COMPUTER SYSTEM MODELLING: A TEST BED FOR NEW SOFTWARE TECHNOLOGIES

Ronald R. Willis

Hughes Aircraft Company

ABSTRACT

This paper discusses a technique in computer system simulation that uses models as effective tools for the study of new software technologies. This model development technique, called representative modelling, and its relationship with real-time program development are surveyed. The significant correlation between these two development processes in addition to reduced costs substantiate the effectiveness of using simulation models for the evaluation of new techniques in software development. An experiment to verify this hypothesis was conducted in which a model was developed using the methods of Top Down Development. This experiment not only indicated that models could be effective development analysis tools, but that computer system simulation is a natural environment for the use of these new software technologies.

INTRODUCTION

This paper proposes an alternative to expensive experimentation in the analysis of new software technologies. New procedures such as top down program development, structured programming, and process construction - even though evidently powerful - require experience to be used effectively and experience means dollars to software managers.

The alternative discussed here is the computer system simulation model. Simulation languages are powerful enough to guarantee development costs for models factors less than the systems they are used to simulate. Therefore, if in the development of a simulation model we can gain experience in the use of new software technologies, then we can guarantee factors less in cost for that experience.

Since the technologies we seek to investigate are generally procedural in nature, we must abandon our old concepts in the use of modelling only as a statistical analysis tool. In order for the computer system simulation model to provide useful evaluations in the design techniques of program development. For example, the model must be thought of as a development analysis tool; i.e., the important product is not at the conclusion of model development, but the development process itself. By changing the emphasis of computer system modelling in this way, we can then consider how model development relates to program development and, thereby, its effectiveness in the evaluation of new software development procedures.

This paper introduces a model development technique called representative modelling which satisfies the required relationship between the two development processes. The most important aspect of this modelling methodology is its inherent ability to preserve the actual sequence of events that occur in the interaction of modelled computer system elements.

The consistent one-to-one translation of program module, data structure, and hardware device not only forces a one-to-one representation of the interaction between these elements, but also forces the modeller to encounter the same problems he would encounter in the development of the actual system. The result of these correspondences between system and model is an understanding of the processes involved in the development of the actual system.

With this tool, then, we are in a position to measure the effectiveness of using simulation models for the evaluation of new software technologies. An experiment was conducted for this purpose and showed that significant insight into the processes of top down development could be obtained through the development of a model. Furthermore, the application of this software technology to model development led to significantly increased model quality and credibility.

TERMINOLOGY

In this paper we confine the topic to real-time computer systems (because of the author's experience in this subset of computer simulation studies and not because the techniques discussed are applicable only in this area).

Real-time systems are defined as systems which are comprised of computers, peripheral equipment and operational programs together performing tasks that have critical response requirements. The system may be for aircraft control, time share services, radar early warning, or production control; there are no restrictions implied in the application. In particular, we are interested in the development of real-time programs, that is, the processes that occur between the definition of system requirements and the resultant operational program.

A real-time computer system is assumed to be comprised of "elements" which interact with each other through "events." Elements are defined to be hardware devices (CPU, disk, channel, memory module, etc.) and data structures (executable program module, file, system data table, etc.). An event is defined to be a communication between any two elements. For example, a disk read-complete interrupt is a disk-CPU event; system data table accesses by program modules are data-program events; tape rewind commands executed in a program module are program-tape events.

Events may occur at various "levels" of detail. For example, on a micro-level a program module is executed by a sequence of individual events in which the CPU accesses an instruction in memory, accesses a data element in memory, stores a result, and transfers control to the next instruction address (ADD X1, X2).

On a macro-level, we may consider only the inputs, the outputs and the time consumed in the execution of a program module and hence, only CPU-program events (execution) and program-data events (accesses) (CALL CORRELATION WITH INDATA, OUTDATA).

We may abstract the level even further by considering entire sequences of macro-level events as one super-level event (EXEC FORTRAN). Corresponding to each level there is a set of required elements which are necessary for proper event sequence control. For example, at the micro-level individual instruction data elements are necessary in order that the proper CPU-instruction event sequence can occur, whereas, at the macro-level, only the input and output message data elements may be necessary.

A computer system simulation model is a computer program written in a simulation language which contains representations for the computer system elements and which, when executed, simulates the events of the actual system. We restrict our attention to models written in discrete event simulation languages such as GASP, GPSS, SIMSCRIPT, CSS, ECSS, and Hughes' Product Line Simulator (PLS).

These languages are felt to have equal applicability to the topic of this paper. The development of a computer system model is the process of translating real-time computer system descriptions into simulation language terms and the subsequent verification of model execution.

REPRESENTATIVE MODELLING

CONCEPTS

In translating a real-time program description into simulation language terms, the modeller has a choice as to how the elements will be represented. This choice can be characterized by the detail into which the modeller translates individual elements and therefore, the corresponding detail into which event occurrences are translated. One such choice is "representative modelling" which is defined in a strict sense as:

> "A straightforward translation from system description to model language so that for every real-world element and event there is a corresponding model element and event."

If the events of a computer system were considered as operations (interactions) on the elements, representative modelling could be described mathematically as a homomorphism; that is, we seek to preserve in the model, sequences of events in the real system such as processor control between program modules. A one-to-one correspondence between the system elements and model elements is also implied in the definition.

