The Functional Life Cycle Model and Its Automation: USE.IT

M. Hamilton and S. Zeldin

Higher Order Software, Inc.

Recently there has been an accelerated awareness of the urgent need for effective system development tools and techniques. Towards this end attempts have been made to develop standard languages for programming and standard techniques for the front end of the development process. Further, there has even been talk of integrating the various processes within a system life cycle. Our thesis, here, is that although these are steps leading in the right direction, they are simply not enough. What we suggest is needed, rather, is a totally new life cycle model; it is based on pure functional needs. This is in contrast to the "event" driven model that has been forced into being based on constraints which are often unnecessary, wasteful, and error prone. The functional model, itself formally defined, not only includes formal techniques for defining the front end, but it also includes techniques which integrate by means of formal methods and automation that front end to the rest of the life cycle of a system. A functional life cycle model has been defined. USE.IT, an implementation of this model, provides for an integrated and automated development process of a system. We discuss, here, the HOS functional model, USE.IT, and the implications of their use.

INTRODUCTION

Software is a set of logical statements which can be and which is intended to be executed by a computer.[1] Software is used for many purposes. In fact, software is used within a particular computer environment to help users of particular applications to prepare other software. We call a system which has been developed in terms of a computer(s) and its interrelated software systems an embedded system. That part of an embedded system which directly relates to the computer and its own embedded software is traditionally thought of as software.

Indeed, a major aspect of our existence is software, since the success or failure of an embedded system is only as good as its software. Software can make or break an economy; it can make the difference between a good relationship and a bad relationship; it can make the difference between getting to a destination or not getting to one; it can make or break companies, projects, or people; it can provide for smooth running bureaucracies or it can make them totally inoperable to the point of bringing everything involved to a standstill; it could prevent or start a world war. In a sense, therefore, it should not seem irresponsible to conclude that software is an integral part of that system which inherently controls society. But software depends on the cornerstone upon which its development is based. That cornerstone is the life cycle model.

The life cycle model is the system or the set of procedures, rules, tools, and techniques used to develop a system. A system developed by a particular life cycle model is one of its target systems. A successful life cycle model for developing embedded systems is a successful model for developing systems, in general; for what we are really talking about is an effective way to

[1]The term software, here, is used in a conventional sense. There are other kinds of sets of logical statements, however. Some of these sets differ only in that they can but are not intended to be executed by a computer. Suppose, however, for a given period of time that a portion of "software" was never executed on a computer and a portion of "non software" was. From a practical point of view, wherein lies the difference? There are still other kinds of logical statements that cannot be directly executed by a computer although there is the possibility of simulating their behaviors with software. Generically speaking, we use software to include the larger domain of a logical system which is computable of which software in a conventional sense is only a part. Our solutions for computable systems, however, cross over into the domain of solutions for non-computable systems as well. Thus, when we later discuss solutions for developing better "software" they can as well be used for developing better logical systems and vice versa.

Address correspondence to M. Hamilton, President, Higher Order Software, Inc., 955 Massachusetts Avenue, Post Office Box 531, Cambridge, MA 02139.

0164-1212/83/010025-38$3.00

think, to communicate our thoughts, and to practically realize those thoughts. The life cycle model includes a myriad of aspects that must first be understood and then integrated (Fig. 1). It starts with the definition of requirements for a particular set of users and ends with the final maintenance of a system that is operable by those users within their own environment.

The life cycle model of a system can make the difference between understanding a system and not understanding it. It can make the difference as to whether it costs millions or billions of dollars to either develop it or to operate it. It can make the difference between that system working or not working. It is clear that a major concerted effort should be taken by embedded system developers to ensure that the life cycle model for developing systems is an effective one. For, if such an effort is not made, the life cycle model will be controlling us, by being out of control, and not us the life cycle model.

THE HISTORICAL LIFE CYCLE MODEL—A PROTOTYPE

We view the conventional, or historical, model as a prototype. That is, it is useful from a historical point of view in that we can learn from it for developing the model of the future; but it should be viewed simply as that, and not as something to be taken seriously as the model of the future.

The historical model established its basic structure almost overnight, over 20 years ago, in order to answer the needs of an extremely fast growing and accelerated hardware technology and its eager users. Its history was and has continued to be influenced not only by events and the timing of events surrounding hardware development but by the politics which surrounded it as well. As a result, patches to development techniques were added to patches of development techniques to adjust to the fast changing environment of hardware and its users. That is, there is a problem to be solved, solutions or partial solutions are attempted on an *ad hoc* basis; they are then incorporated and become firmly locked in as part of the life cycle model. Often, attempts have been made to force individuals to accommodate its inadequacies into general solutions. Many solutions unfortunately have often been implemention dependent in that they are wrapped up with a particular environment's peculiarities. And, of course, many of the solutions are wrapped up with different diverse environments, making it difficult or impossible to integrate them when it comes necessary to do so.

The historical model is a ready candidate on which to perform a "fresh start." In order to start over one must remove any preconceived notions with respect to *how* the existing life cycle model accomplishes its job. He only cares about *what* it should accomplish. Such a commonsense, or functional approach, is helpful in attempting to understand any phenomenon. Here, useful functions are selected, relationships between these functions are determined (and resolved if they are inconsistent), and redundant functions are eliminated [1]. Once this process has been performed it is easier to get an idea of what functions are missing. Although one could interpret such an approach as a relational one, the relations between functions can ultimately be understood in terms of functions.

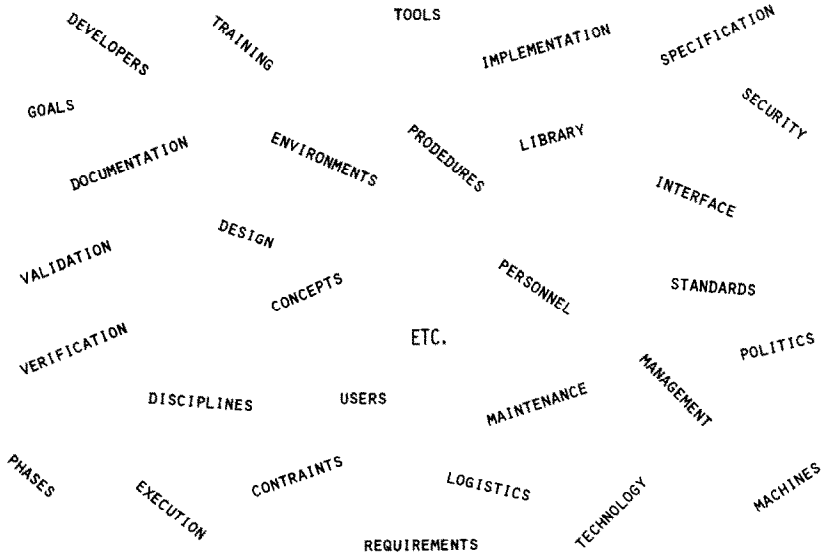


Figure 1. Life cycle model components.

Some Shortfalls of the Historical Life Cycle Model

It is not an unknown fact that the historical life cycle model has serious problems. Over the years, we have collected a list of complaints resulting from experiences of ourselves and others which illustrate this fact further (Fig. 2). The same complaints exist today as they did many years ago. But all is not lost. We have an opportunity to observe just about any type of failure (and to be fair, some successes) imaginable with all of the collective experiences behind us with the use of this prototype model. In the process of doing so, it is helpful to determine why certain procedures exist. This sometimes helps to justify, or at least give a warmer feeling for, the elimination of procedures which functionally do not make sense within the context of an overall systematic process. Do they exist because other procedures which were not working made new and additional procedures necessary as a quick fix? Do they exist because of a peculiar trait that a particular hardware architecture had? Do they exist because of a peculiar trait that a particular human being had? Do they exist because someone many years ago solved a particular problem in a certain way and so everyone else did it that way because that is the only way it had been done before? Do they exist because too many people are already doing it that way to make the investment to change? Do they exist because that is the way people were trained to do it? Do they exist because people are afraid of change? Do they exist because . . .? The answer is "yes" in all cases.

It was not until we defined a functional model that we understood the problems of the historical model as we do now. This is due to the fact that the functional model explicitly pointed out problem areas which were not obvious to us before. It also suggested solutions not theretofore thought possible. In the historical model (Fig. 3) there are manual processes during and between all phases of development. Manual processes encourage the introduction of new errors into a target system. Many of the processes of the historical model are candidates for obsolescence. When manual processes are automated, for example, those processes established to support each of the manual processes can be eliminated. Unnecessary algorithms for each target system are developed and maintained, sometimes for years. A proliferation of sophisticated tools has resulted in order to either answer a particular machine environment's needs or to help manage the horrendous problems that compound developers' problems. These include such tools as higher order languages, compilers, operating systems, and verification aids. Operating systems, for example, are often worrying about uncontrolled interrupts, deadly embraces, etc. that they would not be concerned with if the system, so defined, was defined correctly in the first place. Both compilers and operating systems concern themselves with complex resource allocation algorithms they should have no concern for; again this would not need to be the case if the systems they dealt with were defined differently. Verification tools test over and over again during every system dynamic run for unnecessary errors. In fact, the very use of a technique can eliminate the possibility of particular kinds of errors from even happening [2]. Again, many costly tools have been developed and are now continuously maintained and used, unnecessarily.

Usually a target system in the historical model is described in a different language for each different phase of its development. Why? Probably because as each new phase throughout the evolution of the historical model was thought to be necessary, those creating that phase used or made up their own language. Not only is it necessary to translate a target system from the language of one phase to a language of another phase, but once a given phase of a target system is defined, it must be proven to correspond to a previous phase. This has become an extremely difficult process. Why? Probably because the target system in each of these phases is defined in different languages!

The historical model is inefficient in its ordering of processes. It, for example, leaves the majority of verification and validation until the end. As a result errors live in the target system longer than necessary and these errors encourage new errors. One reason for this is that the "language" has not really been formal enough until the last phase (i.e., coding in software) to verify a system defined in terms of that language. Surprises then occur at the very end of development when it is often too late to do anything about them. Or, a whole new development step, throughout all the phases, must be taken to "fix" a problem, *if* it is found.

Since the historical model is not functionally understood, those systems developed with it are not functionally understood as well. As a result, it is not known when an object within a system can be a static entity or a "constant" within a development process. When, for example, is it no longer necessary to verify an object? Again, everything must be treated as an unknown until the end. Thus, dynamic verification must always be performed at its maximum. And this is impossible in large complex systems.

The languages in the historical life cycle model are traditionally syntax-oriented (i.e., emphasis on how it is being said, not on what is being said) and each language has its own fixed syntax. The fact is, everytime a new syntax arrived, a new semantics method would ar-

- COMPLEX SYSTEMS
- REQUIREMENTS ALWAYS CHANGING: NEW IDEAS OR ERRORS
- UNREALISTIC ESTIMATES OF COMPUTER TIME, MANPOWER CALENDAR TIME, ON-BOARD COMPUTER SPACE AND TIME
- POOR VISIBILITY AND TRACEABILITY
- ASYNCHRONOUS ASPECTS OF SOFTWARE AND ITS INTERFACES
- CLASS OF PROBLEMS INCLUDE THOSE WHICH ARE TIME-CRITICAL AND SELF-CORRECTING (FEEDBACK)
- UNCERTAINTY OF HOW MUCH TO TEST: REDUNDANCIES AND OMISSIONS
- STANDARDS AND DISCIPLINES NOT DEFINED
- AMBIGUOUS, IMPLICIT, TOO DETAILED, OR INCORRECT REQUIREMENTS
- FRAGMENTATION OF PERSONNEL
- UNKNOWN HARDWARE EFFECTS ON SOFTWARE
- SOFTWARE MUST ACCOMMODATE HARDWARE
- MANAGEMENT PROBLEMS INHERENT IN LARGE SYSTEMS; TOO LITTLE OR TOO MUCH
- DIFFICULTY IN MEASURING CORRECTNESS OF SOFTWARE
- SOFTWARE NOT TRANSFERABLE
- SYMPTOMS RATHER THAN ROOT PROBLEMS TREATED
- SYSTEM NOT UNDERSTOOD
- NO INTEGRATED GOALS
- NO INTEGRATED METHODOLOGY
- STRUCTURE OF SYSTEM DEVELOPMENT PROCESS NOT FLEXIBLE ENOUGH TO ENCOURAGE MULTIPLE TECHNOLOGIES, VENDORS, COMPETITIVE INNOVATION AND MULTIPLE SOURCING
- DIFFICULTY IN MEASURING EFFECTIVENESS OF SOFTWARE METHODOLOGY/TOOLS
- COSTLY AND LENGTHLY EFFORTS
- LACK OF SUFFICIENT DOCUMENTATION: TOO LITTLE OR TOO MUCH
- THE PROBLEMS OF PARKINSON'S LAW
- POOR COMMUNICATION
- COMMUNICATION LAGS
- COMMUNICATION INTERFACES NOT DEFINED
- LURKING ERRORS
- "MAN-RATED"
- PROLIFERATION OF LANGUAGES, RULES, SYNTAX, TOOLS, ALGORITHMS
- CANNOT BE VERIFIED IN THE REAL WORLD
- SYSTEM DOES NOT LIVE UP TO EXPECTIONS
- NEED FOR AUTOMATIC ERROR DETECTION SCHEMES
- LACK OF FLEXIBLE RECONFIGURATION SCHEMES DURING DEVELOPMENT AND REAL TIME
- MISUNDERSTANDINGS ABOUT CAPABILITIES OF SUPPORT SYSTEMS
- LOSS OF PROJECT CONTROL
- NO WAY TO INTEGRATE DIVERSE ENVIRONMENTS
- DESIGN BY AUDITORIUM
- PARADOX OF REDUNDANCY MANAGEMENT SCHEMES
- IMPROPER STRUCTURING OF INCENTIVES FOR CONTRACTORS AND GOVERNMENT MANAGEMENT PERSONNEL
- SPECIFIC AND NARROW SCOPED INTERESTS
- DIFFICULTIES IN PERSONNEL ATTITUDES TOWARDS COOPERATION
- FALSE ECONOMIES
- OVER SOPHISTICATION
- CREATION OF "URGENT" PROBLEMS BY FAILURE TO ANTICIPATE TROUBLES OR RESPOND EXPEDITIOUSLY

Figure 2. Typical problems of complex system efforts.

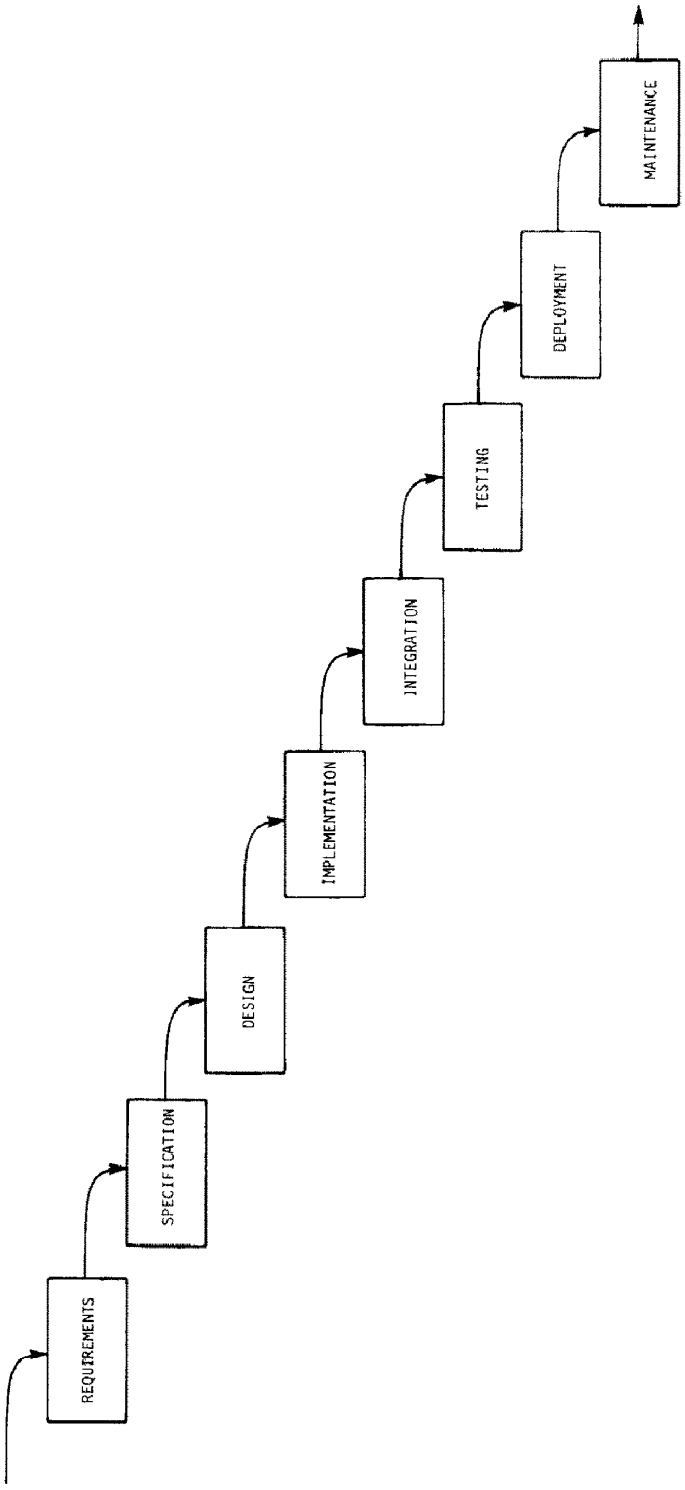


Figure 3. Historical life cycle model.

rive as well. A fixed syntax is not necessary to maintain a desirable state; a fixed semantics is. In a historical method, the opposite seems to hold true.

