A SOFTWARE ENGINEERING MANAGEMENT SYSTEM FOR FLIGHT SIMULATORS

John A. Schepp
The Singer Company
Link Flight Simulation Division
Binghamton, New York 13902

SUMMARY

The complexity of modern simulators has overwhelmed the capacity of conventional approaches to maintain effective cognizance and configuration control. The problem is intensified by the emergence of system engineering techniques which stress functional analysis, requirements allocation, and traceability of design features to requirements. A Software Engineering Management System (SEMS) is described which uses the technology of a relational data base to overcome previous limitations on visibility of software structure and design. Its principles are applicable to the much broader but similar problems of overall simulator development and life cycle management. The principal components of SEMS are:

a) A comprehensive project data base capable of providing multiple user-oriented project views.

b) A suite of processors and protocols to allow production of project-germane information and documentation.

c) Supervisory and monitoring capability for the integration of pre-existing software development facilities into the system.

d) Links to interface geographically separated sites into a unified project control system.

The conceptual development and organization of the system are described and examples of its operation are provided.

Introduction.

The Software Engineering Management System (SEMS) described in this paper had its genesis in a study funded several years ago by a large manufacturer of flight simulators. That effort, called the MicroSimulation Technology (MST) Study, was intended to identify problems which might arise as the industry converted from centralized computer complexes to distributed computation based on microprocessors. Its major finding was that the distributed architecture in itself was quite compatible with the natural organization of flight simulator software; the problems anticipated were essentially the same problems which had caused inefficiency, lost time, and sometimes almost chaos in conventional flight simulator software development. The investigation had been intended to provide simply a go/no-go signal for the conversion to microcomputation, with perhaps recommendations for modest tool development activities. Instead, it became the foundation for a fundamental revision of software engineering concepts, practices and environment within that contractor's facilities.

Furthermore, it is a major theme of this paper that the SEMS, originally conceived and generated for the limited task of simulator software management, contained within itself all the elements required for solution of the far broader and far more complex problem of the management of the total simulator throughout its life cycle, beginning with the proposal stage and continuing throughout its useful career. The same information storage and accessing structures which were found necessary for the original task of software management and configuration control were re-examined and found to be adaptable, with practically no structural change and only a small addition in complexity, to the manifold operations of hardware design, fabrication and test, spares and logistics support, cost and schedule control, etc.

In this paper we shall attempt to show the common problem core which underlies and justifies the expansion of the original system concept to this much broader application. We shall illustrate by specific cases the flexibility with which SEMS addresses the various facets of this common problem.

Scope Of Problem.

Few visitors to the site of a modern large-scale simulator leave unimpressed by the complexity of the equipment they have observed in operation. Even the technically naive quickly observe that the equipment they are viewing contains within itself most of the complexity associated with a highly sophisticated aircraft crammed with every manner of electronic sensing, signal processing, and avionics equipment in the arsenal of modern weaponry. Upon this has been superimposed, first, a complete representation of an entire world of external environment, including not only the physical surroundings, but the perceivable effects of multiple friendly and hostile activities and, second, an elaborate training system whose purpose is to stimulate, observe, react to, and record the trainee's responses during missions which may last for many hours.

The more sophisticated observer, who recognizes the dependence of the operation upon information processing, i.e., computation, may perhaps mentally estimate the millions of instructions per second being carried out to drive the system. His technical background will lead him to check these observations against the amount and type of hardware which he finds within the computer complex. He will be able to verify that he is indeed observing in operation a real-time computation system as elaborate as any to be found in a single isolated functional operation.

However, only the observer who is familiar with the world of flight training simulators will penetrate beyond the reality of the equipment he sees in its full operational form to the events and activities which occurred over prior years in order to bring the simulator into existence. It is this observer who will appreciate that, unlike installations approaching the simulator in complexity which may be in use in other industries, this installation is barely, if at all, one stage beyond prototype, that most of the equipment had to operate properly on the first trial, that its design was begun with missing and incorrect data, that the specifications for its use were incomplete in

defining the design and were probably subject to change after a series of reviews, that these reviews were sometimes conducted by a customer still attempting to reconcile conflicting factions within his own ranks, and that major decisions regarding the use of and selection of important subcontractors were required and executed in the course of the pre-delivery contract years. This last, knowledgeable, observer will hardly need to be reminded that the schedule time allowable for the effort turned out to be inadequate, and that the original cost estimate may have appeared to be naive as the contract unfolded.

The Approach To A Solution.

All of the above factors were well-known to the team of engineers who conducted the MST study. Their recommendations contained three major points. When considered together, these points amount to a restatement of the principle that a very complex and almost ungraspable problem can often best be solved by dividing it into an appropriate number of appropriately related tasks, each of which is in itself amenable to disciplined and effective effort. The three recommendations, only slightly paraphrased, were as follows:

1) Hierarchical Structure. A logical structure must be imposed upon the simulator design, *regardless of how many elements might* eventually be included in that design. For this purpose, a functional tree was proposed, to be organized along strictly hierarchical principles. The hierarchical organization, in which each node of the tree communicates only upward to a parent node or downward to one or more subsidiary child nodes, was selected as the most effective means of maintaining clean interfaces between the functions allocated to each element of the tree.

2) Source files. Each node in the hierarchical tree was to have associated with it a source file which would contain all the information relevant to the hardware/software product associated with that node. These files would contain not only source code, but "...*everything needed to use, interface,* document, test and control that source code...", and all such information would be represented once and only once within the hierarchical tree.

3) Message system. A message system was recommended as the means for transferring data between software modules, with the goal of establishing stable, valid, and coherent computation regardless of changes in simulator load allocations or the occurrence of sequential computations in different computing centers within the simulator. Rigorous scheduling was recommended as the means by which changes in configuration or in computer load allocations could be prevented from introducing unforeseen and sometimes *catastrophic side effects, in either the mini-*computer or the multi-microcomputer environment.

These recommendations became the basis for the initial attempt to design an effective software engineering management system.

Evolution Of SEMS Concept.

The MST recommendations were heavily oriented toward the problem of making a simulator "work". They identified steps to be taken in order to bring the software content of the simulator into a rational form which would permit efficient design, debug, and performance verification. Above all, they emphasized the need for each software engineer to work on his tasks within a well-defined and stable environment. It was thought that the source file and hierarchical structure would lead to the clearest definition of the functions to be performed by each software module. This same structure would provide the basis by which clean interfaces could be defined to establish reliable boundaries within which each engineer could solve his problem. The message system would provide the means by which the stability of these boundaries would be guaranteed regardless of changes in module allocation within the computer complex, timing of frame cycles, etc.

These same principles are directly applicable to the problem of software configuration management. The hierarchical organization of software in itself provides the basis for such management since, by definition, it includes every component of the total simulator software computation load. Furthermore, the interface definitions and message-passing technique provide the means by which the impact of changes made to any software module on other parts of the computation system can be quickly identified. When these are combined with the recommendation for single source files as the repository for all information pertinent to any node of the software hierarchy, the applicability of these concepts as the basis for an effective configuration management system becomes evident. If a way could be devised by which the source files could be guaranteed to contain the appropriate material, and if this information could be kept current with the actual status of the simulator, then the problem of documenting and tracking configuration changes throughout a single simulator's life could be brought under control. Equally important, just as source files and computer loads could change serially in time as a given simulator evolved, multiple implementations could be conceived and maintained in concurrency with a multiplicity of similar but non-identical simulators at various training installations.