In general, then, this model development method seeks to make the diagram below commute. The translation process, f, is a one-to-one mapping, and the operations h_r, on the real-world elements, are preserved as operations, h_m, on the modelled elements; therefore, if x is a real-world element, then $f \circ h_r(x) = h_m \circ f(x)$.

$$\begin{tikzcd}
\text{real-world element}_1 \arrow[r, "f"] \arrow[d, "h_r"] & \text{modelled element}_1 \arrow[d, "h_m"] \\
\text{real-world element}_2 \arrow[r, "f"] & \text{modelled element}_2
\end{tikzcd}$$

The purpose of developing a model will generally limit the level (detail) to which the system is represented in this one-to-one fashion. If the role of the model is only to generate statistical information about the performance of a system, a mathematical queuing model would probably suffice (here the system is represented on a one-to-one basis only to a queue or potential bottleneck level). However, if the model is used to identify program module interface problems (aspects of development procedures) in addition to system performance data, the system must be represented in a one-to-one manner at least to the program module level before problems of this nature can be quantified. On the other hand, it would be absurd to translate detailed system descriptions of every element and event (even if they were available) for models are useful only when they are much cheaper than the real system. If every element and event were modelled we would only be transferring the system to another host machine.

Since simulation at Hughes is used primarily as a tool for reducing software development costs, the level of detail for our computer system simulation models is such that a model satisfies the following three purposes which are discussed in more detail in (1).

1. It serves as a learning tool for the program being developed.
2. It provides useful development information for the software manager.
3. It predicts and aids in the solution of development problems which cause costly overruns.

The structures that accomplish these purposes are what is referred to in this paper as "representative." This level of representation is generally at least to the level of translating available documentation one-for-one into simulation language terms. In cases where specification of critical software control elements has not yet been completed, the representative model will usually have greater detail than the available documentation (design extensions).

Because the purposes are generally software in nature, the model usually does not go into painstaking detail for representing hardware operations within an element; however, for each hardware element in the real system there is always a corresponding element in the model which accomplishes the same function. On the other hand, hardware functions are represented in a very detailed manner in cases of critical event control such as dual processor synchronization and processor interruption.

A model is representative because it is not functional; that is, elements and events are not grouped together into a function which accomplishes the cumulative result of the individual elements and events (again, a question of the structure of representation).

In Figure 1, a software architecture and three sequences of software events are shown to illuminate the fine line between representative and functional. Sequence A is that of the real system being modelled; B, a representative translation into simulation language terms; and C, the author's (unfortunate) functional translation. Sequence C is a grouping of events - a scheduler processing and machine environment restoration - which was specified in this functional manner because:

a. A-priori assumptions for the particular application were that BMOD service requests were always highest priority and hence, scheduled first.
b. The model execution overhead would be very high because of such frequent BMOD service requests.

It is easy to see why Sequence C is functional; the execution of (at least) three program modules was lumped together as one program module execution. This sequence is also functional in that it is not representative to a level great enough to satisfy the purpose of the model. In particular, the purposes that are not satisfied are:

1. The model does not serve as a learning tool - it serves to obscure (at best) the actual

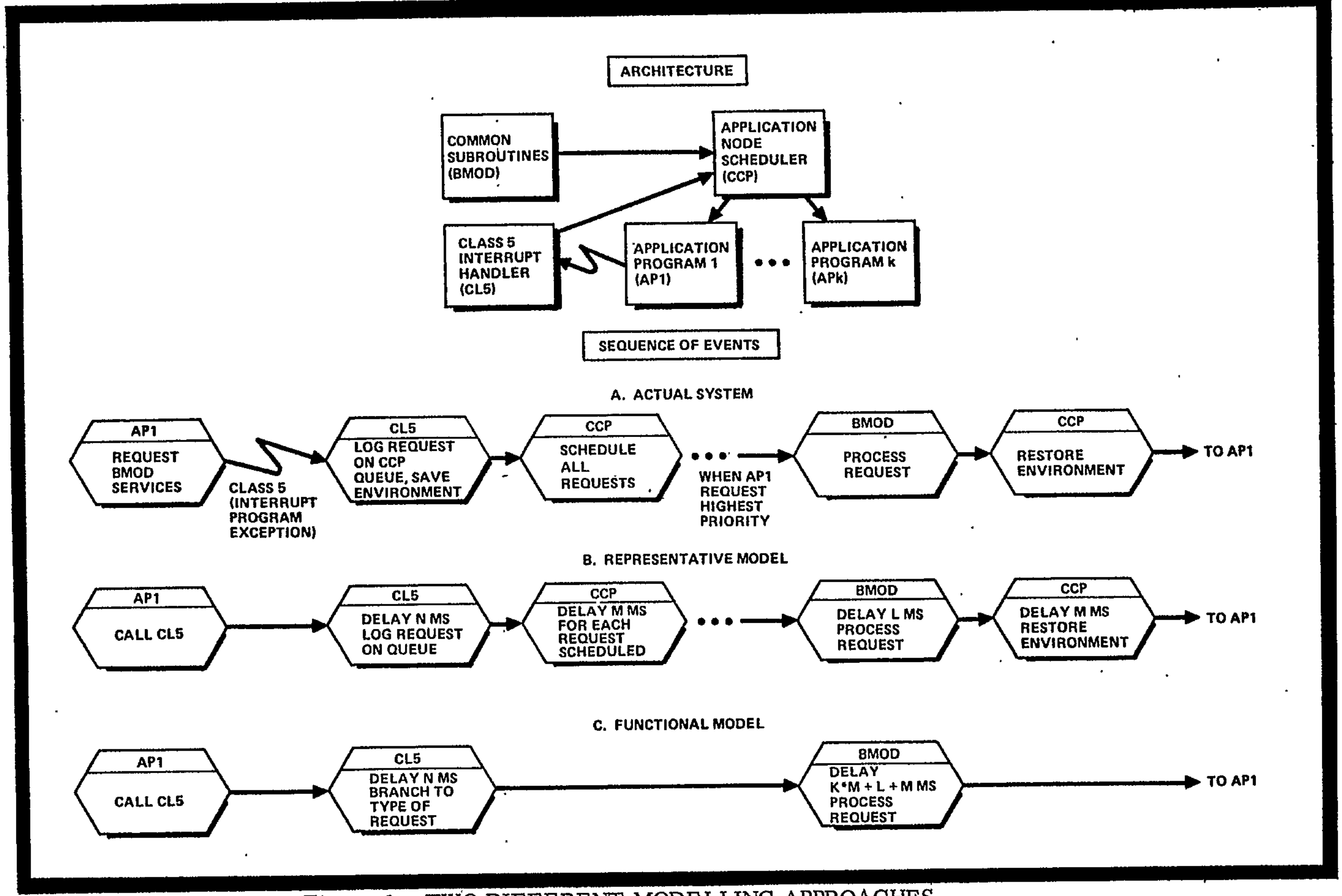
ARCHITECTURE
COMMON SUBROUTINES (BMOD)
APPLICATION NODE SCHEDULER (CCP)
CLASS 5 INTERRUPT HANDLER (CL5)
APPLICATION PROGRAM 1 (AP1)
APPLICATION PROGRAM k (APk)
SEQUENCE OF EVENTS
A. ACTUAL SYSTEM
AP1 REQUEST BMOD SERVICES
CLASS 5 (INTERRUPT PROGRAM EXCEPTION)
CL5 LOG REQUEST ON CCP QUEUE, SAVE ENVIRONMENT
CCP SCHEDULE ALL REQUESTS
WHEN AP1 REQUEST HIGHEST PRIORITY
BMOD PROCESS REQUEST
CCP RESTORE ENVIRONMENT
TO AP1
B. REPRESENTATIVE MODEL
AP1 CALL CL5
CL5 DELAY N MS LOG REQUEST ON QUEUE
CCP DELAY M MS FOR EACH REQUEST SCHEDULED
BMOD DELAY L MS PROCESS REQUEST
CCP DELAY M MS RESTORE ENVIRONMENT
TO AP1
C. FUNCTIONAL MODEL
AP1 CALL CL5
CL5 DELAY N MS BRANCH TO TYPE OF REQUEST
BMOD DELAY K*M + L + M MS PROCESS REQUEST
TO AP1