Within a historical model, the definition of requirements for a system invariably contains system implementation details which depend on either the type of language it will be implemented in or which depend on the type of computer environment it will be executed in. Thus, if there is a new language or a new computer introduced into the target system environment, a target system has to be completely developed again from scratch. Such a situation is indeed wasteful.

Attempts to accommodate the Historical Life Cycle Model

Several attempts have been made by software tool developers to alleviate the problems of embedded system developments. In fact in doing so, all of these tool developers are attempting to alleviate the problems of the existing historical model. They, however, in attempting to provide solutions, make the assumption that they must include as part of their requirements the existence of the historical model as a given.

For example, take the requirements phase, in the traditional sense, which usually tries to address the problem of user needs from the total system point of view. Typical techniques that attempt to address the requirements phase are SADT [3], and PSL/PSA [4]. The use of these techniques, however, can result in error prone requirements and as a result these techniques do not lend themselves as either reliable requirements or as a first step towards reliable code.

When system requirements are "completed," usually marked by the appearance of a requirements "document," the parts of the system informally allocated to software are used as input to the software specification phase. The specification phase deals with software, as opposed to system, requirements. Here, the designer concentrates on defining the target system a little bit better, adding a few requirements for how best to sequence operations, perhaps what computer to use, or other "high level" resource allocation considerations. There are, today, a variety of techniques that are used for this phase: the SREM method of the SDS [5], Warnier Orr [6], HDM [7], Information Hiding [8], Structured Analysis/Design [9] CADES [10], etc.

Each successive phase from design to code adds more and more detail to both the definition of the target system and to the resource allocation of that target system to a computer. Of course, the fact that the resource allocation process itself must be defined (and itself resource allocated) only adds to the problems of what to do at each phase. Recently, tools have been introduced to enable users to create partial applications automatically [11–15]. However, complex logical and data manipulation functions still must be handcoded. To help alleviate the problems here, programs are augmented by sophisticated notational schemes, such as [16, 17], beyond the patience (and time constraints) of the average programmer, or programs are abstracted by symbolic execution techniques [18–20], rather than conceived of abstractly to begin with, or programs are subjected to time consuming test generation programs [21, 22] rather than concentrating on maximizing static analysis.

Similarly, there are other tools and techniques for use in both the earlier and later phases of the historical model. More often than not, these methods exist as a result of or for supporting manual processes. Thus, they are used, for example, to find errors. In fact, the errors would not even be there if the other methods resulting in the errors had accomplished the results they could have accomplished in the first place.

It is true that some of these tools, or combinations thereof, partially solve the problems of parts of the historical model. But, there is no integrated set of tools within the categories discussed above which tackles the whole problem. The root problem that has confronted software tool developers is that they are relating to and depending on an inferior life cycle model. The solution is not to support the historical model but rather to learn from it and then to replace it. With such an approach, history is used to support and not control an evolving life cycle model. This is as it should be.

STEPS TOWARDS THE FUNCTIONAL LIFE CYCLE MODEL

Efforts towards improving software techniques have concentrated, until very recently, on better ways to talk to a computer as opposed to better ways for users and developers to talk to themselves or to each other. If we review the evolutionary nature of software development techniques (Fig. 4) it is apparent that the first and second generations of software development were focused around how to communicate with the computer hardware. By the early 70's however, both users and developers of software were keenly aware of the magnitude of problems that were possible in embedded systems. These problems were due to unreliable software and to lack of formal means for users to communicate to each other or their developers. The result was a split of efforts in both directions, during the third generation, where some people were concentrating on making more reliable higher order programming languages and others were concentrating on making more user-friendly requirements/specifications languages. But the more

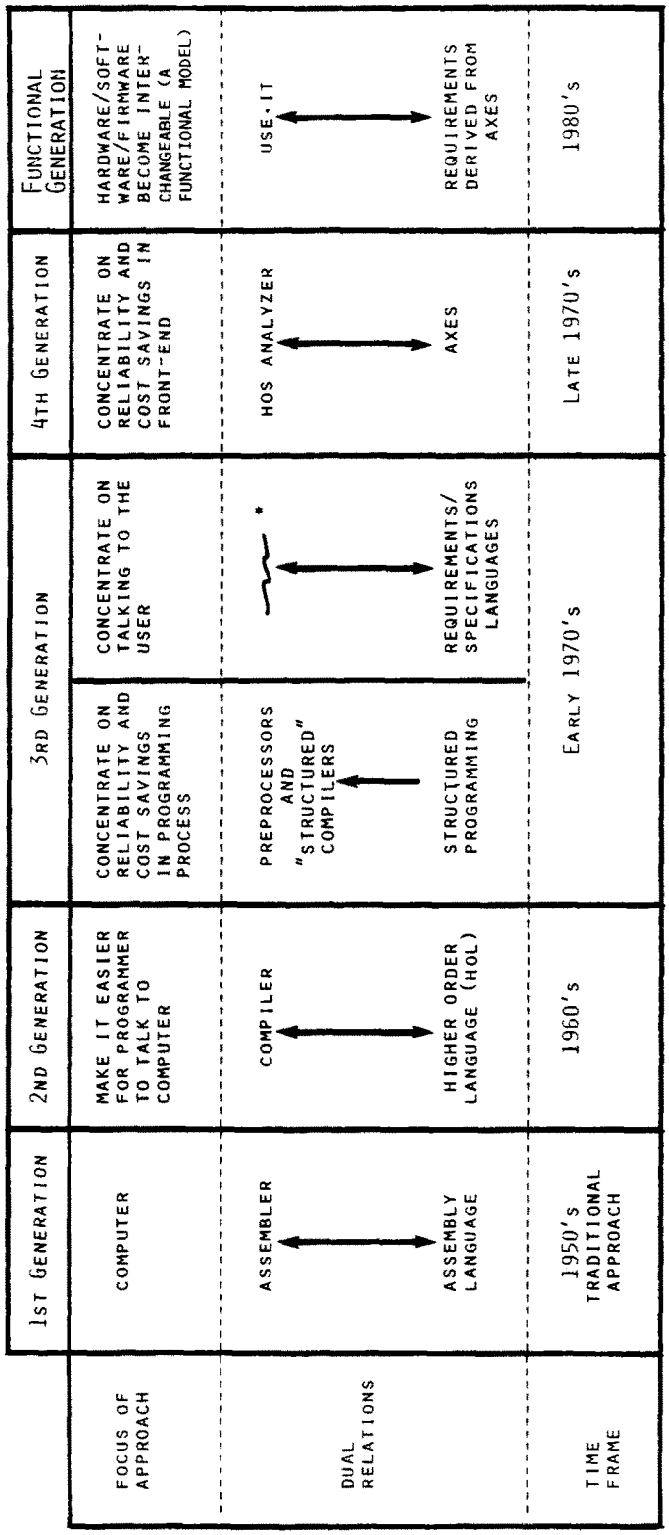


Figure 4. The evolutionary nature of software development techniques.

reliable higher order languages were not user-friendly and they were still not reliable enough, and the user friendly languages were friendly to select groups and they were not reliable at all. It was at this same time that our own staff was concentrating on both aspects of the problem and as a result (see fourth generation) came up with a formal requirements definition language which was intended both to be used at all levels of communication, including the user level, and to address the issues of reliability, not heretofore addressed, particularly at the front end. It was now possible to proceed with the development of the model for the functional generation. With the functional generation, one is able to define systems, whether they be hardware, software, humanware, or some combination, by merely collecting modules from a library. There would only be the *choice* of which "what" should be done and which "how" it should be done by. Once such a choice is made, the "what" and the "how" can be collected from the library of system modules.

PROPERTIES OF FUNCTIONAL GENERATION SYSTEMS

There is a growing awareness, today, of the need for quality metrics with respect to techniques and tools used for systems development [23–25]. In the process of defining the functional life cycle model we found it necessary to establish a set of properties for systems in general. Towards this end, we defined a checklist of desirable properties for a methodology[2] [2], properties of systems from a user point of view [26], properties of systems from a requirements definition point of view [27], and properties of systems as criteria to evaluate development methods [28].

The properties of concern here are those properties that can be used to measure the adequacy of a solution to the problem of developing systems. The result of such a solution is an effective life cycle model for developing a system. The output of an effective life cycle model is a well-formed system.

First and foremost, a system must be scientifically based (Table 1). It follows, then, that an effective system for developing systems, including the well-formed systems that are developed with the developmental system, must be scientifically based. The system for developing systems must formally address practical solutions to problems of development (e.g., elimination of errors, avoiding constraints to creativity, eliminating unnecessary steps, etc.) including the practical solution of providing a means to define scientific-based systems. What this means is that any effective development system must be able to provide a means for expressing the practical properties of any particular application area (financial, avionics, communications, manufacturing, etc.) in a formal way. However, an effective development system cannot guarantee that the designer that uses it will capture all of the practical properties of that designer's application. An effective development system can only guarantee the designer the formal and practical properties of a well-formed system.

Table 1. Scientific System Properties

Property	Definition
Formal	If a system is consistent and logically complete, that system is formal
Practical	To be practical, a system must have applicability to the problem to be solved (i.e., experimental results)

By formal, we mean a system that has the properties shown in Table 2. For example, a method for system development may be consistent and yet apply to only a part of the development process. That same method may be considered formal if it *completely* addresses the *part* of the development process it intended to address. However, if that same method is intended to address the *entire* development process and succeeds to address only one part (or parts of several parts) it may be consistent, but not logically complete and therefore not formal. On the other hand, if a method is inconsistent, then there is no way to show if it is logically complete. Subsequently, such a system is considered informal.

By practical, we mean a system that has the properties shown in Table 3. These practical properties (from the point of view of a system of properties) must themselves be formally defined and practically based. The formal properties in Table 3 are themselves based on the practical requirements for methods which came

Table 2. The Components of Formal

Property	Definition
Consistent	A system is consistent if it can be shown that no assumption of the system contradicts any other assumption of that system. One way to show consistency is to develop a model for the assumptions of the system (e.g., the three primitive control structures of HOS are models of the HOS axioms).
Logically complete	A system is logically complete if the assumptions of the system completely define a *given* set of properties. A logically complete system has a semantic basis (i.e., a way of expressing the meaning of system objects.

[2]"Methodology" is used here in the general sense, i.e., a set of rules that aid a designer in obtaining a solution to a problem.

about from the practical needs based on problems of systems development indicated in the last column of Table 3. If one considers the type of process around which rules or standards are necessary, it is either one of pulling apart or one of putting together objects. The abstraction, integration, and applicative properties all imply a process of going from many objects to one object; decomposition, modularity, and computable all imply a process of going from one object to many objects.

Abstraction and decomposition are inverse properties. In the decomposition of a function, for example, there are many possibilities for a chosen set of subfunctions each of which collectively replace that function. Once a set has been chosen, the function is decomposed. Given only the sets of subfunctions, however, to start with, one must abstract in order to find a function which satisfies each collection of subfunctions. Decomposition is "top-down" (we start, assuming the "top," and work our way down to communicate at the "bottom") and abstraction is "bottom-up," (we start, assuming the existence of the "bottom" and work our way up to communicate at the "top"). We need rules on the way up and rules on the way down. And, in the end, we need to understand *both* the top and the bottom. Both abstraction and decomposition are made up of other properties. Table 4 is an example of component properties for abstraction.

Integration and modularity are also inverse properties. The integration, for example, of one function with another implies that one of those functions has influence over the other (e.g., communication between functions, or one function affects another's timing). The modularity property ensures that each function in the integrated system is able to be selected as a separate entity and stand alone as a self-contained system. Proper integration calls for modular components while the development of each modular component calls for a means to integrate the subcomponents of that modular component. Practically, "reusable" software components have not been possible in traditional systems. The fear of uncovering new interface errors by uncoupling components embedded in larger systems has made even static reconfiguration difficult, if not impossible. How-

Table 3. Practical System Properties

Formal property	Definition	Practical method requirements	Practical user requirements
Abstraction	Abstraction refers to the ability to recognize commonality among a set of objects and then to identify a unique object for which any member of the set can stand for that unique object	Rules for deriving new definitions ultimately in terms of the same primitive mechanisms	Eliminate ambiguity, simplify complexity, flexibility
Decomposition	Decomposition refers to the ability to identify one of the possible sets of subordinates that can stand for a unique object	Rules for separating and relating the "what" and the "how"	
Integration	Integration refers to the ability to connect system components.	Reconfigure component environments	Eliminate interface errors, flexibility, reusable components
Modularity	Modularity refers to the ability to separate system components.	Reconfigure individual components	
Applicative	Applicative refers to functional in the mathematical context of function, i.e., the relationship of input to output. The input object exists. The output object exists. The function is the relation such that each input corresponds to one output.	Functional semantics	Eliminate unnecessary error-prone steps, flexibility, automation
Computable	If an algorithm can be established to allocate to each unique definition object so that it is set up to execute correctly on a machine then that definition is computable.	Functional instantiation	

Table 4. Component Properties of Abstraction

Formal property	Definition	Practical method requirements	Practical user requirements
Data behavior	Data behavior refers to the common relationships that hold between members of a given set of objects regardless of the components or parts, of the objects. For example, the linear ordering relationship among members of the data type time is a behaviorial characteristic of time. Formal data behavior implies that all derived data relationships must be able to be traced to common semantic primitives.	Standard primitive objects	Identify objects
Data structure	Data structure refers to the common relationships among components, or parts of an object. Abstraction of data structure refers to the ability to define common patterns between components of type members without specifying the particular instances that fit the pattern, e.g., a rational can always be replaced by a data structure of two integers. A data structure is consistent if a function can be defined from type to structure. For a data structure to be logically complete, all derived component relationships must be able to be traced to the common semantic constraints of the set of objects for which it is a model.	Standard primitive data relationships	Identify object implementation alternatives
Functional behavior	Functional behavior refers to the relationship between the input and the output of a function. Abstraction of functional behavior refers to the ability to state that relationship between input and output without specifying an algorithm for how that functional relationship will get accomplished. If the same input instance always produces the same output instance then the functional behavior is consistent. For functional behavior to be logically complete *each* input instance must produce an output instance in accordance with the input/output properties.	Standard primitive operations	Identify the "bottom" of a hierarchy
Functional structure	Functional structure refers to the relationship between functions. Abstraction of functional structure refers to the ability to define common patterns between functions without specifying the particular functions that fit the patterns or the particular execution model. For functional structure to be formal, all derived functional relationships must be able to be traced to common semantics primitives.	Standard primitive functional relationships	Identify common functional patterns

with properties of integration and modularity not only static reconfiguration, but also dynamic reconfiguration is not only possible, but is also practical. Again, both integration and modularity are made up of other properties. Table 5 is an example of component properties for modularity.

Whereas applicative is the property of relating one set of instances (domain) to another set of instances (range); computable is the property of relating these instances one pair at a time. That is, if you separate out each "component" of the applicative mode (i.e., specification mode) you get the computable mode (i.e., execution mode): conversely if you put together all components of the computable mode and do them "all at once" (i.e., all instances concurrently) you get the applicative mode. Practically, we need a means to determine what functions are necessary (and, once determined, identify which ones can be automated) and

Table 5. Component Properties for Modularity

Formal property	Definition	Practical method requirements	Practical user requirements
User independent	The ability to define a system with properties from which different user models can be derived. That is, such a system has no knowledge of its users just as a car does not need to know who is driving it to run, or an "add" instruction does not need to know about the "square" or "sum" users.	Diverse users	Design top-down or bottom-up, maintain a multiuser evolving library
Application independent	The ability to define any type of system	Diverse application environments	Eliminates constraints on creativity, eliminates need to learn a new method for each application area
Intent independent	The ability to define a system with analyzable properties from which different models with those properties can be verified.	Diverse readership	Eliminates the need to "return-to the author" for clarification or tie assertion statements of intent to definitions
Resource independent	The ability to define a system with functional properties from which different resource allocation models can be assigned (e.g., one allocation model may be time optimized, another memory optimized).	Diverse optimization of resource utilization	Eliminates the need to tie requirements to resources
Machine independent	The ability to define a system with computable properties from which different models for execution can be derived. A system can be formally computable and yet depend on a particular machine type (e.g., the sequential nature of the structured programming execution-flow control structures) or a particular machine of a type (e.g., company x, model y). If, however, we can establish computational abstractions, different models for the execution can be derived (e.g., the primitive control structures of HOS allow for parallel, sequential or multi-programmed execution models).	Diverse hardware environments	Transport from machine to machine
Syntax independent	The ability to define a system with semantic properties from which different syntactic models can be derived.	Diverse syntactic models	Enhance other methods, encourage user-friendliness

Table 6. Properties of HOS Systems

Formal property	Definition	Practical method requirements	Practical user requirements
Replacement	The ability to establish a relation in a set of objects so that any element can be substituted for any other element with respect to a unique object defined by that relation.	Standards for structure integrity	Assures the "how" conforms to the "what"
Access rights	The ability to locate an element of a given set of variables and once located, the ability to reference or assign the value of said element.	Standards for data flow	Maintain data security
Ordering	The ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements preceeds the other said element.	Standards for timing flow	Inherently be able to identify what can and cannot be done in parallel, while maintaining the ability to unambiguously decide what comes first in cases of potential conflict.
Domain identification	The ability to predict if a function will or will not be able to perform its intended function and to identify unintended from intended phenomena as part of a function definition.	Standards for error detection and recovery	Be prepared for the unexpected, provide outputs

what functions are sufficient (and, once determined, eliminate the redundant and obsolete ones). Only by separating the applicative property from the computable property can we find a means to deal with each effectively.

