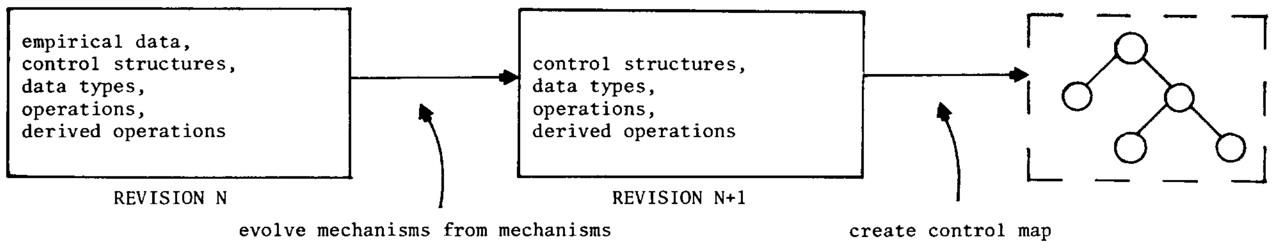


Figure 3. Evolve new AXES derived mechanisms from existing AXES mechanisms.

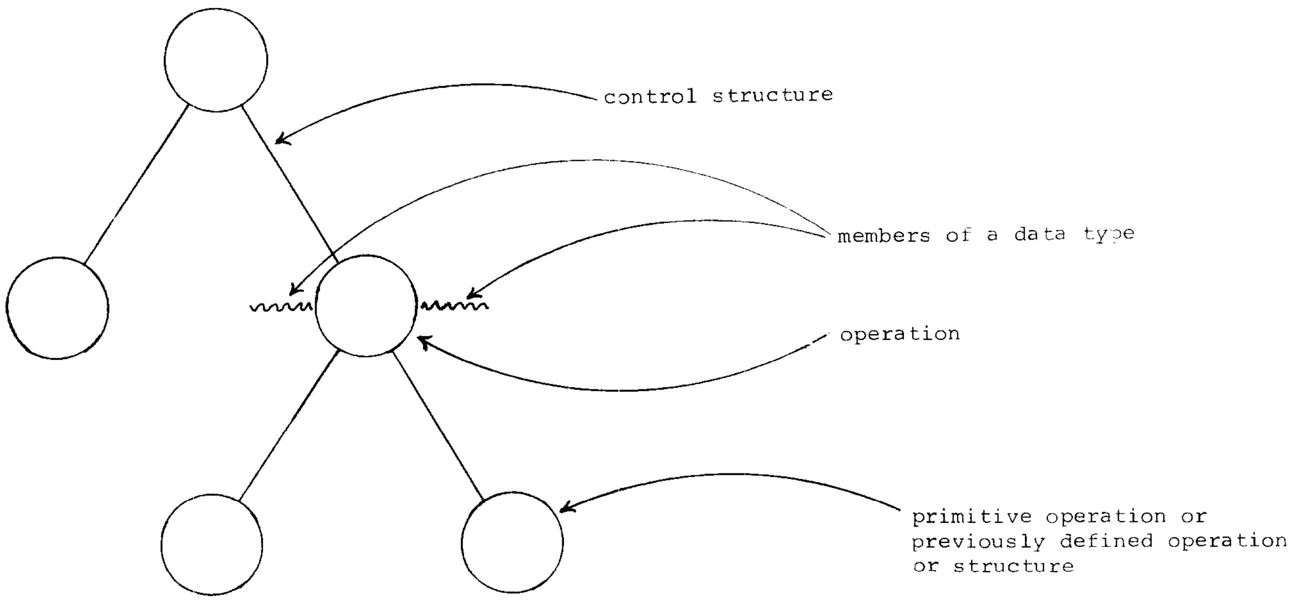

control structure
members of a data type
operation
primitive operation or previously defined operation or structure

Figure 4. Bird's-eye view of control map constructed from AXES library.

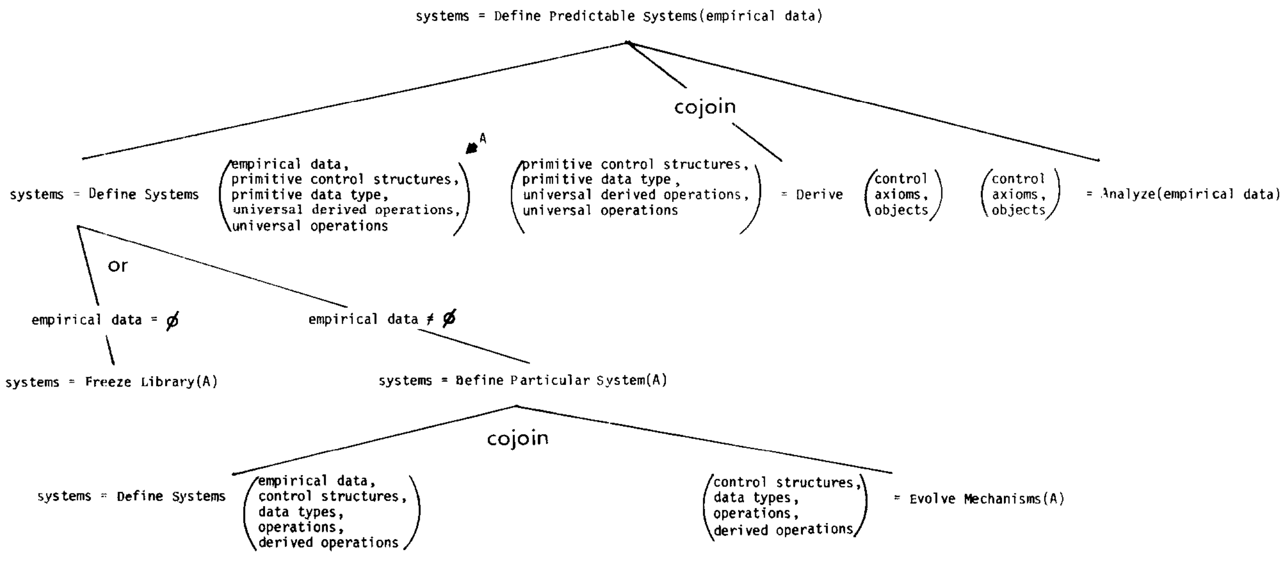
rectly. Changes can be traced and changes can be made locally.

The single-reference, single-assignment property of AXES systems provides for an interesting set of resource allocation alternatives.

An AXES definition can be viewed as a specification that can be directly implemented in terms of a distributed processing environment. Other types of implementations (e.g., multiprogramming and sequential processing) are special cases of a distributed processing environment.

We have found, however, that other aspects of a functional specification should be treated as an integrated whole and *not* artificially separated for the sake of modularity; for such a separation has often resulted in enhancing the errors in a system. That is, at any node in an AXES hierarchy a user is able to identify an object with respect to an integrated set of *aspects of control* that inherently incorporates *types* of definitional *models* and *viewpoints* of those models.

Figure 5. Axes control map defining system of defining AXES systems.


systems = Define Predictable Systems(empirical data)
cojoin
systems = Define Systems (empirical data, primitive control structures, primitive data type, universal derived operations, universal operations)
A
(primitive control structures, primitive data type, universal derived operations, universal operations) = Derive (control axioms, objects) (control axioms, objects) = Analyze(empirical data)
or
empirical data = ∅
empirical data ≠ ∅
systems = Freeze Library(A)
systems = Define Particular System(A)
cojoin
systems = Define Systems (empirical data, control structures, data types, operations, derived operations)
(control structures, data types, operations, derived operations) = Evolve Mechanisms(A)

With respect to aspects of control, the input–output behavior of a system is not treated apart from other aspects, such as ordering (which includes priority, degree of concurrency, and synchronization) and error handling (which includes detection and recovery). With respect to models, every node of an AXES hierarchy represents a controller (the definition of which, in terms of structures, is a user model) that relates functions (the definitions of which, in terms of operations, are a functional model). Those functions in turn relate input to output (the definitions of which, in terms of data types, are an informational model).

Furthermore, every node on an AXES hierarchy is expressed in terms of its viewpoints (i.e., definition, description, implementation, and execution). Whereas each node as an object is *defined* in terms of AXES statements, the *description* of each node exists in the form of the "pencil marks" of AXES statements. The *implementation* of an AXES object is performed by using the description of that object as an input in determining an equivalent form of *definition* of that object for purposes of residing on a particular machine environment. An *execution* of an object occurs when that object is assigned to a name. (In AXES an execution for a particular system begins once an object is assigned to one of its names. A system has completely been executed once objects have been assigned to all of its names. Theoretically, then, one could describe, implement, and execute a system by the very fact that its definition exists!)

The fact that aspects of control, types of models, and viewpoints are inherently integrated with respect to each other at a given node significantly simplifies any given system definition. Furthermore, each user in a development process of an AXES system is able to relate to every node in a unique way (the manager with respect to control, the designer with respect to definition, etc.).

With AXES, any system can be defined in terms of a set of *standard primitives*. The primitive control structures provide rules for the definition of dependent functions (e.g., sequential processing), independent functions (e.g., parallel processing), and selection of functions (e.g., reconfiguration). Combinations of primitives form more abstract control structures. It is also possible to tell when a design has been completed since a complete design is one that has been hierarchically decomposed until all terminal nodes of a control structure represent primitive operations or previously defined structures and operations. Since AXES has a common set of specification primitives (i.e., a common specification "machine"), we envision common tools, such as an analyzer to check for correct interfaces and a resource allocation tool to prepare a specification for a particular machine environment [1].

Although a system can be defined directly with AXES, a more powerful use of AXES can be made by defining systems that are themselves a set of *evolving mechanisms* for defining systems. Thus a set of specification "macros" can collectively form a "language" (or management standards) for defining a particular system or family of systems. It is envisioned that each new system user is able either to use a subset of already defined statements in an AXES based library or to add new statements since the AXES language system provides for extensibility with respect to both structure and data definitions.

AXES provides a user with the capability of using *familiar dialects* for a control structure or data type. Thus, for example, a manufacturing project might have its own set of specification statements to use as a means of standardization, as might an avionics project; but both should be able to intercommunicate since these structures are based on standard primitive mechanisms to which they can both relate.

AXES is intended to provide the mechanisms to define both a *development model* and the management of a system development model, which uses that development model, as systems, since that is, after all, what they both are. Within the context of a complete development process, a means is provided to define management standards, definitions, milestones, disciplines, phases, tools and techniques, and the relationships among all the various components within a development process. A first step in this direction can be found in [1,17].

SOME PRELIMINARIES ON **AXES**

AXES uses the functional notation

$$y = f(x), \qquad (1)$$

where x is the input, y is the output, and f is the operation applied to x to produce y.

In attempting to define a system as a function, we assert that for every value of "x"[2] we expect to produce one and only one value for "y." That is, we expect the system to produce predictably the same result each time we apply f to a particular value.

Now, we must incorporate into our definition a means to identify all of the acceptable inputs and outputs. In AXES, each input and output value is

[2]To differentiate an object from its name, the "use–mention distinction" is used throughout this paper [35]. That is, to form the name of a given name (or written symbol of any kind) we include that name (or symbol) in quotation marks.

associated with a particular set of values, called a *data type*. The syntax for each algebra, or data type definition, is similar to that used by Guttag [21], but the semantics associated with each algebra is similar to the concepts described by Hoare [22]. The semantics of our algebras assumes the *existence* of objects (see Appendix 1). That is, when we define a system, as in (1), we assume the values x and y to exist and that when f is applied to x, y corresponds to the value x.

In many systems, especially large ones, it is often not readily apparent which input values correspond to the system's intended function until the system is decomposed into smaller pieces. Although we start with a large set of "seemingly" acceptable values, a predictive system must be able to identify "truly" acceptable inputs or to produce an indication that a particular function will not be able to perform its intended function. To identify a system's intended function, we make use of a distinguished value, which we call Reject (Figure 6). This distinguished value is a member of every data type. If an input value corresponds to the value Reject as an output, then the function applied to that input is said to have detected an error. A function applied to an input value of which Reject is a component [e.g., the value (1,3, Reject)] may either assign Reject as an output value, or "recover" from the error by assigning an output value other then Reject.