Figure 1. TWO DIFFERENT MODELLING APPROACHES. The functional model lumps together separate processes.

sequence of events and contains hidden a-priori assumptions.

2. The model cannot provide development information in important areas such as the interface between an application and the operating system.

3. The model cannot predict or aid in the solution of cost overrun problems because such design inconsistencies as erroneous module interface assumptions and priority of request scheduling are not identified.

IMPLEMENTATION

The representative model is constructed directly from a system description. A hardware diagram is normally constructed summarizing all external equipment elements and their characteristics; this diagram not only serves as model documentation of the one-to-one translation of hardware elements, but also is a common language between the modeller and model user.

Next, the system software specifications are translated directly into simulation language code; there is no prior analysis required for the interactions of program modules since inconsistent system specifications will be uncovered during the simulation of the model. However, since the modeller must translate one knowledge to another and is, therefore, familiar with the specifications, many inconsistencies are uncovered during the one-to-one translation process.

The most important element of a representative model is the data element. For software models to maintain proper sequences of events, the data necessary to effect required decision branching must be translated in a one-to-one fashion into the model; otherwise, the model becomes functional (e.g., branch to A if RANDOM (1,100) $\geq$ 60) and requires analysis for interpretation of its results.

The modeller usually has liberties in how data is represented in the model, since often in systems under development the location, size, and accessibility of the data are yet unspecified. It is important, however, that each required data element is represented one-for-one in model data elements.

As an example, Figure 2 shows the differences between functional and representative data element modelling. We could use a GPSS SAVEVALUE to indicate a yes/no status bit for program residence rather than statistically analyzing the operation and branching according to some functional value. In this case, the SAVEVALUE represents an actual program residency table data element and can be set or reset during model execution when it is determined that the modelled program is resident or not.

In parallel with data element modelling, the loading environment is considered; the parallelism is necessary in order to be able to define the input load parameters required in the data elements (e.g., branch to B if track report range $\geq$ 100 miles). A useful technique in defining the input load is to determine the value at time of generation; the path of the load element through the system is then deterministic.

The loading environment, if not scenario-driven, is usually stochastic in nature and cannot, therefore, satisfy the representative modelling definition completely. What is important is that if there are load data elements in the real system containing X, Y, Z information necessary for software control, then there are similar data elements in the model with information X', Y', Z'.

The implementation of a model is therefore in four parts: hardware, software, data structure and load environment. As in the actual system, none can be developed separately; but, instead, all are closely tied together in function and, hence, must be developed in parallel. It is difficult to explain in detail the actual one-to-one translation occurring in each of these areas because the representation is intimately associated with the simulation language into which they are translated.

This is the reason that this paper and the definition of representative modelling do not discuss actual

