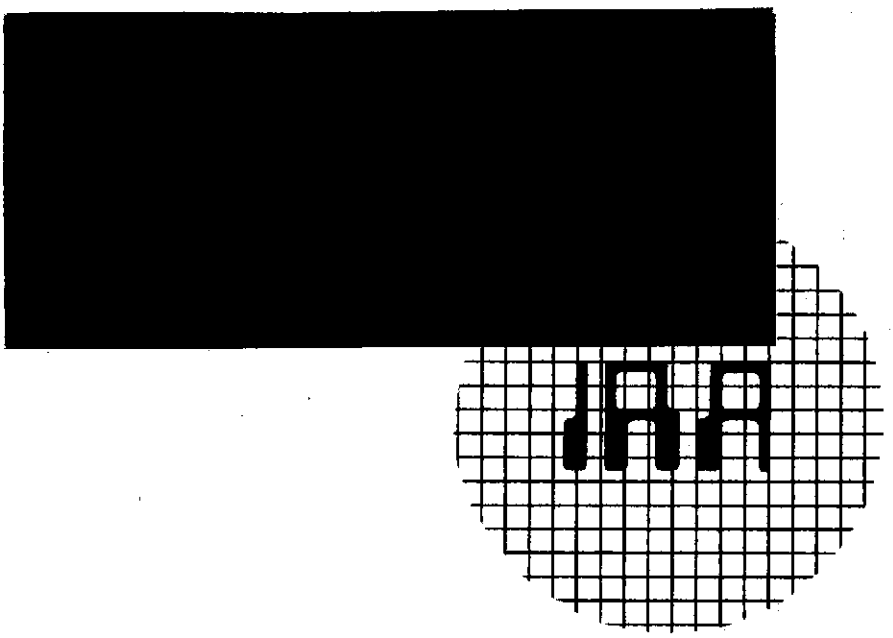
(NASA-CR-120206) SPECIFICATIONS AND PROGRAMS FOR COMPUTER SOFTWARE VALIDATION Final Report (Information Research Associates, Inc.)

N74-21831

CSCL 09B G3/08

Unclas 16809



INFORMATION RESEARCH ASSOCIATES

Final Report

Contract NAS8 28084

Specifications and Programs for Computer Software Validation

Information Research Associates

2200 San Antonio

Austin, Texas 78705

November 20, 1973

Authors

J. C. Browne

R. Kleir

T. Davis

M. Henneman

A. Haller

G. L. Lasseter

Table of Contents

I. Introduction

The output of this contract has been the development of three software products, the Specification Language System, the Fortran Automatic Code Evaluation System and the Array Index Validation System, all of which have been delivered and installed upon the Marshall Space Center, 1108 computer facility and a report covering the design of a hardware/software facility which is capable of simulating code execution in I/O logic etc. of various computer architectures. The format of this final report consists of the detailed study which has obtained through the hardware/software simulation validation complex and small sections detailing the conceptual and functional capabilities of three program products. Detailed documentation of each of the software products was delivered at the time of installation. Reference to this documentation will often be made for elaboration of the summaries given here.

II. Fortran Automatic Code Evaluation System

A. Introduction

A complete description of the Fortran Automatic Code Evaluation System (FACES) including the general description, directions for use, the tables produced by the system and its file structure was delivered with product installation.

FACES is designed to serve as an automatic aid in analyzing and debugging Fortran programs. FACES is a software package which takes as input a Fortran program which may contain many modules (subroutines and functions). The system is composed of two main parts, the Fortran front-end which gathers information about the input program and a set of routines organized as a diagnostic package which evaluates the information and prints warning messages concerning actual or potential errors. All information for the analysis is at present obtained statically from the source code of the Fortran program. Extensions

to the system can be made to include dynamic traces and the run-time analyzers.

The Fortran front-end scans and parses the Fortran input program, gathering information about the source code as parsing is done. A graph structure of each routine analyzed is formed and information concerning interface between routines is gathered. This information is then stored in sets of tables. The second portion of the FACES system consists of a set of diagnostic routines which analyzes the information that is stored in the tables looking for possible danger signs and analyzing for particular types of problems in a program. Each routine may be optionally chosen for execution by the user. The four diagnostic routines developed under this contract are i) verification of correct parameter alignment between routines, ii) verification of COMMON block alignment, iii) analysis of variable initialization, and iv) a trace of the future and history of specified variables.

B. Diagnostic Routines

1. PARAL

The diagnostic routine PARAL may be used to check alignment of all parameters in SUBROUTINE and FUNCTION calls. PARAL is invoked by the statement CALL PARAL (IATOPT) where IATOPT is an option chosen by the user indicating whether array parameters are to be checked for equal dimension. If IATOPT equals zero array dimensions will not be checked. Otherwise array dimensions will be will be checked. PARAL will check the alignment of all argument lists and calls to subroutines and functions that have been analyzed by the Fortran front-end. Each argument list will be checked against the defined parameter list to insure that the following conditions exist:

a. Corresponding parameter lists have the same number of parameters.

b. Corresponding parameters within each list are of the same type.

c. Corresponding array parameters with each list have the same dimensions.

2. TRACEY

The diagnostic routine TRACEY may be used to determine if all variables are initialized before being used in a manner that might presume prior initialization (called an input usage). Those variables which are in COMMON or in a DATA statement or always used as entry parameters are assumed to be handled correctly. The routine is referenced by the statement CALL TRACEY (MDNAM1, MDNAM2). The parameters MDNAM1 and MDNAM2 contain the name (four characters per word, left-justified and blank-filled) of the routine to be analyzed.

Input usages are those in which the variable effects:

a. the value of any other variable

b. the flow of control of the routine

c. the output

From each input usage a backware trace is performed along all possible entry paths. Each such path must pass through an initialization of the variable or a diagnostic is produced.

3. COMBAL

The diagnostic routine COMBAL may be used to verify the alignment of all common blocks in the routines that have been analyzed by the Fortran front-end. It is referenced by the statement CALL COMBAL (NMCK). The parameter NMCK allows the user the option of specifying that the variable names in the COMMON block will be checked for alignment. The alignment conditions that are checked for are the following:

a. Corresponding COMMON blocks must have the same number of entries.

b. Corresponding elements within each COMMON block must have identical dimensions.

c. Corresponding elements within each COMMON block must be the same.

d. Corresponding elements within each COMMON block must have identical names (only if NMCK equals 1).

COMBAL produces output for each COMMON block. The output consists of the COMMON block name, the routines in which it appears and when the above alignments are not met, diagnostic messages associated with these routines.

4. PATHS

The routine PATHS may be used to either trace the history or chart the future of a value of a particular variable at a particular statement in a program. The call statement for PATHS is CALL PATHS (I, MD1, MD2, IV1, IV2, NODE). MD1 and MD2 refer to the name of the module to be considered. IV1 and IV2 contain the name of the variable to be traced. NODE is a statement number from which the trace is initiated. I is a parameter specifying either the backward or forward trace. If I is zero, PATHS will produce a list of all variables which might have effected subject variables value at NODE and associated with each variable the node at which re-evaluation is noted. If I equals one PATHS produces a list of all variables which may be effected by the subject variables value at NODE and the associated node for each re-evaluation. Variables may appear more than once in the output if there is more than one associated node to be listed. It should be noted that the value traced is the value of the subject variable immediately prior to the execution of the statement number NODE.

III. Specification Language System

A. System Overview

The objective of the specification language project is to allow a programmer to verify his coding, by comparing a specification of his algorithm to the object code ultimately produced by the compiler from the program he writes. In order to accomplish its task, the system is divided into three phases:

1. The specification, which is a formalization of the algorithm's flowchart, is given to the system for translation to an internal graph-structure representation. This graph is "folded" to produce a minimal graph.