Properties of HOS systems (Table 6) provide rules for obtaining the practical system properties shown in Table 3. These properties, in turn, are used to define properties of the HOS life cycle model (Table 7). This ensures that systems developed with the life cycle model will have the same properties as those systems from which they came.

We have used these properties as metrics for evaluating development methods that span various aspects of the development life cycle [28]. In addition to using these properties as metrics for development techniques, there are other advantages to having available such properties that, on first glance, may not be so obvious.

Table 7. Properties of HOS Life Cycle

Formal property	Definition	Practical method requirements	Practical user requirements
Management	Management provides for control of a system. Control encompasses which functions are to be performed, input and output rights to data, the ordering of functions, the relationship between input and output, and what it means to have improper input. Consistent control provides for the ability to adhere to each aspect of control in a given system without any aspect conflicting with any other aspect of control. Logically complete with respect to control provides for the ability to trace the chain of command with respect to each aspect of control from level to level and layer to layer of a hierarchy.	Control, goal-driven commands	Eliminate interface errors in chain-of-command

Table 7. (*continued*)

Formal property	Definition	Practical method requirements	Practical user requirements
Definition	A definition of a system states the meaning of that system. A consistent definition of a system is one in which the meaning of that system can only be interpreted in one way. A logically complete system definition is one in which each system object can be traced to primitive semantic objects.	Rapid design rapid prototyping	Eliminate interface errors among users and developers
Analysis	Analysis provides for the capability to determine if a target system is a model of the method used for definition of that target system. Consistent analysis implies that a given target system will always compare to the method used for definition of that target system in the same way. Logically complete analysis provides for the ability to trace a target system as a model of the method used for definition of that target system.	Automatic verification	Eliminate interface errors before implementation
Resource allocation	A resource allocation assigns objects to names and names to objects. A consistent allocation maintains the properties of definition with respect to assignments. Logically complete allocation provides for the ability to trace objects to names and names to objects with respect to properties of definition.	Automatic programming	Eliminate interface errors between definition and machine environment
Execution	Execution provides for the instantiation of a target system. Consistent execution implies that a given model of a target system will always be performed in the same way. Logically complete execution provides for the ability to trace an object as a model of a target system.	The system works	Eliminate interface errors in real-time
Documentation	Documentation provides for a description of a system. A description is a symbolic representation of a system—one step removed from the system itself. A consistent description is one in which the meaning of the symbols used to define a system can be interpreted in one way. A logically complete description provides a means to trace the names of system objects with respect to properties of symbols	Automatic documentation	Eliminate interface errors in describing what is really there

For example, arguments for or against a particular brand of automation can be evaluated by determining the necessity or sufficiency of automating a particular process. A start in this direction (with respect to tools such as HOLs, PDLs, simulators, verification systems, etc.) can be found in [28].

THE FUNCTIONAL LIFE CYCLE MODEL

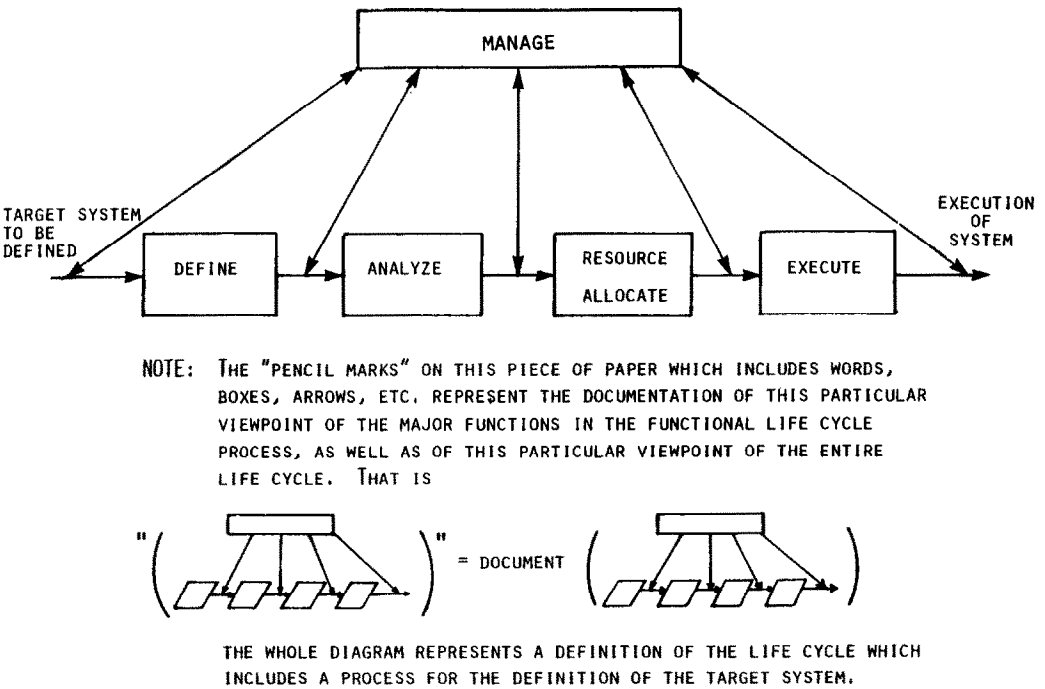
The functional life cycle model is a formal model of the functions and the relationships between those functions which *exist* in a system for effectively developing a system. This could be a model for developing a software system, a hardware system, a system of people, or some combination of software, hardware, and people [27–30].

The functional life cycle model is based on the HOS theory [1,2]. HOS, a systems theory based on analysis of large complex systems development, concerns, among other aspects, the definition of systems so as to eliminate data and timing conflicts. Systems, developed using this model, are themselves viewed by the model as data with respect to that model's functions. In this regard, each system that is developed with an HOS model is inherently forced to be defined in terms of the HOS theory in order to maintain the consistency and logical completeness properties of the model in the execution phase of its own development.

AN EXAMPLE OF THE FUNCTIONAL LIFE CYCLE MODEL

The example of the functional life cycle model described here has six major functions [28]. They are Manage, Define, Analyze, Resource Allocate, Execute, and Document. Although the function labeled "Define" in Figure 5 is a definitional process with respect to the target system being developed by the model, each of the functions in Figure 5 is, generically speaking, a definitional process, in its own right. That is, a definition relates one object to another (e.g., relating a value to its type). But, *with respect* to each given process, there are certain relationships with other processes that have to do with a development process [27]. That is, once a target system definition is completed, the target system is related to other systems within the target system environment in order to complete its development. The relations themselves are other systems within the target system environment. Specifically, once a target system is defined, it is related to a set of instances, or their

Figure 5. A definition of a functional life cycle process.



equivalent, (or analyzed[3]), to "test it out," related to a machine architecture (or resource allocated[4]) to "implement it," related to instantiations (or executed[5]) to "run it," and related to a communications vehicle to describe (or document) it. That which "integrates" or relates relationships between these processes is the management.

Again, generically speaking, every process in the life cycle could be viewed not only as one of definition but as one of management or of verification or of resource allocation or of execution or of documentation, since they are *all* that when viewed *with respect* to the particular relevant target system as a definition (Fig. 6). This is the very reason why people have arguments over such things as requirements, specifications, and implementations, since one person's specifications are another person's requirements. Similarly, one person's specifications are another person's implementation. What is important, then, is to first agree on the target system in question and then to agree on what its relative phases of development are. Take, for example, the definition of a target system. Within that definition each object has to be acknowledged and related to a type (definition), related to "what ifs" to see if it's the right relationship between the object and its type (verification of that definition), related to a "machine" (resource allocation of that verified definition), related to instantiations (execution of the resource allocated, verified definition). Take again, for example, the definition of that same target system. It, itself, could be an instantiation of another system. Now it is an input to the execution process of that other system. All of these definitions, therefore, are what they are as a result of each point of view of that definition with respect to its relationships.

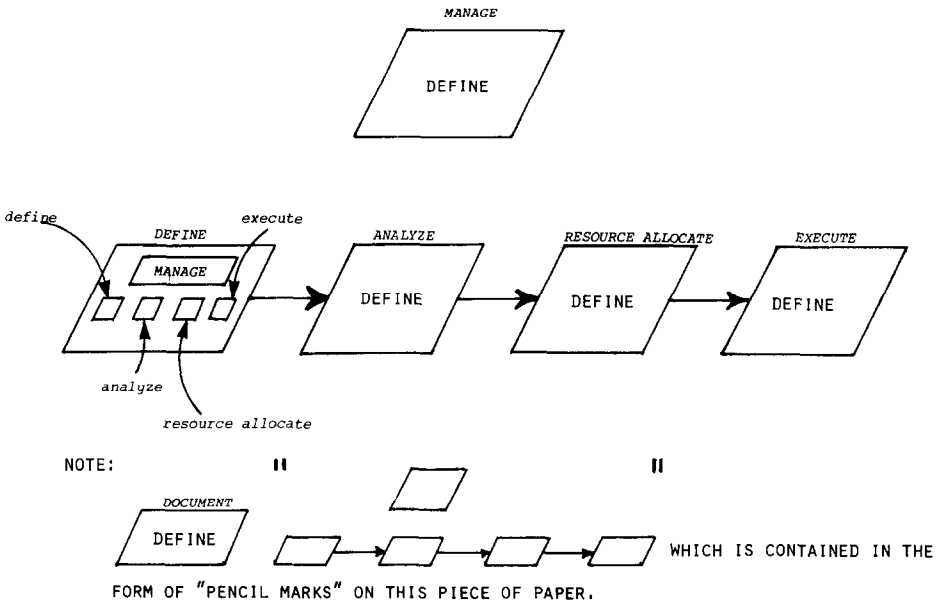
The entire life cycle model, if viewed functionally, has to do with knowing how and when to recognize a problem (or part of a problem) to be solved and then solving it. A major emphasis is placed on being able to tell the difference between a problem and its solutions. This is part of what we refer to as "keeping the layers straight." Understanding each point of view of an object and its relationships goes a long way towards un-

[3]The analysis process involves relating "what if" instances of the target system environment (or the equivalent of having done so) to the target system. An equivalent process might include a guarantee that a set of rules were followed to put together a given system structure. (An analysis process, for example, would ensure that whether or not a bus took off from New Jersey or Boston, it would still arrive at its designated location, that is, if that was a requirement.)

[4]A resource allocation process relates the target system to a machine architecture system (or its equivalent). The machine architecture will then become the place of execution of the target system. It could be a computer, an operating system, or an algorithm within which the target system resides. An assignment of a procedure to memory is an example of a resource allocation process.

[5]The execution process relates instances of the target system environment to a resource allocated target system. The execution process would actively invoke the operation of the system by a user (real or simulated).

Figure 6. Every life cycle process is dependent on viewpoint and in terms of each other: The definition of the life cycle model is made up of definitions in terms of the life cycle model.



derstanding a particular "what" with respect to its "hows" which themselves are "whats" in their own rights. Then the issue of differentiating between a problem and its solution becomes one of relating one definition to another. In the continuing "matching" process, the relationship of each object that was related (where that relationship is itself an object) can then itself be related until a system is finally complete in its development. Relationships are defined in terms of more primitive relationships. Once previously defined relationships are verified, a new relationship, defined in terms of the more primitive ones, need only be verified at its own level of the system definition. In such a way, structure integrity, as the structure evolves, can always be maintained [27].

In the functional life cycle model, the definition is implementation independent and execution independent. These processes *appear* to be sequential (i.e., define, analyze, resource allocate, execute) but they are not necessarily sequential for it is possible, for example, to set up resource allocation and execution of the target system concurrently or provide outputs before inputs when "executing" the model itself.

USE.IT: THE AUTOMATION OF THE FUNCTIONAL LIFE CYCLE MODEL

USE.IT is an automation of the functional model described above. USE.IT is an integrated family of tools for automating a system's life cycle (Fig. 7). That is, there is no need for manual intervention in a system development process once a set of requirements has been stated by a user. Backus alludes to, according to [31], for the future, a functional front-end approach that is inherently computable. USE.IT is currently able to accomplish this task because of the particular set of properties inherent in HOS-based systems [27]. USE.IT can be considered as a "development machine" in that processes in a life cycle, traditionally thought of as manual ones, can now be automated. In fact, the definition process, itself, is supported by automation with USE.IT. That is, USE.IT will develop a system once you tell it what you want to do—correctly, that is. It will not tell you what you want to do. It will, however, help you tell it what you want to do. USE.IT and the systems developed with it are all based on the HOS theory.

The functional life cycle process with USE.IT works as follows.

AXES: Define the Requirements

The first step is to define the requirements. This step is performed with the requirements definition language AXES or with AXES library mechanisms which have themselves been defined with AXES [32,33]. At this time the user, if he wishes, is supported by interactive aids which assist him in defining his requirements either in statement or graphics form. He is reminded, for example, if he states requirements that are inconsistent, incomplete, or redundant. This is especially important because it is at this time that the user is not always sure of what it is he wants to have his system accomplish and AXES assists him in understanding his own requirements. The developer can use the same mechanisms to try to understand the user's requirements. AXES, which is itself based on the HOS theory, provides the user the means to define systems based on the HOS theory. It is this one fact that makes AXES defined systems unique.

Three basic types of mechanisms are used to define systems: data types, functions, and structures. One can define systems with primitive mechanisms and one can define systems with more abstract mechanisms. All abstract mechanisms are ultimately defined in terms of

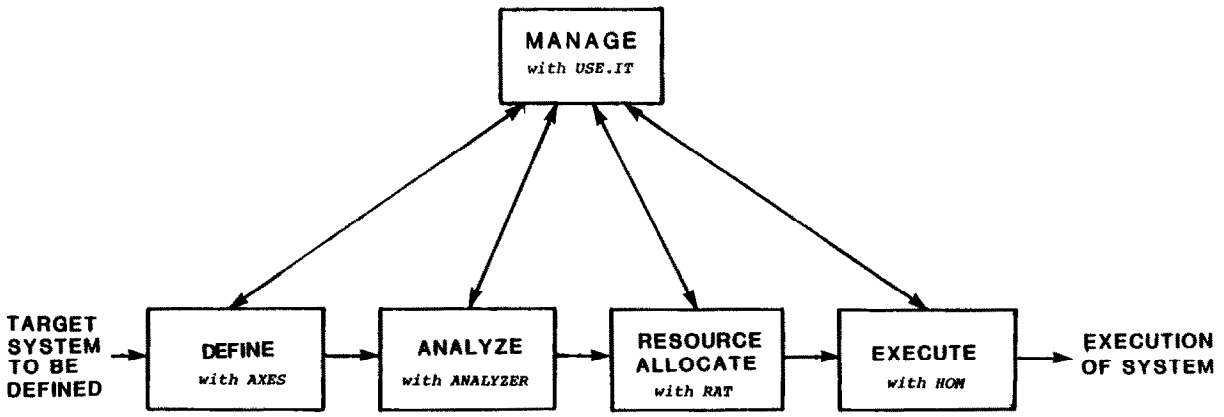


Figure 7. Functional life cycle process with USE.IT.

the primitive mechanisms. A major emphasis of AXES is that it is a language for defining mechanisms for defining systems. Although the mechanisms adher to AXES semantics, the syntax is up to the user. A set of AXES mechanisms forms an AXES library. Use of common mechanisms is obtained by either common use of the same mechanisms or by various derivations from existing mechanisms. Derived mechanisms, in turn, are added to the library for common and standardized use.