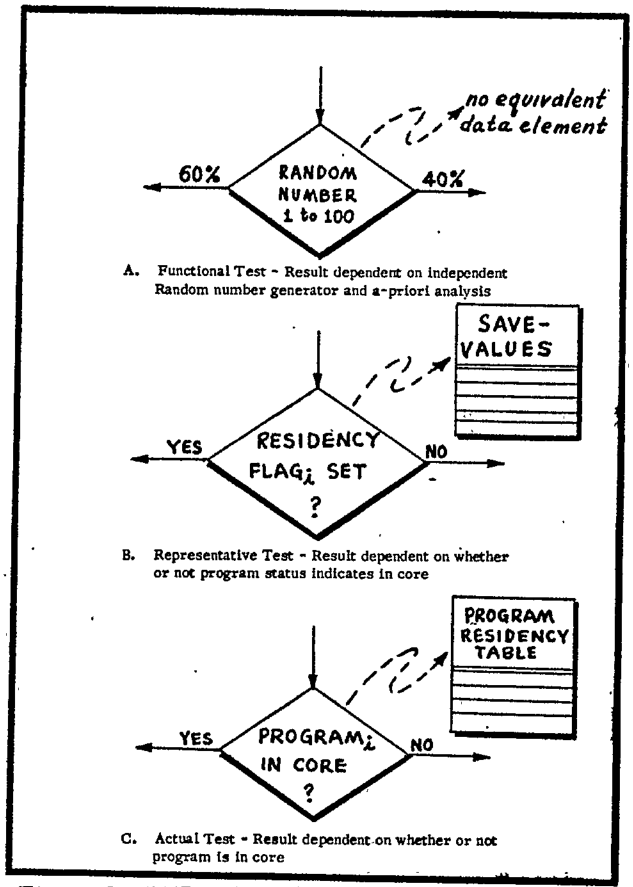
no equivalent data element
60%
RANDOM NUMBER 1 to 100
40%
A. Functional Test - Result dependent on independent Random number generator and a-priori analysis
SAVE-VALUES
YES
RESIDENCY FLAG$_i$ SET ?
NO
B. Representative Test - Result dependent on whether or not program status indicates in core
PROGRAM RESIDENCY TABLE
YES
PROGRAM$_i$ IN CORE ?
NO
C. Actual Test - Result dependent on whether or not program is in core

Figure 2. REPRESENTATION OF DATA ELEMENTS. Representative data elements preserve operation sequence.

implementation - the methodology is independent of the particular simulation language used.

EXPERIENCE

This representative modelling methodology has been used in computer systems models at Hughes for the past four years using PLS. The technique was first employed to serve as a learning tool for our ground systems real-time programmers; it has evolved through use as system design change verification tool, system design tool, a software development management tool (1), and now as tool for evaluation of new software technologies. Experience with representative modelling has shown that the following are generally true:

- The Modeller does not have to apply analytic techniques during the translation process from system to model terms - the translation is a straightforward process that does not necessitate an a-priori understanding of system operation. In addition, system design variations are more easily implemented because no review of assumptions made in generating the model is required.
- Representation should generally be to the level of system documentation available (and, at times, to greater detail) in order to be useful to the particular area requiring analysis. In particular, the control elements of a computer system (scheduler, interrupt handlers, etc.) are elements that should be represented to a great level of detail - possibly to bit setting and testing - in order to have confidence in the correct sequence of model events. However, since often the level of documentation is not consistent; the model in this case will yield results commensurate with the documentation level.
- Model documentation often serves as a significant part of the system documentation since both correspond in logical description (and labelling conventions). In addition, the model and model documentation serve as communication tools between the modeller and system designer/manager and, as a consequence of this communication, the modeller learns the system he is modelling.
- Representative models generally use more core and execute longer than analytically translated models. However, a technique used at Hughes is to investigate a system's behavior for a short interval of time at peak loading conditions. This technique not only satisfies the purpose of the model to characterize the system elements and identify bottlenecks, but also reduces the effect of a longer running model.
- It has been found that the development of models using a one-to-one translation process leads to understanding about the development of the system being modelled (and consequently the purpose of this paper). An outcome of this phenomenon is that testing and validation procedures used in model development serve as guidelines for testing and validation for the actual system. This outcome is significant when considering the effort normally applied to proving program correctness and also opens a new area of investigation for future uses of computer system modelling.

Limitations of representative modelling generally result from increased model development costs and execution rates. These limitations are far outweighed by such benefits as: (1) the use described in this paper - a test bed for new software technologies; (2) in-program development models are used primarily for design verification and hence are generally being changed often - representative models are easily changed and re-verified because there are no re-analysis steps to go through as in functional models; (3) representative models serve the purposes stated previously of reduced software development costs; (4) representative models have long development times but are easily verified because their operation "looks like" the actual system.

A DEVELOPMENT ANALYSIS TOOL

CHARACTERISTICS

To be able to say that a particular development analysis tool is good or bad, we must first identify what it is that is to be evaluated by the tool. For this purpose we seek to understand what it is about a particular development procedure that makes it better or worse than other procedures (i.e., the characteristics of the development process which lead to a better or worse product).

We evaluate a new software development procedure by investigating answers to the following questions:

1. Are development costs reduced?
2. Is software performance improved?
3. Is the software more reliable?
4. Is software maintainability and reusability improved?
5. Is the procedure applicable?

The characteristics, then, seem to fall into two categories: reduced software development costs and improved software quality. Therefore, a tool for development analysis must be able to comparatively establish values for these two categories.

REQUIREMENTS

What is needed by a development analysis tool to be able to characterize software cost and software quality?

A significant portion of cost seems to be intimately related to the completeness of design in inter-element interactions. It is the integration phase of development (the first phase in which software elements must interact with each other and with actual hardware) in which schedules are normally not met, manpower is inefficiently increased, and software changes or hardware add-ons become necessary (2). If we seek to evaluate the cost of a new software development procedure we must therefore investigate, in detail, the processes involved in the integration of software elements.

Although software quality is dependent on many factors, those to which modelling can contribute directly are generally dependent on performance characteristics such as reliability, reusability, and sensitivity to changes. The latter two are often degraded because intimate knowledge of the particular system is built into program operation; i. e., there are a-priori conditions "understood" between communicating elements. Software reliability is improved by consistent, organized procedures of testing both at element level and element interface level, and so must also be dependent on the interaction of elements.