2. The object code produced by the Fortran program is decompiled into the same internal graph representation, and is again folded.

3. The two graphs are compared, to insure that the graphs are identical, and to assure that the actions along the graph arcs are compatible.

B. System Status

1. Specification Language Processor

A complete description of the Specification Language is given in the previously submitted documentation. The program to translate Specification Language input to internal graph representation is completely operational. It produces as output two data structures: the graph itself, and a list of reverse Polish commands which represent the actions to be performed along the arcs of the graph. Within the graph, a node is considered to be a machine state, and an arc going from one node to another represents the action which must be taken in order to effect the corresponding state change. The graph structure is described in detail in Section 2.c of the internal documentation for the Flowchart Translator. The reverse Polish code is described in Section 1.d in the same document and in other places referred to by that section.

2. Graph folder

The graph folder, also complete and operational, produces from an internal graph representation of an algorithm, a "minimal" form of the same graph. That is, the graph produced by this phase retains from the original graph nodes of only four types: (1) nodes with no predecessor, (2) nodes with no successor, (3) nodes with more than one predecessor, (4) nodes with more than one successor. When a string of one-entry, one-exit nodes is encountered, the actions necessary to make the set of transitions are packed into one arc, the arc coming from the previous "important" node. In this way, redundancies in representation are eliminated, and the need for storage is minimized. The action representations, however, are kept in their original form for use at the compari-

son phase.

3. Decompiler

It was not possible to produce a decompiler which was as complete as had been originally hoped. There were several reasons for this, the primary one being the lack of system documentation by the Univac Corporation concerning its compiler and its relocatable code format. The fact that the Fortran compilers were of different versions at Marshall Center and at the computer installation on which we developed the programs was also a factor.

The decompiler is neither complete nor "perfect". It is rigorously provable that to produce a "perfect" decompilation of Univac 1108 Fortran V object code is impossible. One example of a feature which prevents perfect decompilation is the compiler's use of temporary locations. When a subroutine call, such as CALL SUB (I + 7) is made, the value "I + 7" is computed and stored in a temporary. It is impossible to tell -- given only the object code -- whether that computation actually was the compiler's use of a temporary, or the programmer's having pre-calculated the value and placed it into his own temporary location.

Sections of the decompiler which are incomplete include the handling of complex and double-precision arithmetic. This arithmetic is handled, but in a very simple manner, as if the values were single-precision real numbers. The Univac 1108 Fortran V "FLD" function is not provided for. The primary reason is that this function is not in ANSI Fortran; because of this omission, the only shift instructions which are decompiled are those which shift through 36 bits -- thereby producing a register to register transfer -- and those which effect a multiply or a divide.

Finally, because of the temporaries problem, it was decided not to expend the effort involved in doing an analysis of the temporary locations. The temporary storage analysis would have been prohibitively complex, and (as

previously stated) could never have been complete. The primary result of this omission is that when a value is stored in a temporary location, the decompiler assumes that that store is into a program variable. An extra node and arc are therefore produced in the graph, representing the store operation which the decompiler thinks it found. If a complete analysis of the temporary locations were possible, this store could be treated as a store into another register, and code not be generated at that time.

4. Graph comparer

Because of the limitations on the decompiler, the graph comparer was also produced in a restricted form. Since for the more elaborate arithmetic expressions good code generation may not be assumed, the rigorous comparison of actions would not be meaningful. Therefore, when an arc is to be compared to an arc on the corresponding graph, the only thing that is checked is the code representing the type of the action. For example, if a 14 (assignment) matches a 14 on the other graph, the comparer is satisfied. There are problems with this approach, but given the time allocated and the decompiler restrictions, it was felt that to do more would not have been justified.

The program is complete with respect to actual graph comparison. The heart of the comparer is a section of code which can call itself recursively; when called, this section is given a pointer to an edge in the graph. It will then compare that edge, and all succeeding edges, down to a node which has more than one exit (that is, a decision node). At that point, it will first recursively call itself to establish a correspondence between the edges out of the node on one graph and the edges out of the node on the other. Once this correspondence is established, it must then recursively call itself once again for each edge out of the multi-exit node, to effect a comparison between the corresponding edges and all <u>their</u> successors down to a multi-exit node. (Etc.)

The error messages produced by the comparer are listed and des-

cribed along with a description of the control cards necessary to operate the entire system in the documentation previously submitted. Note that because of the decompiler limitations, error messages will be produced which are not really there. For example, since a storage into a temporary location will produce a node and an arc on the Fortran graph, the lack of a corresponding node and arc on the flowchart graph will produce an error. The comparer will, however, be able to recover from this error and continue its comparison unhampered.

C. Conclusions

1. Specification Language

a. The Specification Language can be a valid tool in the representation and validation of algorithms. In its present form, with punched-card input, it is not as effective as it might be.

b. Flowcharts are essentially graphical entities; therefore, input to the system should be graphical.

c. In order to effectively specify an algorithm, facilities must be provided to specify every detail of it. (This version omits such details as formats.)

2. Decompiler

a. To produce a "perfect" decompiler is impossible.

b. To produce a complete decompiler would require much more effort than was budgeted to the entire project for this year.

c. For working, tested Fortran compilers such as that on the Univac 1108, the effort which must be dedicated to producing a decompiler is not justified. With a probability far greater than .99, any errors which are introduced into algorithm are introduced before compilation. With new untested compilers -- such as that for the SUMC -- however, decompilation might be a valid tool for verification.

d. To produce the best decompilation possible, the original com-

piler should retain and pass on information about how it did its job; for example, its name table would be extremely helpful.

3. Comparison

a. It is possible to compare program graphs in a straightforward manner.

b. To compare program actions, however, requires far greater effort. Many compilers use special "tricks" wherever they can. The sign of a value is particularly hard to pin down; for example, the 1108 Fortran statement "A = B - C" produces the following code: load C, subtract B, store negative into A.

IV. Array Index Validation System

A. Purpose

The purpose of the Array Validation System is to determine the validity of array references within a DO loop. The system uses output in the form of tables from FACES. The tables used are SYMTAB, USETAB, etc. These tables are described in the documentation provided for FACES.

B. Capabilities

The capabilities of the current system are limited to checking array references which use DO loop indices as subscripts. This includes the following two cases:

Case 1:

```
      DIMENSION ARRAY (5)
        .
        .
        .
      DO  10  I = 1, 6, 2
   10 ARRAY (I) = - - - -
        .
        .
        .
```

In this case the terminal parameter (upper bound index) is greater than the declared dimension of the array. This problem is handled in the following

manner. The following formula is used to determine the number of times the loop is executed:

DO _____ I = K, L, M

$$\text{Max}\left(1, \left\lceil \frac{L - K + 1}{M} \right\rceil\right). \qquad (1)$$

Combining this result with

$$K + (\text{Max} - 1)\, M \qquad (2)$$

yields the highest value the array index can achieve.

```
Case 2:     DIMENSION ARRAY (5)
              .
              .
              .
            DO  10  I = IV1, IV2, IV3
         10 ARRAY (I)  = - - - - -
```

In this case the program derives sufficient conditions for non-violation of array bounds and prints them out.

V. SOFTWARE VALIDATION COMPLEX

Objective

The Software Validation Complex (SVC) is a tool designed for enhancing the reliability and performance characteristics of a computational system. These design goals require investigating the operation of hardware, software, and the cooperative environment within the objective system. The final result would be a test bed environment for the operation and instrumentation of a projected system. Considerable attention is required to operational philosophy before the physical system can be realized. The resulting system is expected to mechanize the development of future systems and to change existing systems.