Interpretation of MST Recommendations.

The MST study was convincing in its case histories and in its analysis of the simulator design problem, and its recommendations were fully in accord with recognized sound engineering practices. It was considered imperative that they be put into practice. However, early attempts to implement these recommendations disclosed a huge gap between their fundamental simplicity and the complexities inherent in the simulator environment. The bridging of this gap required the conceptual development which is the basis of the SEMS configuration.

For the purposes of this paper, we will treat the third MST recommendation -- the use of a message system technique to control transactions between computing units - as a problem in scheduling, *accompanied perhaps by special* hardware considerations. This is not a trivial problem, and if carried out thoroughly will in itself make a significant contribution to proper distribution of software among computing facilities, and eliminate many mysteries which presently plague simulator debug, checkout, and modification activities. However, its implementation is outside the scope of the SEMS.

Let us turn now to the recommendations at the very core of SEMS: a functional hierarchical

structure for the entire simulator, and a single source file for each node of the hierarchy, containing all information relevant to that node.

Hierarchical Structure(s) If we are to organize the simulator -- and especially the simulator software -- as a hierarchy, it is necessary to ask, "Upon what basis?", and the answer turns out to be, "It depends on what you want to use it for." Although this paper is principally about software and its attendant problems of development and management, we may find some conclusions about software hierarchies less surprising if we first look at the principle as it applies to general simulator design.

The upper part of Figure 1 shows a sequence of stages through which any simulator must pass. Of course, each of these activities may be done well or poorly. In the past they have not generally been carried out with the clear definition shown in the figure. But the logic of the progression is indisputable, so we may well ask why this logical sequence has not been a standard practice in simulation. The answer suggested in this paper is that standard data manipulation techniques have been incapable of effectively translating the results of each design stage so that they could serve as useful inputs to the succeeding stages. Whenever identification between successive design phases has even been attempted, the mechanism has been inadequate. First, it has failed to correlate the findings, decisions, or designs established in one phase with the information needed in succeeding phases. Second, it has not been possible to maintain anything like concurrency of working information available to the various stages. It is these failures which SEMS had to address in order to promote and support the logical sequence of the figure.

Now let us concentrate on hierarchies, starting at the input to the system process in Figure 1. By its formal structure, a Prime Item Development Specification, for example, is automatically a hierarchy. Its numbered paragraphs will provide the form for the hierarchical tree regardless of its contents. So, in turn, are the MIL-Specs, Contractor Standards, the Statement of Work, and other typical input documents. Aircraft Data and Mission and Training Data are likely to be collections of parallel data packages.

The first task in a systems approach consists of the evaluation of the requirements expressed in these defining documents and their translation into a set of functions which must be supported by the system. This process produces a functional hierarchy; the tree structure reflects the progressively finer decomposition of functions until their implementation is both apparent and assignable to a specific element of the system being synthesized. To make this possible, we must generate a system hierarchy in parallel with the generation of the functional hierarchy. This system hierarchy will not be simply an overlay on the functional hierarchy with different captions in the boxes. For example, suppose that our system synthesis creates a node whose purpose is to interface, say, a 1553 avionics bus with a simulator computer. Thereafter we will assign to that node all relevant bus communication resulting from trainee and simulator behavior identified in many parts of the functional hierarchy. Note that by this process, as well as by the interpretation of the defining documents to form the functional description of the system, we experience a loss of cohesion, or congruence, between the structures of the input documents and the organized hierarchies of the functional and system trees.

This same loss of cohesion occurs in the next stage, in which the system tree is decomposed further. For the sake of simplicity, we discuss only two further decomposition trees, devoted respectively to the simulator hardware hierarchy and the software hierarchy. In practice, the complete system will

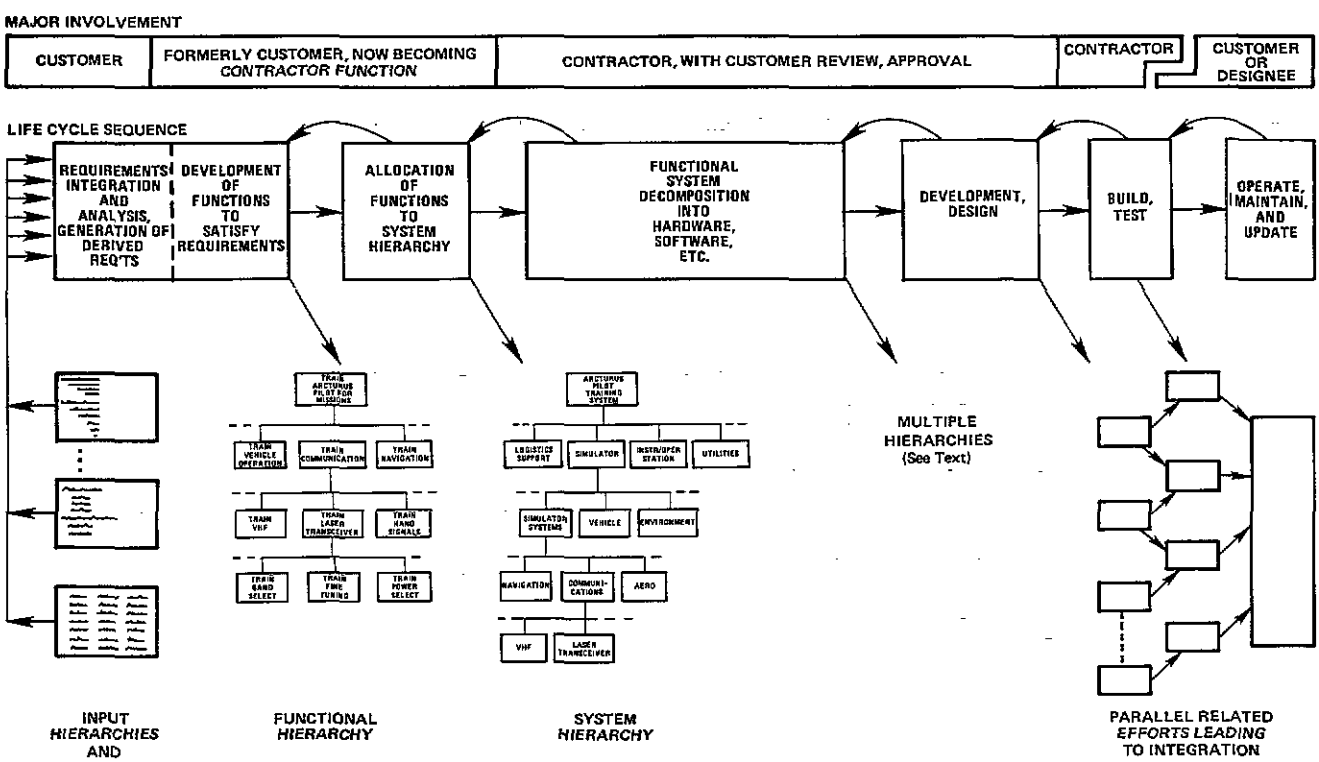


Figure 1. Simulator Life Cycle Stages

include many other aspects of the total contract, including logistics, installation, documentation, training, etc. However, even limiting ourselves to the two trees named, we again face the problems we saw in the creation of the system hierarchy (and these problems would appear in any of the other trees which must be generated in an actual simulator contract). In general, neither the software nor hardware hierarchies will map one-to-one with the system hierarchy.