It is the author's hypothesis that the fundamental requirement of a development analysis tool is its ability to characterize inter-element interactions. Indeed, the most interesting problems in software development (deadlock, communicating parallel processes, program schemata) and new software technologies (top down development, structured programming, process construction) deal with inter-element interactions in some way.

We should, therefore, seek to characterize an entire system operation as a sequence of individual element interaction events where we understand "element" to be program module, data structure, or hardware device. If these events occur "correctly" once, then the software will be said to be error free; if in all cases, then reliable.

If the number of incorrect event occurrences is reduced (during integration), then software development costs will be reduced; and if event occurrences are independent then the software will be reusable. In this way, we can then approach a method by which a particular development procedure can be comparatively analyzed.

THE TOOL

We recall from the discussion on representative modelling that models developed by this method preserve the actual sequence of events that occur in the interaction of modelled computer system elements. Therefore, if we agree that a particular development procedure is characterized fundamentally by the manner in which the elements interact, then models developed in this one-to-one fashion have the potential to analyze new development procedures.

What is missing is that we are dealing with models and not the actual system. However, we recall from the discussion on experience in using representative modelling that the modeller becomes knowledgeable about the system being developed while developing a model of that system; i. e., the modeller is forced to encounter the same problems that he would encounter in the development of the actual system.

Let's consider a simplified example of this "forcing" nature (this occurred recently at Hughes on a program development project in which computer simulation was being used for design verification). The two program modules shown in Figure 3, a scheduler and a scheduled application program (PROGRAM j), interact with each other through common data elements that are used as indicators for scheduling requests. The data elements are decremented in one module and tested for a "less-than" condition in the other.

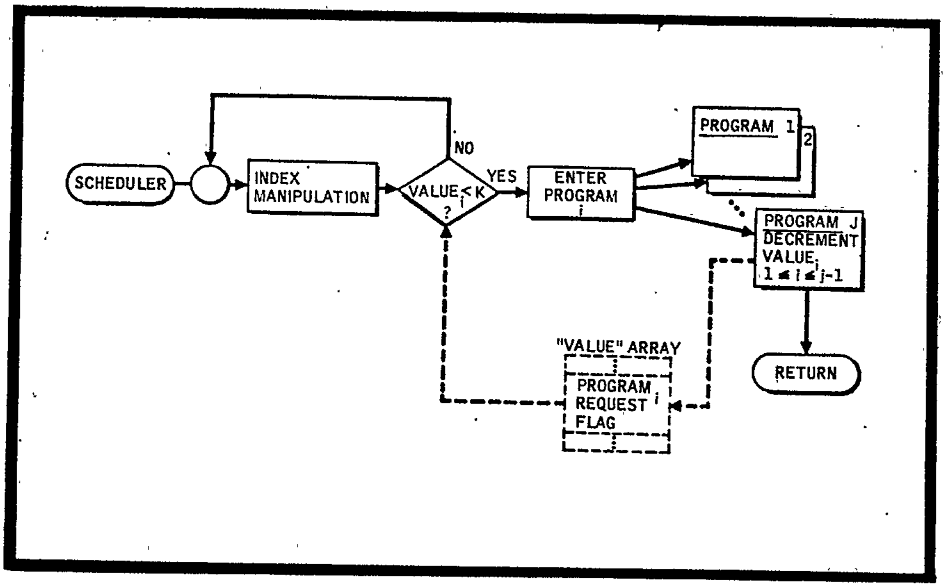

Figure 3. INTER-ELEMENT INTERACTION PROBLEM.
Unassumed negative values caused processor control errors.

A-priori assumptions led the designer to dismiss the possibility of negative values. If negative values occurred, an undesirable sequence of scheduling events would result. These elements were translated one-for-one into the model along with an independent design change which negated the designer's a-priori assumptions. When simulated, negative values occurred and resulted in the same undesirable sequence of events.

The modeller, faced with these undesirable statistics, was then forced to determine the problem and help in the design of a solution. Even though simple, this example points out how interaction problems in program development can be uncovered when using this straightforward modelling approach.

We can now appreciate the inherent ability of representative modelling to analyze a new development procedure - the consistent one-to-one translation of program module, data structure, and hardware device not only forces a characterization of the interaction between these elements, but also forces the modeller to encounter the same problems he would encounter in the development of the actual system.

In this manner, significant insight into the merits of a new development procedure can be gained from experience with its application to the development of a model. Even more, recalling that design variations are easily implemented in a one-to-one model, refinements to the negative aspects of a development procedure can be tested in the modelling environment.

AN EXPERIMENT WITH TOP DOWN DEVELOPMENT

Recently, there has been interest at Hughes to investigate new software technologies such as structured programming, top down development and process construction, in order to increase the quality of our real-time programs and reduce software development costs. Even though these methodologies may evidently be worthwhile, their evaluation in use on real-time programs and even more importantly, how they are implemented, are risky experimentations on milestone-oriented projects.

Hughes, however, has been experimenting with computer system simulation as a management tool for software development (1). Models are developed as management tools for real-time programs to provide insight into areas such as concept feasibility, design viability, integration problem analysis, and design change performance analysis. These areas of analysis by simulation are seen to overlap with the areas of interest in the evaluation of new software development procedures. Perhaps by only shifting the emphasis of the modelling activity, simulation could become a development analysis tool.

A study contract for an air operations computer system became the first experiment in using computer system simulation as a tool for development procedure analysis. The first stages of software design were carried out using the concepts of Top Down Development (3, 4) which is concisely defined by the following: * * * * * * * * * * * * * * *

A program is a tree structure whose leaves are unique modules which implement the design requirements. The program is developed in a top down manner by designing, testing and integrating each level from the top-most element in such a way that design decisions as to how the requirements are implemented are postponed to as low a level as possible.