Within this global frame work, the investigations to be reported in this section involve the groundwork and first attempt at the design of such a system. The subjects to be discussed encompass the topics of:

1) Investigation of computational processes
2) Identification of feature expression
3) Implementation requirements
4) Experimental system analysis
5) Proposed extensions to current results.

In modeling the computational system, it is desirable that the exposition adjust to the detail level desired by the investigator. The features to be examined should be presented with the fidelity

present in the actual system, but unnecessary details below the level of inspection should not impede the analysis process. If possible, the model should telescope automatically to the desired level allowing reversibility to detailed investigation if necessary. Furthermore, the entire process should not be required to be in one particular level of detail; rather, the portions not under investigation should contract to the most computationally efficient form to expedite the investigation.

Results from the model can be used to institute a modular simulator for support activities on modular machines. Just as the actual hardware is physically constructed, so should the support simulator. To institute this aim, some design discipline must be exercised in the production of the simulator. Interface requirements must be identified and undesigned portions accommodated. This approach would track the development and permit rapid reaction to design changes in the actual system. The final version of the resulting simulator would closely replicate the final hardware version.

Toward these ends, the control of the system must be vested in the user through some more transparent medium than an elaborate design language. The user should be free to incorporate useful constructions into the system as required to perform tasks considered clumsy with the existing constructions. Rules for the incorporation of new building blocks should be simple and clearly explained. Supervisory system requirements should be driven from the user's description of

the target process rather than requiring explicit supplemental information auxiliary to the target process description.

To accommodate the modularity requirements and construction simplicity, a representation format was chosen which implicitly indicated the construction through local connections among modular units. Input and output interfaces are manifested by connection omission in the description. Automatic recognition of these points assists the integration of independently developed subunits.

II. Principles of Operation

To realize the goals of flexibility, modularity, and construction ease, some aspects of computer system operations must be investigated. The results of this investigation are coupled with operational philosophies commensurate with distributed control of the system. This permits the user described target system to operate on a flexible host machine capable of reconfiguration to fit the target system under inspection.

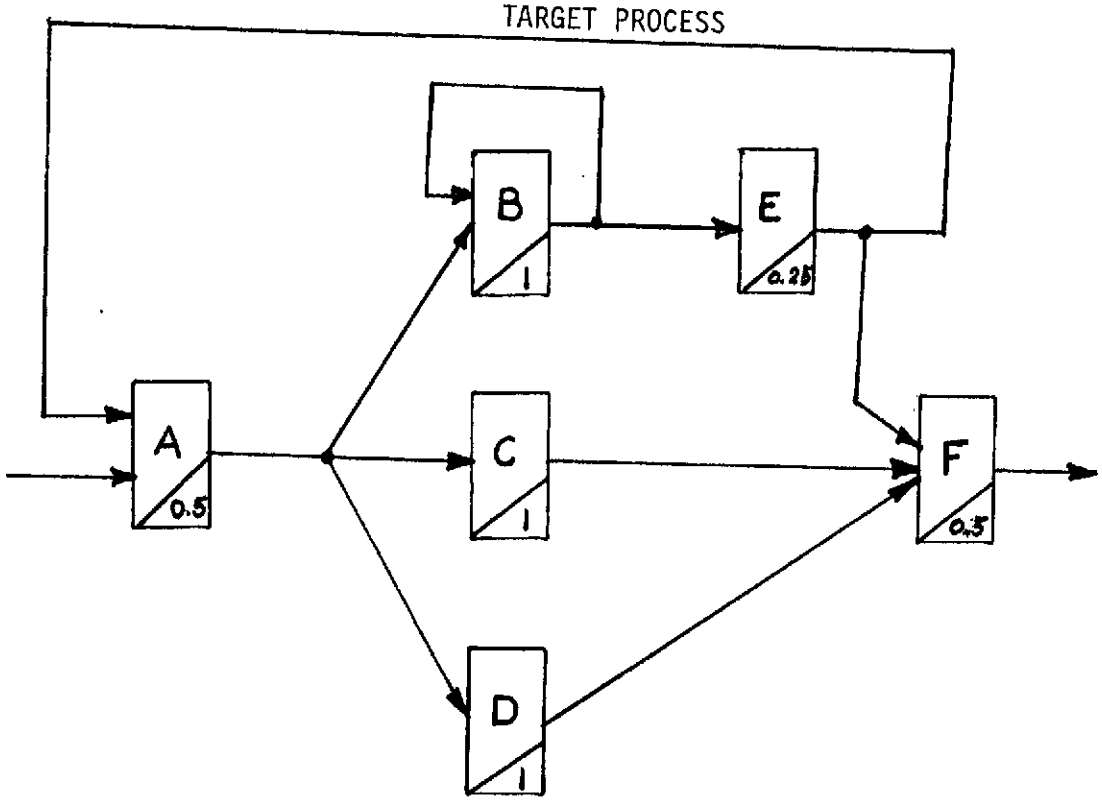
Characteristics of Computational Processes

Computational processes, whether implemented in hardware or software, share common characteristics. The medium employed, however, affects the operational methodology and attack even when the purpose is similar. These media differences contribute to the operational characteristics and limitation frequently observed in hardware or software systems. In an effort to avoid these pitfalls, the features of computational processes were examined to identify desirable properties which might serve the objectives of SVC and origins of failure mechanisms to avoid.

The primary objective of any computational process is to move information through a set of transformations which produce the desired results. This is accomplished by

1) Establishing a network of potential information flow among the transforming components.
2) Dynamically controlling the flow to particular paths consistent with the aims of the process.

EXAMPLE INFORMATION FLOW MODEL OF
TARGET PROCESS



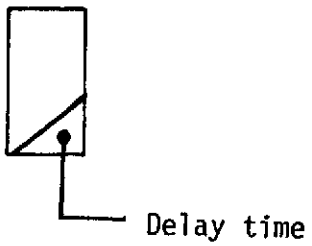


FIGURE 1

To illustrate, consider the example process depicted in Figure 1. The lines correspond to potential information flow paths while the boxes represent transformations performed on the information present on incoming lines. This format represents only the potential for flow in the process, much as the syntax of a language indicates potential legitimate sentences of a language. For example, it is impossible from the static representation of the process to determine whether blocks B, C, and D operate simultaneously, or by some prior agreement (i.e., control), only one operates at a time. Similarly, this static representation would be the same for both a serial system in which information progresses from A to C to F, or for a parallel system in which A, C, and F are all simultaneously active on different information moving through the system in a pipeline fashion. Notice that the information flow diagram of Figure 1 might represent any of the three programs of Figure 2.

The actual flow of information is governed by a control structure. Conceptually, actual information flow occurs at the intersection of the control system and the information flow paths. For example, one process might assign value to a quantity which appears as an input to another process. Unless the second process is activated, however, actual information flow does not occur. Rather, the information is destroyed on the next execution of the first process.

In general, hardware implementations of computational processes have enjoyed more integration success among separately developed parallel subsystems. The operational properties of hardware and software systems

```
A = INPUT + E
B = B + A
E = B**2
C = 10 + A
D = A - 5
OUTPUT = E**2 + 2*C + D
```

(a)

```
   A = INPUT + E
   IF( A ) 10, 20, 30
10 B = B + A
   E = B**2
   GO TO 100
20 C = 10 + A
   GO TO 100
30 D = A - 5
100 OUTPUT = E**2 + 2*C + D
```

(b)

```
   A = INPUT + E
   IF ( A ) 10, 20, 30
10 B = B + A
   E = B**2
   C = D = 0
   GO TO 100
20 C = 10 + A
   E = D = 0
   GO TO 100
30 D = A - 5
   C = E = 0
100 OUTPUT = E**2 + 2*C + D
```

(c)

CANDIDATE PROGRAMS FOR EXAMPLE INFORMATION FLOW MODEL

FIGURE 2

were characterized in an effort to identify the reasons for this capability to incorporate these features in software systems.