There is a hierarchical breakdown of use of mechanisms in the AXES library. That is, any system can be defined in terms of the primitives (Fig. 8). There is also a set of abstract mechanisms, defined in terms of the primitives, that are universal in nature in that any system definition process can be accelerated by the use of these more abstract mechanisms. There are families of systems which share in common a particular set of even more abstract mechanisms than the universal ones. In fact, the very fact that a set of systems share a set of mechanisms, in common, that another set does not share, in common, is one way of distinguishing these two sets of systems. This process determines which systems fall into distinct families of systems or a distinct family or a distinct system. As we define a wide variety of diverse system types with AXES we continue to learn more about "natural" functional divisions in regard to hierarchical families of systems.

Although AXES is a language, AXES is unlike software specification languages. In fact AXES can be used for specifying systems other than software, such as hardware and people systems. Interactive AXES provides decision support mechanisms. If a user, for example, has a syntax or a language that is ambiguous, AXES interactively works with the user in order to make it unambiguous [39]. Thus, it is possible for a user to use his own syntax or existing front-end syntax oriented techniques for user friendly reasons. In doing so, whatever means of communication is employed, the resultant definitions in terms of that communication vehicle will have the same rigor as AXES. In addition, definitions in terms of more than one kind of communication vehicle can be integrated as if they were part of the same system. With this capability, AXES provides a means for diverse users and for users and developers to speak the "same" language.

AXES specifications can also be transformed to other representations by translation of a proper subset of its properties. Automated means can provide projections of an AXES specification in terms of data flow diagrams, priority diagrams, structured design diagrams [40,41], syntax oriented techniques, and other representations such as Higher Order Languages, or machine languages.

Although AXES is a language, it is not a programming language. Not only is AXES a nonprocedural language, but a set of AXES statements allows for many options of implementation both in nonsoftware environments or at the programming stage of development in software environments. That is, from one definition in AXES a system could either reside in, e.g., a distributed or a sequential environment or it could reside in, e.g., an ADA or FORTRAN environment or it could reside directly on various computer architecture environments. It is thus an implementation independent language.

Analyzer: Analyze the Requirements

Once the requirements have been defined with the AXES component, the Analyzer component of USE.IT analyzes the AXES defined requirements [1,42,43]. Once the Analyzer (with interaction from the user if the Analyzer finds a problem) has completed its job, the requirements are consistent and logically complete. The Analyzer insures logical completeness by detecting missing functions or missing data and by guaranteeing that the hierarchical definition stops at primitive operations on algebraically defined data types, insures consistency by enforcing correct interfaces and correct data flow (thus data and timing conflicts are resolved), and integrates system modules by checking across independently developed modules and checking definitions of "library" modules.

RAT: "Program" the Requirements

The next step is performed by the Resource Allocation Tool (RAT) [1,44]. Here, a given analyzed AXES specification is itself treated as data and transformed to another representation. Different representations generated in this manner are referred to as layers of implementation.

Although the RAT produces code automatically, the RAT is not just a code generator. The RAT is an automatic programmer. That is, the RAT reads in unambiguous requirements from any problem domain, received from the Analyzer, and produces source code from those requirements. The code produced by the RAT could be an HOL source code, machine language code or, for that matter, commands to a robot. Boehm [45] referred to Automatic Programming as

> The ultimate in program generation capability, in which a user begins to specify his derived information processing activity to an automatic programming system, which then asks him questions to resolve ambiguities, clarify relationships and converge on a particular program specification. The system then automatically generates a program that implements the specification.

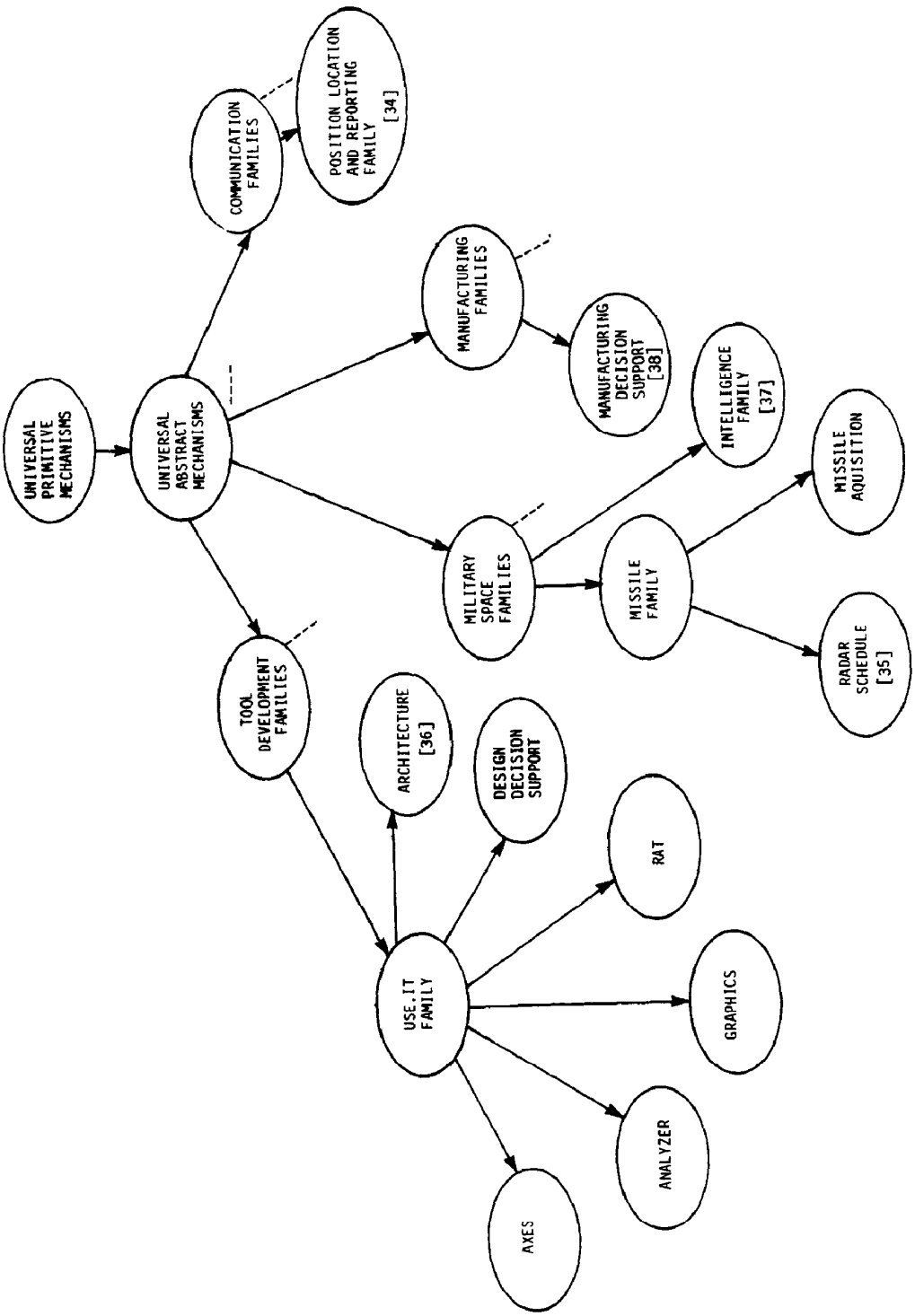


Figure 8. Examples of some families of AXES mechanisms.

He then goes on to say that " . . . automatic programming systems are still somewhat beyond the current frontier of the state-of-the-art." But the reason that USE.IT is able to automatically program is because the Analyzer ensures that the RAT receives unambiguous requirements.

The RAT also provides the end-user with the capability to reconfigure to any language or machine environment desired, whenever desired, without modifying the requirements definition. Since the Analyzer has guaranteed that the requirements used by the RAT are consistent, the automatic programs produced by the RAT are also consistent. Not only are the initial requirements defined by the user guaranteed to be interface error free after the "programming" phase of development, they are also guaranteed to be the *same* ones the user defined.

The RAT generates simulations from control maps involving unimplemented primitive operations, generates efficient implementations from control maps involving implemented primitives, provides an advanced capability for implementing basic primitives in an HOL (e.g., FORTRAN) that far exceeds the capability of that HOL by itself, and permits inclusion of existing HOL packages into the user library as external operations. Whereas in a historical model the programmer goes through a design and code phase manually, in the functional model the developer designs by *choosing* which RAT he wants. The RAT then performs his coding automatically.

The *same* set of requirements that has been "ratted" to one environment (e.g., FORTRAN) can be ratted to another environment (e.g., ADA). This means, for example, that developers who are anxious to start to use the ADA DOD standard language but who do not have the compiler and other support tools yet available can define their requirements in AXES and rat them to FORTRAN or to some other HOL environment until ADA is available. They can then simply rat them to ADA when ADA is ready. It also means that developed systems are never obsolete just because there is a new language or a new computer system introduced within an organization.

HOM: Execute the Program

If "ratting" produces HOL code, compilation, of course, is necessary before execution. If machine code were "ratted," this would not be a necessary step.

The final step in the USE.IT software development process is the execution step itself where a Higher Order Machine, the HOM, is the particular machine configuration that executes the "ratted" requirements.

Documentation with USE.IT

USE.IT is self documenting in that the AXES front end produces a documented hierarchy of the requirements for the user. The analyzer produces documented error messages if there are errors, and documents the fact that there are no errors, if there are no errors. The RAT will produce documented code if asked to do so. Additionally, a plotter, which is one of the USE.IT support tools, will produce documented plotted output of the system requirements if requested to do so.

Management with USE.IT

Management properties are inherent within USE.IT. That is, in a historical model management is something to be contended with, additionally or after the fact; management in the functional model is part of the model, itself. Each function is a manager, both in the target system development and in the target system. Each integration of functions, itself a function, is a higher level manager than the functions it integrates. The more abstract a definition becomes, the less there is for human management to perform with respect to that system [30].

Some management issues, in the use of USE.IT, still remain to be resolved by individual project managers. In our own experiences, we find ourselves resolving issues such as:

What happens if a user(s) submits the same mechanism (semantically) more than once in different syntactical forms?

What happens if the same name is given to two different mechanisms?

When should more abstract mechanisms be constructed for common use when combinations of mechanisms are used or could be used quite frequently?

Should users be forbidden to use combinations of more primitive mechanisms when more abstract mechanisms exist?

Should seemingly arbitrary rules exist such as limitations of length in English descriptions?

Which parts of the syntax should always be standardized? That is, if one person turns in graphics and another English, should both be accepted, neither, or should each person be asked to submit all known syntax forms for each mechanism?

If a better syntactical form is realized, should all existing mechanisms be updated to report the change?

What aspects of mechanism building should be frozen for a given project development?

Which mechanisms or aspects of mechanisms should be

coordinated with other management in the project before they become part of the ongoing language of that project?

What are the various categories and dimensions of categories that should or could exist in the library? Should they, for example, be categorized by project, basic types of mechanisms (e.g., data types), layer of development (e.g., requirement layer vs code layer), type of mechanism with various syntaxes, or by hierarchical breakdown of use (by application family)?

Each issue, however, and its resolution is far more understandable in terms of a functional model than it is in terms of a historical model. And, there are simpler and fewer issues to resolve.

The AXES Library: A Functional Language And A Set Of Management Standards

With the functional model, any set of mechanisms and any set of syntax can be selected or defined provided that it is acceptable to the particular environment where that syntax is to be used. Just as freedom is with respect to any other phenomenon, however, freedom in this case, as well, is something to be respected. That is, if there is freedom with respect to the particular mechanisms chosen, or in the use of syntax for those mechanisms, that freedom should be capitalized on but not misused. The responsibility of such a process, should this freedom be exercised, should be one of project management.

Whatever the case, the very choice of each mechanism and the syntax that goes with it determine not only the language that is to be used on the project for defining the requirements, but, it also determines, to a large extent, the way that the people who are involved on the project relate to each other and how they think, individually or collectively. That is, as new mechanisms and syntaxes are added or deleted, the language used for both the system being developed and by the people working on it, evolves as well. In such a way the sophistication of both the methods used on the project and the people on the project evolves together. There is, therefore, no need to live with an obsolete method of communicating, since the new means have been defined in terms of existing ones; yet the previous work does not have to be thrown out. We thus have a language where the language is what it is, since by doing it becomes. In essence, then, the state of the library is the state of the language; the state of the language is the state of the management standards; the library is the language; the language is the set of management standards.

The Integration of USE.IT

In its final and integrated form, USE.IT provides for an automated life cycle process which eliminates the need for manual intervention. It not only provides a strict separation of the specification of a system from its implementation, but it permits a totally automated implementation of a system from a completely machine-independent specification. A friendly interface with automated decision support is provided by the user friendly package, the system is specified with AXES, the definition produced is checked for consistency by the analyzer, the verified definition is resource allocated by the RAT, and the verified, resource allocated system definition is executed on the HOM. USE.IT is not restricted to a particular language or machine environment or to the type of user friendliness desired at the front end of a development process. The use of USE.IT is not restricted to any application area. The USE.IT components, themselves, are defined in terms of AXES. And, all configurations of USE.IT have the same basic core and standardized units (Fig. 9). Once a set of requirements is defined, USE.IT is able to completely "develop" a system.

The functional life cycle model, upon which USE.IT is based, is a departure from the historical life cycle model. We summarize here differences between the functional model and our own experience on APOLLO [1]. Whereas in a traditional model, the majority of errors found are interface errors, in a functional model there are no interface errors. In a traditional model, these errors are either found manually or by dynamic runs (usually after implementation); in a functional model these errors are found by automatic and static analysis (before implementation). In a traditional model requirements are known for being inconsistent; in a functional model they are guaranteed to be consistent. In a traditional model, programming is manual; in a functional model the programming is automatic. In a traditional model there is no guarantee of maintaining the function integrity of the requirements after implementation; in a functional model there is a guarantee of maintaining function integrity of the requirements after implementation. In a traditional model requirements from different types of users are defined with different types of requirements definition techniques and cannot be integrated; in a functional model, requirements definitions (and thus their implementations) can be integrated. The traditional model is known not to be cost effective. Conservative estimates are that the functional model with USE.IT could cut costs by at least 75% when compared to the traditional model (see section on productivity below).

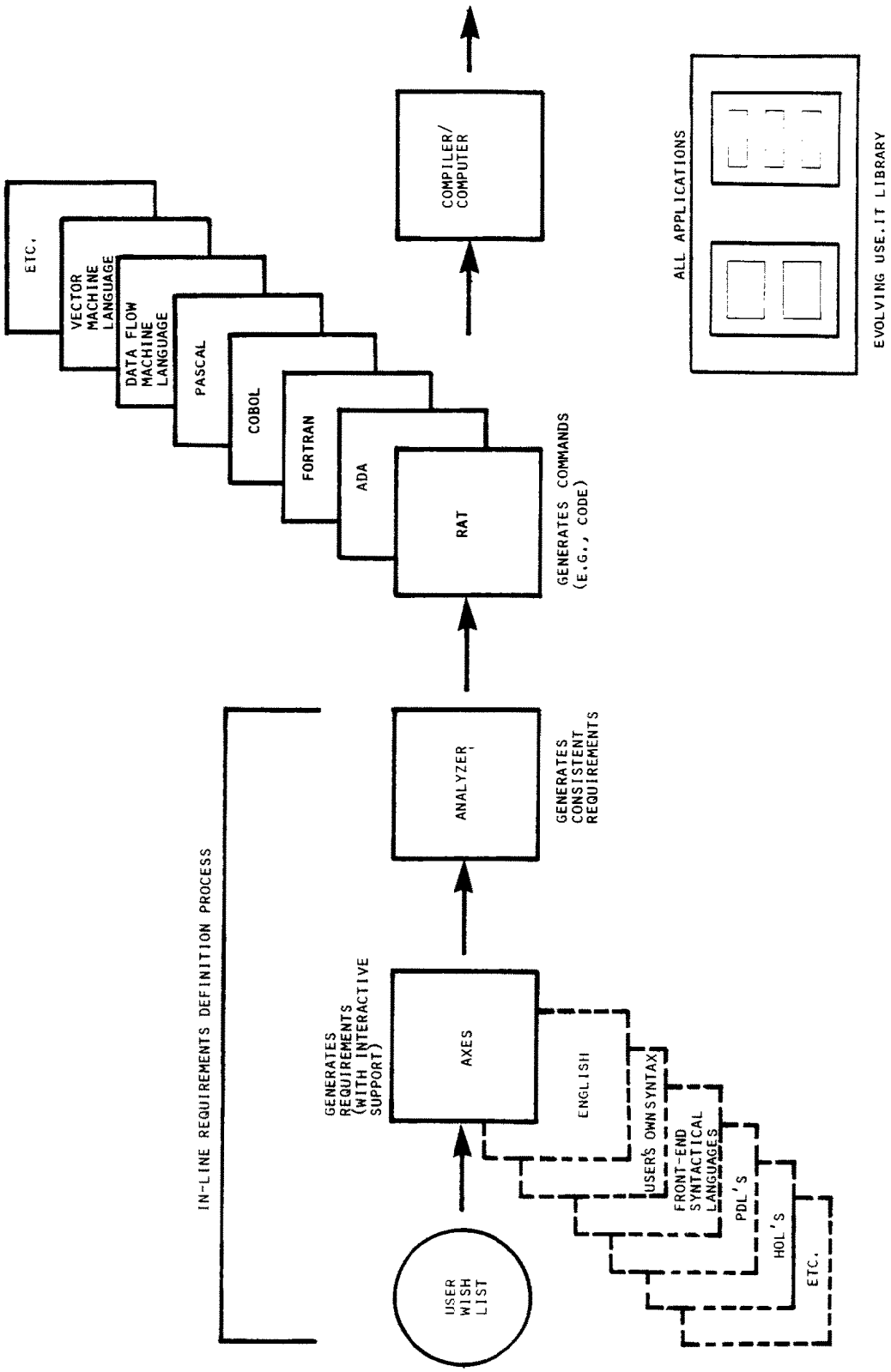


Figure 9. USE.IT scenarios.