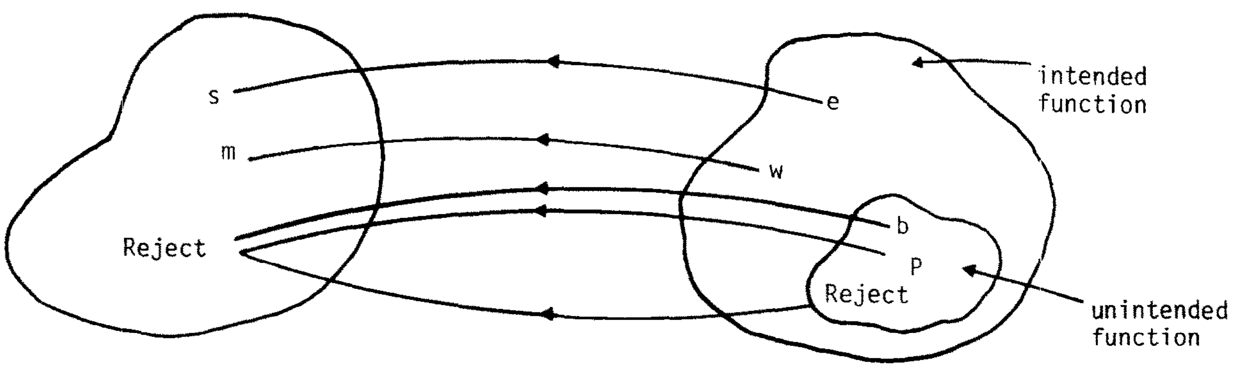
Once we have identified all acceptable inputs and outputs of our system, we need a means to describe the relationship between the input and output, sometimes called the performance of the function. Relations on a set of operations give rise to a hierarchical structure, like the structure appearing in Figure 7. At each node in our hierarchy we shall put a function, with the intent that at any level of our hierarchy (a level is a set of immediate dominated nodes with respect to a particular node, sometimes called a step of refinement), we can relate the functions at that level to the function at the node immediately dominating them.

We need a set of rules to determine a level, and a set of rules to determine whether we want to create a level. To determine a level, we want all the functions at the nodes of a level to be necessary and sufficient for the replacement of the function at the node directly controlling these functions (Figure 8). This will ensure that we get no more or no less than we want, i.e., that our level is logically complete.

As we continue to build our hierarchy, each level completely replacing the function at the node directly above it, we must be able to define each point at which we want to stop. We stop when we reach a function whose behavior, i.e., its input and output relation, has been defined in terms of other operations on a defined type, and our specification is complete when we determine each stopping point. Now, if we know the behavior of each function at a bottom level and how it relates to the other functions at that same level, we know the behavior of the node directly above it. With the same reasoning, we know the behavior of the functions at each level successively closer to the root, or top node; similarly, we end up knowing the behavior of the root function itself. Thus the behavior of the top node is ultimately determined by the behavior of the collective set of bottom nodes (Figure 9).

Now we also want to assure logical consistency for a level. Since our intent, in the end, is to understand the behavior of the function at the top node, every time we talk about a value of that function we want to assure ourselves that we are talking about the same value at the level directly dominated by that function; that is, we want to be able to determine which values match up with which functions. To talk about a value we use its name, or variable. We want

Figure 6. Acceptable vs intended values of a function.



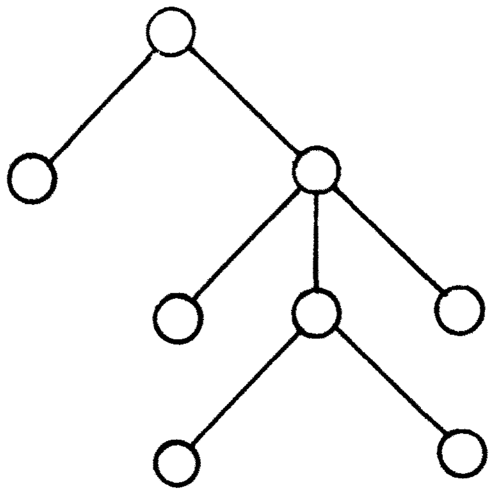


Figure 7. Hierarchical system structure.

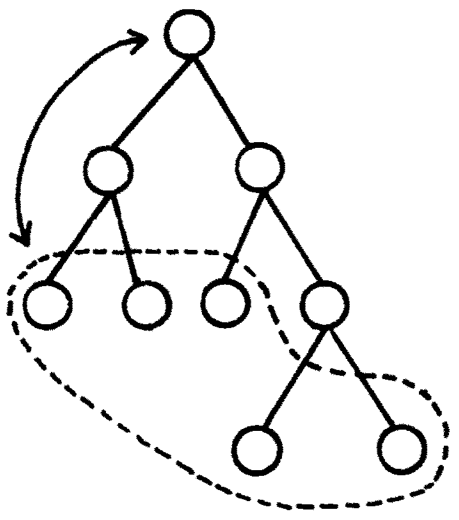


Figure 9. Endpoint completeness.

to be consistent about input variables (Figure 10) and output variables (Figure 11). To avoid specification errors in naming values, a particular name is always associated with the same value as we travel down the hierarchy.

We also want to be able to determine which functions are more important than others. For example, a function is always more important than the functions at the level dominated by that function, and at a particular level each function is assigned an importance with respect to each other function at that level (Figure 12). Among other things, we can use this information to implement specific timing relationships, both relative and absolute, without conflict.

The above concepts, defined in terms of axioms [2], are inherent in every AXES defined mechanism. Now let us see what all this means if we try to specify a particular function. For the purpose of demonstration we select the function that is to produce the greatest common divisor (GCD) of two natural numbers.

To define GCD implicitly, we have the following AXES definition [19, Appendix IV].

Derived Operation: n_3 = GCD(n_1, n_2);
 where n_1, n_2, n_3, n **are** Naturals;
 Factor (GCD(n_1, n_2), n_1) = True; (2)
 Factor (GCD(n_1, n_2), n_2) = True; (3)
 Entails (And(And(Factor(n, n_1), Factor(n, n_2)),
 Not(?Equal?(n, Zero))), Factor(n, GCD(n_1, n_2)))
 = True; (4)
end GCD;

Each operation in terms of which GCD is defined is checked to determine if it has been previously defined. Each defined operation must eventually be able to be traced to a definition of a primitive operation on a defined type (Figure 13). This could be performed automatically. Here, GCD is defined in terms of Factor, an operation on two naturals that

Figure 8. Level completeness.

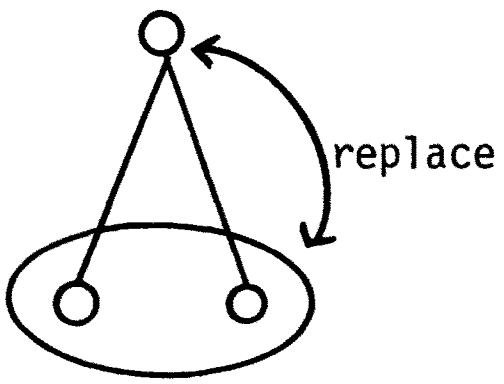
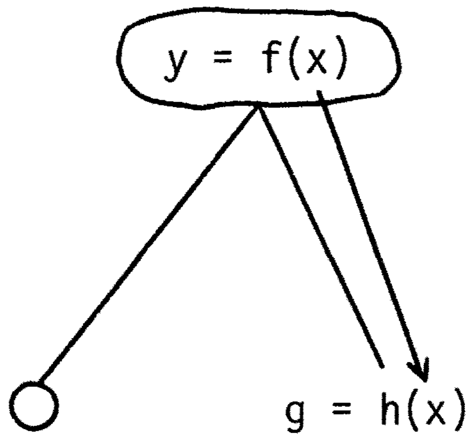


Figure 10. Tracing input names.



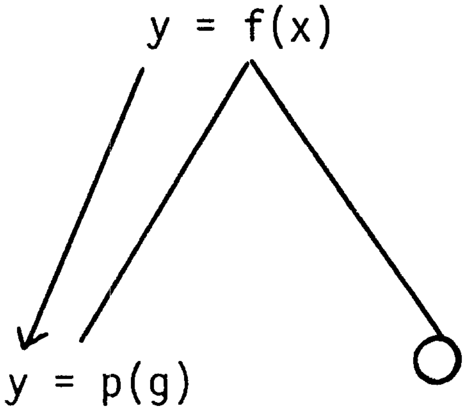


Figure 11. Tracing output names.

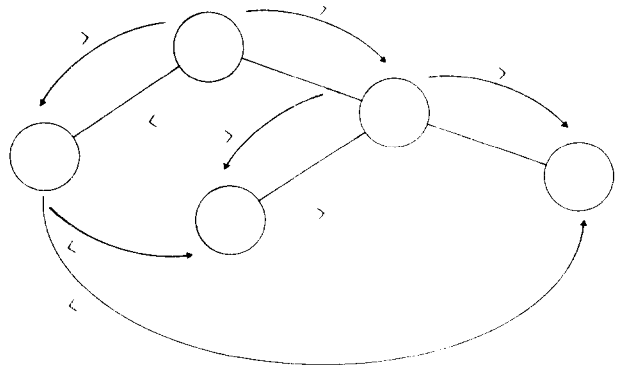


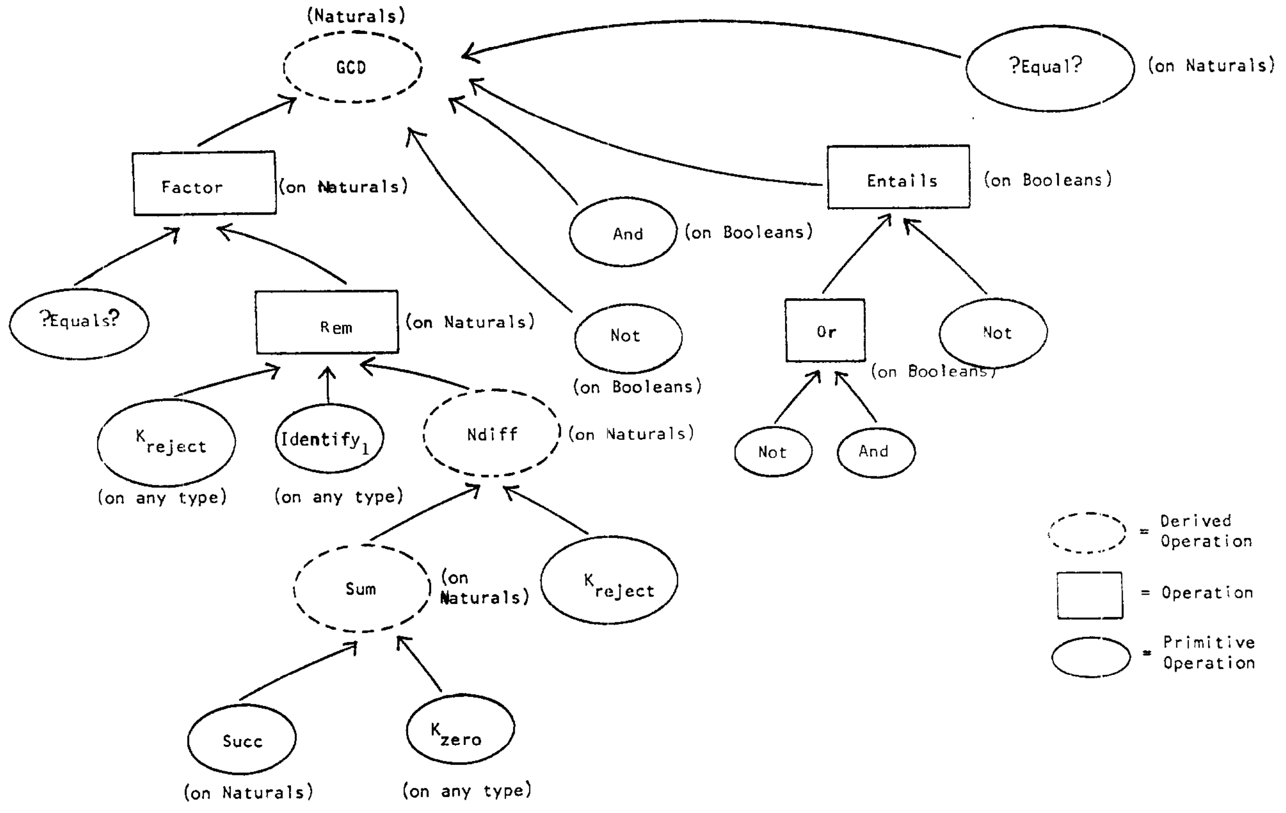
Figure 12. Complete ordering relationships.

produces a Boolean, which tells us when one natural is a factor of another; Entails, an operation on two Booleans that produces a Boolean, provides a notion of entailment; And, an operation on two Booleans that produces a Boolean, and Not, an operation on a Boolean that produces a Boolean, have the usual logical meaning on Booleans; and ?Equal?, which is a primitive operation on two naturals, provides us with a notion of equality for naturals.