The postponing of design decisions is aided with the use of module stubs which serve only as a point of control acceptance, possible time consumption, and control return; these stubs become the topmost elements when the current level has been verified correct.
* * * * * * * * * * * * * * *

The significant results from this experiment are summarized by the following:

1. Computer system simulation models employing the concepts of representative modelling have the ability to evaluate new software development technologies.

2. Models developed in a top down manner share in the same benefits that result from top down development of real-time programs.

THE SYSTEM MODELLED

The system being studied is a multi-node network of communicating devices employed as an air operations facility (Figure 4). There are two processor nodes, air operations control and air operations control support, which receive and process messages originating from external hardware devices, operator entry equipment, and other air operations processors.

The purpose of the system is to monitor, coordinate and control the various flight patterns by receiving and processing tracking and status information messages in response. Since the system is in the design concept phase of development, the modelling activity is employed to quantify system characteristics such as numbers and types of aircraft that can be controlled successfully, response requirements, processor capabilities, file storage requirements, inter-node data link capacities, degradation effects in system operations, and other typical system attributes.

The hardware portion of the model is translated into the simulation language directly from Figure 4. For each operator entry node, the model contains a "Processor"*which can generate messages automatically by any distribution, route message through its node along legal paths, and has logic to model operator responses. Likewise, each aircraft node is a processor which generates track and status information messages, routes messages, and can have logic which models hardware responses (e.g., "bank left"). The two air control operations processors are models of actual CPUs and memory systems which execute the software logic being

*A device which executes programmed simulation language logic.

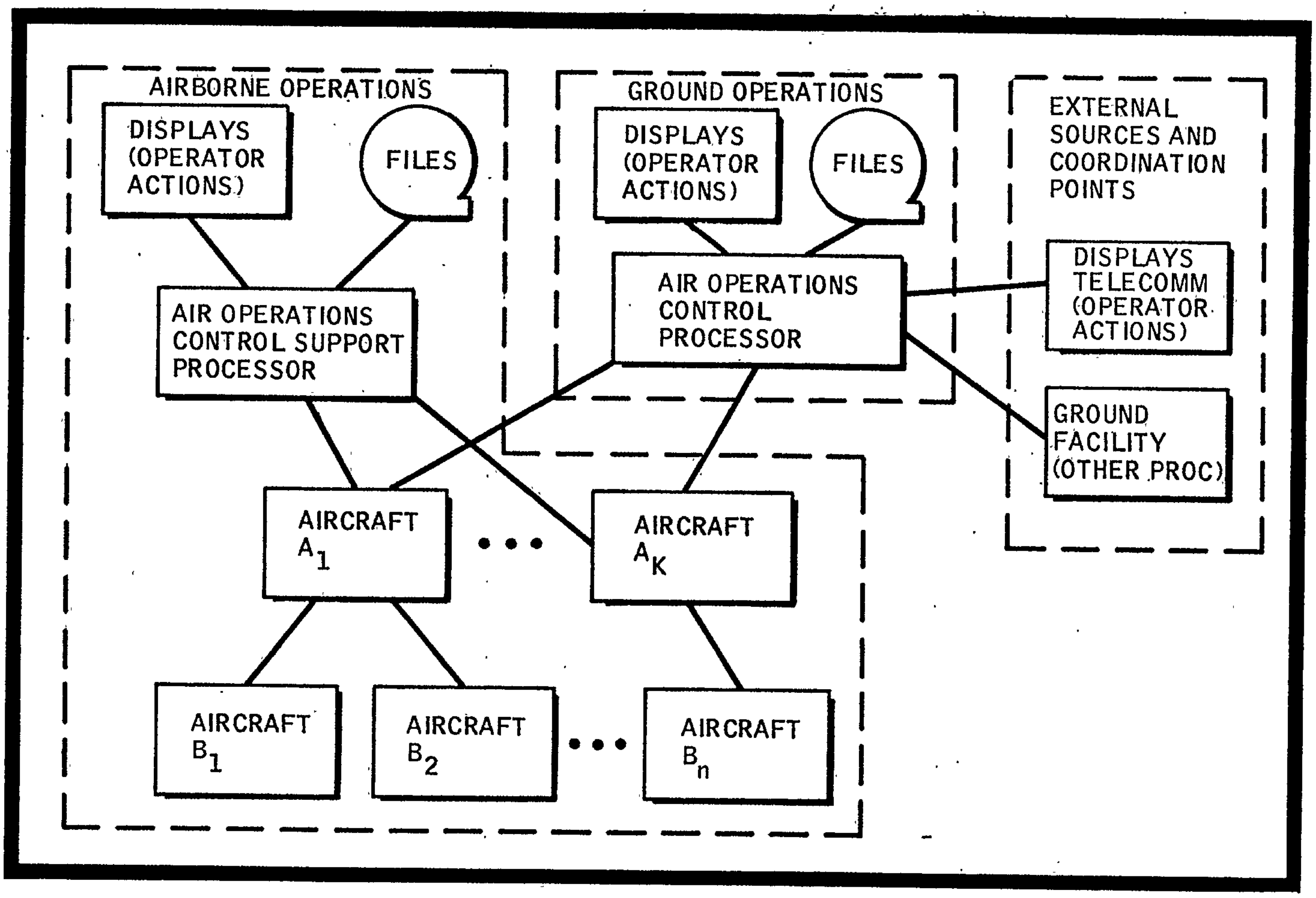
AIRBORNE OPERATIONS
DISPLAYS (OPERATOR ACTIONS)
FILES
AIR OPERATIONS CONTROL SUPPORT PROCESSOR
GROUND OPERATIONS
DISPLAYS (OPERATOR ACTIONS)
FILES
AIR OPERATIONS CONTROL PROCESSOR
EXTERNAL SOURCES AND COORDINATION POINTS
DISPLAYS TELECOMM (OPERATOR ACTIONS)
GROUND FACILITY (OTHER PROC)
AIRCRAFT A1
AIRCRAFT AK
AIRCRAFT B1
AIRCRAFT B2
AIRCRAFT Bn

Figure 4. AIR-OPERATIONS CONTROL SYSTEM CONFIGURATION.

developed as part of this study. Finally, the nodes are interconnected with channels and communication links which act as queuing devices (when busy) and transmission delays.