The Development of USE.IT

USE.IT was defined with AXES. It was first implemented in PASCAL and FORTRAN by conventional means. Now that this internal bootstraping process has taken place, we are now in the process of redeveloping USE.IT in terms of itself.

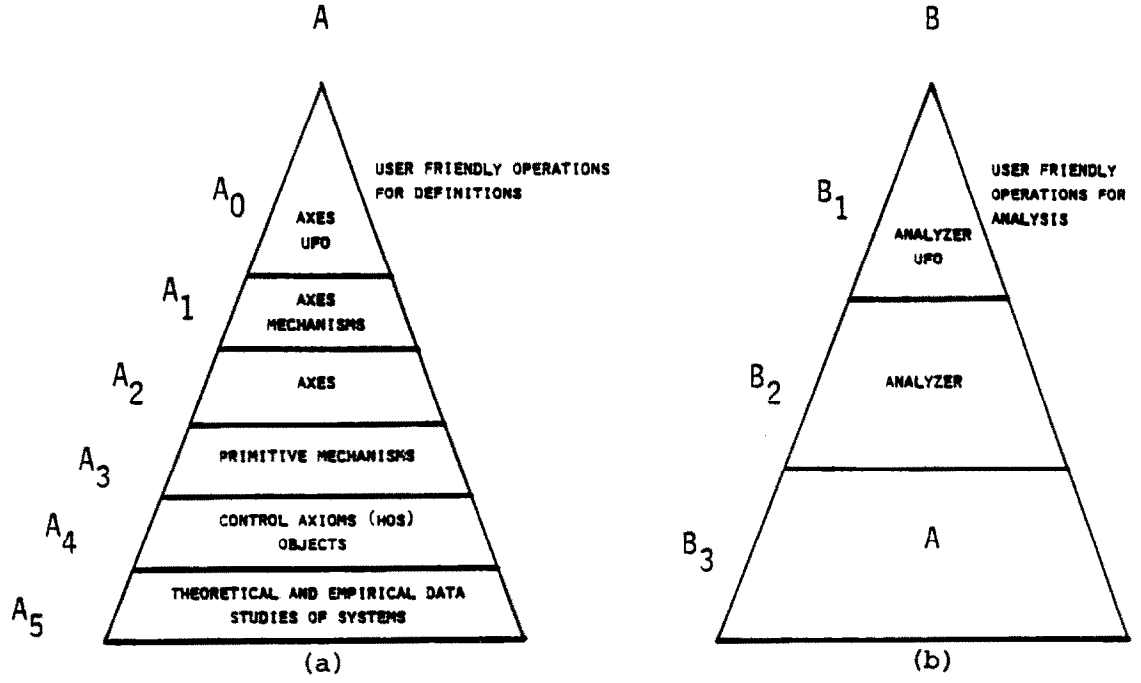
Figure 10 summarizes the HOS life cycle evolvement. Figure 10(a) summarizes the evolvement of the AXES user friendly package for defining a system. Here, empirical data from experiences of developing large systems, existing technologies, theoretical studies, etc., A_5, was used to derive axioms and objects for defining systems A_4 [1]. Primitive mechanisms, in turn, for defining data types, functions, and structures were derived from A_4 and A_5 [2]. Again, based on A_3–A_5, the AXES technique A_2 itself was derived [33]. Similarly, AXES mechanisms were created based upon A_2–A_5 and user friendly operations (UFO) were created based upon A_1–A_5 [2]. The user need only interact with A_0 since the other functions that A_0 is derived from are already an integral part of the automated life cycle process. And as time goes on, the interaction with the AXES UFO will be minimized even further.

Figure 10(b) illustrates the evolvement of the Analyzer user friendly package for analyzing a system definition. Here B_2 is shown to be derived from A_0–A_5, although A, itself, must go through B as a next step. Likewise, the analyzer, itself, is a system which must go through the whole development process on a target system just like its target system. Likewise, it is a subset of the AXES mechanisms, but it needed information of A_2–A_5 to help determine which of those mechanisms were to be related or, in fact, identified. But the AXES mechanisms have to be analyzed. Thus the Analyzer is capable of analyzing itself. (Of course, initially there is a bootstrap process.)

Figure 10(c) illustrates the evolvement of the RAT user friendly package for resource allocating a system definition to a particular machine architecture or set of resources. Here C_1 is shown to be derived from C_2 and C_3. The RAT process, itself, is a system and therefore must go through the process of development just like the target system. That is, the RAT is derived from B which is derived from A_0–A_5, since it is defined using AXES, analyzed using the analyzer, and finally itself "ratted" in order to "rat" other systems. Figure 10(d) illustrates the evolvement of the higher-order machine for execution of a target system [46]. Here, it is shown that the machine itself is both based on and defined, analyzed, and resource allocated with the HOS life cycle model.

Here, then, is a life cycle process, itself a system developed with its own model, where the functions and their relationships are well understood. It is recursive in nature from several viewpoints, where the target system and the machine itself will eventually be developed, derived from and using the same principles (Fig. 11). We can take advantage of this fact in many ways. In the resource allocation process, for example, systems are

Figure 10. HOS life cycle evolvement. (a) The HOS life cycle definition language: AXES. (b) The HOS life cycle analyzer function: Analyze.



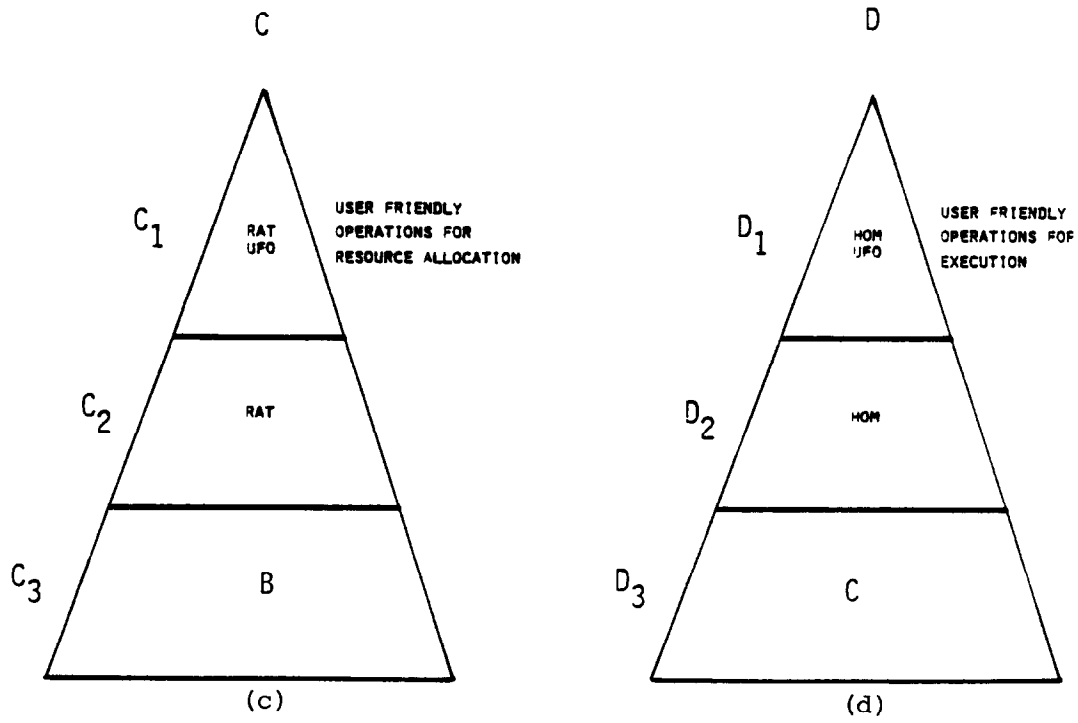


Figure 10. (*continued*) (c) The HOS life cycle resource allocation function: RAT. (d) The HOS life cycle execution function: HOM.

related which have the same generic properties. Thus a complete system and its environment are able to capitalize on the benefits one gains from a functional life cycle model. As another example, evolving machine independent definitions can be useful. One use is operating systems which can be viewed as layers of intermediate machine architectures. Such layers come in handy as "break off" points for machine transferability requirements or constraints [30]. Figure 12 illustrates these concepts within the life cycle model. Here, we show an operating system as an intermediate machine concept where a target system is an instantiation of something to be run by that machine. Similar breakdowns can be defined with algorithms, in general.

THE DEVELOPMENT OF THE LIFE CYCLE MODEL

The life cycle model is a complex real-time large-scale system just like a radar system or a missile system. USE.IT, for example, is one part of the life cycle. Sophisticated concepts, therefore, such as communicating asynchronous and concurrent processes, synchronization, and reconfiguration must be contended with in defining and developing such a system. Examples of structures that are used for these types of phenomenon can be found in [47]. Figure 13 illustrates the use of HOS for defining a small part of a life cycle model. Such a part could exist in, for example, the definition process of the life cycle model. Here we show the definition of a communicating, asynchronous, concurrent structure, the EXCHANGE structure [47]. This structure can be used in any problem domain. We have taken the EXCHANGE and annotated it with life cycle model terms for purposes of demonstration, only.

In this particular illustration the structure is named "Xchevery" [Fig. 13(a)]. Here we have a user and a developer defining requirements. The user and the developer each go through several iterations of the definition process. Both are processing concurrently and asynchronously. Occasionally each is to update his own set of requirements with the other's, but it is up to each individual process when this event takes place. This process continues until Preliminary Design Review (PDR) timing determines that it stop. We show here a "development" of this particular part of the life cycle model, as opposed to developing something with it. (We will show later, in this paper an example of a system developed with USE.IT.)

Since the system Xchevery is specified in terms of AXES it is defined with structures, functions, and members of algebraically defined data types. Some of these units are primitive and others are defined in terms of primitives. For example, in Figure 13(a) the structures used are the primitive structure, JOIN, and the abstract structures COOR and COINCLUDE [2]. The functions are on each node of the hierarchy. Here, for

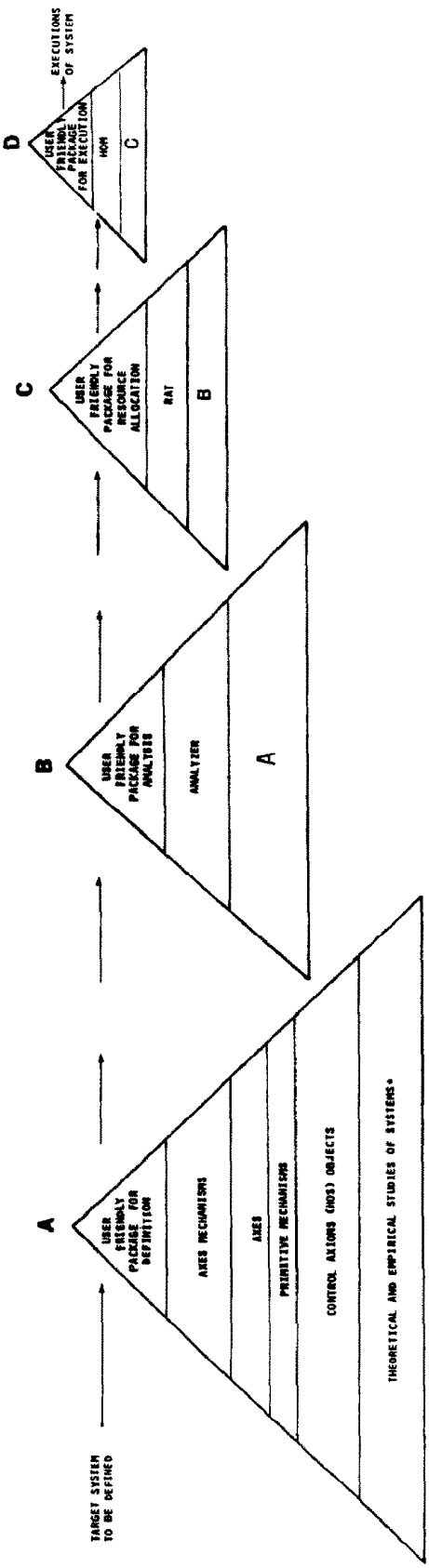
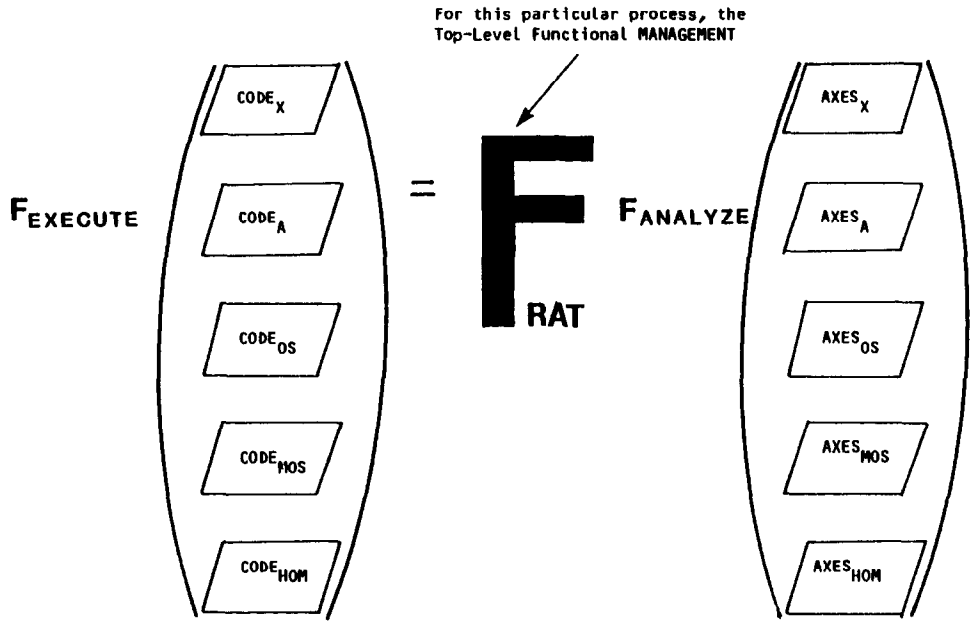


Figure 11. HOS life cycle evolvement.



A BI-DIRECTIONAL FUNCTIONAL LOOK AT THE HOS LIFE CYCLE

NOTE 1: X represents input to the target system
A represents target system
OS represents machine independent operating system
MOS represents machine dependent operating system
HOM represents higher order machine for HOS

NOTE 2: In addition to any target system, OS, MOS, HOM, execute, analyze, RAT and AXES could as well be X or A (being or doing) depending on view point with respect to the development process.

NOTE 3: Each F is a manager

Figure 12. A bidirectional functional look at the HOS life cycle.

example, Clonel is a universal function in that it is applicable to any data type [2]; Xchevery is a recursive function (see third level) in that it is invoked by a function, Update, which is invoked by Xchevery on the top node. The data types in this system are Ordered Sets [2] and Naturals [33]. R, for example, is a member of Ordered Set and n is a member of Natural. Xchevery controls the functions immediately below it with a COOR structure, a structure for making a decision. That means that either Clonel or Update will be performed. If Clonel is performed, the most recent requirements are frozen and the process of defining requirements is complete. If Update is performed, then Update controls the functions immediately below it with a JOIN structure. The JOIN is used for communication of processes. Here, the output of Incorp is received by Xchevery as input for another recursive round of Xchevery. Incorp controls its lower level functions with a COINCLUDE structure, a structure used for parallel processing. Here, both $\text{Decision}_{\text{Developer}}$ and $\text{Decision}_{\text{User}}$ are able to be processed concurrently. Each of these functions make use of the structure, Decision [Fig. 13(b)]. Decision has one variable function F and an operation on Ordered Sets, Integrate, which creates one Ordered Set from two Ordered Sets as inputs. In using DECISION, Xchevery either "plugs" User into F or Developer. Note that although, for example, $\text{Decision}_{\text{Developer}}$ has two inputs, its lower level function, Developer, when plugged into F in Decision can begin as soon as it receives its own input. Likewise this is true with $\text{Decision}_{\text{User}}$. Note that a structure can be used, once defined, as a more abstract entity [Fig. 13(c)]. For more detailed discussion of how to interpret the meaning and use of many of the structured mechanisms used here see [2].