Software Properties

Software systems accomplish control through a program counter concept. The fundamental representation format of a software system (i.e., program listing) indicates the sequence of operations to be performed. There is typically a partitioning of software operations into control and computational classes. The two classes interact through conditional branch operations in which computations influence the flow of control. Similarly, control flow influences computations. Typically, the control format contains some default "next operation" sequence (usually the next sequential location) which occurs unless an override in the form of an explicit control operation occurs.

Flow of information in software is accomplished through variables. The program description representation indicated the potential flow of information through a forward chaining mechanism. That is, variables are assigned value in one location and the appearance of the variable in a subsequent statement indicates a potential coupling with the assignment. Given a usage of a variable, it is usually necessary to track the control flow to determine where the information originated. The potential information flow usually is derived by auxiliary information (e.g., a cross reference list) to aid determination of the actual flow.

Historically, the program operations have been serially executed. That is, additional instructions beyond the normal process procedure

description have been introduced to override the assumed serial operation. This has been accomplished with "fork" and "join" operations imbedded in the source code. In the absence of these constructions, it is assumed that at most one operation is active. Virtual parallelism is accomplished using partitioned data in which one task at a time is executed. Results of these partitioned tasks are subsequently used by later tasks.

Hardware Properties

Hardware systems have been represented primarily by a data flow format (e.g., schematic diagram) indicating the interconnection of hardware components. Each component in the system introduces more "variables" for holding information since electrical considerations reduce the possibilities for sharing these resources among several modules. When sharing does occur, some implied protocol governs the shared resource (e.g., wired ORing or ANDing). Typically, each unit in the hardware organization operates in a combinatorial fashion, reacting to input stimulation. This produces "event fronts" which ripple through the interconnected network producing the desired results at the output lines.

Control of hardware processes is accomplished by two methods:

1) Iterative stable states
2) Time advance.

A stable state is achieved when the input change of all networks causes the same computational result to appear on the output as the current

value of the output. In effect, the event front arriving at networks are suppressed since the resulting output remains the same value. Inductively, the absence of change implies that all downstream components will maintain their same values, hence the combinational results will stabilize. It is easy to see that in a large network without feedback, this stable state is eventually obtained.

With feedback present, the second control mode is required. In time controlled circuits, the stable state is forced by some circuit which relies on the passage of time rather than formal electronic input to cause an output change. These components, commonly clocks, have distributed outputs such that at a minimum, all feedback circuits subject to unstable state transitions have their feedback temporarily disconnected. The disconnection is produced by identifying some function of the clocks to predominate in the switching circuit. For example, a clocked register will not respond to input data line changes unless the clock is "on" (i.e., the output value remains the same if the clock is "off"). In effect, the change of time causes the hardware network to stabilize by selectively causing suppression of the event fronts.

Control of the hardware process displays characteristics of a time window in which a subsection of the system is active at any particular time. Usually, the portions that are active in adjacent time slots are also physically adjacent in data coupling, allowing each section to produce results which are subsequently used by the next section. This type of operation is called "phasing" such that results move from one station to the next in adjacent time slots. Through

subordination of some clocks to other clocks, a hierarchy in the control structure can be instituted to any level of structure. The interaction in this hierarchy is usually depicted in a timing diagram for the system in which the time ordered sequence of the clocks is illustrated. This information, coupled with the data representation schematic, allows the investigator to trace the dynamic action in the machine.

Hardware systems, unlike software, are intrinsically parallel in operational nature. Considerable design effort is required to accomplish the phasing in hardware systems (e.g., achieve the sequential behavior present in software). Furthermore, for some span of time (called the delay time), the inputs to a hardware module and the output values emanating from the module are inconsistent with the switching function. Although inconsistency is also present in software systems (e.g., a subroutine during execution), the event is rarely considered significant since control is usually not available to execute any external activity. (Note: This phenomenon is significant when parallel processors are used. It leads directly to Bernstein's parallelism conditions.)

Program Integration Error Sources

In an effort to identify the problem sources in integrating separately developed software modules, some investigation of trouble sources indicate features to be avoided in establishing operation principles on the SVC. These limitations contribute to malfunctions in both the initial system implementation and in malfunction experiences

during extension and modification of existing systems.

1) Program Variable Usage. In general, the storage of information in software systems is not a one to one mapping onto the variables used to physically hold the values. That is, in hardware, introduction of new components automatically introduces additional "variables" (i.e., signal lines) which must be deliberately joined with existing lines to cause interaction with previous results. In programs, concern for space economy promotes the reuse of previously established variables in the system. This results in discarding information at points in the program where it might be later required. Similarly, the intermodule communication methods through passed parameters and common data promote the undisciplined modification of passed information. Reducing variable space requirements through reuse of storage locations is similar to minimization of switching functions. The resulting implementation is efficient for the application at hand but very difficult to extend and modify.

2) Program State Implication of Program Counter. To identify the state of the process in software, the current location of the program counter often serves to represent the aggregate of all conditional testing previously performed. That is, since the program is executing at point A, all logical conditions leading to A must have been satisfied to get there. Frequently, the variables used for state testing have been modified. Retesting the state or ascertaining the conditions present at that point becomes difficult. It is not surprising that logically inconsistent processing may be introduced in the system in

this environment.

3) Failure to Grant Control. To modify the software system, correct calculation code must be inserted and the programmer must assure that control is granted to that section at the appropriate time. Thus, there is a twofold opportunity for error: 1) malformation of the computation being introduced, and 2) incorrect warping of the existing control structure. In hardware, the second effect is moderated since the introduction of additional computations is driven by the combinatorial events at the terminals of the new processor.

4) Incompleteness of Process. The completeness of a software system is expressed by absence of code for the "don't care" condition. In effect, the absence of code indicates that an occurrence is not significant in the system. Thus, it is impossible to distinguish between those activities which are legitimately ignored and those which result from omissions in the design. In particular, programmers tend to think only in the direction of positive occurrences. That is, activity is triggered when some conditions occur. The activity required when the event reverses is frequently not treated. Consider the case of a decode function in which one of several lines is set depending upon the input value. While it is obvious that the line corresponding to the current line must be set when the appropriate input appears, reset of the previously set line may receive secondary consideration. In fact, many cases can be presented in which the necessity for reset can be programmed around.

Basic SVC Concepts

From the observations presented, principles of operation in SVC were defined to avoid some pitfalls. These decisions strongly influence both the reliability of constructed systems, flexibility of modifications, and distribution of the computational workload on an expandable host machine. In general, operational characteristics were adopted which most closely approximate the activities found in hardware since these display more natural integration, parallelism, and modification properties.

1) Paramount Fidelity. The consideration for operational fidelity of the target system was considered of primary importance. In particular, fidelity was considered more important than efficiency or minimum simulation cost. For this reason, the current system contains a deliberate excess of activity to avoid suppressing values which might be required in a modified version of the process description. Optimization to reduce the excess activity is considered a postdescription phase.

2) Event Stimulated Simulation. The prime moving control mechanism in the SVC description is the change of value. If some information link changes value, activities connected to this information are initiated. Thus, the incorporation of new activities is reduced to inserting them in the appropriate network location and allowing their locally contained operational characteristics and activity at the interface to initiate the new process.

3) No Distinction between "Data" and "Control" Information. In simulation systems which attempt to partition the control and data functions of information lines, there are occasions in which process expression becomes extremely awkward. For example, consider an iterative process controlled by the error between the current solution and the desired value. The current result is both data and control since it determines both whether another iteration is required and what value is to be passed on to later sections. This is not to say that strict partitioning of control and data cannot be used on SVC, but rather, it is not a process requirement. The concept of "control" appears to be a mutual agreement among modules that certain values will modify the behavior of given components. It is largely a device for human conceptualization of the system and used to organize subsections into an analytically tractable form. Control is instituted in SVC through local interpretation by each module in the system. The interpretation of the incoming values and implications of output values remain with the user.