This is most obviously seen in the case of hardware (Fig. 2). We realize intuitively (more accurately, by knowledge of its use) that the hardware hierarchy, if it is to be meaningful, must be based upon a packaging concept which has been designed to permit rational procurement, kitting, fabrication, assembly, test, maintenance, etc., and must promote the efficiency of these operations without an excessive penalty in hardware component cost. In Fig. 2 the dotted boxes show some of the hardware elements in which a single system element, such as "Laser Transceiver", would appear. Like it or not, the hardware hierarchy must support and document the simulator package concept, and would require a complex network to map to the functional system tree. It is a separate and unique hierarchical structure.

But does the same thing needs to be said about the software tree? After all, the software is there to execute functions that implement the simulation requirements and these functions have been logically arranged in the system hierarchy. If the software tree is not homologous with the system tree (allowing null software functions if necessary), where have we gone wrong? The answer, of course, is in the use of information within the system hierarchy. Any number of functions within that hierarchy may be users of a single piece of information generated within the software tree, and we are not so foolish as to require that such information be generated redundantly at many places within the computing system, just so that the software hierarchy can be a faithful shadow of its functional system counterpart. Instead, we

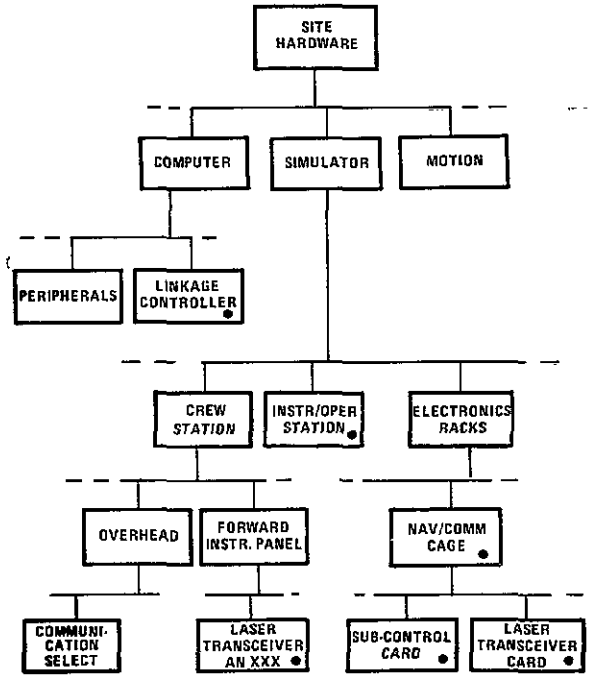


Figure 2. Hardware Hierarchy

indicate the need for appropriate software inputs (specifying their ranges and accuracies) in the system tree, and leave to the software design the problem of organizing the computing functions for optimum speed and efficiency. A software tree derived from a functionally-oriented system hierarchy will mimic the structure of that hierarchy more closely than the hardware tree, but in general will not be homologous with it (Fig. 3).

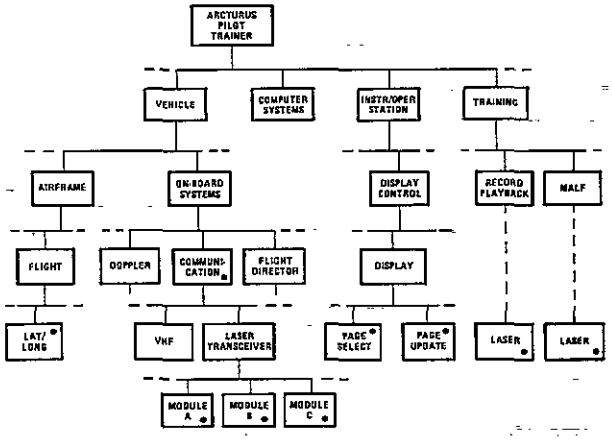


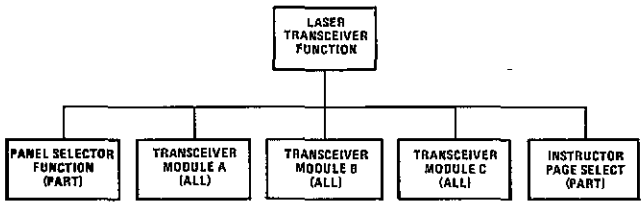
Figure 3. Software Hierarchy

Once it becomes clear that the software tree does not emerge automatically from the painstakingly constructed (and all-important) functional system hierarchy, we must find a rational basis for its construction. It may as well be stated here that many a simulator has been developed with a software "tree" consisting of a top level identified as the Computer Program Configuration Item, a bottom level consisting of, say, 3,000 software modules, and nothing in between of semantic significance. It is suggested that this may have been a "default solution" arrived at because of inability to choose between more useful, but seemingly incompatible, software structures. For example (see Figure 4):

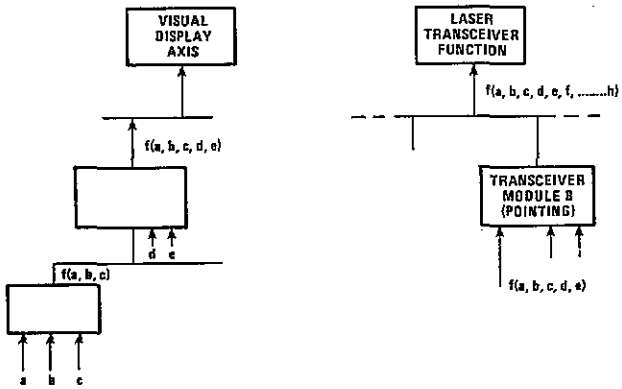
a) We could associate software with well-defined functional entities within the simulator: a specific navigation system, a particular version of a flight control or autopilot system, a functional bus structure, etc. This would map closely to parts of the general system hierarchy and be almost identical to parts of the hardware hierarchy. It would have the great merit of identifying chunks of software modules which could be reused whenever a particular system was to be simulated, or which had to be revised whenever the system was updated, and would have very little use otherwise.

b) We could organize a software tree to minimize interfaces. This would provide the most efficient computation (and is, to a great extent, the state to which present design has evolved), but would degrade identification with aircraft system entities and would present great difficulties when tracking and maintaining control over system modifications. It would have little direct use in securing or maintaining stable computing environments, since it is not primarily time- or sequence-oriented.

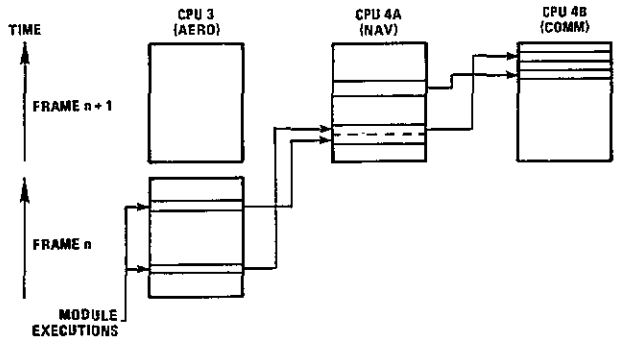
c) We could attempt to emphasize computational stability and clarity of input and output



a. BY FUNCTIONAL SYSTEM



b. BY MINIMUM INTERFACE



c. BY COMPUTING SEQUENCE

Figure 4. Some Possible Software Organizations

conditions by organizing the software in sets of serially operating modules. This attempt will quickly break down in a tangled web of interconnecting networks rather than the clean hierarchical structure we are seeking, but it is worth mentioning because it is the organization which software engineers are finally forced to derive, ad hoc, when confronted with the most timeconsuming and frustrating problems of simulator integration and debug.