Each statement about GCD in the derived operation definition is an assertion about GCD. The set of statements about GCD must itself be shown to be consistent with the axioms of the type natural from which it is derived. The proof that GCD is consistent with these axioms is performed manually.

The technique of defining derived operations in AXES was introduced to limit the complexity of defining a type. The idea here is to define a type with the *least* number of axioms required to characterize the behavior of the objects of a type: then we can build

Figure 13. Tracing definitions of operations to primitive operations on a type.



on our basic definitions and simplify our task of proving the consistency of a set of axioms. In a sense, we wind up building a hierarchy of axioms. Without a concept of derived operations we would either have to limit the number of operations allowable on a type, as suggested in languages like CLU [23], which might make a large system specification quite cumbersome to understand, or add a few more axioms to our type definition each time we introduced a new operation, as suggested by Guttag [24], thereby imposing on ourselves the task of proving the consistency of possibly hundreds of axioms in a large system environment.

```
Operation: $y = \mathrm{GCD}(x_0, y_0)$;
        where $(x_0, y_0, y)$ are Naturals,
            $(x_1, y_1)$ are Naturals;
  GCD: $y = A(x_0, y_0)\big|_{x \neq 0 \text{ Or } y \neq 0}$        or
                                          $y = \mathrm{Reject}\big|_{x_0 = 0 \text{ And } y = 0}$ ;
  A: $y = y_0\big|_{x_0 = 0}$                      coor
                                          $y = B(x_0, y_0)\big|_{x_0 \neq 0}$ ;
  B: $y = A(x_1, y_1)$                     join
                                          $(x_1, y_1) = C(x_0, y_0)$;
  C: $(x_1, y_1) = D(x_0, y_0)\big|_{y_0 \geq x_0}$         or
                                          $(x_1, y_1) = \mathrm{Xch}(x_0, y_0)\big|_{y_0 < x_0}$ ;
  D: $x_1 = x_0$                           coinclude
                                          $y_1 = y_0 - x_0$;
end GCD;
```

The explicit algorithm shown in the operation above, introduced by Manna and Waldinger [25], is defined here in terms of structures that relate operations. 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 that 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 that 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 the term "function" is slightly different from that in mathematics.

In the operation definition for GCD, a hierarchy of functions is obtained (Figure 14) by using defined structures and "plugging in" particular operations and particular variables to represent the inputs and outputs. With respect to the GCD definition, A, B, Clone$_1$, C, D, Xch, K_{Reject}, and Ndiff are functions. With respect to GCD as an object to be used, GCD is an operation because a user can supply his own particular input and output variables to use GCD as a function for another system definition. [Note that the alternative forms for Clone$_1$ and K_{Reject} are used in the corresponding AXES statements (see Appendix 2) and an alternative infix form for Ndiff using the symbol "$-$" for "Ndiff" is used in the AXES description.]

The particular structures used in the GCD operation definition are **or**, **coor**, **join**, and **coinclude.** The definitions for these structures can be found in Appendix 2. **or** and **join** are two of the three primitive structures. The third primitive structure, which was not used for GCD, is **include,** the definition of which also appears in Appendix 2.

All of the nonprimitive structures used to define GCD explicitly are defined in terms of the primitive structures. For example, in Figure 15 the **coor** structure is built from the **join**, **or**, and **each** structures. The first level of decomposition for GCD is defined in terms of the primitive **or** structure. In this case the **or** is being used to define the relationship among GCD, A, and K_{Reject}.

In using the set partition control structure (or AXES "**or**" statement) for the relationship among GCD, A, and K_{Reject}, we can check that the input and output to GCD is the same as the input and output to both A and K_{Reject}. In this case, A and K_{Reject} are partial functions of GCD. The control schema for set partition assumes the existence of data type Property (of T) (see [19]), where T is a type. A property is something that maps other things onto truth values. In Figure 14, where **or** is used for GCD, A and K_{Reject} "$x_0 \neq 0$ Or $y_0 \neq 0$" is a particular property on naturals and "$x_0 = 0$ And $y_0 = 0$" is another particular property on naturals. For a set partition, the two properties are mutually exclusive, but one or the other must apply for any value of the input set of a function at the node controlling a level.

To decide whether to decompose functions K_{Reject} and A, we determine whether either function has already been defined. Since K_{Reject} is an already defined operation on any type (it produces a Reject value for any input; see Appendix 2), we know we need not decompose it. A, on the other hand, has not been defined elsewhere, so we proceed to decompose it. The level of decomposition for A is defined in terms of the nonprimitive **coor** structure.

Only B must be decomposed as we proceed down the hierarchy associated with GCD because Clone$_1$ is

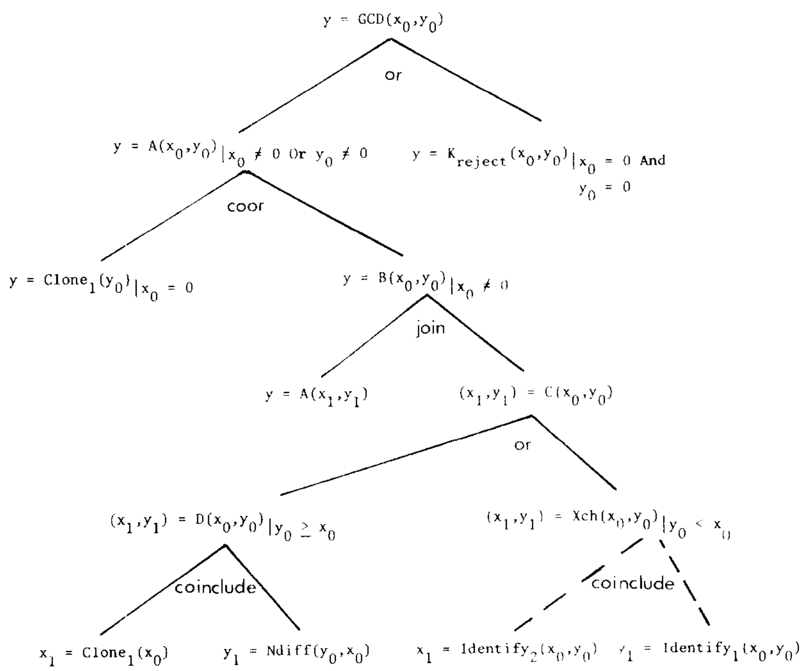


Figure 14. A control map equivalent to the AXES statements for the GCD operation.

a defined operation on any type (it provides a notion of corresponding the same value; see Appendix 2). *B* is related to *A* and *C* by means of an AXES "**join**" statement.

In using the composition control structure (or AXES "**join**" statement) in defining the relationship among functions *B*, *A*, and *C*, we can check the following: the input to *B* must appear as input to *C*; the output of *C* must appear as input to *A*; and *A* must produce the output for *B*.

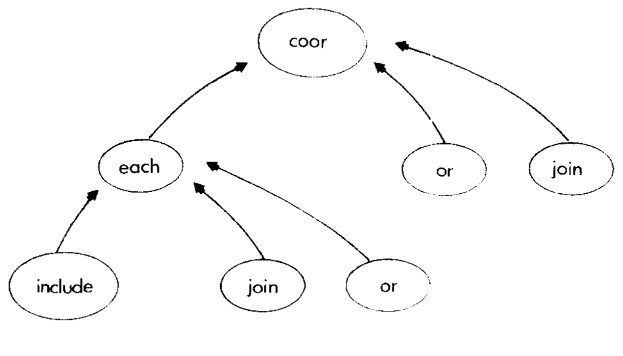
In the operation definition of GCD, note that recursive functions are formed by combining control structures (see "*A*" in Figure 14). In this case, the total hierarchy is formed dynamically, where each occurrence of "*A*" requires a different input value. Although we statically check to assure that there is *some* input value that will produce an output, proof that the chosen algorithm will find that input cannot always be checked. A good discussion of this problem can be found in [25]. If an operation has a corresponding derived operation definition, we can use this information to help prove the possibility of termination.

We continue to decompose each function at each level until we reach the point at which a previously defined operation or structure appears. In the GCD case, we check Clone$_1$, Ndiff, K_{Reject}, and Xch. Ndiff has already been mentioned as an operation on naturals. Xch is an operation that exchanges the ordering of an input. Although previously defined [26] in terms of operations Identify$_1$ and Identify$_2$ (whose definitions appear in Appendix 2), we show the Xch definition in Figure 14 (with dotted lines for information purposes only). A previously defined operation need not be decomposed each time it is used.

The technique of defining structures in AXES was introduced to limit the complexity of interface definitions among systems. Interface correctness can be checked statically by comparing the use of a structure to its definition.

We can extract certain computational properties from the GCD operation definition and use these properties to implement our specification in a programming language. A representative implementation is shown in Figure 16 graphically in terms of an HOS structured design diagram [27], which is now automated as a Universal Flowcharter [28,29]. We shall make the same assumptions as Manna and Waldinger

Figure 15. Tracing definitions of structure to three primitive control structures.



do; that is, the programming language used has integer types, but not naturals.

Although the restriction to naturals is asserted in [25], we explicitly include area 3 of Figure 16 to avoid misuse (an often occurring event during development of large programs). We check our input to GCD (Figure 14) with our input to Program B by area 1 and our output by area 4. If x_0 and y_0 both have the value of "0," then there is no greatest common divisor. In this case, we have implemented the specified K_{Reject} function of Figure 14 as an error message (area 3). Again, leaving area 6 as an assertion in the form only of a comment could cause an interface problem.

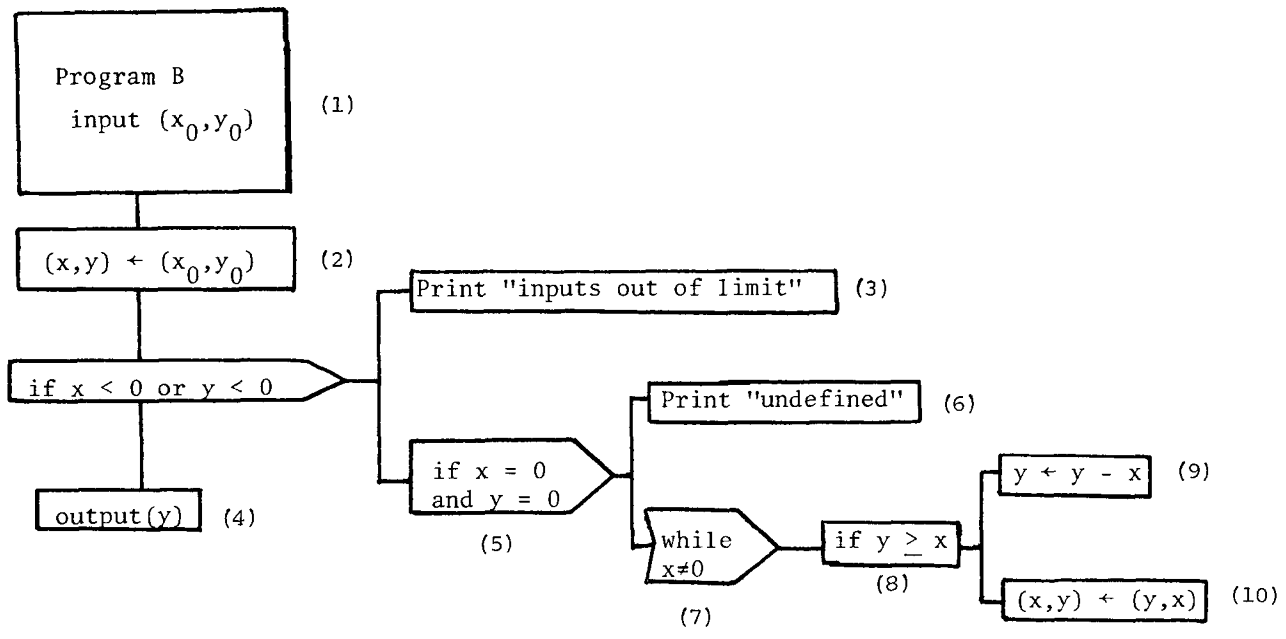
Note that each time the recursive function A of Figure 14 is to be invoked, the specification indicates that the initial values are no longer needed once the next invocation of "A" is to be executed. We make use of this fact in Program B by allocating the temporary variables "x" and "y" for each new value. Areas 7–10 of Program B implement function A from our specification of Figure 14.