Similarly, the SVC support supervisor system is completely ignorant of the semantic interpretation among modules. It assumes that the description is a collection of combinatorial processes connected in a selfmanaging fashion. The supervisor's primary duty is to serialize parallel activity and assure the computational fidelity with the actual process. With this approach, all control functions are resident with the user who expresses the operational properties indirectly through describing the target subsystem. (i.e. no additional control information is required.)

4) Reduction of Program Counter Implications. The program counter is used exclusively to provide a physical processor to logical processes. There is no implication in the system as to the state of the target description beyond which unit is currently active. All that can be inferred from this information is that an input line to the active module has changed value.

5) Module Interface Rules. The rules for interfacing two modules under the supervisory system are very simple; the simplicity of these rules, however, precludes policing consistent semantic interpretation between the two modules. The interconnecting link must be the same size. For reasons which will be amplified later, each module must have unique output lines rather than output to other common lines. This requirement is usually satisfied by hardware components or when exceptions occur, some defined protocol (e.g., OR or AND function) occurs which encompasses all contributing module output values. Each module is prohibited from modifying "input" lines unless these are expressly connected to outputs of that module. The effects of these requirements and suggestions on relaxation of them are discussed in Implementation Considerations and Experimental Results sections.

6) The User Maintains Complete Control. The user's description of his target system is the fundamental definition of system control. No directive information is required by SVC to properly control the system. The user, at his option, may express semantic properties about the target system operation characteristics to improve simulation time,

however, such information is only a simulation performance enhancement and has nothing to do with fidelity of the operation. Auxiliary semantic information permits the suppression of system events which either are not of interest to the user or will be dynamically rejected by the target process.

In keeping with user control of the simulation, instrumentation of the target system is included with the process description. The SVC system draws no distinction between monitor and operational modules; however, the former is typified by having no output lines.

Usage Scenario

A major application of SVC is found in system development using a top down design and bottom up implementation. In this situation, the user performs an initial design which indicates the scope and nature of the resulting system. He then reviews the availability of components to implement the system and identifies those which currently exist. Missing components are constructed by connecting available components. Each subunit is individually tested and adjusted for proper operation. Integration is performed by expressing local connectivity among the separate subunits. The process continues with expanding subunit scope until the totality of the system is constructed.

To accomplish this design scenario, the user must have available information describing existing components from which to draw. The operational properties of these components depend upon the type of system he is constructing. For example, the primitive constructions for an operating system, computer hardware, queuing models, etc., would

differ substantially. It is important, however, that these components are managed in a compatible fashion to permit a heterogeneous system simulation. Appropriate semantic coupling of components in this type of system is, of course, the user's responsibility.

As the SVC is used, constructions are developed from previous designs. In some sense, the presence of these component models indicates previous design needs and some prior experience and/or availability is indirectly adaptive to the usage requirements of the installation.

The library constructions are probably provided by personnel familiar with programming on SVC and perhaps not the user. Similarly, users of previously developed library components should be concerned with the internal operations of the module. For this to be practicable, some expression of module design limits is required to prevent misapplication by a subsequent user. This characterization will also assist the designer in formulating his requirements and changes necessary to adapt modules to his needs. This liaison is accomplished by appending each library component with a simple descriptor summary indicating the interface requirements necessary to successfully apply it. When the user invokes the module, this description is compared with the interface information of the user's description to insure the module is being properly used.

The user's target system description contains both the functional components of the projected system and instrumentation modules for his behavioral investigations. While this description is most expedient with an interactive system, the basis of the description is local

interconnection of modules drawn from the library. The description in symbolic form is automatically translated into a host runnable form which executes under the supervision of the host executive. The description can be rerun without translation with different input data. If the system is to be modified, the symbolic form is adjusted and retranslated. When the subsystem is satisfactorily designed, instrumentation is removed from the description and the subsystem is stored away pending integration with other sections.

With the intended usage scenario and principles of operation established, actual techniques required to implement this system are presented in the next section.

III. Implementation Considerations

To bridge the gap between previously presented concepts and an operational system, considerable attention must be directed toward further foundations required in applying the host language and host machine. These factors impact the system's actual utility from the user's standpoint. The analysis of this section produces postponed decisions necessary to produce the experimental system presented in the next section.

Distribution of Responsibilities

Serial Execution of Parallel Processes

Because the intent of SVC is to permit a general purpose simulation test bed, even with a flexible host configuration, at some point the target process degree of parallelism will exceed that of the host. Thus, the simulation system will have to perform parallel processes in a serial fashion. Care must be taken to assure that simultaneous target events correlate with results in the actual target process.

To illustrate the requirements, consider the activities in a typical target transaction on SVC. Assume the simulated target process is that illustrated in Figure 1. Suppose further that module A has just completed its computation and the evaluation has resulted in a different value from the previous output. In general, this change requires that modules B, C, and D be executed to produce their result. From previous discussions in the last section, we do not know from the

potential information flow diagram whether all three or only one of the connected modules should execute. Each module of the network has an associated delay time. Suppose that B, C, and D all have different times and that module F expects a particular arrival pattern at its inputs. Thus, after the execution of the appropriate modules in the set of B, C, and D, the arrival pattern to subsequent modules must be coordinated. In general, the execution sequence must be managed and the data values on the various target process lines must be correlated with time.

Since, by assumption, the supervisory routine has no knowledge of module semantics, the determination of input value significance will be made by the individual modules. That is, when an input change to a module is detected, the module is executed. A portion of this execution is an activation analysis in which the module determines whether an execution of the mathematical function is appropriate. If an execution is not appropriate, control is returned to the supervisor. If execution is necessary, the function is evaluated.

If the result of the evaluation is different from the current output, successor modules must be permitted to activate. If the evaluation produces the same value present on the output line, no downstream activity is required; control is returned to the supervisor. Notice that the supervisor is never aware of the difference between modules which choose not to activate and those which activate and produce the same result.

Similarly, it is important that the individual modules not be required to know the connectivity of the processing network. This assumption greatly simplifies coding general purpose routines for use in the SVC library. Target process connectivity is managed by the resident supervisor.

Recognition of changed output values would be most easily performed by the functional module. Individual modules are each aware of the number of outputs and the data format of each output. Lacking global connectivity information, however, the module can take little action. Furthermore, incorporating the same change analysis code in each functional module would increase the code bulk considerably. Therefore, a compromise was instituted to interface all functional modules with the supervisory routine.

A service routine is called at the completion of evaluation execution; it compares the resulting evaluation with the information presently on the corresponding output. If they differ, the service routine informs the supervisor. Control is returned to the processing routine after change inspection. This exchange occurs for each functional module output.

Data Management

When parallel processes are serialized, considerable attention is required to avoid the untimely destruction of data values. Suppose, for example, in the process of Figure 1, that A and B are parallel processes which execute serially. Suppose that A is selected for execution

first and, as a result of its execution, the output value is changed. Obviously, A cannot be permitted to directly place the new result on the output line since B is still to be evaluated. The execution of B requires that the "old" information be available. In theory, the new value of A will be available after the delay time of A and not at the current time. Thus, the necessity for data buffering arises because we are effectively executing values in advance of current time.

In general, every target description component might be in simultaneous execution, requiring the temporary holding of output information until the appropriate time. To accomplish this in each individual module would require a complete duplication of the target process data base. Usually, only a subset of the information will have both "old" and "new" values at any point in time; our impass is that the particular subset varies dynamically and is difficult to predict. Thus, a data holding manager is introduced in the supervisory system which receives the new data output values from each module and queues these along with the time at which they become "valid".