We are left with the conclusion that in software, as in all else in the simulator, a single simple hierarchy cannot do the job. If this concept breaks down, what becomes of the source file concept intended to contain all data about each hierarchical node? And how then do we solve our problems of management and control?

Conceptual Development of SEMS.

It seemed clear that achieving the SEMS objectives would require the use in some manner of a massive data base which would form the core of the system. The mere size of the data base was not a major concern - given the magnitude and criticality of the problem being addressed, the cost of disc and tape drives and the computers to run them was a very secondary consideration. The real problem was that the data base was to be the sole source of information for all inquiries about entities under system control. We have seen some of the difficulties inherent in this simple-sounding proviso. It implies access to the information in ways almost as varied as the number of users. Not only would the users' needs differ greatly, they would change continually during the life cycle of the simulator.

The key step in the SEMS development was the conversion of the problem elements which have been illuminated in the foregoing analysis into a meaningful statement of requirements to be placed on the data base subsystem. These requirements amounted to a summarization and clarification of the problem:

1. The data base had to be capable of storing information which would be accrued from a number of sources. Some of this information would be relatively stable, such as customer specifications. Some would be highly volatile, especially at critical stages of a simulator's life: software interfaces, data, test results, etc. The data base would have to be capable of accepting additions, deletions, and changes at arbitrary times. These could affect not only the information contents, but the information categories. Moreover, in the useful life of the data base an undeterminable number of branches and alternate versions of the information stored would have to be accepted and preserved in a traceable manner.

2. Information from the data base had to be retrievable in a wide variety of arbitrary combinations and contexts. The number of these contexts was not predictable in advance; the philosophy was that since all the information was to be in the data base, it must be possible to use that information for any purpose that would come to hand.

3. The data base had to function in "real time" for both the insertion and retrieval of information. By this was meant that it was not to be considered simply as an archival record, but as a readily accessible store and source of information needed in the day-to-day development and operation of the simulator. Certain operations involving the data base were to be conducted from on-line terminals and its ability to perform its functions was to be commensurate with the timing implied thereby.

These requirements are extremely demanding, and in some cases almost contradictory. Conventional data base systems were entirely inadequate to meet the SEMS requirement. The solution was found in the emerging technology of the relational data base (RDB) concept. This concept has been widely discussed for a number of years, but only recently have systems appeared which support its practical application to complex cases.

The selection of the appropriate type of data base has been crucial to the success of the SEMS system and it is worth reviewing the alternatives which were considered prior to that selection.

Conventional data bases depend upon fixed file systems. They are classified according to their file structures, which may have either a hierarchical or network topology. In the hierarchical structure each

node communicates only to the node above it or the nodes directly below, so that an inverted tree-like pattern emerges. A variation of this allows for less rigidly structured communication paths, including those which communicate between elements at the same level and to elements at higher or lower levels of the data base structure which are not on the same branch of the tree. Such topologies lead to the network form of data base architecture.

In considering the appropriate form of data base for the SEMS application three points weighed heavily against either of the above structures.

1. Repeated attempts had been made to dissect a typical simulator system and software package into a single hierarchical form. In each case it was found that the appropriate structure differed, depending upon the point of view and needs of a given user. Thus, different hierarchies were indicated, depending upon whether one was interested in tracking specification requirements, development or readiness status of modules within a full simulator load, allocations of software modules within simulator CPU's, I/O and simulator hardware complements, minimization of interfaces across major computing modules for frequently used math model terms, documentation of status according to varying customer requirements or formats, etc. What emerged from these considerations was the appreciation that not one, but a number of, hierarchical (and other) arrangements of data were needed if all of the hopes for the SEMS system were to be realized.

2. Conventional file structures tend to be rigid in their organization and file arrangements. Typically, an addition of a new category of information and the establishment of new files requires major reorganization of the entire system. In the case of SEMS, it was recognized from the beginning that the system must be capable of growth beyond its initial software implementation because it was hoped to form the nucleus of a total systems engineering approach - a hope which is now being implemented.

3. In either the hierarchical or network data base models users are required to navigate along fixed access paths to reach data, and in some cases these paths can be of considerable complexity. These paths are built into the file structure. Pointers defining these paths are within the files and file records themselves. The result is that response time of the system may be affected seriously and unpredictably by adjustments to the structure of the data base system as user needs evolve, with little recourse except sweeping redesign.

A relational data base solves these problems by providing for the creation and storage of data base "relations". These relations are, in effect, easily constructed paths, reconfigurable as necessary, by which items stored in the data base can be accessed. The concept and implementation of the "relation" does significantly more than this, since in addition to providing the path between segments of the data base it also provides identification of the particular items which are available at each end of the path. The data base designer can establish any combination of paths which might be appropriate to the initial user needs and then modify or add paths as system applications emerge.

Relational Data Base. The relational data base has superficial resemblance to a conventional system of files, records, and fields in which information is stored, but it is equipped with processors which can locate and retrieve information corresponding to logical combinations of the contents of the fields.

Figure 5a illustrates the nomenclature involved. The entire two-dimensional array is known as the "relation". Each successive line within the array constitutes an "instance" of the relation. The number of lines is arbitrary, and, since an indexing and search mechanism is provided within the system, there is no requirement that instances be inserted in any particular order. The columns of the array are known as "attributes". Their selection is a matter of the use to which the system will be put, as we will see in specific examples. The number of attributes in any given relation is arbitrary. An attribute of one relation may appear as an attribute in another relation. If this is permitted within the particular relational data base, means should be provided to ensure that the attribute's contents are consistent in each appearance.

We will first consider relations in which we use only the information stored within the relation itself. The cases selected for illustration are taken from the actual set of relations provided with SEMS.

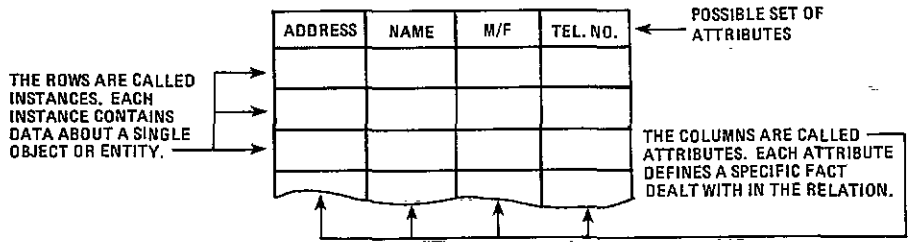
If we look first at the relation called "software-hierarchy" (Figure 5b), we see immediately the possibility of searching the entire file and detecting all node connections. We could start with either identification attribute at any instance and use the adjoining attribute as the pointer to the next instance. The processors provided with the system make it possible to perform this operation, and thus, for example, reconstruct the entire software hierarchy of Figure 3.

Suppose now that we look at the software-hierarchy relation in conjunction with the software-software relation (Figure 5c). If our task were to completely remove a functional system element (node) from the hierarchy of Figure 3, we could search the relational data base to see what software units are called by the deleted system element, and we could find out whether or not these are used by any other element. This is information completely lacking in the hierarchical structure itself, and yet it is vital to the functioning of the hierarchy. By reference to the task-library and library-software relations (Figures 5d and 5e), we can immediately discover the means and effect of deleting the system from computer loads.