There are basic assumptions implied in this implementation that may not be correct assumptions for all applications: (1) The expression "$(x, y) \leftarrow (y, x)$" implements the Xch operation. An example of misinterpretation of this expression would be a compiler which would first store the initial value of "y" and "x" and then take the *new* value of "x" and store that in "y." (2) Single statement restart capability is either not required or, if required, is an inherent compiler capability. This type of ultrareliability is often required, for example, in aerospace real-time applications. Suppose the expression "$(x, y) \leftarrow (y, x)$" were executed and a restart occurred before the program counter advanced to the next statement. Without restart protection, area 10 would be executed over again with the *new* value of "y" and "x" and could, under some conditions, give the incorrect results. These two assumptions would have to be validated as "interface" correct for our particular application. If another implementation is desired for special applications, we start with the *same* specification (of Figure 14) and use the computational *properties* of that specification to derive a new implementation.

We have found that the same design techniques that are used to design a layer, where that design process supports the verification for errors within that layer, can serve the dual purpose of supporting the design and verification processes between development layers (e.g., between specification and the implementation of that specification).

It is in such a process, that of going from one layer to another, that we are made more aware of the significance of the separation of the "what" from the "how." For not only is it the case that the conventional specification process today is more complex than it need be because it confuses the specification with implementation considerations, but it is also the case that the conventional implementation process is more complex than it need be because its specifica-

Figure 16. Graphic description of GCD implementation.



tion is confused with implementation considerations (and, more often than not, considerations that are unrealistic, incompatible with a particular implementation environment, unfeasible, or technologically out of date) or because some specification information is completely missing.

Since an AXES control hierarchy includes all of the information about the objects and the relationships of those objects in a given system, if we wanted to implement a specification in terms of, for example, a software program (such as the GCD one), we could make use of such a specification on a one-to-one basis with any of its possible implementations.

For any implementation, any function on a control map could be implemented as a procedure, a process, or as a set of in-line statements within a procedure or process. One implementation of a given specification could be multiprogrammed, another multiprocessed, and still another sequential. Values, variables, and data types can be directly translated into programming language representations of these objects.

The definition of operations on data types provides not only the set of operations that are allowable in an implementation, but also serves as a basis for checking correctness of intent. If, however, an operation is implemented as a subroutine, decisions affecting data transfer, such as "CALL by name" or "CALL by value," could vary from implementation to implementation.

The layer and level relationships with respect to communication and resource allocation can be used in the assignment of input and output access rights, data flow, functions that are to be invoked, error detection and recovery procedures, and order of execution of implemented modules.

The data flow can be traced directly on the control map in terms of access rights assignments (i.e., input can be traced down and output can be traced up the control map), which suggests, of course, that the access rights themselves can be readily determined for any given implementation. For example, with respect to scope, a variable only needs to be declared at the level where it first appears. That same variable "local" to the level of the controller above it can also be implemented as such.

It is not possible with the use of conventional computers always to maintain a single-assignment, single-reference status when going from a specification to an implementation, but it is possible to resource-allocate more efficiently an implementation when its specification is defined with single-assignment, single-reference properties. This is true since the status of any "location" is *always* known. Thus, a reuse or a sharing of a particular location can not only be determined, but a location can always be shared when it is safe to do so.

Since every node on the control map explicitly states all input and output variables, it is possible for an implementation to be set up to implement alternative plans in the case of a failure.

Priorities can be determined readily for a particular implementation since there are some very specific rules to be followed. (For example, a controller always has a higher priority than the functions it controls.) Thus, a master sequencer-type of executive, in an implementation, would always be forced to maintain a higher priority than the functions (or processes) it invoked. Other types of ordering considerations and their alternatives, such as timing, are also readily apparent. It is clear, for example, that in the implementation of a primitive composition control structure some data from one function must be computed before the other function is initiated; whereas in a primitive set partition structure, only one of the functions need be processed for a given performance pass. Similarly, a primitive class partition would allow for more than one function to be performed at a given time should it be desirable to do so. These facts are directly translatable to the various ordering options that are available in a specification for the processing of those functions in a given implementation.

A REAL-WORLD EXAMPLE

As an illustration of how AXES can be used to represent functionally a system so as to lead the way towards a reliable and efficient implementation, we include here specifications for a satellite navigation system called **navpak**. This system is intended to update navigational parameters of Earth-referenced satellites with imaging data transmitted to the ground from the satellite. The ultimate aim is to be able to determine the orbit and attitude of the satellite precisely enough so that the imaging data can be used to answer user queries, such as "To what landmark am I pointing?" or "Where is Florida?"

This example is intended to provide a specification of the interaction, or relationship, among system components for the case in which the orbit and attitude of the satellite is not precisely known. In this case, the imaging data is used to determine orbit and attitude state estimations from landmark observations. 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 [30].

The process of determining orbit and attitude can be done with varying degrees of automation. The least automated approach is one in which the landmark observation is obtained manually by displaying the imaging data (retrieved from data available on files) directly on graphics devices. In this case, the correlation function (i.e., correlating the geographic coordinates and the coordinates of the displayed image) is intended to be performed by a human user. The function relating the observation to the state of the satellite is to be performed by a computer. Here, the computer processing includes the computation of the landmark time from the coordinates of the displayed image, 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 a given scene, or set of images), computing the uncertainty of the observation, and, upon user request, an orbit–attitude–covariance matrix update based on a classical "weighted least squares" statistical estimation algorithm [31].

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 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, manual interaction with the processing system is essential so that a person can make the ultimate decision as to whether a particular observation should be incorporated or not. The system described here is designed for automatic processing with the capability for manual override at crucial processing decisions.

The system structure (or set of functional relationships) is as follows.

```
Structure: $y$ = Navpak($x, s, l, c$);
   where $x, y$ are States (of Satellites),
         $s, s'$ are Ordered Sets (of Images),
         $l, l'$ are Ordered Sets (of Places),
         $c, c'$ are Ordered Sets (of Ordered sets (of
            Images)),
         $\theta$ is an Option,
         $x', x''$ are States (of Satellites),
         $l_1$ is a Place,
         $l_2$ is an Ordered Set (of Places);
  Navpak: $y = f_1(x, s, l, c, \theta)$   cojoin
                                 $\theta$ = Choose($x, s, l$);
  $f_1$: $y = f_2(x, s, l, c)\big|_{\theta=\text{Enter}}$
                                 or
                                 $y = B(x, s, l, c)\big|_{\theta=\text{Proceed}}$
                                 coor   $y = x\big|_{\theta=\text{Terminate}}$   ;
  $f_2$: $y$ = Navpak($x', s', l', c$)   cojoin
                                 $(x', s', l')$ = Override($x, s, l$);
  $B$: $y = f_3(x, s, c, l_1, l_2)$         cojoin   $l_1$ = First(l)
                                 coinclude
                                          $l_2$ = Second(l);
  $f_3$: $y$ = Navpak($x'', s, l_2, c$)   cojoin
                 $x''$ = extract/filter$_{\text{Intervene,QA}}(x, s, l_1, c)$
                                 failure          $x'' = x$;
syntax: Choose to Override $x, s, l, c$ and aid automatic
        correlation by Intervene or qualify with QA to
        obtain $y$.
end Navpak;
```

Figure 17 shows the hierarchy of functions for **navpak**, a projection from the AXES definition.

We begin with an initial estimate of the state of the satellite x; a preselected "scene" or set of images of a portion of Earth s; a set of predetermined landmarks or Earth-based locations l; and a set of image sets that have been previously identified as images of particular Earth landmarks c. The intent is to produce a new state estimate y.

navpak is related to its offspring, f_1 and Choose, by a **cojoin** structure. Choose examines x, s, and l and, based on these values, will produce a value of type Option, which consequently gets used as input to f_1, which in turn produces y. Function f_1 is related to its offspring f_2, B, and Clone$_1$ by **or** and **coor** structures. In AXES, a function can be replaced by its next most immediate level of decomposition by simply inserting the level description appropriately in an AXES statement, as in the decomposition for f_1. In the case of f_1, the determination as to which function is to be performed is dependent on the properties "θ = Enter," "θ = Proceed," and "θ = Terminate." Related to its offspring, **navpak** and Override, by a **cojoin** structure, f_2 provides the opportunity to select new data with Override and then to go through the same procedure recursively until the data are acceptable to "Proceed" to B or "Terminate" accepting the initial state value as the best state estimate. To Proceed at B entails using the first landmark to update the state by **extract/filter** (if **extract/filter** fails, the initial data are salvaged for the next try) and then the remaining set of landmarks, along with the new state estimate x'', is resubmitted to the next recursive instance of **navpak**. The **failure** and **extract/filter** structures, as well as some operation definitions described in this section, can be found in Appendix 2.

Each leaf of **navpak** is either a previously defined AXES operation [in this example, Clone$_1$ is an operation for any type, First and Second are primitive operations on type Ordered Set (of T)], a recursive in-

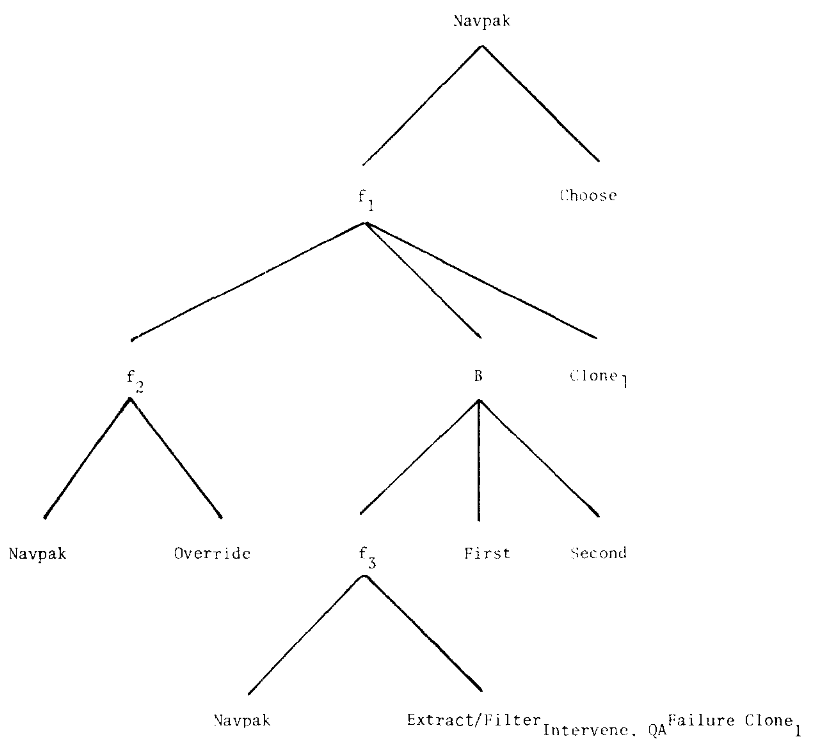


Figure 17. navpak.

vocation (**navpak** itself is recursive), or an unspecified function referred to in the user defined syntax (in this example, Intervene, QA, Choose, and Override are unspecified functions and are referred to in the syntax statement appearing at the end of the AXES definition for **navpak**).

Each variable is identified with a previously defined data type [in this example, we refer each variable to State (of T), Ordered Set (of T), Satellite, Image, Place, or Option]. 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 implementation. Data types used for **navpak** are discussed in Appendix 1.

Once designed and verified, a structure is *used* for an operation definition by identifying particular operations for the unspecified functions and particular variables for those variables mentioned in the user defined syntax. For example, particular operations for Choose, Override, Intervene, and QA and particular variables for x, s, l, and c would be identified when *using* **navpak** for an operation definition.

Particular operations can then be allocated, either manually or automatically, to particular resources. For example, particular Choose and Override operations would most likely be assigned to human operators in a **navpak** implementation, whereas First and Second would most likely be allocated for computer processing.