An example of a USE.IT development session with these user requirements is shown in Figure 14. Here the updated requirements to Xchevery are Xchevery itself. To illustrate, further, the "generics" of systems, we have included (Fig. 15) excerpts of a USE.IT session of a system which uses the same Exchange structure as we used (i.e., Xchevery) for the life cycle model for asynchronous and concurrent communication between a pilot, operator, plan, radar, and missile.

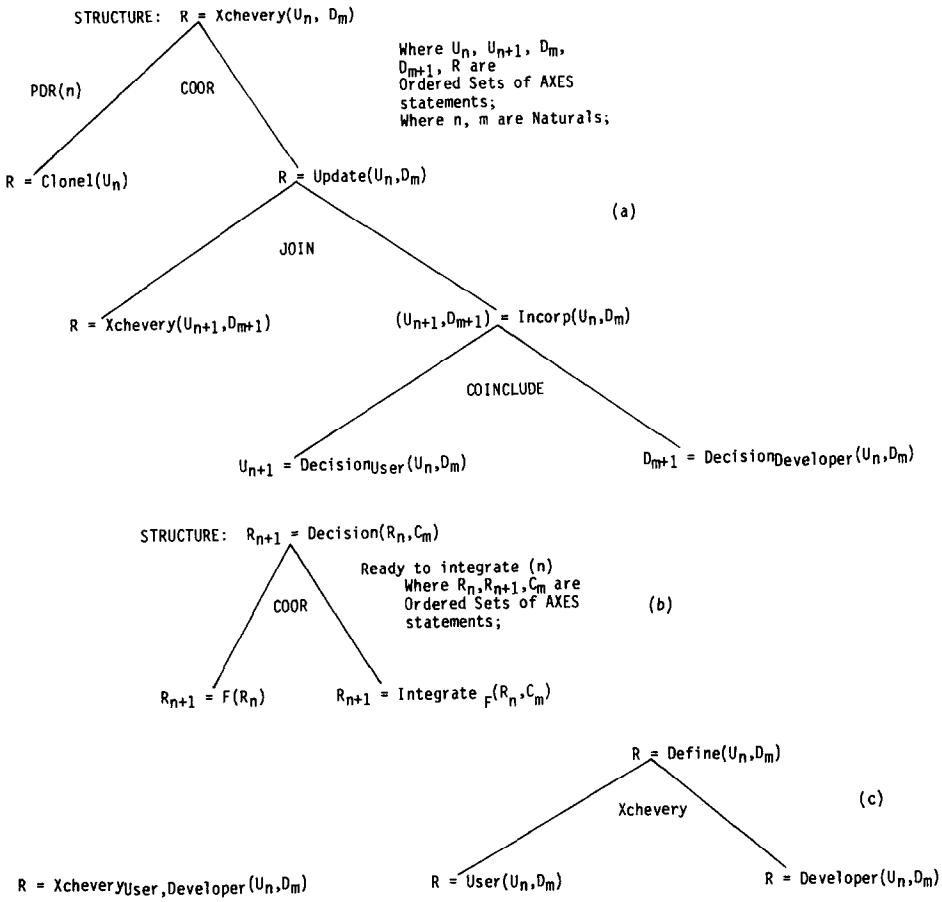
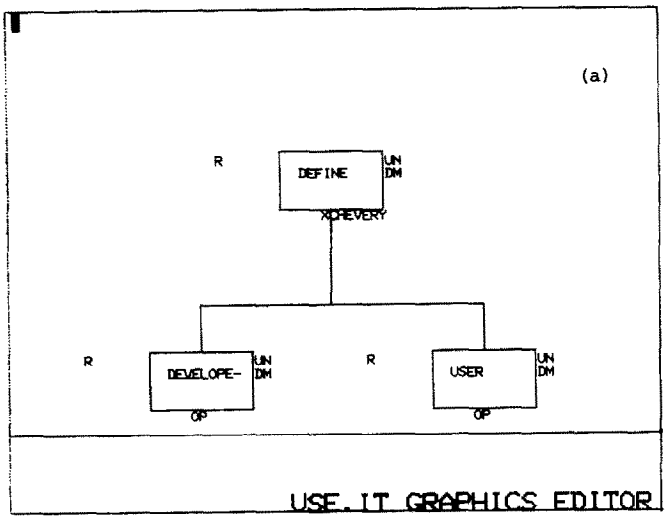


Figure 13. An example of the use of HOS for defining a part of its own life cycle. (a) Structure for defining communicating processes which are synchronous and concurrent. (b) Structure for deciding when to incorporate other requirements into your own. (c) Two syntax options for use of structural Xchevery.

THE TRANSITION FROM THE HISTORICAL MODEL TO THE FUNCTIONAL MODEL

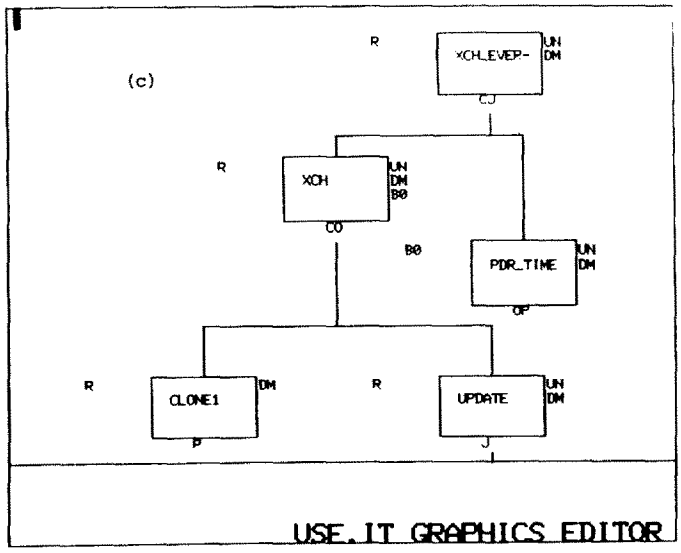
The introduction of the functional life cycle model within an organization which is now using the historical model is not unlike the introduction of computers in the fifties when organizations were using mechanical calculators. The introduction of a functional model, in systems already deployed or far down the line in the development process, is not an easy job. In some cases, in fact, it is almost impossible. There then becomes the unenviable choice of either fixing existing systems or redoing them completely over again. Often, it is much more cost effective in the long run to start over, although such a fact is not obvious until it is too late in that much time and money is wasted in finding this out by attempting to fix a system first. This is often true when requirements for a particular system are in a state of flux. For those systems which have more or less stabilized, or for those systems which are near obsolescence, a complete redo may not be such a wise choice. But for those systems which are near the front end in their development or which are changing almost as often as to behave as a new system, there appears to be no reason why the use of newer techniques would not be a major consideration (see discussion of Table 10).

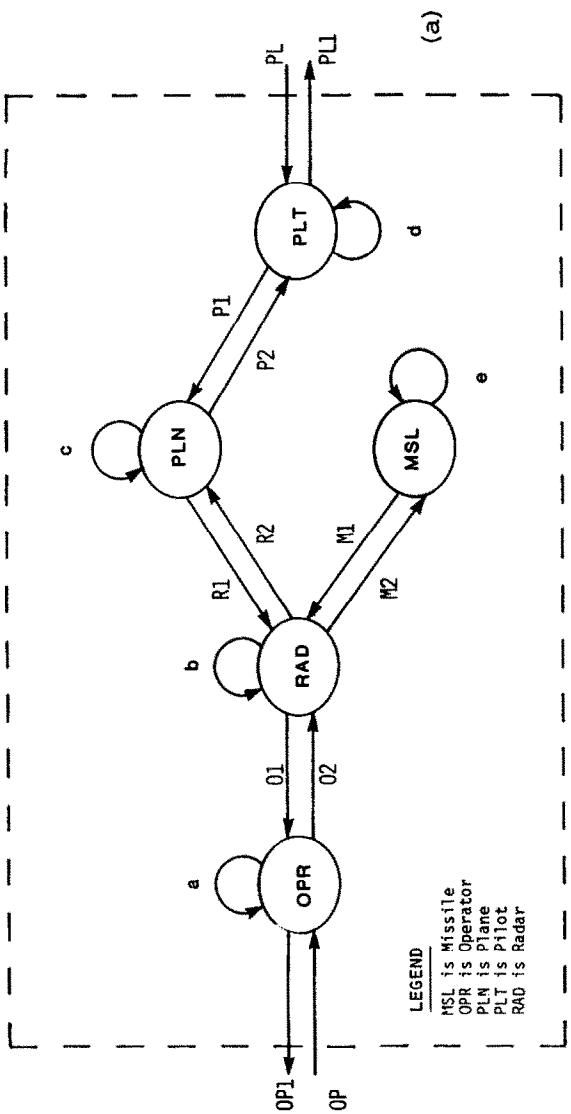
With those systems where a decision has been made not to change over to new techniques there are still some benefits that can be obtained by using the new techniques as a support tool to existing techniques. An attempt can be made, for example, to take an ongoing specification to a parallel prototype one using the new techniques. Such an effort is very effective as a front-end verification and validation tool, since many errors can be uncovered in the translation process. Not only does this process find errors, but it finds errors without running the system dynamically and it finds them *early*. The new converted specification can also be used as a means of understanding the original specification if it is still considered desirable to keep it intact as the primary system. And, eventually if the primary system has had so many changes as to be unwieldy, there is always the back-up system to transfer over to should such a trans-

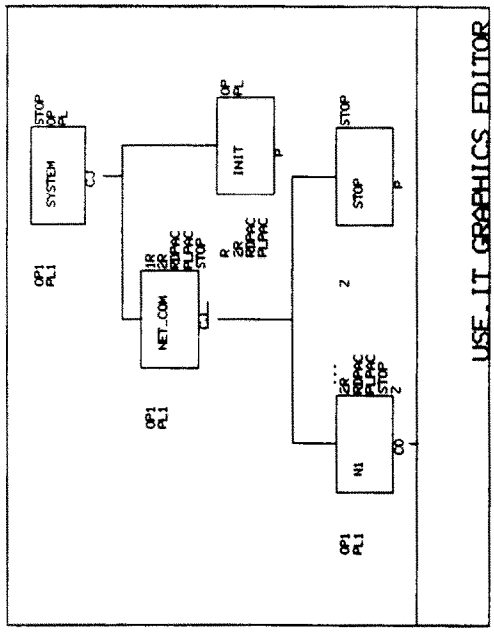



```
C  CODE BEGINS:
       IF('V0002'.EQ.'V0012')GO TO 90010
       J=V0002(1)
       DO 90011 I=1,J
90011  V0012(I)=V0002(I)                    (b)
90010  CONTINUE
930    CONTINUE
       WRITE(6,90020)
90020  FORMAT(' ENTER USER COMMAND:',$)
90031  READ(5,90030)K,(INSTR(I),I=1,80)
90036  FORMAT(Q,80A1)
       IF(K.EQ.0)GOTO 90035
90033  DO 90034 I=1,K
       L=INSTR(I)
       L=L-(256*(L/256))
90034  V0003(K+2-I)=L
90035  V0003(1)=K+1
       ENCODE(32,90041,INSTR)
90041  FORMAT('$')
       DO 90042 I=1,32
       IF(INSTR(I).EQ.' '.OR.INSTR(I).EQ.0)GO TO 90042
       K=I
```

Figure 14. Xchevery development session with USE.IT [49]. (a) The analyzed definition. (b) Excerpts of FORTRAN code (automatically produced by USE.IT). (c) Some results of execution.




(a)
PL
PL1
PLT
d
P1
P2
PLN
c
MSL
e
R1
R2
M1
M2
b
RAD
O1
O2
a
OPR
OP1
OP
LEGEND
MSL is Missile
OPR is Operator
PLN is Plane
PLT is Pilot
RAD is Radar


USE_IT GRAPHICS EDITOR
SYSTEM
INIT
NET_COM
N1
STOP
EXIT
N2
N3
NET_COM
PLN_PLT_SYS
OP_RD_MS_SYS

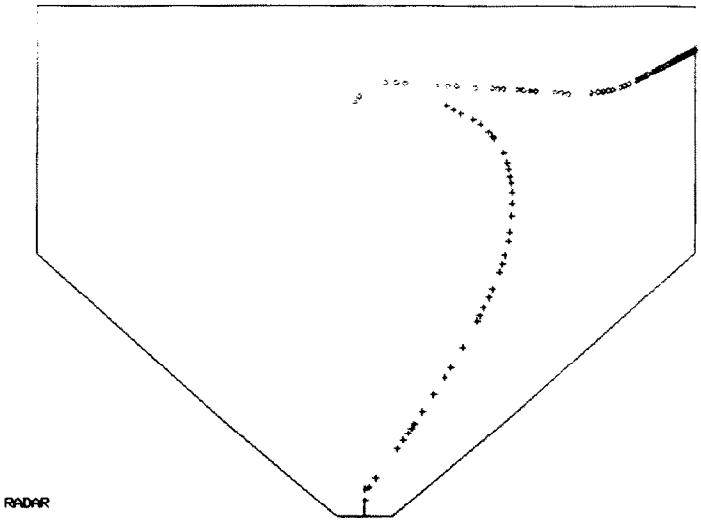
(b)

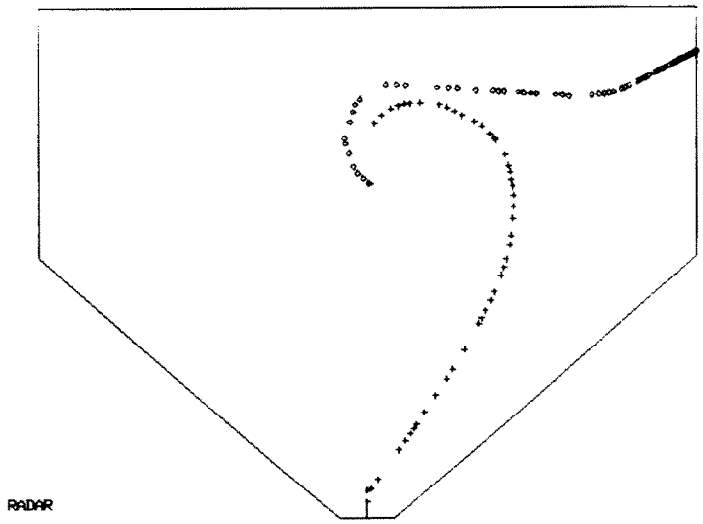
```
C  CODE BEGINS:
      V0003=-1.
      V0001(1)=V0002(1)*V0003
      V0001(2)=V0002(2)*V0003
      V0001(3)=V0002(3)*V0003
      V0001(1)=V0005(1)+V0001(1)
      V0001(2)=V0001(2)+V0005(2)
      V0001(3)=V0001(3)+V0005(3)
      V0006=(V0001(1)*V0001(1)+V0001(2)*V0001(2)+V0001(3)*V0001(3))**0
     1 .5
      V0008=.1
      V0007=0
      IF(V0008.GT.V0006)V0007=1
      IF (V0007.EQ.1)GO TO 291
      V0006=V0010/V0006
      V0001(1)=V0001(1)*V0006
      V0001(2)=V0001(2)*V0006
      V0001(3)=V0001(3)*V0006
      V0012(1)=V0002(1)+V0001(1)
      V0012(2)=V0001(2)+V0002(2)
      V0012(3)=V0001(3)+V0002(3)
      V0013(3)=V0014(3)
      V0013(2)=V0014(2)
      V0013(1)=V0014(1)
```

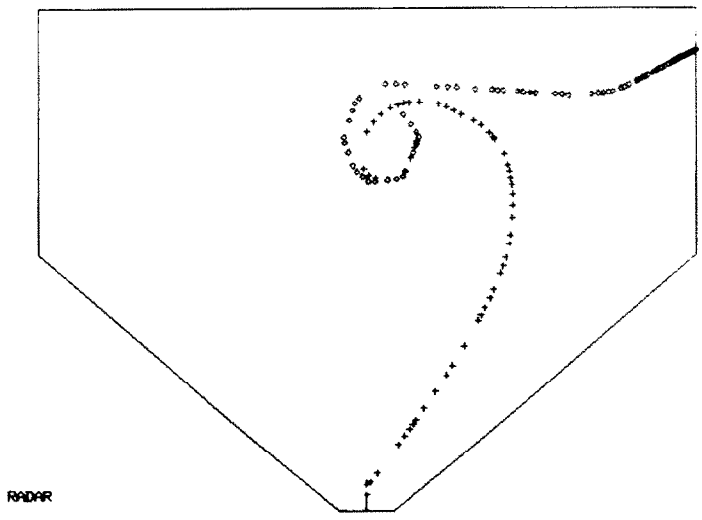
(c)

```
      V0034(1)=V0007(1)
50    CONTINUE
      CALL CONVEC(0.,0.,0.,V0010)
      V0009=1
      IF((V0010(1).NE.V0034(1)).OR.(V0034(2).NE.V0010(2)))V0009=0
      IF(V0010(3).NE.V0034(3))V0009=0
      IF (V0009.EQ.1)GO TO 71
      V0011=V0034(1)
      V0013=5.
      V0012=0
      IF(V0011.EQ.V0013)V0012=1
      IF (V0012.EQ.1)GO TO 121
      CALL CONVEC(1.,0.,0.,V0017)
      CALL RECEIVE(V0033,V0034,V0017,V0014,V0015,V0016)
      CALL LAUNCH(V0015,V0029,V0018)
      IF (V0018.EQ.1)GO TO 181
      CALL OBSERVE(V0015,V0014,V0030,V0031,V0028,V0029,V0032,V0016,V00
     1 20,V0021,V0022,V0023,V0024,V0025)
      GO TO 182
181   CONTINUE
      CALL ACQUIRE(V0014,V0015,V0029,V0028,V0032,V0030,V0031,V0020,V00
     1 16,V0024,V0025,V0021,V0022,V0023)
182   CONTINUE
```

Figure 15. A missile acquisition system taken from Ref. 49. (a) The problem. (b) The top of the definition (automatically analyzed by USE.IT). Invocation only control map and complete control map. (c) Excerpts of FORTRAN code (automatically produced by USE.IT).