Furthermore, we are, of course, interested in the inputs and outputs to any given software unit. Another set of relations is provided which are devoted to the symbols used in software. One of these is shown as Figure 5f. By reference to its attributes, logically combined with the software-hierarchy attributes, we can determine which software units contribute terms to the system being deleted (and perhaps may now also be deletable) and which units expect output from the system being deleted (and therefore probably require modification in turn).

It is apparent that the powerful processing techniques associated with the relational data base concept allow us to interrogate a data base in precisely those varied contexts for whch we must be prepared.

One more step must be provided to arrive at the final SEMS data base concept. The examples given



a. NOMENCLATURE OF RELATION

NODE IDENT	HIGHER NODE IDENT	NODE DESCRIPTOR

b. SOFTWARE–HIERARCHY RELATION

CALLING SW UNIT IDENT.	CALLED SW UNIT IDENT.

c. SOFTWARE–SOFTWARE RELATION

TASK IDENT	LIBRARY IDENT	ORDER OF LIBRARY SCAN

d. TASK–LIBRARY RELATION

LIBRARY IDENT	SW IDENT	SW REV LEVEL	SW ENTRY DATE	SW ENTRY TIME

e. LIBRARY–SOFTWARE RELATION

SYMBOL IDENT	SW IDENT	NO. TIMES SYM. READ	NO. TIMES SYM STRD	SYM USE (I, O, BOTH)	DOCUMENT I/O IDENT

f. SYMBOL–SOFTWARE RELATION

SW IDENT	DATE INTO CONFIG CTRL	(OTHER ATTRIBUTES)	UNIX HOME DIR./ PATH NAME

g. SOFTWARE–STATIC RELATION

Figure 5. Relation Examples

thus far have shown cases in which the information of immediate interest is contained in the relational data base itself. However, we are concerned with huge masses of information which will be required to fully describe the simulator, or even its software aspects, throughout its life cycle. Much of this information tends to be used and reused as bulk entities, such as software source code modules, or is relatively stable and fixed in structure, such as Prime Item Development Specifications and Statements of Work.

In theory it would be possible to include such information within the relational data base, simply calling out whole blocks of text as attributes. However, this would be extremely inefficient. It would increase storage requirements immensely, reduce flexibility, reduce the possibility of finding relevant query-related information in concurrent immediate storage, and in general be a misuse of a very elegant capability. The solution is to provide a second data base, devoted entirely to mass storage of all project-related information in conventional file structures. We can then use attributes of the relational data base as pointers to these files. In the SEMS, this function is fulfilled by a file system operating under the UNIX* operating system, and therefore designated as the UNIX Data Base.

One example of this usage is provided in Figure 5g: it is the obvious one of identifying the location in the mass storage system of the actual code represented by a software unit identification.

The unification of the two data bases completed the essential conceptual development of the SEMS. The operation of the system is summarized graphically in Figure 6, which illustrates the capability that has been provided to supply the many types of information which we have seen necessary for the effective development of the system and its subsequent life cycle management.

*UNIX is a registered trademark of Bell Laboratories.

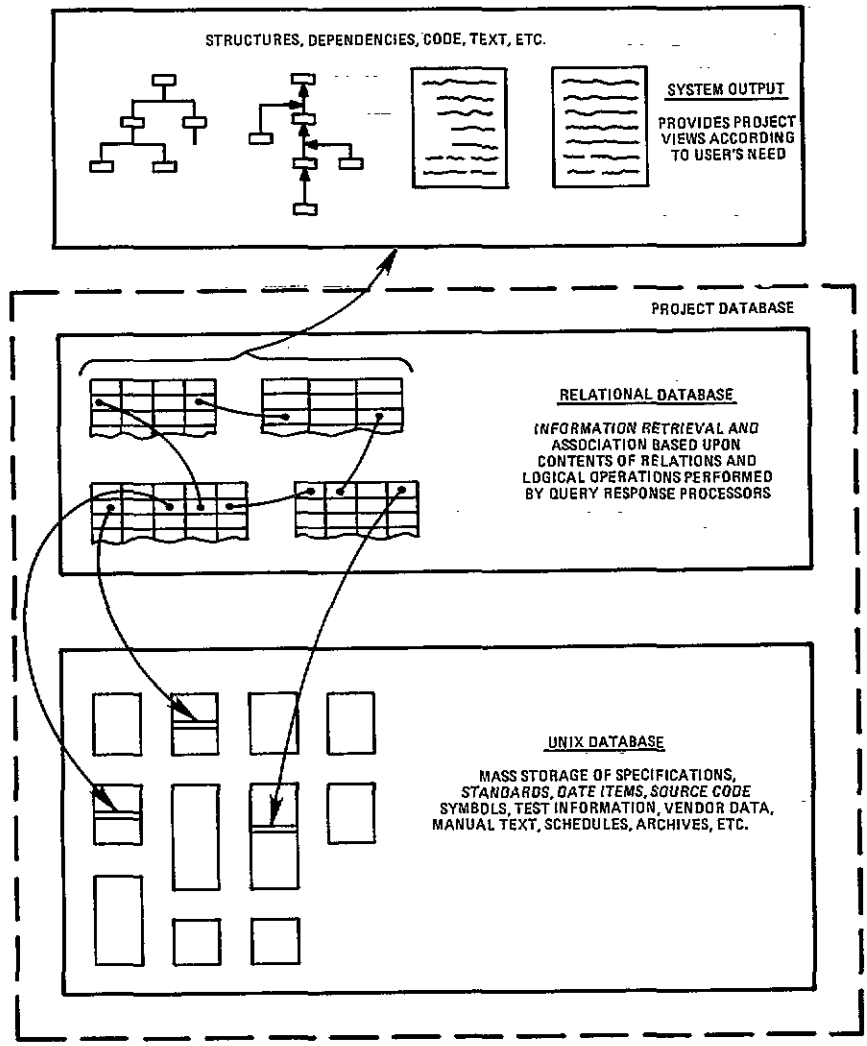


Figure 6. SEMS Database Functions

Definition Of Software Engineering Management System.

The relational data base concept as a powerful tool for information storage and structured retrieval was applied as the nucleus of an engineering management system oriented initially toward the problems of software development and configuration control. The domain of the system was taken to include the following capabilities:

1. Management of software development in accordance with requirements allocated to software from a functional system tree.
2. Verification that the products of development (source code) function to requirements.
3. Maintenance of test criteria traceable to allocated requirements, to accomplish performance verification.
4. Establishment and maintenance of software configuration control oriented toward single simulators or multiple simulator installations.
5. Maintenance of archival records of software history.
6. Management of change activity in accordance with standard protocols.
7. Management of software-related schedules, resource commitment and resource expenditure.
8. Production of reports to document status and results of above activities.

These broad objectives led to the identification of particular capabilities which were to be embodied in SEMS:

1. Internal maintenance of the software structural hierarchy.
2. Internal maintenance of interface definitions and symbol usage.
3. Internal checks for the integrity and consistency of computer modules at each level of the software tree (symbols, scaling, etc.).

4. Rejection of auxiliary information channels having no reference within SEMS.

5. Integration with a Software Development Facility including a supervisory capability over SDF activities.

6. Maintenance of status information regarding every module under SEMS control.

7. Internal maintenance of schedule, task, and responsibility data.

8. Capacity to serve as a permanent repository for all released software.

9. Capability to produce standardized documentation from internally stored data.

10. Capability to produce specialized reports in response to arbitrary queries.

11. Incorporation of audit, trace, and recovery capability for all configurations and revision levels of any program under SEMS control.