To drive the model, messages are defined by specifying message type, length, priority, originating node and inter-node routing information. Messages can be generated automatically by giving a message generation rate and inter-arrival modifier, or message generation can result from a system action (e.g., receipt of a message).

The routing of messages through the system is part of the model logic and is performed automatically as a result of the message routing definition. By defining sets of message definitions in this manner, the various system configurations can be characterized by message priority, by the rates of message generation at each node, and by the routing of messages through the system.

For example, the loss of the air operations control support processor can be modelled by "turning off" all message generation from within it and by rerouting messages that it should process to the second control processor.

IMPLEMENTATION

The software subsystem was designed using the above definition of top down development and translated into the model using the techniques of representative modelling. Because the system being studied was in the design concept phase, no particular language or operating system had been chosen - instead, a large subset of each was to be studied independently after questions on configuration and capability had first been solved. This was significant in the study, for this allowed the software subsystem design to proceed without a-priori restrictions which might influence the objectivity of the results. By definition, we sought first to develop Level One.

The first level software design was arrived at by placing the scheduler at the "root" of the program tree and dividing the remaining responsibilities into common programming interests (Figure 5). In order to conceptualize a real-time program in this manner, we hypothetically chose a philosophy of operation in which the occurrence of an interrupt causes a request to be logged in the scheduler for interrupt processing.

That is, the scheduler conceptually becomes the root of the tree in that it schedules all task processing - including interrupts.

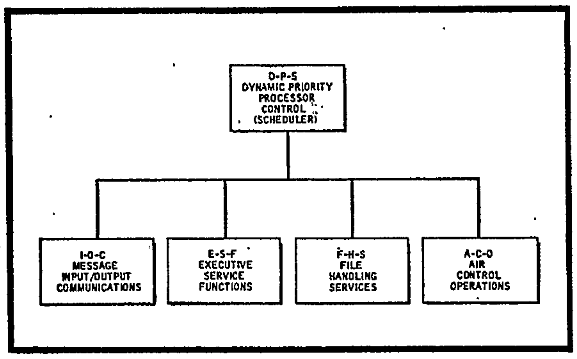

Figure 5. FIRST LEVEL ONE SOFTWARE DESIGN. The stubs are IOC, ESF, FHS, and ACO.

The four program modules IOC, ESF, FHS, and ACO were at this stage, by definition, stubs which, when called, would consume time and return to the scheduler.

By the definition of top down development, the next step is to design and test this Level One architecture. Since no OS had been specified, the resulting scheduler, DPS, was designed to be a "best case" scheduler of single-entry task requests - a dynamic priority scheduling scheme in which the priority of a request is determined (dynamically) when it is generated (5).

The resulting design was translated directly into the model in order to be tested. At this point it was obvious that just defining DPS, itself, would not be sufficient. The scheduling queue, actual task requests, a program module entrance scheme, and at least a gross data structure would be required to be able to execute the model. Therefore, there had to be parallel software and model development in the areas of data structures and programming standards.

Since the hardware and load definitions for the model were being developed in parallel with the software, message inputs were processed and used as a task request generation scheme (Figure 6) - this necessitated a deviation from the strict top down development process in that the design of message I/O handlers (message processing submodules of IOC) was required.

Finally, the configuration shown in Figure 6 was exercised until designed properly and validated for correct control sequencing between the Level One program modules.

The next step in the development process was, by definition, to expand each Level One stub into a Level Two node with submodule stubs. At this point, and throughout the remaining parallel development of software and model, it was recognized that the structured design of the software and subsequent one-for-one representation in the model was causing a structured design of the model!

The software was being designed, translated into the model, then tested for design completeness - without having to code the actual system. Even more, the quality of the design process itself was being evaluated automatically in the design process of the model - a natural evaluation tool for development procedures.

The final design of the software subsystem reached three levels, having a total of 53 program module stubs. Because of the necessity for a complete design of a module at stub level, some of these Level Three modules implemented a system design requirement (a leaf on the program tree).

The difficult implementation aspects of top down design seemed to be at Level One. At this level, the philosophy of operation had to be decided; module interfacing, data structure, and execution sequence were part of this decision. As stated below, once these problems