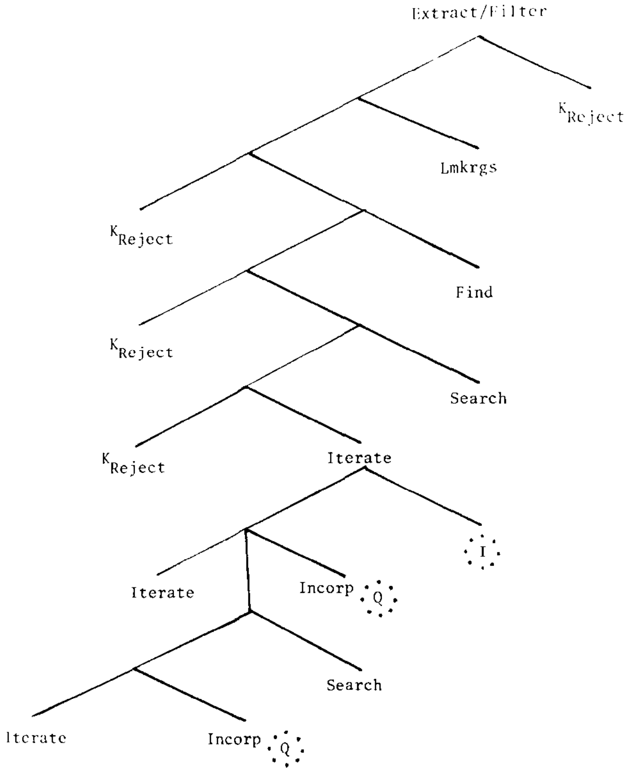
We could also use **navpak** to define another structure. In such a case, for example, K_{Constant} operations could be "plugged in" for Choose and Override indicating that the "use" of the use of **navpak** would be the assignment of names of objects. This use of **navpak** would ensure that the ultimate decisions would be accomplished by manual interaction with the processing system.

extract/filter is itself defined as an AXES structure in this example. **extract/filter** determines whether the landmark measurement is suitable to be used to update the estimated state of the satellite. In Figure 18 a projection of the specification for **extract/filter** is shown. Each operation that appears at a leaf node is a specified AXES operation except for the two operations circled by dots, I and Q. These two operations are the unspecified functions of **extract/filter**. The syntax selected here for this structure is not as English in character as it is functional, as compared to the suggested syntax for **navpak**. Different syntactic forms, including those which are graphical, may be chosen for structures, depending on user preference. The **navpak** structure *uses* **extract/filter** by substituting "Intervene" for "I" and "QA" for "Q." (Other structures, such as **or, cojoin,** and **failure** are also used to define **navpak.**) In this case, we are *using* structures to define yet another structure. In a similar way, the **extract/filter** definition uses, for example, the **incorp** structure (see Appendix 2).

The landmark extraction function Lmkrgs (see Figure 18) integrates the vehicle state and covariance matrix to the time of the landmark and defines the uncertainty in the measurement. The landmark extraction function may reject the measurement automatically if it cannot find the landmark in the chosen scene. Lmkrgs is a rather lengthy operation, discussed in terms of the data types vectors, matrices, scalars, time, and angles in [32]. Of interest is the fact that many submodules and groupings of submodules were able to be used over and over again, both within the definition of Lmkrgs itself as well as for various other operations within **navpak**, specifically, operations Final and Update for the **incorp** structure (see Appendix 2).

If the landmark extraction is not successful, the measurement is rejected. If the landmark extraction is successful, the measurement is automatically correlated with a preprocessed image of the landmark (see operation Find in Figure 18 and the expansion of Find in terms of operations on Ordered Sets and Images, Appendix 2) in an attempt to obtain a better measurement time within the uncertainty "window." (See operation Search in Figure 18, in which a region is superimposed on the scene and the chip is matched with the images in the window of the scene.) The correlation is functionally related to the intensity of the image and the intensity of the chip for each particular location in the region being searched. The specification for Search appears in Appendix 2, along with structure definitions used to specify Search. If a negative correlation is found, the measurement is rejected automatically. If a positive correlation is determined (see the operation Iterate, Figure 18), the user has the option to incorporate the measurement immediately by specifying a particular *I* function; if this option is not exercised, automatic processing continues until the region is considered completely searched. If at any step of this process the landmark is rejected, the error filters back up the **extract/filter** structure. Subsequently, if **extract/filter** fails, error recovery is achieved, as seen in the higher-level definition for the **navpak** structure.

Figure 18. extract/filter$_{I,Q}$.



At each "better" correlation, the user may decide to incorporate the measurement (see the operation Incorp, Figure 18). When the "best" correlation is determined, the measured landmark is compared to a computed landmark. The computed landmark is used to construct a new region and the measurement is tested to see whether it can be found within the new region. If the measurement is not within the new region, the measurement is rejected. If the measurement is computed to be successful, the user may decide to reject the measurement if not satisfied with the results. This is accomplished by specifying a particular *Q* function. If this option is not exercised, 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.

EXPERIENCES WITH THE APPLICATION OF HOS

HOS has now been employed by our own staff in several different types of application. They include those that were familiar to our engineers as well as some that were not familiar at the beginning of a project. There was direct involvement in some applications whereas in others, involvement only on parallel efforts. Both original designs and redesigns have been prepared. Likewise, in-line verification has been performed on some projects, independent verification on others. In all of these experiences, a conscious attempt has been made to analyze ourselves and others in order to enhance either our own techniques or theirs.

Throughout this process, certain trends, patterns, or common experiences have taken place. Phenomena have been observed, both with respect to the design and verification processes and to the other processes, all of which are directly related to design and verification.

Some Experiences on Specific Projects

One of the first projects was a respecification of the Apollo Guidance Computer (AGC) operating system (OS), an application familiar to us [26]. Unfortunately, we had a great deal of difficulty reconstructing the pieces. This was due mostly to the fact that the AGC OS was poorly documented. Our only solution for completely understanding the system (which included our own results of various design processes, including our own coding and our own verification) was to go back and pour over the original code, which was very clever and difficult to understand. When we began this effort, we thought there was little in the AGC OS upon which we could improve. This attitude was partly a result of the fact that no errors were found for several years within the OS itself. However, when we attempted to respecify the OS, we discovered that many of the development errors that occurred in the application programs using the OS would not have occurred if the AGC OS had certain other inherent properties; for although the AGC OS had properties of hidden data, it did not have properties of hidden timing. From this effort, we therefore determined that the AXES methods were very helpful in demonstrating more reliable design goals with respect to interfaces between application programs and the systems software that executes these programs.

With respect to another project, Position Locator Reporting System (PLRS), our charter was to select the most complex module, specify that module with AXES, and demonstrate the advantages of applying an effective methodology. We did just that. This was the first effort in which we attempted to use AXES in an ongoing project. Not only was our aim to demonstrate its effectiveness, but also to perform this task without impacting schedules or deliverables. In this process, however, we determined that the use of an effective methodology can benefit not only a new project, but also an ongoing project that already employs a different methodology [33].

When our engineers began this effort, the ongoing project engineers were just completing the design of their specifications and were about to embark into a design phase that would result in the implementation of computer code. As a result of our respecification to one module in their system, it was possible to have an understanding of the system and the methods used to develop that system. Recommendations for specific ways of enhancing both their system and the methods to develop that system were made, although this particular system was being developed with methods that were beyond the sophistication of most conventional systems today. In the process of defining standards for the chosen module (i.e., common structures, functions, and data types), it was determined that many of these standards were not only applicable to other modules in the system but to a family of systems within which this one resided (i.e., other communication network systems). During the same respecification process, 16 categories of questionable areas, such as unanswered questions, inconsistencies, incompleteness, and redundancies were determined. This was not only a demonstration of the advantages of using an effective methodology, but this information could be directly applied for the next phase of development. It is our own opinion, however, that many of these problem areas would have been uncovered prior to our involvement had an attempt been made during the specification phase to integrate the top levels of the specification from the beginning. (This same phenomenon was observed in the Navpak project as well, and Navpak had been around a lot longer than PLRS. In fact, a "working" implementation for it already existed. In this case, the integration of specifications was often missing since the problem was too "familiar" to the Navpak engineers.)

One of the more interesting sets of observations made was that involving a project for which a software system was conceptualized and then developed to completion by our staff. This involved the design of the Universal Flowcharter in AXES, which was implemented in PASCAL [28,29]. The programmers, who implemented the flowcharter, determined the design of the code by using AXES specifications as a guide. There were several different engineers on the project. Some of them were involved throughout the project; others only came in during the programming stage. Although our charter was to build a universal flowcharter, we were asked to apply AXES whenever possible. We had the unenviable position of attempting to design something that had never been done before, provide a design in light of continuously changing requirements (this was as a result of both designing a new concept and designing that concept for universal use), deliver and implement that design in terms of well-defined deliverables, and use a methodology (which was our own) throughout all phases of development (when this had never been done before). We were also observing ourselves continuously to see how effectively we were dealing with all of these considerations.

As delivery dates got closer, some designers panicked and decided to start implementing before all of the data types were rigorously defined and therefore before the control maps were completely defined. Others forced themselves to complete the control maps for a particular specification unit before implementation began. Of that set of modules that was completed, some had to be changed after implementation began (e.g., some data types were too specific and were better suited for another machine environment; others needed to be defined in more detail). We did find, however, that any errors that occurred in implementation were in those areas where the specification was *not complete* before implementation. That is, if all changes were negotiated and specified, chances of an error in implementation were almost nonexistent. The other modules (i.e., those that were not completely defined) were not only error prone, but took much longer to debug than those modules whose specifications were completed at least once before implementation.

The Navpak project had the most implementation details embedded in its specifications. A possible reason for this fact was that the Navpak system was already implemented in at least one form, and it is often the case that engineers update specifications further with implementation considerations when more is thought to be known about the implementation. One of the potential problems they would have, therefore, would occur when they wanted to make a change to their existing system; for each time there would be a change, it could be necessary to redesign, or at least retest, the whole system. This could be the case, for example, if a new user option were to be incorporated. This situation is typical of conventional methods and is a good example of how a design problem can affect the verification process in more than one iteration of a particular phase of development. It is for this reason that we chose to discuss a portion of this particular system in more detail.

Although each of these projects has had its own interesting aspects, it has also been quite interesting to observe the commonalities that occur among projects. The process of applying a methodology to each project has certain common elements, and the results of that process also have certain common elements. For example, the common process of defining AXES modules within each given project produces the common result of identification of commonality between modules in that project. As a result, new structures, functions, and data types are defined and can be added to the general AXES library, as well as to the project specific AXES library. Errors, in particular interface errors, are always found within existing systems, whether they exist as requirements or as completed code. In these projects, a comparison of the old and new versions of a given module is always made. One cumulative result of all these efforts is the list of properties that are recommended for a methodology (discussed in an earlier section) as well as sets of project specific recommendations based upon that list. For every ongoing project, a minimum set of recommendations is always made, if it is not too late to make some incremental changes. For every project just starting up, a more complete set of recommendations is made. An example of one set of recommendations is shown in Table 1.

Certain advantages, as a result of using a more formalized approach, can be directly related to making life easier for the designers and verifiers on a project, as well as for the managers, implementers, and documenters. Some of these will be discussed below.

Acceleration of the Learning Process

The engineers who performed work on these projects needed to go through a learning process of some sort. This varied from learning a new application, to learning about someone else's module on a familiar application, to relearning one's own module after some time had elapsed. On these projects that had applications with which we were most unfamiliar, such as PLRS, we were able to take advantage of such a shortcoming in order to test our methods as a learning technique. Our method of understanding, in this case, was first to attempt to construct a control map; by doing so, we were able to determine existing functions and their relationships. This process not only provided us with an accelerated means of asking the questions that should be asked to construct the definition of a module, but it also became clear that this was a technique for prompting questions that otherwise might never have been asked; for during this process we found that there were areas in the documentation that were either not clear enough, missing, inconsistent, redundant, or not integrated with other areas.

The fact that we were able to use the control map technique as an accelerated learning process for ourselves suggested to us that this same technique could

Table 1. Recommendations of Standards

Definition of design goals: For example, definition of interfaces should be made in the specification phase: i.e., integrate from the beginning.

Rules for design and verification: Specifications should be defined hierarchically, and rules (e.g., those that accompany the control map) should be followed with respect to how one level in the hierarchy relates to the function directly above it. These rules should include ways of defining the invocation of a set of functions, input and output flow, input and output access rights, error detection and recovery, and ordering.

Interface specification document: For every system a standard dictionary (or library) should exist that provides common meanings, ways of saying things, ways of doing things, mechanisms for defining a system, system modules, and support tools and techniques. An evolving dictionary is recommended that includes a set of
definitions of terms
formally defined data types
formally defined control structures
system functions

User manual: A user manual should be provided that contains checklists and explains (1) how users interpret the standards in the interface specification document; (2) how designers design modules to add to the "library" of the interface specification document; and (3) how managers define new standards for system development that in turn can be converted, by the designers, to modules for incorporation into the interface specification document.

User guide to implementation: If specifications contain certain consistent properties, one can take advantage of these properties by understanding their consequences with respect to implementation. Given that there are standards for specifying, it would expedite the implementation process if standards for specifying were defined to go from a specification to an implementation. The user guide should include standards for (1) going from the specification (e.g., a control map) to a computer allocation; (2) reallocating functions to a computer, and (3) providing for reconfiguration of functions in real time.

Definition of development model: The definition of a development model is most helpful to the manager, who is responsible for integrating all the phases of development. In addition to the above recommendations, the development model should define phases of development and how to integrate them; disciplines (such as management, design, verification, implementation, and documentation); and an integrated application of tools and techniques that are to be used, and how and when they are to be used throughout the development process.

be used as a learning tool, for example, for those people new to a project; a manager learning about the work of the people in his project; designers and verifiers learning about each other's modules in the same project; implementers learning about the specification from which they are building; and users, such as maintenance people, learning about the system they are using or changing.