Time Management

Since time in the target description is controlled to achieve virtual parallelism, some system component must be responsible for the advancing time. Clearly, the supervisor is the only component with enough information to effect this decision. The supervisory system knows the module connectivity and is informed of changing values with their times. Using this data, modules are scheduled for execution at

a particular virtual time. Time is advanced when all activities scheduled for the present time are exhausted. The amount of advance can be determined by examining the times associated with recorded changes rather than a fixed increment of advance. That is, after the execution of all necessary modules, the changed outputs are scanned for the closest event to the current time. This becomes the new time and modules connected to the changed output lines are scheduled for execution.

The summary of activities required in the execution of the target system simulation is illustrated in the table of Figure 3.

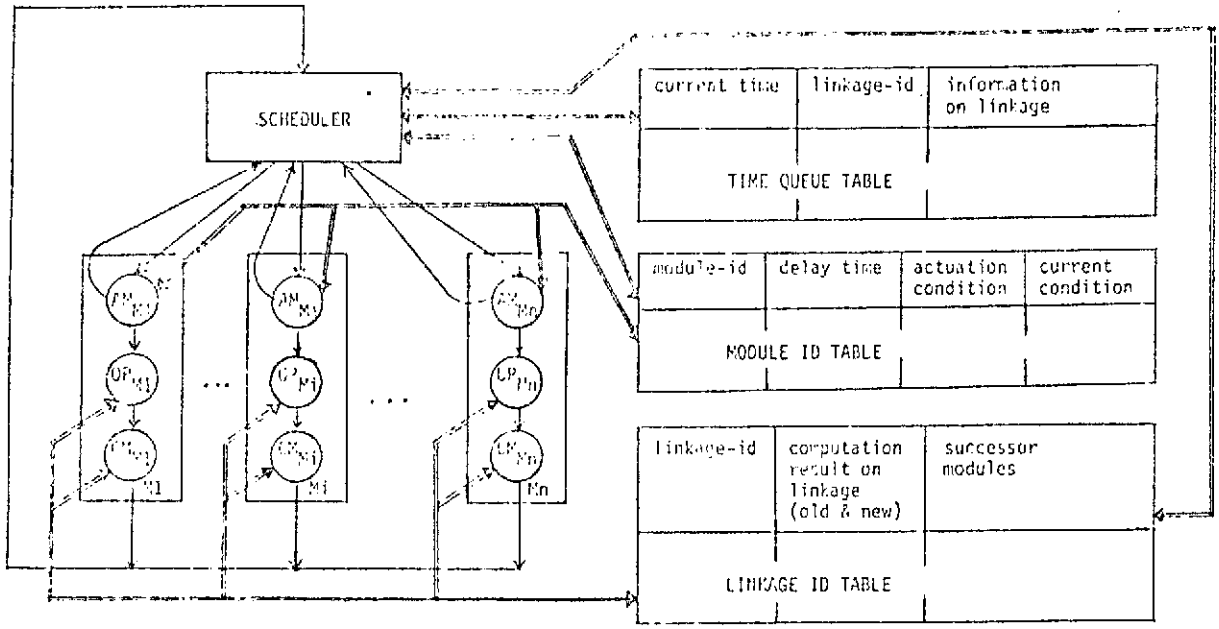
System Flexibility

The SVC system adaptation to a variety of requirements is governed by host machine characteristics, language capabilities, and match of target process to host process.

Resolving Target/Host Mismatches

To resolve target and host mismatches, some operation principles must be defined in the host machine. These usually involve trade-offs in the implementation features which may move the mismatches around the system rather than eradicate them. For the moment, attention is restricted to the simulation of a target computer on a given host machine.

The simulation of one computer on another machine is demonstratively easier if the host equipment is more powerful than the target. If the converse is true, some subset of target machine features is not



◆-◆ : communication line to data base
---> : Inquiry and report line
Mi : Module i
AM_{Mi} : Actuation Monitor for Mi
OP_{Mi} : OPeration of Mi
CM_{Mi} : Change Monitor for Mi

$$Mi = AM_{Mi} + OP_{Mi} + CM_{Mi}$$

NOT REPRODUCIBLE

(a)

SUMMARY OF RESPONSIBILITIES

ACTIVITY	PERFORMED BY
Schedule Module Execution	Supervisor
Advance Simulated Time	Supervisor
Update Information Paths	Supervisor
Compute Logical Output Function	Functional Module
Compute Information Availability Time	Functional Module
Identify Actuation Condition	Functional Module
Identify Change of Information	Change Monitor

(b)

SVC OPERATIONAL OVERVIEW

FIGURE 3

supportable on the host. With a flexible host, capabilities can be expanded to exceed those of the target. The host machine superiority does not imply that all features of the host supersede those of the target machine. For example, the host hardware may have more storage than the target machine, but the word size of the host might be shorter than the target. Thus some dynamic host activities are required to reconcile these differences. The methods employed for this matching will affect the coding of module functions, but should not affect the user's conceptualization. The standards adopted will require recognition by all functional modules of the information interchange format to insure an orderly system. With this cooperation, any module can communicate results to any other module.

The features which commonly differ among machines are enumerated as follows:

1. Word Size. Differences in word size can be reconciled by the usual technique of using multiple host words to accommodate target machine information which exceed the host machine word width. Several implications accrue from this decision. For example, the functional routines to perform target system operational activities can be modularized to permit arbitrary information sizes. The actual size in a particular instance will be established at allocation time. These techniques frequently require special treatment for the first word and last word, but general treatment for all interior words.

Since the information must be exchanged between modules using the host structure, a format convention for mapping onto host words must be

established for use by all modules. The format selected for the SVC experiment system excluded the host machine sign bit. This packing avoids complications in some (e.g., add, subtract) routines, increases problems in others (e.g., shift, mask), and is ignored in some cases (e.g., AND, OR). If the format is burdensome, individual functions may internally reformat incoming information to modularize the operation. At the interface, however, the format is standardized.

2. Value Coding. While the concepts of bits and concatenating bits into words is standardly used in all computer systems, the value represented by a particular pattern of bits may vary. Among possible interpretation forms, one's complement, two's complement, and sign magnitude are the most prevalent selections. Since the number of varieties is limited, exhaustive coverage of alternatives through unique functional modules tailored to these values is feasible. The major question is what form to use for the target description information linkages; that is, should the information be kept in target or host interpretation. Interconnection problems to the user are minimized by keeping the information in target form. Conversion to host value representation is performed for those functions which utilize host facilities assuming this form.

3. Formal Conventions. Beyond the integer representation of values, the formats of higher constructions (i.e., floating point numbers, complex numbers, double precision representations) vary widely. The provision for all possible variations of these constructions would be prohibitively expensive. Thus, the decision was made to restrict

the functions implemented to integer representations. Higher constructions could be functionally constructed on an ad hoc basis from these lower forms. The majority of interpretations will easily fit into positional and value interpretation variations which can be treated independently.

4. Bit Numbering. The number of bit positions within a construction varies from machine to machine. Sometimes the interpretation may vary within the same machine. For a quantity of n bits, the most frequent representations are numbering from 1 to n, from 0 to n-1, with the highest number bit being most significant, or with the lowest numbered bit most significant. The number of bits within the target machine or the numbering of words for that matter, primarily impacts the descriptive symbolic process in which the user first expresses the target process. This variation was not treated extensively; however, provisions for the frequent forms encountered could be provided to translate any convention selected by the user to the form required by the host processes.