(d)

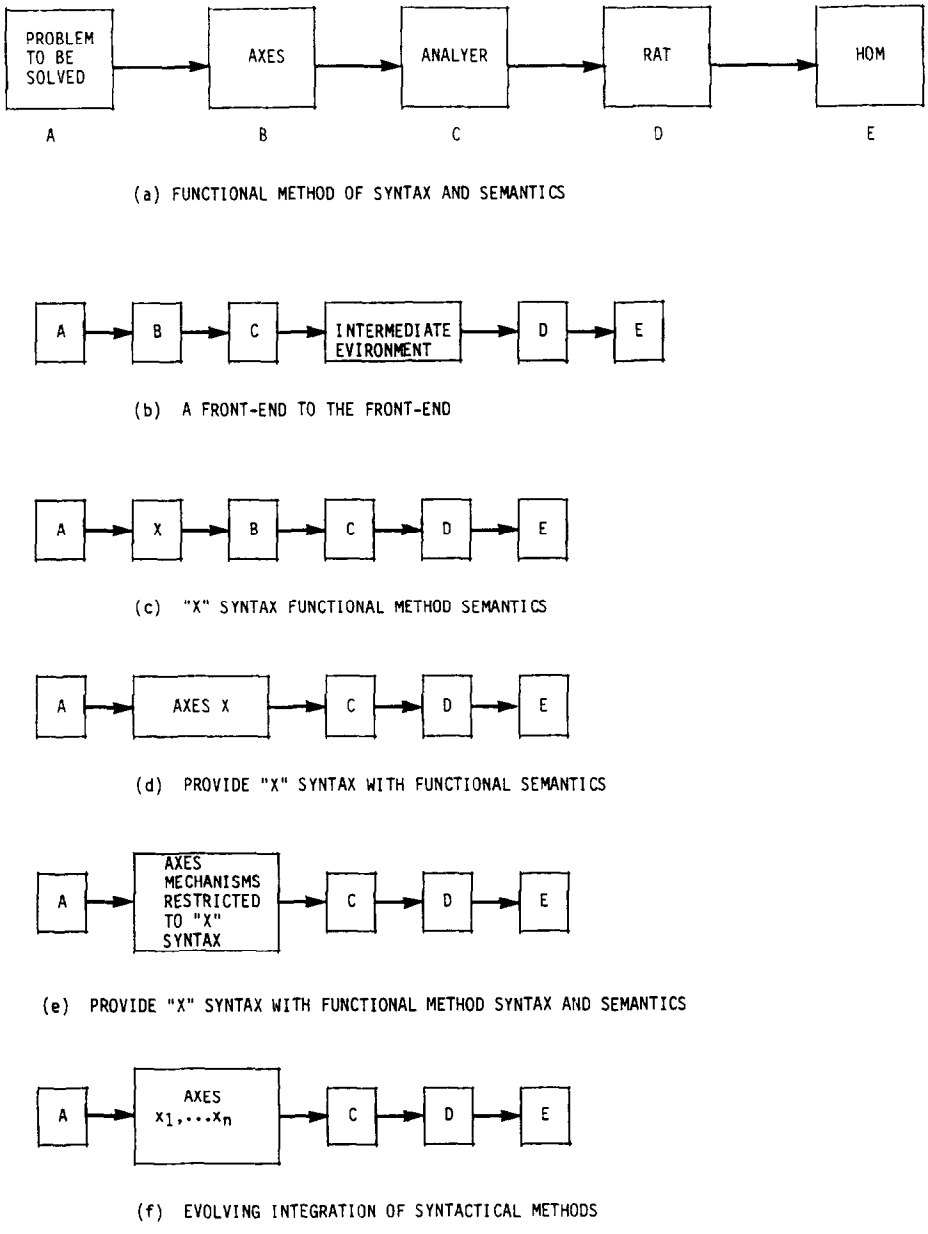
Figure 15 (*continued*) (d) Some results of execution.

fer be considered necessary. We have found that such an exercise can be useful for verification and validation purposes even when a specification has been implemented within a particular machine environment. In this case, several errors can be detected that could be fixed both in the original specification and in its resulting code.

There are several "incremental" methods for starting over with USE.IT. These are applicable to situations where system developers want to start a target system's development from scratch but where there are reasons, political or otherwise, for holding on to an existing tool, such as a language, to use with the functional model. In most cases the easiest and most cost effective method is to use the pure functional model without attempting to complicate it. To "start from scratch" means bringing in others to help develop the system or training existing developers. This initial investment is minor within the overall context of developing a system. A summary follows of the various alternatives, starting with the pure functional one (Fig. 16).

Functional method syntax and semantics approach: [Fig. 16(a)] the most direct since it corresponds directly to the functional life cycle model. In this case the developers use both the syntax and semantic rules of the functional approach [37].

Figure 16. Methods for starting over.



A front-end to the front-end approach: [Fig. 16(b)] the target system is defined with the new syntax and semantics rules of the functional approach. The completed definition is then translated to a definition, which still follows the functional rules, but it is described in terms of a more familiar syntax, so that the developers may still "talk" or "think" in the same "language," syntactically [50].

In this case there is the initial investment of building a translator, and it is necessary to have the definition (and changes to it) provided by those who are familiar with the syntax of the functional method. A potential shortcoming is that the actual physical process of defining a system gives a familiarity not acquired by learning one already provided. Thus the "developers" as opposed to the front-end "definers," in this case, might start with an unnecessary hardship. This approach, however, could appear to the developers to have less of a transient then that initially provided by the "cold turkey" approach.

"X" syntax, functional method semantics approach: [Fig. 16(c)] the developer is still able to use the syntax he is most familiar with, but he must follow the rules with the use of this "X" syntax. There is more than one way of accomplishing this [39]. With the Direct Method, the developer learns the rules and is taught to write system definitions using similar syntax; but he must follow the rules provided to him for use with that syntax. With the Indirect Method, the developer uses the same syntax and he does not necessarily follow the functional semantic rules (just like he always has before). Once the definition has been completed or while he is in the process of defining the system, a user friendly module assists him by asking him enough questions for him to provide answers, thus forcing the definition to follow the same rules as those followed in the direct method. Once the definition is determined to be a good one by the decision support system, the definition is converted to the syntax of the functional method.

Provide "X" syntax with functional semantics approach: [Fig. 16(d)] defines a compiler for the "X" syntax defined with the functional method and replaces the old compiler [51]. There is training here in that the functional compiler will provide a new set of meanings to the "X language".

Provide "X" syntax with functional method syntax and semantics approach: [Fig. 16(e)] the "X" syntax is maintained by forcing the "new" mechanisms defined with the functional method syntax and semantics to be as familiar as possible by corresponding them to the "X" language. This method, however, is being short changed, if newer, more abstract mechanisms are not also constructed. These new mechanisms add new syntax to the "language".

Evolving integration of syntactical methods approach: [Fig. 16(f)] the developers could choose to speak in the language desired (i.e., the old or the new). An algorithm would then be constructed to combine the functional syntax with the "X" syntax(es).

RESULTS

It is no coincidence that the HOS life cycle model is a functional one. The process of determining properties necessary for a functional model has been an interactive one with that of defining the HOS model. This process has not taken place overnight. It began when we were involved in all aspects of development of a very complex and large-scale real-time avionics system. At the time we were only interested in which things we should do differently for the next applications or in which things we should keep on doing since we could not find any reason to want to change them. The first analysis led to a beginning towards understanding generic properties of a system [1] and later towards understanding the generic properties of a system for the development of a system [27].

We began by attempting to understand the properties of our own software system and its development. We extended this analysis to one of understanding a larger system within which the software resided. The result was a theory for defining systems. At this time we had only a set of definitions and axioms with which to define systems. A definitional process was time consuming since each definitional step required theorem proving exercises. Although we were successful in the application of the theory, in that it worked in providing unambiguous definitions, we found that using these formal methods, directly, to define each level of a hierarchical definition was a very user unfriendly approach to take—in fact, so unfriendly that we were the only ones to use the theory for some time.To remedy this situation, we attempted to communicate, more effectively, the procedures that we went through to others. To do so required a next step of analysis. It did not take long to realize that as we defined systems, certain common patterns would occur over and over again. Each of these patterns, a proof in itself, was then adopted for future systems as a standard for definition. The first mechanisms to evolve were the primitive mechanisms (primitive structures, functions, and data types) for defining

a system [1,2]. Another result was the demonstrated ability to integrate these mechanisms which consisted of both generation and behavior types of definitions (i.e., structures and functions with data types). We then proceeded to build more abstract mechanisms in terms of the primitive ones. With these mechanisms we had a more effective means of conveying to others how one goes about defining an HOS-based system [1]. We found, however, that our procedures were still not user friendly enough for most system designers and that there was only a handful of us who proceeded to use our theory in its early stages. We discovered, however, at this early time that the properties of these systems were very different than those of conventionally defined systems. In particular they served as a unique vehicle for automation. We also knew that few would benefit from them if we could not convey these properties even more effectively to others. It was at this time that we decided to accelerate the automation of the methodology, both as an end in itself, and as a means of conveying to others what was possible with the theory. First we defined a conceptual life cycle model for the Army [29] and a requirements definition language for the Navy [33]. For the Army we defined the practical aspects of our theory for a new life cycle approach. For the Navy we attempted to make possible the use of that approach for a large class of users by developing the language AXES.

Aside from a stepup in user friendliness, the key considerations which were introduced at the time we defined AXES were those of variable syntax, extensibility for all types of mechanisms, abstraction, ability to integrate data types, structures and functions at any level of abstraction, and the ability to define all mechanisms in terms of the HOS primitives.

Once the AXES component was developed, there were numerous experiences in applying the HOS theory as a definitional method. Earlier efforts were without automated aids (see discussion of some of these efforts in [2,27]). Some of these experiences were with software, only, systems, some were systems of which software was a part, and some were systems without software. Others were systems for developing systems. In these earlier efforts HOS was used only as a means to define systems in such a way as to be free of ambiguities. Not only did this provide a way of helping one user to convey the meaning of his part of a system to another user, but it also helped a user understand the meaning of his own part of the system, especially in terms of the larger part within which it resided.

We suspected that unambiguous requirements would make the developer's life a lot easier; this in turn would make the user's life a lot easier. Our suspicions proved out when our staff defined a radar system in AXES and turned it over to a different organization to develop the system [35]. On this project, we demonstrated, and verified more completely than before, that programming, indeed, could be almost a one-for-one process from requirements if requirements were defined in an unambiguous state.

The first tool to be automated was the Analyzer. The second tool to be automated was the RAT. The advent of its automation completed the automation of the life cycle model, USE.IT, for its first complete configuration. The RAT (together with AXES and the Analyzer) has now been demonstrated within large system applications.

Our first application demonstration of a complete USE.IT configuration, in its prototype state, and outside our own organization, was in the manufacturing environment. Here we demonstrated the feasibility of an automated life cycle concept to a system which consisted of an interactive human operator within a decision support environment of hardware and software subsystems [38].

Another application, the ASAS system, consisted of a demonstration of a module which was a part of a battlefield intelligence system environment [37]. Our staff defined a set of requirements taken from existing documentation. This documentation consisted of a mixture of English, equations, and SREM. The requirements defined a module which would result in approximately 10,000 lines of code if they were resource allocated to all software. A major part of this module was defined with AXES and analyzed with the Analyzer, removing all inconsistencies in this part of the requirements. For demonstration purposes, several hundred lines of FORTRAN code were produced automatically, and executed from software portions of the system requirements.

Several organizations, other than our own customers, for whom we performed experiments or developed prototypes have had some experience with the AXES and the Analyzer components. We are at this time actively in the process of installing complete configurations of USE.IT, which have in addition, the RAT, in the government, university, and commercial environments. Although at this early date we can not report any extensive statistics in the use of USE.IT by anyone other than ourselves, we have had some preliminary experiences with customers, to date, which we believe would be of interest in that they do, we believe, give a hint as to the ramifications of the use of an automated functional life cycle model for future applications, particularly the large and complex ones.

One commercial organization, for example, in the communications environment gave us user's require-

ments for what we will call a real-time CLOCK problem. Our staff defined the user's requirements in AXES and developed it with USE.IT as the user observed the process. The results of this problem are documented in [52].

A member of another organization decided to take this same problem and write the program for it in order that he could compare USE.IT's productivity to his own. The resources used by our staff were 4 man hours to define the problem. USE.IT was used to "develop" it. The resources used by the manual method to develop it were 3 man days where the programmer claims to be a 5000-lines-a-month programmer.

A third organization gave our staff a nontrivial real-time asynchronous communicating concurrent processing radar system problem. The results of this problem are documented in [49]. Some of the results of the life cycle process of this problem are illustrated in Figure 15. Our staff spent a total of 24 man-hours to define the problem. USE.IT was used to "develop" it, resulting in 800–1000 lines of FORTRAN code, varying with changes as requested by the user. If we compare the time with USE.IT to an average programmer's time (by DOD standards) such a programmer would have spent 80 man-days for just the implementation part of this example.

Still another organization gave us a problem related to manufacturing of buildings. Our staff defined and developed this application in 11 man days. It resulted in approximately 10,000 lines of FORTRAN code [53]. Our customer estimated that by conventional standards it would have taken them approximately two years to do the same job. Other more recent experiences have been documented such as [54,55].

It became apparent that some organizations and applications would not be able to take advantage of USE.IT unless there was an effective means to hook up the front end of their environments to the front end of USE.IT and the back end of their environments to the back end of USE.IT. Our initial attempts with the front end were with PSL/PSA, SREM, and IDEF. Here we were able to show that users could "speak their own language", but yet gain the rigor that is required for unambiguous communication and thus the automatic programming of USE.IT. Our initial attempts with the back end involved a research version of the RAT which automatically programs in APL, PASCAL, and LISP. They were followed by the production version of the RAT which now automatically programs in FORTRAN and PASCAL. Other more commercially oriented organizations who are involved with very large data bases are often dependent upon existing data bases as well as with data base handling mechanisms. We are now in the process of working with these types of organizations by helping them to "plug in" to USE.IT. This involves the interfacing of USE.IT defined systems with data base management systems as external operations in the USE.IT system. Methods for defining a data base with USE.IT are discussed in [56].

We have also worked with intermediate stages of development by attempting to put the rigor of AXES into higher order languages. For, example partial semantics of ADA was defined in terms of AXES [56]. This sets the stage for an ADA RAT as well.

The HOS model has been applied to several different types of systems at various levels of its own life cycle model development. Not only did these experiences help us to understand properties of systems, including systems of developing systems, users who develop systems, and tools for developing systems, they also helped us to enhance the methods that were being used to understand these properties.

The next step is to apply USE.IT to a system which is comparable in magnitude and complexity to the ones from which it was "derived" [1]. For it is here that we believe its true virtues will be made known.

PRODUCTIVITY AND USE.IT

It is now not unusual in today's computerized society to spend millions of dollars for a single software project. One recent interview with a relatively small insurance company has indicated present expenditures of 40 million dollars per year on software development. Further, this company is discovering that the projected number of software programmers required to fulfill their needs in the near future is just not available.

Initial cost savings with USE.IT can be attributed to automatic detection of a large class of errors, to automatic programming and to a design technique that accelerates the design process.

Table 8, a typical breakdown of classical life cycle

Table 8. Classical Life Cycle Costs (derived from [57])

Phase	Activity		
	Design	Analysis	Total
Initial Development	10	15	25%
Requirements & Design (R + D)	5	5	10
Programming (P)	5	[a]	5
Verification + Validation (V + V)	[a]	10	10
Maintenance	27	48	75%
Residual errors		7.5	7.5
R + D	13.5	13.5	27
P	13.5	[a]	13.5
V + V	[a]	27	27
Total development	37	63	100

[a]Analysis cost data for programming phase and design cost data for V + V phase included in analysis cost data for V + V phase (assumption based on available data).

costs for a large software project, was derived data from [57].