Acceleration of the Specification Process

In the process of constructing various specifications, we found that the control map technique was quite effective in expediting what are often considered to be design processes. In those projects for which we were given the task of defining an alternative module to an existing specification, the existing specification was, for all practical purposes, thought to be complete. But it was necessary for us to design more explicitly function definitions, including data definitions, as well as the integration of these functions. We were able to determine, for example, types of design trade-offs; design decisions with respect to interface correctness (i.e., verification before the fact); common use of specification modules (data types, operations, and structures); more powerful and simpler ways of conveying specifications; when each specification module was complete; how to integrate modules safely; common rules (or management standards) of communication between modules; methods of defining a system so that changes could be made safely; and the effects of those changes traceable within the design and during the design process.

Our findings were that these methods not only supported a designer in providing designs more quickly, but also helped to point out things that might otherwise have been completely forgotten.

Verification and Validation Aid

Within our various efforts for which there was an existing module with which to start, several errors were discovered by the two-step process of formal definition of (1) the data types that were used and (2) the structure (or organization) of the existing module. Because problematic areas were detected early, later development phases were able to benefit: those problems that had not been forestalled were not only able to be detected sooner, but were also prevented from surfacing later or propagating into worse problems.

Establishment of Design Goals

In the process of understanding a module on an existing project, especially a large or complex one, it would always have been helpful if the specification had been concerned more with the definition of the relationships of specified functions (particularly at the top level). The control map technique forced us to consider integration of the functions in the system from the very beginning. Such a design philosophy, if applied, not only aids in understanding a design but eliminates integration problems that would subsequently show up in later development stages. Thus, if a specification were integrated, its implementation would be able to be an evolvement rather than a "redo," as is usually the case, especially in the development of a conventional system.

Enhancement of Existing Techniques

We found that it was possible to indicate certain problem areas or demonstrate ways of making certain improvements to an ongoing project and do so without impacting schedules or milestones if necessary (it always was in our case). Those types of improvement included enhanced methods of error location, the actual discovery of errors, and off-line methods for providing the engineer greater (or more quickly obtained) visibility. (An automated graphics tool would be an example of an add-on feature that would not necessarily have to halt progress during a system development.)

Management Visibility

In those projects in which we were asked to look at a part of a system, we were able to determine a "feel" for the state or health of the specifications of the system in general. For example, a better idea could be formed of the types of interface problem that needed to be resolved before a specification could be successfully implemented. Those steps were determined that would be necessary before a specification could be called complete, and certain recommendations were determined that were thought to be helpful in providing a more reliable specification more efficiently in the future.

The Need for Constructive Standardization

Put simply, the most urgent need on any large-system development process is that of *constructive* standardization. Some standardization, if it is effective, is certainly better than none at all; but if a project is already in development, it is not usually possible to apply an ideal and complete set of standards. However, it is possible to use incrementally those standards that would enhance the development process either by finding errors or by accelerating remaining phases of development. We did this on one very large software effort with uncompromising schedules. For example, we discovered that many interface errors took place in the implementation phase when programmers would use instructions in an unstructured language, such as "GOTO + 3." Errors would creep in when someone would come along, often the same programmer, and inadvertently insert a card between the GOTO instruction and the location at which it should have gone. Once we discovered the amount of errors that resulted, we enforced by standardization the use of instructions such as "GOTO A" rather than "GOTO + 3." As a result, such errors never happened again. The same sort of introduction of standards could take place in any project. We have found that it is too easy to want to hurry the design process in order to meet deliverables. As a result, we too often hesitate to introduce additional standards into a system development process. But hindsight and recent experience, both of our own and of others, have demonstrated that in the end it pays to organize first and build later, especially when involved in the development of large and complex systems.

SUMMARY

In order to change to new and standard techniques, there is always the initial investment that is necessary for defining and developing a model, or subsets thereof for systems in general. We believe that a step in this direction has already been accomplished.

Given AXES and the AXES library as a first step, a second step is to define a set of additional structures, operations, and data types that are necessary for defining a particular family of systems. Once the initial investment has been made to establish what in essence is a way of organizing the development of a system with standards and mechanisms to accomplish that organization, the payoffs should be quite apparent. Design time during the requirements/specifications phase should be no greater than (in fact, we suspect, much less than) with conventional techniques. Implementation designs should take considerably less time than with conventional practices since it is possible to perform such a process on an almost one-for-one basis. We suspect that the largest savings will be realized within the verification processes since most of the recommended techniques provide standards that should eliminate errors before the fact, and it is just these very types of error for which one spends so much time looking today.

APPENDIX 1. Some Data Type Definitions

The following universal primitive operations are defined for any type T and can be assumed to apply to each new type definition:

Boolean = Equals (t_1, t_2);
 (t_1, t_2) = Clone$_2$ (t);
t_3 = Identify$_1$ (t_1, t_2);
t_1 = Identify$_2$ (t_1, t_2);

The axioms that characterize these operations and therefore apply to any type are

where t_1, t_2, t_3, t **are** Ts;
Equals (t, t) = True; (1)
Equals (t_1, t_2) = Equals (t_2, t_1) (2)
Entails ((Equals (t_1, t_2) And Equals (t_2, t_3)),
 Equals (t_1, t_3)) = True; (3)
Equals (Identify$_1$ (t_1, t_2), t_1) = True; (4)
Equals (Identify$_2$ (t_1, t_2), t_2) = True; (5)
Identify$_1$ (Clone$_2$ (t)) = t; (6)
Identify$_2$ (Clone$_2$ (t)) = t; (7)

The first three axioms characterize "equality" as an equivalence relation in terms of type Boolean, which was characterized by Cushing [19 (Appendix 4)]. The fourth property of equality, replaceability, is already fixed simultaneously with the introduction of a type T (e.g., this allows us to use the "=" in each axiom definition), assuming that equality can be defined for a particular type by defining a particular equivalence relation [this must, of course, satisfy axioms (1–3) on any type] on an already known type (one that presupposes equality).

Axioms (4) and (5) characterize the ability to choose, or identify, a particular object. Axioms (6) and (7) characterize the Clone$_2$ operation, which provides for the ability to rename the same object.

We often make use of a special case of the Identify$_1$ operation, which we call the K_{constant} operation. When the first argument of Identify$_1$ is a constant, Identify$_1$ can be viewed as an operation on one argument of type T.

$K_{\text{constant}}(t)$ = Identify$_1$(constant, t)

An alternative way of writing any K_{constant} operation in AXES is simply to use the constant itself. For example,

"$y = K_1(t)$" is equivalent to "$y = 1$."

This alternative form appears often in the AXES definitions throughout this paper.

Type Ordered Set (of T) makes possible the selection of values from a set of objects in a particular order. The property we want to characterize here is simply the ability to distinguish which is first from "all the rest." Ordered sets can be implemented as files, lists, or arrays, for example

Data Type: Ordered Set (of T);
primitive operations:
 t = First(ordered set$_1$);
 ordered set$_2$ = Second(ordered set$_1$);
 Boolean = OEquals(ordered set$_1$, ordered set$_2$);
axioms:
  where t **is a** T,
   (a, b) **are** Ordered Sets (of T),
   Nullo **is a** constant Ordered Set (of T);
 First(Nullo) = Reject;
 Second(Nullo) = Reject;
 OEquals(a, b) = Equals(First(a), First(b))
   And OEquals(Second(a), Second(b));
end Ordered Set (of T);

The first two axioms define the error conditions for an Ordered Set (of T), and the third axiom provides a concept of equality for Ordered Sets (of T). Ordered Set (of T) is a parameterized type in that, in its use, "T" can be replaced with the name of a particular type. In the Navpak specification, for example, we used Ordered Set (of State (of T)) as a particular use of this type.

The algebra associated with State (of T), itself a parameterized type, is a heterogeneous algebra in terms of types Time and Boolean. Time was characterized in [26].

Having a specification for Time and Boolean, we can now define State (of T) as follows:

Data Type: State (of T);
primitive operations:
 time = Stime(state);
 t = Correspondent(state);
 state$_2$ = Ssucc(state$_1$);
 Boolean = Sequals(state$_1$, state$_2$);
axioms:
  where (s_1, s_2) **are** States (of T),
   time **is a** Time,
   t **is a** T;
 Precedes?(Stime(s_1), Stime(Ssucc(s_1)) = True;
 Equals(Correspondent(s_1), Correspondent(s_2))
  = False $\subset$ Stime(s_1) $\neq$ Stime(s_2) = True;
 Sequals(s_1, s_2) = Equals(Stime(s_1), Stime(s_2))
  And Equals(Correspondent(s_1), Correspondent(s_2));
end State (of T);

The first of these axioms characterizes the time dependence of each State (of T), in terms of the previously defined AXES operation, Precedes?. Precedes? is an operation on two values of type time that produces a Boolean. It provides the notion of being able to determine if one time precedes another. The second axiom imposes a functional relationship between time dependence and the particular t of a State (of T) in that two different states cannot be associated with the same time. The third axiom characterizes equality of a State (of T) in terms of its components. In the Navpak example we used State (of Satellite) as a particular State (of T).

Satellite itself, then, must be defined as a type. The type definition given for Satellite is more analogous to a data structure definition than a behavioral definition in that it only says that two Satellites are equal if their components are equal and that there are four components of a Satellite that will characterize the type. To make this type more useful, the primitive operations specified (and perhaps a few additional ones that would have to be defined) would have to be related by means of the particular approximation to the equations of motion to be used for Navpak.

Data Type: Satellite;
primitive operations:
 vector = Position(satellite);
 vector = Velocity(satellite);

vector = Attitude(satellite);
matrix = Covariance(satellite);
Boolean = Stequals(satellite$_1$, satellite$_2$);
axioms:
where (s_1, s_2) **are** Satellites;
Stequals(s_1, s_2) = Equals(Positions(s_1), Position(s_2))
And Equals(Velocity(s_1), Velocity(s_2))
And Equals(Attitude(s_1), Attitude(s_2))
And Equals(Covariance(s_1),
Covariance(s_2));
end Satellite;

Types Vector and Matrix are discussed in [32]. The same sort of data structure definition is supplied for type Image since the only characteristics we were able to abstract from the information we had on hand at the time of this project was that an image was an object that had a particular intensity and associated location.

Data Type: Image;
primitive operations:
scalar = Intensity(image);
place = Location(image);
Boolean = Iequals(image$_1$, image$_2$);
axioms:
where i_1, i_2 **are** images;
IEquals(i_1, i_2) = Equals(Intensity(i_1), Intensity(i_2))
And Equals(Location(i_1), Location(i_2));
end Image;

Type Scalar is defined in [32]. Type Place was defined as part of a project now in progress for Defense Civil Preparedness Agency (DCPA) [35], where it was necessary to define a geographic coordinate system in order to distribute food, fuel, and other resources to various regions within the United States.

APPENDIX 2. Some Structure and Operation Definitions

Specifications for the specific Navpak structures **extract/filter, incorp**, and the operations Find and Search appear in this appendix. More general AXES structure definitions, which were used to define these specifications, are also included.

The primitive control structures form the basis for defining other control structures in AXES. The use of AXES syntax and associated rules for the primitive control structures follow:

For composition, if $y = f_0(x)$,
f_0: $y = f_2(g)$ **join** $g = f_1(x)$;

(See Figure A1.)

1. One and only one offspring (specifically, f_1 in this example) receives access rights to the input data x from f_0.
2. One and only one offspring (specifically, f_2 in this example) has access rights to deliver the output data y for f_0.
3. All other input and output data that will be produced by offspring, controlled by f_0, will reside in *local* variables (specifically, "g" in this example). Local variable "g" provides communication between the offspring f_2 and f_1.

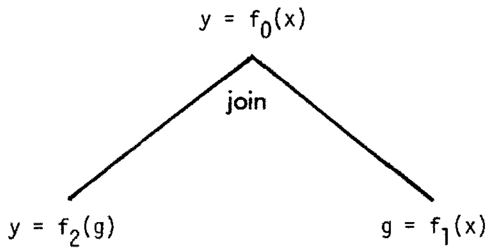


Figure A1. Composition.

4. Every offspring is specified to be invoked once and only once in each process of performing its parent's corresponding function.
5. Every local variable must exist both as an input variable for one and only one function and as an output variable for one and only one different function on the same level.