Host Target Capability Gap

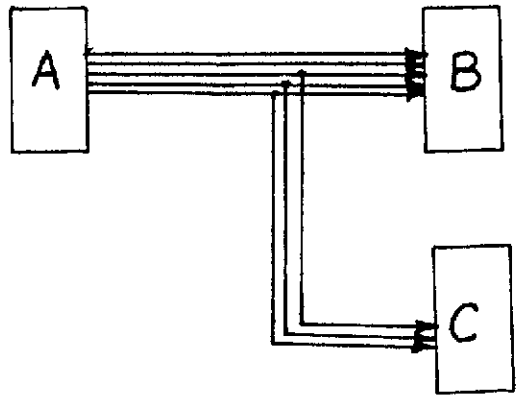
The capability gap, or processing capabilities of the target system which are unnatural or awkward in the host machine, will limit the flexibility and efficiency of the resulting simulation. Consider, for example, the simulation of hardware systems in software. As previously mentioned, the basic operation assumptions and execution characteristics differ, but in addition, the microscopic flexibility

of the target medium introduces manipulations which can cause difficulty.

In particular, hardware systems often exercise the option to select a subset of information lines to form another signal. This permutation of existing bits is quite easily performed in hardware but difficult to simulate in software. For example, consider the selection of every odd numbered bit from a register collected to form a signal of half the original size. This is easily accomplished in hardware, but if the register is represented as a word of memory in the simulation, the extraction of this new signal will require quite a few computations to obtain the permutation.

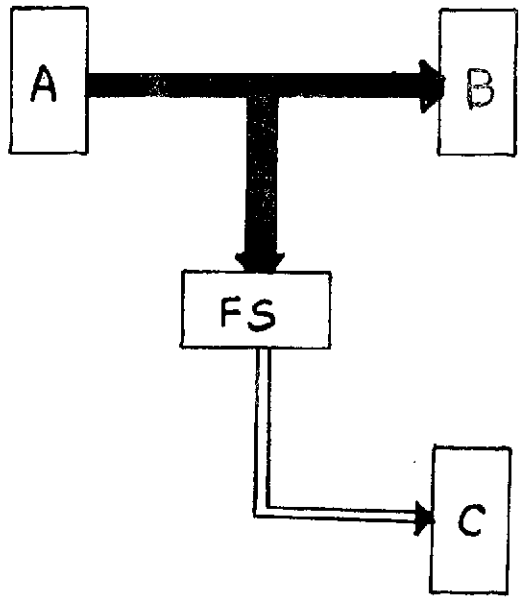
Usually, the subset selected is a strict subset of contiguous bits--this process will be called field selection. The field selection process can be applied with other logical processes to obtain all the permutations available through wiring selection. The example of Figure 4 indicates how the field selection process could be implemented on SVC. Notice that field selection is an activity which is performed in zero time on the actual target. Field selection is simply a vehicle to accomplish hardware-like activity in software.

In large part, necessity for introducing these normalizing constructions demands some behavior restraints such as prohibiting inputs from being outputs of a module. Consider modifying the example of Figure 4 into the construction of Figure 5a. Suppose the subset signal out of the field selection gate were modified in value by the functional process in module C. This would imply that the subset of the signal



PHYSICAL PROCESS

(a)



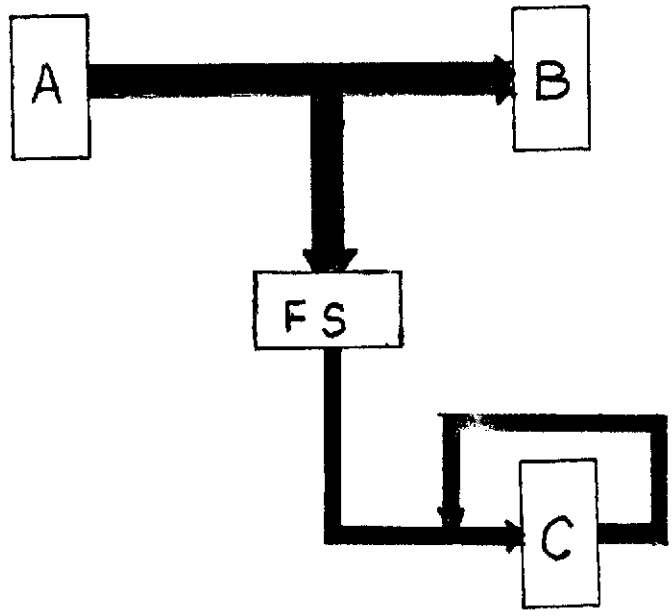
F.S. = Field Selection

SVC MODEL

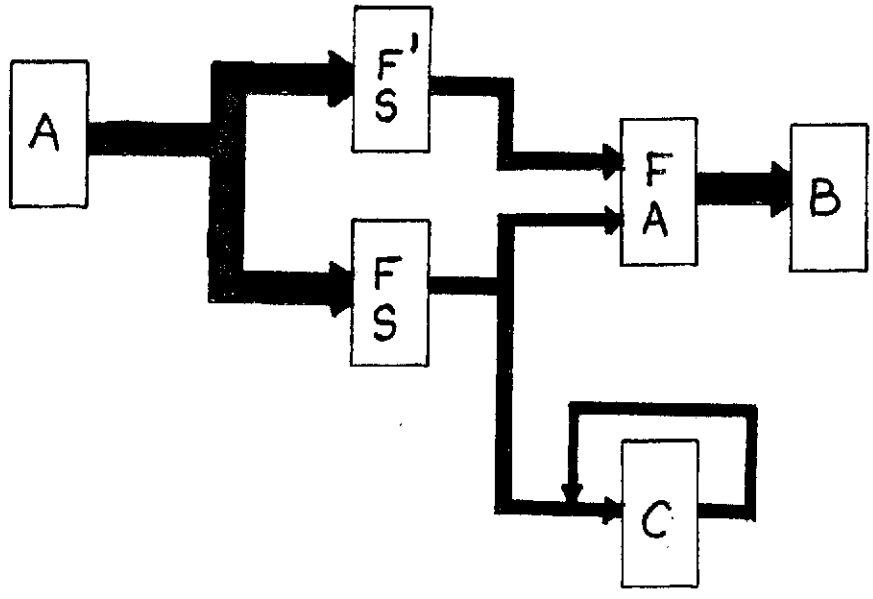
(b)

FIGURE 4

OUTPUT TO INPUT COUPLING HAZARD



(a)



REQUIRED MODEL

(b)

FIGURE 5

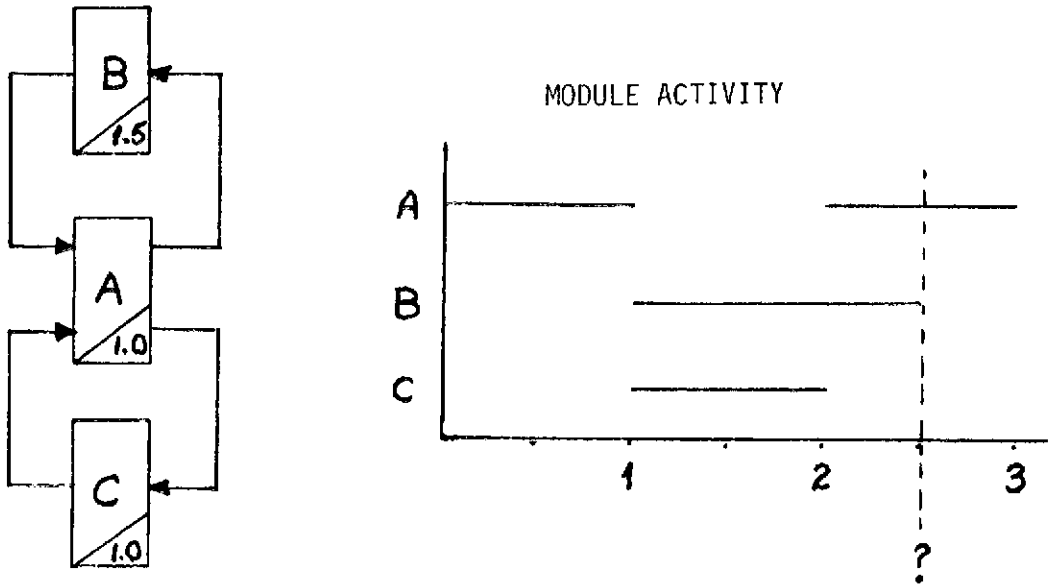
path from A to B, from which the field was extracted, must be made equal to the value of the subfield. For each occurrence, a structure illustrated in Figure 5b would be required.