This particular set of classical statistics [57] separates the software life cycle into four phases where the requirements and design, programming, and verification and validation phases are associated with initial development and maintenance begins when the system is "operational". Each phase has associated with it two activities: design and analysis. Costs are associated with each of these two activities for each phase. (Note that activities within a phase and the phases themselves sometimes overlap, a problem often encountered in classical life cycle models.) With respect to maintenance costs, we make the assumption here that maintenance is a reiteration of the initial development process in addition to fixing residual errors.

Consider, now, the impact of USE.IT with respect to Table 8 (like life cycle costs).

With the USE.IT tool, the design activity of the programming phase, which accounts for 18.5% of software costs, would be done automatically. In addition, where the analysis activity of all phases of present software projects accounts for approximately 63% of software costs, the USE.IT tool would automatically perform 75% of that function. Finally, we must not overlook that most of the time the user, or customer, of software products does not get what is really needed, because with traditional methods there is no way to communicate those needs adequately between the user and developer. This part of the development process, referred to as the design activity during the Requirements and Design phase, accounts for another 18.5% of software expenditures. With USE.IT, unambiguous definition and rapid prototyping are part of the process, resulting in an estimated minimum savings of 50%, while ensuring the customer gets what is wanted.

One isolated example, with an insurance company, of USE.IT cost savings estimates is presented as a simplified summary in Table 9. Assuming these estimates, should the insurance company change over to functional techniques? According to Table 10, the most difficult decision point would occur when the insurance company has just reached the 30 million dollars expenditure point.

Table 9. Estimated Life Cycle Cost Savings With USE.IT[a]

	Historical costs	Costs with USE.IT
R + D design @ 18.5%	$ 7,400,000	$ 3,700,000
Programming Design @ 18.5%	7,400,000	0
Analysis (all phases) @ 63%	25,200,000	6,300,000
Total costs	$40,000,000	$10,000,000

[a]Case example of annual savings to a small insurance company with present annual software development costs of 40 million dollars.

Last, but not least, USE.IT provides a means to achieve "reuseable" software. New software projects today have no means to use parts of old systems to build new ones because the old parts are embedded in the old system so as to depend on other parts. Because cost implications of "reuseable" software to the development of new systems has not been exploited to date, a minimum cost savings of 75% can be projected over a project life cycle. Recent small samplings of actual USE.IT applications (see Results) have shown greater productivity then estimated above (Table 11).

SUMMARY

The theory exists. The technique for using it exists. Its automation exists. The AXES library will continue to evolve. We are currently concentrating on providing education for developers and future developers which focuses on a different way of thinking about systems. A major step in this direction has also been taken by Martin in his forthcoming book [58].

A most influential rationale to use the functional life cycle model, to organizations, is that which has to do with cost savings. Careful analysis should be performed in the area of cost savings in order to fully realize its impact. There are a lot of claims being bandied about in the area of productivity today, especially since productivity is such a popular subject. It is important to make sure that statistics are being used as they should be—with care. If for example, a development of a system can be accomplished with a new method in $\frac{1}{10}$ of the time it would have been developed with an older method, one gains by being able to accomplish the job with 90% savings. This, however, is an *increase* in productivity in 1000%. If, however, one takes a slice of the development process and compares the time of an automated tool which takes, say, 1 minute to produce source code to 1 year of a programmer's time, the increase in productivity could be viewed as 525,600%!

With the historical life cycle model, there is both obsolescence in the tools and techniques that are being used to develop systems and in the algorithms that are being developed with, or because of, these tools and techniques. How many times have design techniques been used in particular applications only because one was forced into them before and they are now ingrained habits? Is it really necessary, for example, in the software engineering field to design data base management systems, operating systems, or avionics systems as they are designed today? Many areas of research or follow-

Table 10. Trade-offs for Starting Over with USE.IT

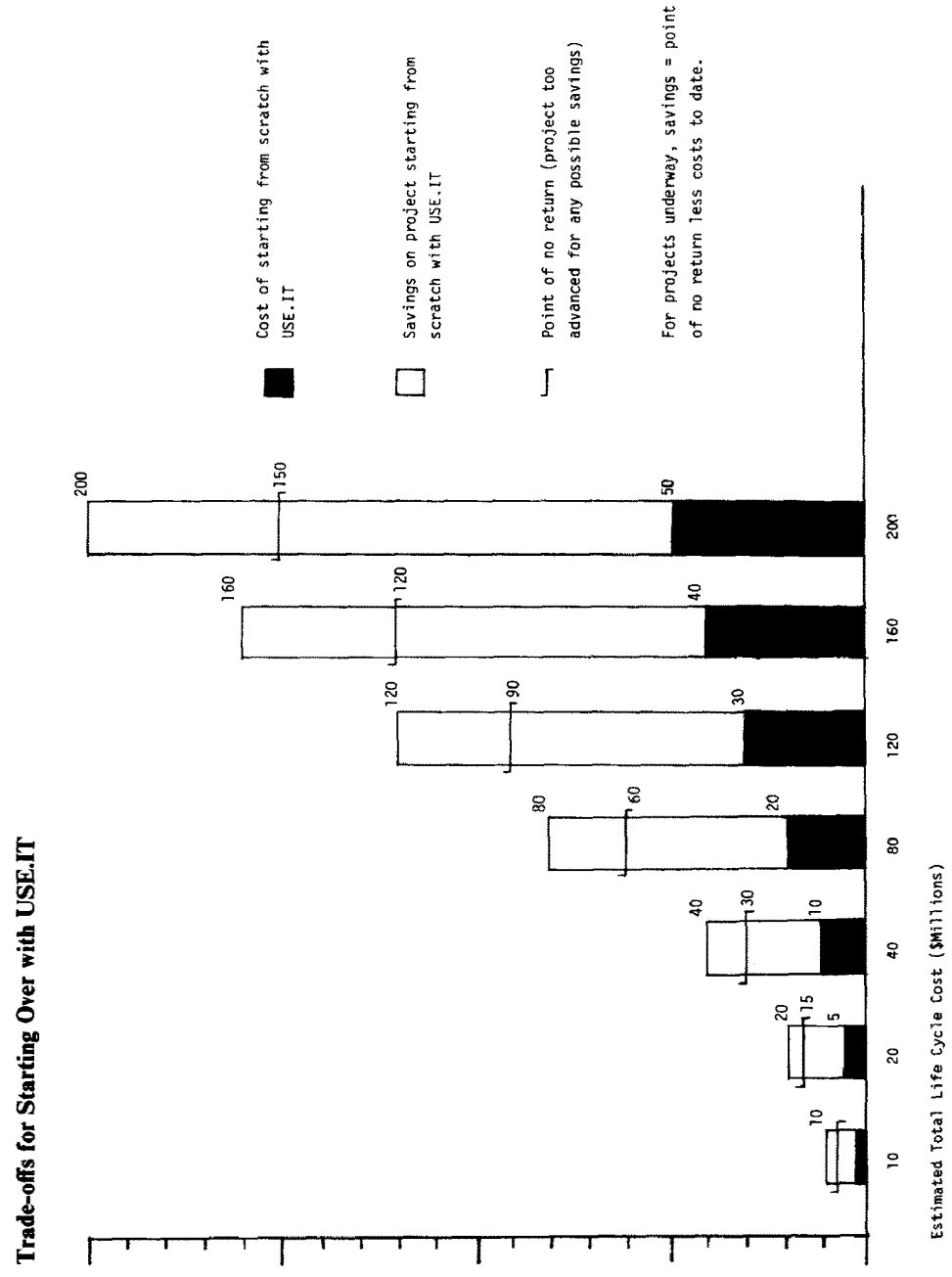


Table 11. Actual Life Cycle Cost Savings with USE.IT

	Life cycle manpower			
	Example 1	Example 2	Example 3	Example 4
With USE.IT	4 hr[a]	3 days[a]	2 days[a]	11 days[a]
Without USE.IT	24 hr[a]	80 days[b]	60 days[b]	1,000 days[b]
Productivity increase	600%	2,700%	3,000%	9,091%
Cost savings	83%	96%	96%	99%

[a]Actual.

[b]Estimate using DOD standard of 10 lines of code produced by an average programmer per day.

on to research relating to the historical model are pursued that may themselves not be necessary. Is extensive work in areas such as *special* techniques for each of the areas of concurrent processing, asynchronous processing, proof of correctness, or theorem proving necessary? Or should there be a different focus in each of these areas than there is today? We suspect that, given a clean slate as a basic foundation, that one of the first areas of research should be to find out where such obsolescence truly exists. Such obsolescence, however, cannot be found unless those who look for it know what they are looking for.

REFERENCES

1. M. Hamilton and S. Zeldin, Higher Order Software—A Methodology for Defining Software, *IEEE Trans.* SE-2, (1976).
2. M. Hamilton and S. Zeldin, The Relationship Between Design and Verification, *J. Systems Software* 1, 29–56 (1979).
3. D. Ross, Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Trans. Software Engineering* SE-3, 16–34 (1977).
4. D. Teichroew and E. A. Hershey III, PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, *IEEE Trans. Software Engineering* SE-3, 16–34 (1977).
5. C. G. Davis and C. R. Vick, The Software Development System, *Proc. 2nd International Conference on Software Engineering,* October 1976, Addendum pp. 27–43.
6. J. D. Warnier, *Logical Construction on Programs,* Van Nostrand Reinhold Company, New York, 1974.
7. L. Robinson et al., Proof Techniques for Hierarchically Structured Programs, *Comm. ACM,* 20, 271–283 (1977).
8. D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, in *Tutorial on Software Design Techniques Second Edition,* edited by Peter Freeman and Anthony I. Wasserman, IEEE Computer Society, IEEE Catalogue Number 76CH1145-2C, 1977, pp. 131–136.
9. E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Yourdon Press, New York, 1978.
10. R. A. Snowdon, An Experience-Based Assessment of Development Systems, in *Software Development Tools* (W. E. Riddle and R. E. Fairley, eds.), Springer, Heidelberg, 1980, pp. 64–75.
11. J. Martin, *Application Development Without Programmers,* Prentice Hall, Englewood, NJ, 1982.
12. FOCUS, Information Builders, Inc.
13. Nomad, National CSS, Inc., Wilton, CT.
14. Ramis II, Mathematica, Inc., Princeton Junction, NY.
15. Application Development Facility, IBM, Armonk, NY
16. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press, Cambridge, MA, 1977.
17. J. de Bakker, *Mathematical Theory of Program Correctness,* Prentice-Hall, Englewood Cliffs, NJ, 1980.
18. W. E. Howden, DISSECT—A Symbolic Evaluation and Program Testing System, in *Tutorial: Automated Tools for Software Engineering,* IEEE Catalog Number EHO 150-3, Library of Congress Catalog Number 79-91320, IEEE Computer Society, NY, 1979, pp. 207–210.
19. J. C. King, Symbolic Execution and Programming Testing, *Commun. Ass. Comput. Mach.* 19, 385–394, (1976).
20. L. A. Clarke, A System To Generate Test Data and Symbolically Execute Programs, in *Tutorial: Automated Tools for Software Engineering,* IEEE Catalog Number EHO 150-3, Library of Congress Catalog Number 79-91320, IEEE Computer Society, NY, 1979, pp. 211–218.
21. D. J. Reiffer and R. L. Ettenger, Test Tools: Are They a Cure-All?, TR-0075(5112)-5, The Aerospace Corporation of America, El Segundo, CA, 1974.
22. E. F. Miller, *Methodology for Comprehensive Software Testing,* AD-A 01311, Available from NTIS, (1975).
23. L. Druffel, "The Need for a Programming Discipline to Support the ADA* Programming Support Environment," IFIP Working Conference on Automated Tools for Information System Design and Development, (1982).
24. Attendees of SIGSOFT First Software Engineering Symposium on Tool and Methodology Evaluation, "Proposals for Tool and Methodology Evaluation Experiments," Software Engineering Notes, ACM, (1982).
25. B. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.
26. M. Hamilton and S. Zeldin, Properties of User Requirements, in *Formal Models and Practical Tools for Information Systems Design* (H. J. Schneider, ed.), North Holland, Amsterdam, 1979.
27. M. Hamilton and S. Zeldin, "Requirements Definition within Acquisition and Its Relationship to Post-Deployment Software Support (PDSS)," TR-22, Higher Order Software, Inc., Cambridge, MA (1979).
28. M. Hamilton and S. Zeldin, "A Functional Approach to the Life Cycle Model: Towards a Development Support

System for DOT," Technical Report No. 31 prepared for SDC Integrated Services, Higher Order Software, Inc., Cambridge, MA (1981).
29. M. Hamilton and S. Zeldin, "Integrated Software Development System/Higher Order Software Conceptual Description," Higher Order Software, Inc., Cambridge, MA (1976).
30. M. Hamilton and S. Zeldin, "The Manager as an Abstract Systems Engineer," Digest of *Papers,* Fall COMPCON 77, Washington, DC, IEEE Computer Society Cat. No. 77CH1258-3C (1977).
31. W. Schatz, "Rebel With a Cause," Datamation, (1981).
32. 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, MA (1976).
33. M. Hamilton and S. Zeldin, "AXES Syntax Description," TR-4, Higher Order Software, Inc., Cambridge, MA (1976).
34. Higher Order Software, Inc., "The Application of HOS to PLRS," TR-12, Cambridge, MA (1977).
35. R. Hackler, An AXES Specification of a Radar Scheduler, *Proceedings, Fourteenth Hawaii International Conference on System Sciences* 1 (1981).
36. A. Razdow, Introduction to the Application of HOS to Hardware Design: The CORDIC Algorithm, *Proceedings, Institute for Defense Analyses, Summer Study on Hardware Description Languages,* National Academy of Sciences, Woods Hole Study Center, Woods Hole, MA (1981).
37. R. Hackler, "An HOS View of ASAS," Technical Report No. 32, Higher Order Software, Inc. (1981).
38. "Integrated Decision Support System (IDSS) Functional Requirements and Preliminary Design," Technical Report No. 30 prepared for the U.S. Air Force Materials Laboratory, Wright-Patterson Air Force Base, Higher Order Software, Inc., (1981).
39. Higher Order Software, Inc. "Computable IDEF: A Major Step Towards Increasing Productivity," Cambridge, MA (1981).
40. M. Hamilton and S. Zeldin, "Top-down/bottom-up, Structured Programming and Program Structuring," Charles Stark Draper Laboratory, Cambridge, MA, Rev. 1, Doc. E-2728 (1972).
41. 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.
42. J. Rosenbaum and C. Early, "FAME: Front-End Analysis and Modeling Environment," *Proceedings,* NBS/IEEE/ACM Software Tool Fair sponsored by the National Bureau of Standards, San Diego, CA (1981).
43. R. Hackler and A. Razdow, Analyzer project, Higher Order Software, Inc., Cambridge, MA (1981).
44. A. Razdow and R. Hackler, APL RAT project, Higher Order Software, Inc., Cambridge, MA (1981).
45. B. Boehm, Keeping a Lid on Software Costs, *Computerworld* (1982).
46. W. Heath, *A Higher Order Machine (HOM) for Higher Order Software (HOS)* in "Techniques for Operating System Machines," TR-7, Higher Order Software, Inc., Cambridge, MA (1977).
47. M. Hamilton, "The ADA Environment As A System," *Proceedings of the ADA Environment Workshop,* sponsored by DoD High Order Language Working Group, Harbor Island, San Diego, California, (1976).
48. Prepared by R. Hackler, HOS, Inc. (1982).
49. R. Hackler, A. Razdow, and B. Wright, "An Example of Asynchronous and Concurrent Communication with USE.IT: A Missile Acquisition System," Educational Series No. 10, Higher Order Software, Inc. (1982).
50. J. Rosenbaum, FAME interface to PSL/PSA project, Higher Order Software, Inc. (1980).
51. R. Smaby, "Specifying ADA Semantics in HOS," TR-34, Higher Order Software, Inc., Cambridge, MA (1982).
52. Higher Order Software, Inc., "Annotated Model of a Digital Clock," Educational Series No. 1, Cambridge, MA (1982).
53. Higher Order Software, Inc., "USE.IT for Building Buildings," Educational Series No. 2, Cambridge, MA (1982).
54. A. Razdow and G. Goates, "Using AXES as a Hardware Description Language," Educational Series No. 8, Higher Order Software, Inc., Cambridge, MA (1982).
55. R. Hackler, "An HOS Simulation of a TCAC-Like System," Educational Series No. 9, Higher Order Software, Inc., Cambridge, MA (1982).
56. R. Hackler and R. Smaby, "Information Modeling in HOS," Higher Order Software, Inc. Educational Series No. 11, Cambridge, MA (1982).
57. D. K. Lloyd and M. Lipow, *Reliability: Management Methods and Mathematics,* Prentic-Hall, Englewood Cliffs, NJ, 1962.
58. J. Martin, *Program Design Which is Provably Correct,* Savant Research Studies, England, 1982.