For Class partition, if $(y_1, y_2) = f_0(x_1, x_2)$,
f_0: $y_1 = f_1(x_1)$ **include** $y_2 = f_2(x_2)$;

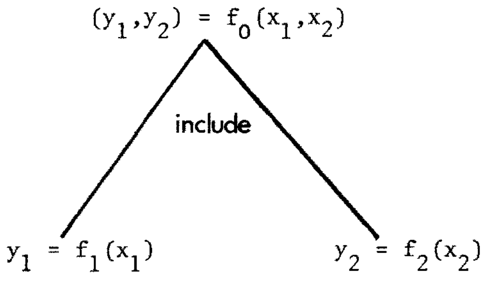
(See Figure A2.)

1. All offspring of f_0 are granted permission to receive input values taken from a partitioned variable in the set of the parent's corresponding function domain variables, such that each offspring's set of input variables collectively represents the parent's corresponding function input variables.
2. All offspring of f_0 are granted permission to produce output values for a partitioned variable in the set of the parent's corresponding function range variables, such that the sets of each offspring's output variables collectively represent the parent's corresponding function variables.
3. Each offspring is specified to be invoked per input value received for each process of performing its parent's corresponding function.
4. There is no communication between offspring.

For set partition, if $y = f_0(x)$,
f_0: $y = f_2(x)\big|_{\text{property}}$ **or** $y = f_1(x)\big|_{\text{Pnot(property)}}$;

(See Figure A3.)

Figure A2. Class partition.



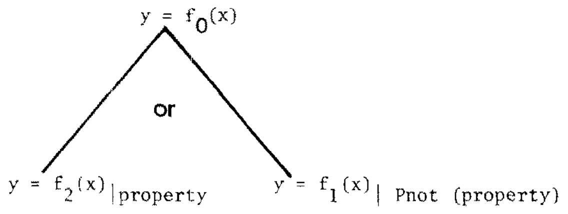


Figure A3. Set partition.

1. Every offspring of the parent at f_0 is granted permission to produce output values of "y."
2. All offspring of the parent at f_0 are granted permission to receive input values from the variable "x."
3. Only one offspring is specified to be invoked per input value received for each process of performing its parent's corresponding function.
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.
5. There is no communication between offspring.

In the above definitions x, y, y_1, y_2, x_1, x_2 are ordered sets of variables; f_0, f_1, f_2 are functions; property is of type Property (of T) [19]; and Pnot is a primitive operation on type property whose result is a property exclusive of its input argument.

One structure, the **each** structure, is intended to be able to perform the same operation on each member of an ordered set of objects. Similar structures have been useful on other projects, such as [33] and [28].

```
Structure: $y$ = Each($x$, $b$):
                where $x$, $y$ are Ordered Sets(of $T$),
                      $b$ is of some type;
  Each: $y$ = Nullo$\Big|_{\text{First}(x)=\text{Reject}}$    or  $y = f_1(x, b)\Big|_{\text{First}(x)\neq\text{Reject}}$ ;
  $f_1$: $y$ = Combine($y_1$, $y_2$)             join ($y_1$, $y_2$) = $f_2(x, b)$;
  $f_2$: ($y_1$, $y_2$) = $f_3(a_1, b_1, a_2, b_2)$  join    ($b_1$, $b_2$) = Clone$_2$($b$)
                                        include ($a_1$, $a_2$) = Clone$_2$($x$);
  $f_3$: $y_1$ = $F(a', b')$                  include $y_2$ = Each($a''$, $b''$)
                                            join    $a'$ = First($a_1$)
                                            include $a''$ = Second($a_2$)
                                            include $b'$ = Clone$_1$($b_1$)
                                            include $b''$ = Clone$_1$($b_2$);
syntax: $y$ = $F$([$x$], $b$);
end Each;
```

The **each** structure has one unspecified operation F. First and Second are primitive operations on type Ordered Set (of T), defined in Appendix 1. Clone$_2$ is a primitive operation on any type and is also defined in Appendix 1. Combine is a derived operation on type Ordered Set (of T), the specification of which follows:

```
Derived Operation: $y$ = Combine($a$, $b$):
            where $a$ is a $T$,
                  ($b$, $y$) are Ordered Sets (of $T$);
  First(Combine($a$, $b$)) = $a$;
  Second(Combine($a$, $b$)) = $b$;
end Combine;
```

The **each** structure can be used to define the **cojoin** structure, which provides the ability to select components of an input set of a function that serves as common input for dependent subfunctions; similarly, the **coinclude** structure provides the same ability for independent subfunctions, and the **coor** structure provides the same capability for a selection among subfunctions. In each of the following definitions, some type is an ordered set of variables. The notation "id$_{[b]}(x)$" is an alternative form for the notation "id([b], x)" used to indicate that the user of the structure is to supply the value for "b" as a constant, thereby specifying particular id functions as mappings associated with "x" only. An implicit specification of "a" or "b" occurs when the id function is "performed" by simply replacing "$x_{[a]}$" or "$x_{[b]}$," respectively, by a particular subset of variables of "x" in the use of this structure.

```
Structure: $y$ = Cojoin($x$);
        where $x$, $y$, $g$, $x_{[a]}$, $x_{[b]}$, $x_1$, $x_2$ are of some type,
              $a$, $b$ are Ordered Sets (of Naturals);
  Cojoin: $y = A(x_{[b]}, g)$     join     ($x_{[b]}$, $g$) = $f_1(x)$;
  $f_1$: ($x_{[b]}$, $g$) = $f_2(x_1, x_2)$  join     ($x_1$, $x_2$) = Clone$_2$($x$);
  $f_2$: $x_{[b]}$ = id$_{[b]}$($x_1$)         include $g = B(x_{[a]})$
                                      join $x_{[a]}$ = id$_{[a]}$($x_2$);
syntax: $y = A(x_{[b]}, g)$       cojoin   $g = B(x_{[a]})$;
end Cojoin;
```

The specification of id, which is a derived operation on Ordered Set (of T) and Naturals, follows:

```
Derived Operation: $t$ = id$_n(\theta)$;
              where $n$ is a Natural,
                    $\theta$ is an Ordered Set (of $T$),
                    $t$ is a $T$:
  id$_n(\theta)$ = id$_{n-1}$(Second($\theta\Big|_{n>1}$))  or First($\theta\Big|_{n=1}$)  or Reject$\Big|_{n=0}$ ;
end id;
```

```
Structure: ($y_1$, $y_2$) = Coinclude($x$):
        where $x$, $x_1$, $x_2$, $y_1$, $y_2$, $x_{[a]}$, $x_{[b]}$ are of some type,
              $a$, $b$ are Ordered Sets (of Naturals):
  Coinclude: ($y_1$, $y_2$) = $f_1(x_1, x_2)$ join     ($x_1$, $x_2$) = Clone$_2$($x$);
  $f_1$: $y_1 = A(x_{[a]})$                join     $x_{[a]}$ = id$_{[a]}$($x_1$)
                                             include $y_2 = B(x_{[b]})$
                                             join $x_{[b]}$ = id$_{[b]}$($x_2$);
syntax: $y_1 = A(x_{[a]})$               coinclude $y_2 = B(x_{[b]})$;
end Coinclude;
```

```
Structure: $y$ = Coor($x$):
          where $x_1$, $y_1$, $x_{[a]}$, $x_{[b]}$ are of some type,
                property is a Property (of $T$),
                ($a$, $b$) are Ordered Sets (of Naturals):
  Coor: $y = f_1(x)\Big|_{\text{Has(property},x)=\text{True}}$             or

                                 $y = f_2(x)\Big|_{\text{Has(property},x)=\text{False}}$ ;
  $f_1$: $y = A(x_{[a]})$                      join     $x_{[a]}$ = id$_{[a]}$($x$);

  $f_2$: $y = B(x_{[b]})$                      join     $x_{[b]}$ = id$_{[b]}$($x$);
syntax: $y = A(x_{[a]})\Big|_{\text{property}}$              coor

                                   $y = B(x_{[b]})\Big|_{\text{Pnot(property)}}$ ;
end Coor;
```

Has is a primitive operation on type property [19] that provides a notion of associating a particular property with a value.

The **failure** structure, the definition of which follows, provides for the ability to "recover" from a "detected" error. The definition uses the **cojoin, coor, join,** and **each** structures.

```
**Structure:** $y$ = Failure($x$);
        **where** ($x$, $g$, $y$, $x_{[a]}$) **are of some type,**
            $a$ **is an** Ordered Set (of Naturals);
  Failure: $y = f_1(x, g)$                    **cojoin** $g = E(x)$;
  $f_1$: $y$ = Clone$_1(g)\big|_{g\neq\text{Reject}}$        **coor**   $y = f_2(x)\big|_{g=\text{Reject}}$ ;
  $f_2$: $y = F(x_{[a]})$                   **join**   $x_{[a]} = \text{id}_{[a]}(x)$;
**syntax:** $y = E(x)$                    **failure** $y = F(x_{[a]})$;
**end** Failure;
```

The operation definition for Clone$_1$, which is also used to define the **failure** structure, is defined in terms of the **join** structure and primitive operations on any type.

```
**Operation:** $u'$ = Clone$_1(u)$;
        **where** $u_1$, $u_2$, $u$, $u'$ **are** $T$s;
  Clone$_1$: $u'$ = Identify$_1(u_1, u_2)$   **Join** $(u_1, u_2)$ = Clone$_2(u)$;
**end** Clone$_1$;
```

An alternative way of writing the Clone$_1$ operation in AXES is simply to omit writing the operation itself; e.g.,

"y = Clone$_1(x)$" is equivalent to "$y = x$."

This alternative form appears often in AXES definitions throughout this paper.

every is a structure that requires at least two members of an Ordered Set (of T) as input and successively performs the same operation on the result of the operation performed on the first two members and the next member, as in the sum of a set of naturals or the product of a set of rationals.

```
**Structure:** $y$ = Every($x$);
        **where** $x$, $x_1$, $x_2'$, $x_2$, $g$, $y$, $x_2''$ **are of some type;**
  Every: $y$ = Reject$\big|_{x_2=\text{Reject}}$      **or**    $y = f_0(x_1, x_2\big|_{x_2\neq\text{Reject}})$
                                  **join** $(x_1, x_2) = S(x)$;
  $f_0$: $y = F(x_1, x_2'\big|_{x''_2=\text{Reject}})$   **coor**  $y = f_1(x_1, x_2', x_2''\big|_{x''_2\neq\text{Reject}})$
                            **cojoin** $(x_2', x_2'') = S(x_2)$;
  $f_1$: $y = F(x_1, g)$               **cojoin** $g = f_0(x_2', x_2'')$;
**syntax:** $y = f<x>$;
**end** Every;
```

The **every** structure uses operation S, which produces the first and second component of an Ordered Set, the specification of which follows:

```
**Operation:** $(x_1, x_2) = S(x)$;
        **where** $(x_1, x_2)$ **are** Ordered Sets (of $T$),
              $x_1$ **is a** $T$;
  $S$: $x_1$ = First($x$)   **coinclude** $x_2$ = Second($x$);
**end** $S$;
```