It was a requirement of the SEMS development that not only would all of the above capabilities be provided in SEMS, but that they would automatically be operational without reliance upon the diligence, responsibility, or familiarity of the using personnel. In order to ensure this, input/output processors and file manipulation processors were required which would provide the sole interface available between SEMS and users. These processors facilitate the organization of information and the manipulation of the system by users; they also automatically maintain cognizance of the operations being performed on all objects under SEMS jurisdiction.

SEMS Configuration.

A SEMS configuration consists of:

1) A project data base.

2) A supporting computer facility.

3) A software development facility.

4) SEMS processors.

The following sections describe the major components of SEMS in sufficient detail to provide an understanding of its operation.

SEMS Data base. The SEMS Project Data Base is comprised of the two separate structures known as the UNIX Data Base and the Relational Data Base. From the broadest point of view, the UNIX Data Base is a mass storage system and the Relational Data Base contains the intelligence which permits the SEMS to function efficiently with respect to user demands. The Relational Data Base may be thought of as a collection of reconfigurable pointers to the various items stored in the UNIX Data Base. However, it also is a store of information in its own right.

UNIX Data Base Description. Table 1 lists the top levels of the UNIX Data Base directory.

The global segment contains those documents which are primary references for software and systems development engineering. It contains, first, all specifications and standards applicable to simulator

Table 1. Major UNIX Database Directories

GLOBAL DIRECTORIES	PROJECT-SPECIFIC DIRECTORIES			
	GENERAL AND SELF-DESCRIPTIVE	REQ'TS, SYSTEM DEVELOPMENT	SOFTWARE-SPECIFIC	
			TEST AND VERIFY	SOURCE CODE
DOCUMENT LIBRARY CUSTOMER-GENERATED STANDARDS SPECIFICATIONS COMPANY-GENERATED STANDARDS PROCEDURES REPORTS UTILITY PROGRAMS DEVICE LIBRARIES GENERAL USER'S *DIRECTORIES* SYSTEM SOURCE CODE GENERAL-PURPOSE	USER UTILITIES USER DATA UNIX ADMINISTRATION UNIX ADMINISTRATION UTILITIES PROJECT ARCHIVE RELATIONAL DB ADMINISTRATION RELATIONAL DB DESCRIPTION PROJECT DB SOURCE CODE SYSTEM SCREEN *REPORT* RELATIONAL DB TABLE DEFINITION TABLE INPUTS	CUSTOMER-GENERATED REQ'TS PRIME ITEM DEVELOPMENT SPEC DATA ITEM DESCRIPTION STATEMENT OF WORK CONTRACT WORK BREAKDOWN STRUCTURE o o o SUPPLIER-GENERATED REQ'TS REQUIREMENTS ANALYSIS PROD/FUNCTIONAL ALLOCATION INTERFACE DESIGN SPECS INTERFACE CONTROL DOC UNIT DEVELOPMENT FOLDERS *MM REPORTS* TRADE STUDIES SUBCONTRACTOR/PURCHASE SOW'S SPECS PART 1 SPECS PART 2 SPECS TECHNICAL PERFORMANCE MEASUREMENT RISK ASSESSMENT PLANS ENGINEERING CONFIGURATION MANAGEMENT *QUALITY CONTROL* INTEGRATED LOGISTICS MANUFACTURING o o o	MATH MODEL TEST SOURCE MMT DRIVER NAV-COM FLIGHT o o o MMT DIRECTIVES o o o o MMT RESULTS o o o o REAL-TIME DATA FILES o o o o SEND TO TARGET OVERBRIDGE INCLUDE FILES CONSTANTS FILES SYMBOL DICTIONARY SOURCE FILES CRT PAGE FILES *MISC. DATA FILES* RECEIVED FROM TARGET	REAL-TIME SOURCE CODE NAV-COM FLIGHT o o o DRLMS FIRMWARE PROM SOURCE CODE PROM FIRMWARE PAL SOURCE CODE PAL FIRMWARE VENDOR SOURCE CODE SUBROUTINE/FUNCTIONAL SOURCE

design and fabrication. These arise from two sources: major customers such as the Department of Defense, and the contractor engineering division.

The remaining files within the global segment of the UNIX data base are devoted to software which defines the system operation itself, the device drivers which provide access to the SEMS peripheral facilities, the software related to the target computers (operating systems, utilities, etc.) for which SEMS-controlled programs are intended, and general utility programs for operation of the SEMS system.

The project-applicable segment contains directories and files whose contents are unique to a given project. It is a self-contained, self-supporting record of current and archival information. For example, the permanent record of all real-time software, including code definitions, symbol directories, interface conditions, data instructions, and all revisions or applicability levels ever generated in the course of a simulator life are maintained permanently within the UNIX Data Base project segments. The classifications of the project-specific directories shown in Table 1 are purely for convenience in summarizing functions. There is no distinction made in the actual data base between the various directories. Much of the table is self-explanatory, but a few points are worth noting. In the column headed "General and Self-Descriptive" it can be seen that some directories are devoted to the relational data base associated with the project. It has already been pointed out that this is an evolving system. As more capabilities are added to SEMS, the relational data base will change. It is defined within each project's UNIX Data Base so that the process of elaboration and expansion can be carried out as opportunity and need arise, without disturbing simulator projects which are already under current levels of SEMS control.

In the column headed "Requirements, System Development", some directories are shown which are not part of the baseline software engineering management function for which SEMS was originally intended. These directories are in varying states of development at the present time. Those associated with requirements and functional allocations were added at the beginning of 1985 and are currently being implemented as SEMS capabilities. Those directories associated with subcontractor relationships and planning activities are future capabilities which have not yet been implemented or integrated within SEMS.

The collection of directories identified as "test and verify" are essential in fulfilling the original objective of SEMS to provide effective software management and configuration control. Each of the directories shown can and does contain files at various revision levels and configuration identifications to match varying site and facilities peculiarities. The relational data base contains the information which identifies the correct version of any file which is to be used for particular purposes of test, verification, change control, etc. The relational data base operates upon the directories identified in this column of the UNIX Data Base to assure that all software activity takes place in the correct environment with respect to symbols, data files, constants, initial conditions, interface timing, etc. A similar provision is made to accept source code or other software-related data from a SEMS-controlled facility with assurance that the conditions under which the information was generated or validated are documented.

The remaining column deals with directories which store source code itself. This source code is always stored as individual modules; task and load-building are functions of the relational data base and the UNIX Data Base retains only the software units, with archival records of their revision levels as they occur.

Relational Data Base. The effectiveness of the Relational Data Base is measured largely by the rate at which it can process queries to retrieve information. For a given system this is heavily dependent on the structure of the data model, and in designing this structure some compromise is necessary between conflicting ideals.

From the point of view of system compactness and immunity to accidental corruption (by internal inconsistency) of the data base, it is desirable to minimize the number of appearances of any given attribute (such as a software unit identification number) in the data base. In principle, this is possible, since in principle it is possible to obtain information related to an attribute from any number of other relations by a properly constructed query which will specify a logical chain of joins, unions, negations, etc., with the rest of the data base relations until the desired set of attributes is retrieved. The price we would have to pay is the time required for complex processing, and time is at a premium in data base operation.