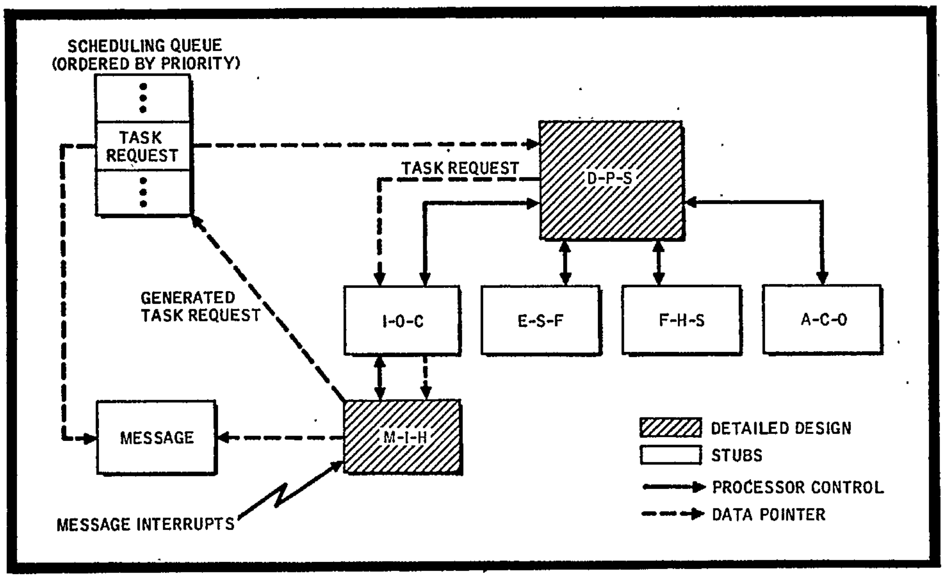

Figure 6. FINAL LEVEL ONE SOFTWARE DESIGN. A Level Three module (MIH) was necessary to test DPS design.

were resolved, the succeeding levels of development proceeded with few problems.

RESULTS

Some observations made during the evaluation of the top down development process were:

The "Root" of a Real-Time Program - Considering the scheduler, the root of the program tree seemed to correspond with most functional definitions of structured programming and fit very well in this real-time program design effort.

Design Viability - The success of a design seemed to be dependent on the ability to predict interface requirements for programs at a lower level. Because of this, the model employed a message data structure whose contents were dependent on the varying requirements. A pointer to the message was passed to every successor-level program. As requirements were uncovered, message parameters were defined.

Design Verification - The verification of each level design is a testing of the control structure. A similar level of design for the data structure is therefore necessary.

Programming Standards - The top down design of the software system caused a top down establishing of programming standards, restrictions, and assumptions. This top down approach caused a uniformity among the lower-level program modules which resulted in many fewer violations of individual module standards. Furthermore, the top down establishing of standards aided in design by presenting a common base of assumptions from which to work.

Design Expansion - As stubs became the current design level, their expansion (design) was made much easier knowing that all higher levels were verified. This knowledge permitted the designer to concentrate on his individual task as if it were the only program in the system.

Program Management - The expansion of stubs into submodules using the criteria of common programming interests established a program of development management structure (criteria for these expansions are discussed further in (6, 7, 8)).

Parallel Data Structure Development - In parallel with the model software design, the hardware, load and data structure designs were accomplished in a similar structured manner. Like benefits occurred in these areas.

Design Feasibility - A recurring problem with this development process seemed to be that at Level n, for example, the requirements of a detailed program design at some level $m>n$ were needed. Time permitted no other solution than to deviate from a strict structured approach and design the required module. This is an area that needs further study.

TOP DOWN MODEL DEVELOPMENT

As stated previously, the top down development of a real-time program and subsequent one-to-one translation into simulation language terms causes a top down development of the model. Though this new development technology may have been established without regard to computer system simulation models, there are evident independent benefits in its use for model design and development.

After all, models are nothing more than software programs themselves which, in development, must face many similar problems as in real-time programs (indeed, the reason why models can serve as development analysis tools). Hence, models can not only be used to evaluate design procedures but can also share in the benefits of better software quality by using the procedures which are being evaluated.

The experience gained from the experiment in top down development has led this author to now proclaim a new technique in model construction (in addition to representative modelling). Of course, if in the design of the system being modelled top down development procedures are used, then the model will be developed in a like manner if one-to-one translation techniques are used.

But often, models are after-the-fact tools of analysis (as opposed to the use discussed in this paper for design analysis) and, as such, must be constructed using existing system descriptions. In these latter cases, top down design of models can result in the same benefits as those gained in top down design of real-time programs; i.e., better software quality, reduced costs, and more credible results.

Is it possible to develop a model in a top down manner when the system has been completely designed and specified? In two recent experiences in which models were being developed from existing system descriptions, the experience gained from the air operations model was used successfully in specifying a top down model development procedure.

Recognizing that the levels of development are actually levels of control structure, one is then able to "step back" and pull out succeeding levels of control structure starting, of course, with Level One.

Each level is completely designed (in Model language), tested, and verified before going to the next level just as if the system were being designed again in a top down manner. But, the expansion into next level nodes is a translation from the system description into the model rather than a design.

CONCLUDING REMARKS

In this author's opinion, the most important result from the use of top down development is the assurance of program correctness which results from verification at each level of design. The author's experience in several simulation studies and in the development of a real-time program has shown that models tend to be much more difficult to verify for correctness than the programs which they represent. And yet, simulation results often plan an important role in the decisions for expensive computer system design changes.

It then seems to be indicated that the processes of top down development should be standard procedures in model design in order to obtain more credible computer simulation results.

The experiment discussed in this paper, even though not of significant magnitude, led to significant results both for real-time program development and for simulation model development. It would seem likely, then, that more significant results are forthcoming if models are used further in this same endeavor.

BIBLIOGRAPHY

1. Camp, John W. and Sullivan, R. L., The Use of Simulation in Real-Time Program Development, Proc. 6th Annual Simulation Symposium (1973) pg. 127
2. Aviation Week, pg. 31, February 9, 1970
3. Mills, Harlen, Top Down Programming in Large Systems, IBM Federal Systems Division
4. Liskov, B. H., A Design Methodology for Reliable Software Systems, Proc. FJCC (1972) pg. 191-199
5. Liu, C. L. and Layland, James W., Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, JACM 1 (1973) pg. 46
6. Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM 12 (1972) pg. 1053-1058
7. Baker, F. T., System Quality Through Structured Programming, Proc. FJCC (1972) pg. 339-343
8. Dijkstra, E. W., The Structure of the "THE" Multiprogramming System, Comm. ACM 11 (1968) pg. 341-346