The **extract/filter** structure, discussed in the main text of this paper, is specified using structures (of which **cojoin, coor,** and **coinclude** have been specified above and **incorp** is specified below), data types previously defined (see Appendix 1), and operations (of which Find and Search are specified here and Lmkrgs is discussed in length in [32] and in summary in the main text of this paper).

```
**Structure:** $y$ = Extract/Filter($x$, $s$, $m$, $c$);
    **where** $x$, $y$, $x_m$ **are** States (of Satellites),
          $x$, chip **are** Ordered Sets (of Images),
          $m$ **is a** place,
          $c$, ellipse, ellipse$'$, ellipse$''$
            **are** Ordered Sets (of Ordered Sets (of Images)),
          $s$ **is an** Ordered Set (of Images),
          $p_1$, $p_2$ **are** Scalars;
          $\theta$ **is an** Option;
  Extract/Filter: $y = f_1(x, s, m, c\big|_{m\neq\text{Reject}})$          **coor**
                                        $y$ = Reject$\big|_{m=\text{Reject}}$ ;
  $f_1$: $y = f_2(x_m, s, m, c, \text{ellipse})$                   **cojoin**
                         $(x_m, \text{ellipse})$ = Lmkrgs$(x, s, m)$;
  $f_2$: $y$ = Reject$\big|_{x_m=\text{Reject}}$                          **coor**
                      $y = f_3(x_m, s, m, c, \text{ellipse}\big|_{x_m\neq\text{Reject}})$;
  $f_3$: $y = f_4(x_m, s, m, \text{chip}, \text{ellipse})$                **cojoin**
                                   chip = Find$(m, c)$;
  $f_4$: $x'$ = Reject$\big|_{\text{chip}=\text{Reject}}$                       **coor**
                    $x' = f_5(x_m, s, m, \text{chip}, \text{ellipse}\big|_{\text{chip}\neq\text{Reject}})$;
  $f_5$: $y = f_6(x_m, s, m, \text{chip}, \text{ellipse}', p_1)$            **cojoin**
                    $(p, \text{ellipse}')$ = Search(chip, ellipse);
  $f_6$: $y$ = Reject$\big|_{p_1\leq 0}$                             **coor**
               $y'$ = Iterate$(x_m, s, m, \text{chip}, \text{ellipse}', p_1\big|_{p_1>0})$;
  Iterate: $y = N(x_m, s, m, \text{chip}, \text{ellipse}', p_1, \theta)$      **cojoin**
                                          $\theta = I(p_1)$;
  $N$: $y$ = Iterate$(x_m, s, m, \text{chip}, \text{ellipse}', p_1\big|_{\theta=\text{Enter}})$     **coor**
             $y = R(x_m, s, m, \text{chip}, \text{ellipse}', p_1\big|_{\theta=\text{Proceed}})$
                                          **coor**
             $y$ = **incorp**$_Q(x_m, s, m, \text{ellipse}'\big|_{\theta=\text{Terminate}})$;
  $R$: $y = W(x_m, s, m, \text{chip}, \text{ellipse}'', p_1, p_2)$          **cojoin**
                 $(\text{ellipse}'', p_2)$ = Search(chip, ellipse$'$);
  $W$: $y$ = Iterate$(x_m, s, m, \text{chip}, \text{ellipse}'', p_2\big|_{p_2-p_1\leq 0})$    **coor**
               $y$ = **incorp**$_Q(x_m, s, m, \text{ellipse}''\big|_{p_2-p_1>0})$;
**syntax:** $x'$ = **extract/filter**$_{I,Q}(x, s, m, c)$;
**end** Extract/Filter;
```

```
**Structure:** $x'$ = Incorp($x$, $s$, $m$, $e$);
    **where** $x$, $x'$, $x_t$ **are** States (of Satellites),
          $e$, $e'$ **are** Ordered Sets (of Ordered Sets (of Images)),
          $\theta$ **is an** Option,
          $m$ **is a** Place,
          $s$ **is an** Ordered Set (of Images);
  Incorp: $x' = f_1(x_t, e, e')$            **cojoin**
                           $(x_t, e')$ = Final$(x, s, m, e)$;
  $f_1$: $x'$ = Reject$\big|_{\text{Test}(e,e')\leq 0}$            **coor**
                      $x'$ = Assure$(x_t, e, e')\big|_{\text{Test}(e,e')>0}$;
  Assure: $x' = f_2(x_t, e, e', \theta)$         **cojoin**     $\theta = F(e, e')$;
  $f_2$: $x'$ = Assure$(x_t, e, e'\big|_{\theta=\text{Enter}})$     **coor**
                        $x'$ = Update$(x, e, e'\big|_{\theta=\text{Proceed}})$
                          **coor**
                                $x'$ = Reject$\big|_{\theta=\text{Terminate}}$ ;
**syntax:** $x'$ = **incorp**$_F(x, s, m, e)$;
**end** Incorp;
```

incorp is a structure intended to incorporate a measurement and update the estimated state of a satellite.

In **incorp**, Test is a scalar valued operation that checks the quality assurance of the measurement based on predetermined criteria [32], and operation F is a user supplied operation that may impose additional quality assurance checks. F could be allocated to a human operator, for example, whose "better judgment" would be the additional quality assurance function.

```
Operation: Chip = Find($m$, $c$);
  where $m$ is a place,
        $c_2$, $c$ are Ordered Sets (of Ordered Sets (of Images)),
        $l_2$, $c_1$, chip are Ordered Sets (of Images),
        $l_1$ is an Image,
        $g$ is a place,
        $b$, $b'$ are Booleans;
 Find: chip = $f_1(m_1, c_1, c_2)$  cojoin $(c_1, c_2) = S(c)$;
 $f_1$: chip = $\text{Reject}\Big|_{c_1=\text{Reject}}$       coor  chip = $f_2(m, c_1, c_2\Big|_{c_1\neq\text{Reject}}$  );
 $f_2$: chip = $f_3(m, c_1, c_2, b)$  cojoin $b$ = Locate$(m, c_1)$;
 $f_3$: chip = $c_1\Big|_{b=\text{True}}$              coor  chip = Find$(m, c_2\Big|_{b=\text{False}}$  );
 Locate: $b = f_4(m, l_1, l_2)$     cojoin $(l_1, l_2) = S(c_1)$;
 $f_4$: $b$ = False$_{l_1=\text{Reject}}$            coor  $b = f_5(m, l_1, l_2{}_{l_1\neq\text{Reject}}$  );
 $f_5$: $b = f_6(m, l_2, b')$        cojoin $b'$ = Equals$(m, g)$
                                cojoin $g$ = Location$(l_1)$;
 $f_6$: $b$ = True$\Big|_{b'=\text{True}}$              coor  $b$ = Locate$(m, l_2\Big|_{b'=\text{False}}$  );
end Find;
```

Find is an operation that "finds" the set of images that contains place m in a set of sets of images c.

```
Operation: $(p, e')$ = Search(chip, $e$);
 where chip is an Ordered Set (of Images),
       $R$, $e$, $e'$ are Ordered Sets (of Ordered Sets (of Images)),
       $p$, $p'$ are Scalars,
       $\xi$, $\chi$ are Ordered Sets (of Scalars);
 Search: $(p, e') = f_1(\text{chip}, e\Big|_{\text{First}(e)\neq\text{Reject}}$  )
                              coor        $p = 0$
                              coinclude   $e' = e\Big|_{\text{First}(e)=\text{Reject}}$  ;
 $f_1$: $(p, e' = f_2(\text{chip}, R, p'))$
                              cojoin      $(R, p') = f_3(\text{chip}, e)$;
 $f_2$: $(p, e')$ = Search(chip, $R\Big|_{p'\neq 0}$  )
                              coor        $p = p'$
                              coinclude   $e' = R\Big|_{p'>0}$  ;
 $f_3$: $R$ = Second$(e)$               coinclude   $p' = f_4(\xi, \chi)$
                              join        $\xi$ = Intensity[First $(e)$]
                              include     $\chi$ = Intensity[chip];
 $f_4$: $p' = \dfrac{\Sigma<[\xi*\chi]>}{\Sigma<[\xi*\xi]>*\Sigma<[\chi*\chi]>}$;
 end Search;
```

Search is an operation that matches a preselected set of images with a component of a set of sets of images when a positive correction p is found. The **each** and **every** structures are used to define p. In the definition of Search, "Σ" is an alternative symbol for the sum operation on Scalars and "$*$" is an alternative symbol for the product operation on Scalars.

ACKNOWLEDGMENT

We would like to thank Barry Boehm of TRW for a most helpful review of this paper.

REFERENCES

1. M. Hamilton and S. Zeldin, Integrated Software Development System/Higher Order Software Conceptual Description, TR-3, Higher Order Software, Inc., Cambridge, Massachusetts, November 1976.
2. M. Hamilton and S. Zeldin, Higher Order Software—A Methodology for Defining Software, *IEEE Trans. on Software Engineering* SE-2 (1), 9–32 (1976).
3. C. V. Ramamoorthy and H. H. So, Appendix to Requirements Engineering Research Recommendations, *Software Requirements and Specifications: Status and Perspectives* (August 1977).
4. M. A. Jackson, *Principles of Program Design*, Academic Press, New York, 1975.
5. R. F. Bridge and E. W. Thompson, A Module Interface Specification Language, Information Systems Research Laboratory, University of Texas at Austin, Technical Report No. 163, December, 1974.
6. J. E. Horowitz Guttag and D. Musser, The Design of Data Structure Specifications, *Proc. 2nd International Conference on Software Engineering*, October 1976, pp. 414–420.
7. L. Robinson and R. C. Holt, Formal Specifications for Solutions to Synchronization Problems, Computer Science Group, Stanford Research Institute, 1975.
8. Computer Sciences Corporation, A Users Guide to the Threads Management System, City, State, November 1973.
9. M. W. Alford, *R*-Nets: A Graph Model for Real-Time Software Requirements, *Proc. MRI Symposium on Computer Software Engineering*, April 1976, pp. 97–108.
10. C. G. Davis and C. R. Vick, The Software Development System, *Proc. 2nd International Conference on Software Engineering*, October 1976, Addendum pp. 27–43.
11. Hughes Aircraft Company, 1975 IR&D Structured Design Methodology, Vol. II: Structured Design, FR 76-17-289, Fullerton, California, 1975.
12. D. Teichroew and E. A. Hershey III, PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, *IEEE Trans. on Software Engineering* SE-3 (1), 41–48 (1977).
13. IBM, HIPO: Design Aid and Documentation Tool, IBM SR20-9413-0, Bethesda, Maryland, 1973.
14. D. Ross, Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Trans. on Software Engineering* SE-3 (1) 16–34 (1977).
15. M. L. Wilson, The Information Automat Approach to Design and Implementation of Computer-Based Systems, Report IBM-FSD, IBM, Bethesda, Maryland, June 1975.
16. M. Hamilton and S. Zeldin, Reliability in Terms of Pre-

dictability, *Proceedings, COMPSAC '78*, Chicago, Illinois, IEEE Computer Society Cat. No. 78CH1338-3C, November, 1978.
17. M. Hamilton and S. Zeldin, The Manager as an Abstract Systems Engineer, *Digest of Papers, Fall COMPCON 77*, Washington, D.C., IEEE Computer Society Cat. No. 77CH1258-3C, September 1977.
18. M. W. Cashman, An Interview with Prof. Edsger W. Dijkstra, *Datamation* 23 (5), 164–166 (1977).
19. M. Hamilton and S. Zeldin, AXES Syntax Description, TR-4, Higher Order Software, Inc., Cambridge, Massachusetts, December 1976.
20. M. Hamilton and S. Zeldin, The Foundations of AXES: A Specification Language Based on Completeness of Control, Doc. R-964, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, March 1976.
21. J. Guttag, The Specification and Application to Programming of Abstract Data Types, Univ. of Toronto Technical Report CSRG-59, September 1975.
22. C. A. R. Hoare, An Axiomatic Approach to Computer Programming, *CACM* 12, 576–580 (1969).
23. B. H. Liskov and S. N. Zilles, Specification Techniques for Data Abstractions, *IEEE Trans. on Software Engineering* 1 (1), 7–9 (1975).
24. J. V. Guttag, E. Horowitz, and D. Musser, Some Extensions to Algebraic Specifications, in *Proc. of an ACM Conference on Language Design for Reliable Software* (D. B. Wortman, ed.), Raleigh, North Carolina, Association for Computing Machinery, New York, March 1977.
25. Z. Manna and R. Waldinger, The Logic of Computer Programming, *IEEE Trans. on Software Engineering* SE-4 (3) 199–229 (1978).
26. Higher Order Software, Inc., Techniques for Operating System Machines, TR-7, Cambridge, Massachusetts, July 1977.
27. M. Hamilton, and S. Zeldin, Top–Down/Bottom–Up, Structured Programming and Program Structuring, Rev. 1. Doc. E-2728, Charles Stark Draper Laboratory, Inc., December 1972.
28. D. Harel and R. Pankiewicz, The Universal Flowcharter, TR-11, Higher Order Software, Inc., Cambridge, Massachusetts, November 1977.
29. J. Rood, T. To, and D. Harel, A Universal Flowcharter, *Proceedings of the NASA/AIAA Workshop on Tools for Embedded Computer Systems Software*, Hampton, Virginia, November 7–8, 1978, pp. 41–44.
30. A. F. Fuchs, C. E. Velez, and C. C. Goad, Orbit and Attitude State Recoveries from Landmark Data, *The Journal of Astronautical Sciences* XXIII (4), 369–381 (1975).
31. Computer Sciences Corporation, Navpak Design for Landsat and Kalman Filter Applications, CSC/TM-77/6012, Arlington, Virginia, January 1977.
32. Higher Order Software, Inc., A Demonstration of AXES for Navpak, TR-9, Cambridge, Massachusetts, September 1977.
33. Higher Order Software, Inc., The Application of HOS to PLRS, TR-12, Cambridge, Massachusetts, November 1977.
34. S. Cushing, Geographically Distributed Systems in Higher Order Software, DCPA Memo No. 7, Higher Order Software, Inc., Cambridge, Massachusetts (in preparation).
35. J. R. Searle, review of J. M. Sadock, *Towards a Linguistic Theory of Speech Acts, Language* 52, 1976.