In practice, we know in advance most of the queries in which we will be interested. It is possible to shorten greatly the chain of inferences required, and thereby improve system throughput capacity, if we design relations so that most queries - and perhaps all "standard" queries - need to operate on only a few relations. The system can be preconditioned to behave in this way if we allow selected attributes to appear identically in several relations. Then, when we make a query which concerns such an attribute, we enter the data base in the relation which has been designated as efficient for the purpose of that query. Of course, this does not preclude "non-standard" interrogation of the data base by means of any correctly formulated query; in those cases response time is not an important consideration.

In the SEMS Relational Data Base attributes are allowed to exist in multiple relations. Optimizing the system consists largely of identifying cases where this is advisable, in conjunction with the general problem of designing system-effective relations.

When the decision to permit this practice was made, the possibility of conflicting versions of the same attribute was recognized. Every effort has been made to prevent this before occurrence, e.g., by giving the system knowledge of all such redundancies, with automatic multiple entries on change, etc. As a back-up, however, the system runs a background processor whose function is to endlessly scan multiple insertions and flag any discrepancies which are detected.

The entities of greatest interest in the baseline Relational Data Base, devoted primarily to software management, are identified as software units. These are such things as modules, subroutines, functions, data files, directives for compiling, math model drivers, initialization files, etc. Each is identified by a unit identification number (ID).

Although software symbols are not in themselves configuration units, many of the attributes needed

to describe the nature of the symbol and its usage are the same as those for software units, so symbol relationships are a part of the Relational Data Base. Of course, cross-reference relations exist between symbols and software units.

In the initial baseline SEMS, relations were provided for activities related to the placement of a new software unit under configuration control, for the control and management of change of controlled software, and to support and facilitate processes of load-building, configuration management, development and verification of modules, site support, etc. The versatility of the system is shown by the relative ease with which an entire requirements tracking data model was developed at the beginning of 1985, building upon the experience gained in the earlier software management design. The requirements tracking system became operational for practical test in slightly more than four months. The remaining effort has been concentrated on integrating its data model into the baseline SEMS and beginning the process of tuning the combined Relational Data Base for efficient throughput. This experience provided confidence upon which to plan the expansion of SEMS into an overall simulator management capability.

Supporting Computer Facility.

During the SEMS development, and through at least mid-1985, the selection of a supporting computer was contingent upon the particular relational data base system used (and vice versa), because of vendor compatibility considerations. Both hardware and software relational data base systems have been evaluated and exercised successfully. The baseline SEMS installation uses a SEL 32/87 computer supporting the Mistress/32, Version 1.3, relational Data Base System.

Improvements in data base processing rates are a continuing goal and are expected to materialize as data base systems evolve. Therefore, the baseline configuration is subject to change. To allow for this, the SEMS processors, which are the key to efficient convertability, have been written to maximize their portability.

Software Development Facility. (SDF)

Software Development Facilities had been in use for several years prior to the undertaking of the SEMS project.

The original function of the SDF was to provide an off-line environment in which simulator real-time programs could be developed independently of the stage of development or construction of the actual simulator. Associated with the SDF computers themselves are a number of local and remote terminals and peripherals such as disc systems, and tape units, all linked by a central MICOM switch. With this arrangement, a software development engineer can perform his functions in a real-time interactive manner at any of the system terminals.

The key to the effectiveness of the SDF is in the assortment of development tools which have been provided to facilitate the work. These tools allow the generation of new software source code, or the modification of source code in an old program, and provide for compilation and running of such modules. A Math Model Test program then allows the engineer to test these programs prior to their integration with a simulator. Equipped with these tools, the software engineer finds it convenient and effective to accomplish both alpha testing, on single software modules, and beta testing, on multiple modules linked to perform a larger function.

The SDF's had been developed as independent, self-contained facilities. The specifications, interface definition, and reference data for use in each module were all made available for the software engineer working in the SDF by manual and essentially uncontrolled means. Therefore, the validity of the development effort was dependent upon the diligence with which the software engineer assured himself of the accuracy of his information, and the consistency with which changes and updates of information, possibly originating in totally different areas of development, were reflected into his working documents. Furthermore, there was no assurance of consistency and coherence between multiple parallel efforts in terms of the interface understandings and, indeed, of the actual symbol consistency of variables crossing software module boundaries. Although tools were available within the SDF for checking symbol validity, consistency, interface specifications, etc., such tools could be bypassed at will and, therefore, heavy reliance was placed on the judgment and thoroughness of individual software engineers. In addition, any results which had been verified under conditions equivalent to partial load testing could not take into account contingencies which might result from full load operation and could not demonstrate the validity of the software under the actual conditions of memory and time allocations which would occur in the complete simulator.

SDF Integration into SEMS. Integration was accomplished by means of a software bridge constructed between the SEMS itself and the existing Software Development Facilities. It was determined that SEMS should not replace the Software Development Facilities, which were functioning very effectively within their narrow domain. What was needed was the superposition of a supervisory layer upon activities taking place within the SDFs. Moreover, this did not have to be applicable to every conceivable activity which could be engaged in at the SDFs, but only those which had an intended or potential impact upon the software under SEMS control.

Thus, there is no restriction whatever upon a software engineer requesting, from his SDF terminal, a copy of any configuration unit which is stored within the data base for any simulator under configuration control. (Passwords and special secure facilities are provided and enforced in order to maintain security in those cases where sensitive data is involved in the information requested.) Having obtained a copy of the SEMS file, an engineer has complete freedom to experiment in any way he wishes, and may run any variation of math model testing that he wishes to, choosing interfaces and variables at will. If his activity is essentially "scratchpad" in nature, he may observe and obtain copy of results according to his individual preferences.

The SEMS control over software configuration comes into play if, for example, the user is complying with an officially approved software change request or discrepancy report. In these cases, he would normally have indicated his intention at the time of asking for a copy of the configuration unit in question. Had he done so, the configuration unit would have been locked against further attempts at change, and official records would be in the file concerning his objective, activity, and the document to which he is responding. In the event that he failed to do so, he will be prevented by the SEMS protocols from entering whatever changes he may have decided upon into the system. That is, he will be

unable to insert the changed configuration unit into the official SEMS data base. Retroactively, he must now apply for approval and permission to make the change. As a consequence of this approval, the SEMS protocol then requires that before he can execute an official test procedure to verify that the changed CU meets the requirements spelled out in the change document, he must access and employ the symbols, variable ranges, and interface conditions which are available in the SEMS data base as applicable to the CU in question. When his change is completed (in his view) and submitted for approval to the data base manager of configuration control, it is automatically accompanied by verification that the CU has actually been tested according to the conditions which are defined within the data base as valid for the application in question. Since there is no way to build an effective simulator load except by using configuration units which are maintained within the approved SEMS data base, arbitrary, undocumented, or improperly tested real-time software programs cannot be used to bypass the SEMS control system.

SEMS Processors.

The processors provide the internal system software by which SEMS functions. Their operation is, in general, transparent to the user, who is required only to know the functional capability of the system and to identify what service he is requesting. Utilities used to create and modify these processors are available only to the SEMS operating personnel, and are not discussed here.

Data Base Processors. These consist of the Configuration Unit Identification Processor and the Configuration Unit Control Processor. When a configuration unit is first placed under SEMS control, the Identification Processor opens all files and relations necessary for basic identification; specific relations defining the CU with respect to the functional system tree, etc., are expected and their absence is recorded within SEMS until the deficiency is corrected. The Control Processor manages and records all subsequent activity with respect to the CU.

Document Processors. These consist of three processors which are used to prepare the formats, relation paths, relation logic and output channel commands used to provide screen or hard copy displays of standard SEMS reports. They are used primarily by SEMS operating personnel to create reports beyond those already in the standard report library as SEMS usage expands, or in response to particular project needs.

SEMS Report Processors. The SEMS data base includes management and status information regarding all realtime software within a given simulator, in all of that simulator's possible configurations. It has already been indicated that this information constitutes the only valid source for both the content and the status of individual software efforts. Therefore, SEMS can be the source of all reports and other documentation which may have to be generated in connection with development or maintenance activity.

The structure of the relational data base is amenable to this usage. Indeed, one of the criteria by which the data model for the relational data base is constructed is that efficient pathways be provided for the chaining of information in the forms which are required for the various report and documentation functions. To assist in the preparation of reports, a set of report processors has been provided in SEMS. By means of these processors a user, when he addresses the system and indicates the type of report he is preparing, is provided with prompting and help screens (to the extent that his participation is necessary). Upon his answering these prompts, the SEMS processors automatically execute the necessary search, retrieval, formatting, and print functions to prepare the report in question.

It should be emphasized that these report processors are fully automated for the preparation of any of a sizable list of reports and documents which are recognized as standard requirements for the development and management of the simulator. In addition, of course, the relational data base makes possible the retrieval of information in any arbitrary form for special purposes, via a standard query language processor.

SEMS In Operation.

The SEMS is installed in a facility which is managed and maintained by trained personnel as would be the case in any computation center. Data base administrators and software engineers/programmers are responsible for the design and maintenance of utilities, data base processors, screens, etc. Their intervention is not required for normal access to the system by its end users.

Several communication channels are provided for user access to SEMS:

a. Local terminals within the SEMS facility for access to the SEMS data bases via the interface processors, and peripheral equipment for generation of hard copy printouts.

b. Remote facilities similar to the above, which can be located within a project or project control area.

c. The software bridge between the SEMS and the SDF, for users working in the software development facility.

d. Secure modems and standard communication lines to permit communication between SEMS facilities at different geographical locations.

The first three of these communication channels were made available in the initial SEMS development. The fourth is a part of the planned SEMS expansion. Its purpose is to allow SEMS installations at the various contractor facilities to interact cooperatively in the development and maintenance of large systems. It can provide the same service between the contractor's facility and field installations under a simulator user's control.

Initialization of SEMS Project Data Base. When a simulator is placed under the SEMS system, a descriptive code number is issued to be identified with all entries and operations associated with that project. In addition, the project manager provides the data base administrator with an initial listing of authorized personnel and their functions within the project. Identifying codes are then issued by the data base administrator, and these identifications are kept current throughout the life of the project.

The process of entering data can start at any time after initialization of project access to the system. For projects which begin after the first installation of SEMS, it is recommended that this

start during the preproposal cycle where possible. Modern proposal activities for large-scale simulators are so extensive that a comprehensive management system such as SEMS can greatly facilitate much of the data organization and scheduling which is a key part of the proposal cycle. It is then available for immediate use at the beginning of a contract. Typically, this data entry would be carried out at dedicated stations within the proposal or project work areas, at terminals linked to the SEMS system.

During initial implementation it is assumed that the "standard" relational data base structure would be used. Therefore, no additional effort is needed to develop or establish the system; the software used for its operation and management as well as the full structure of software files which have been described above are simply copied over into the disc storage which will be private to the particular simulator project.

Typical SEMS Procedure. For the purposes of this paper, only a simple configuration unit development and subsequent change cycle will be summarized to illustrate the operation of SEMS.

It is a characteristic of the system that any item of information, once officially accepted into the UNIX Data Base or the Relational Data Base, is made a part of a permanent archival record. Thus, the historical development of systems and their correlary information packages are always retrievable and reconstruction of a prior state is always possible regardless of any activity which may have taken place with respect to a project. However, scratch pad files are provided in SEMS for use during the preliminary phases of any project activity, if desired. These allow software engineers to experiment with solutions to a problem without inhibition, but make the results immediately available for error-free entry into the formal project data base when the SEMS conditions have been satisfied.

The process of archiving begins with the assignment of a valid identification number and revision level to the data item. Thereafter, the only operation which can take place without setting flags and initiating other evidence of on-going activity is the simple extraction of a copy of the file for purposes which need not be further elaborated or explained. Any software engineer having an interest in the CU may receive a copy by transmittal over the SEMS-SDF bridge. Such copies have no effect on the SEMS directories. If the purpose of extracting a copy of the CU was to modify it and reinsert it into the data base, the software engineer will find that SEMS refuses to accept the revised CU unless proper procedures have been followed.

An engineer responsible for the development of a given configuration unit is likely to be aware of past practice in producing similar units and may elect to modify or use directly an existing verified and tested software unit to satisfy his requirement. (In a future extension of the SEMS, it is anticipated that such cases of "standard" requirements will be identified by the systems engineering process and likely candidates will be offered automatically to the development engineer.) If he elects to begin with a previously tested and verified module, he may have this copied to his work station by requesting a copy of the module from the software library.

When the software unit under consideration is ready for test, the Relational Data Base is used to search for and copy over to the work station its previously defined test procedure. The test procedure will have been inserted into the UNIX Data Base by means of the same system control procedures which were exercised over the creation of the CU itself. If these procedures are used by the development engineer to exercise the CU for which he is responsible, the results can automatically be compared with the standard value and tolerance, and the performance of the CU thereby verified. At the conclusion of this effort, therefore, the development engineer is ready to submit the CU for acceptance and for incorporation into the simulator software real-time directory.

Revision of a CU already under SEMS control must begin with notification to the system that such modification is required. For SEMS to accept such notification, it must be given the identification of the formal cause for the change activity. This might be in the form of a discrepancy report, a specification revision, etc. Furthermore, the system will not accept the validity of change processing being undertaken on a CU because of the existence of such a document unless it has also been notified of the individual to whom responsibility for the change has been assigned. Under these conditions, the responsible software engineer may request a copy of the CU, this time indicating that his purpose is to accomplish the indicated problem correction. In response to this condition, the SEMS will transmit copies of the CU, and will also flag the data item so that other possible users of the CU will be made aware of the change activity then in process. Only one such withdrawal-for-change activity can take place at any time with respect to any given data item.

At this point SEMS expects that a modified version of the data item within its directories will be submitted at a future time. In fact, such resubmittal will not occur until the revised data item has been processed through the same acceptance procedures as accompanied the initial submittal of the original CU. At such time as the authorized and accepted resubmittal is made, the preceding version of the data item is archived, a new revision level is assigned to the changed version, and it becomes the working copy within the directory system. At the same time, users who had requested copies of the CU while it was in the change process are automatically addressed via the UNIX mail system and informed that the change process of which they had previously been warned has now been completed.

ABOUT THE AUTHOR

John Schepp is a Member of the Technical Staff at the Link Flight Simulation Division of The Singer Company, Binghamton, New York. He is principally concerned with the planning and execution of the Link IR&D program, under which the software engineering management system described in this paper was conceived and developed. Mr. Schepp holds Bachelor of Science and Master of Science degrees in Electrical Engineering from Columbia University.