Another problem with the simulation of hardware on software based systems is that electrical characteristics which control the system are now absent. The necessity for maintaining the delay characteristics of hardware components has already been discussed. To understand another problem, consider the illustration of Figure 6. Here the different lengths of delay in the feedback loops cause a proliferation of events until one event arrives at the input terminals of module A before it has theoretically finished the previous response. The questions to be resolved are: 1) How many of the generated results are useful? and 2) What should be done in the event that a new input stimulation appears while a unit is currently active?

The first question cannot be answered without interpreting the semantic interchange of neighboring system components. This requires additional knowledge about the event front in the description which transcend individual module boundaries.

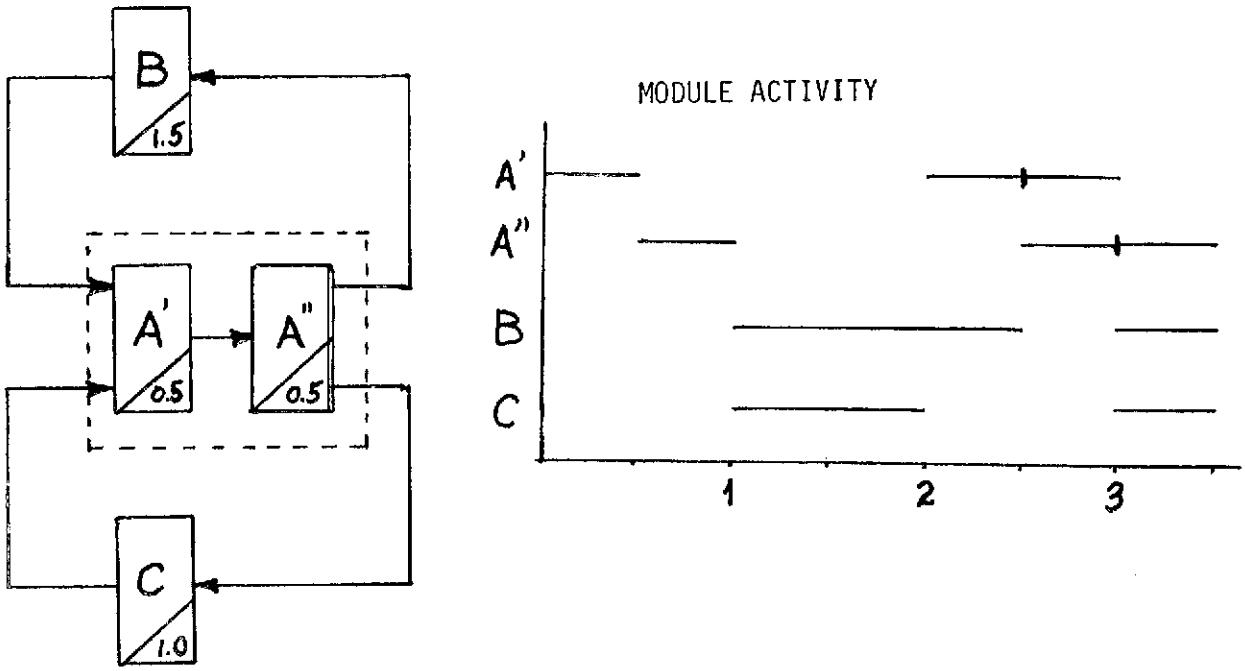
For the second question, if an event arrives before the functional unit has completed the reaction to the last arrival, several interpretations are possible. These include:

1) Physically concurrent events whose timing difference is negligible.
2) Pipeline operation where the new arrival begins an event front and outputs are not synchronized with input arrivals.



INPUT ARRIVAL HAZARD

(a)



PIPELINED ACTIVITY MODEL

FIGURE 6

In the first case, physical reaction is limited by the "rolloff frequency" of the unit. That is, if events occur more rapidly than the unit can respond, all events in the time frame of the rolloff frequency period are treated as coincident in time. Thus several arrivals result in only one output from the functional module.

If pipelining is present, the SVC system will require that the stages of the pipe be broken out into functional units to distinguish this from the rolloff frequency case (see Figure 6b). In this fashion, an output from one stage occurs before the next pipeline input begins, although the event front has not traversed the full length of the pipe.

At first glance, this requirement appears to demand more process description and have adverse effects on abstraction. Further examination indicates that the activities required at the pipeline state interface would actually be required internal to the piped representation if it were implemented as a single unit. Thus, no additional processing effort is being introduced: the required activity is simply being moved from the interior to the exterior.

Implementing Language Limitations

The implementation discussion has indicated a need for flexible and adaptive constructions in the functional modules and their linkage; availability in the language selected will govern the degree of automated adaptation and simulation efficiency impact when dynamic methods must be used to supplement language flexible features. For the example system, FORTRAN was selected to implement the system since the purpose

was more expository than operational. While simulation efficiency was not the primary consideration, the impact of selecting FORTRAN on system flexibility was not fully appreciated at the onset.

Most FORTRAN limitations stem from the requirement for prematurely binding constructions at compile time. For example, it is impossible to construct pure FORTRAN routines with a variable number of formal parameter inputs. An alternative is to present the number of inputs to the routine in vector form as one of the formal parameters. The vector representation requires elements to occupy contiguous memory locations and additional execution time to link the list dynamically. Contiguous storage could be allocated for each individual module independently; however, when several modules require access to the same information, several modules may impose conflicting requirements which cannot be simultaneously satisfied.

Similarly, the only convenient allocatable storage medium in FORTRAN is the array. Therefore, the target system information linkages were assigned from the user description to array locations with the information coupled to the processing routines through formal parameters.

Some of the needed flexibility was obtained using PARAMETER statements to tailor modular subroutines. To accommodate different sized information lines, standard quantums were established (e.g., 2, 4, 8, etc.) from which the user must select one or connect several in a subnetwork to obtain the desired sized routine.

Similarly, full macro expansion requires text editing of the basic routine and selective incorporation or omission of executable sections

to customize functional modules. These capabilities are absent in FORTRAN.

Another problem to resolve is whether to couple the target machine information to the functional subroutines through formal parameters or COMMON data areas. Clearly, COMMON is the most efficient dynamic linkage, but its use inhibits the sharing of execution code. For example, several target machine adders may have identical processing capabilities. They differ only in the information lines upon which they operate. If COMMON is used to link operands to the adders, each adder must be represented as a physically separate routine. Formal parameters permit one subroutine to perform all the adder functions with one subroutine copy.

Furthermore, the use of COMMON directly increases the possible access to information by the subroutine without proper description to the global supervisor; this can produce undesirable error conditions and can lead to a proliferation of ad hoc techniques which violate operational assumptions. Rather, the appearance of awkward operations should promote some analysis of the underlying principles to indicate desirable philosophy adjustments.

User Interface with SVC

Features must be provided in SVC to promote assistance rather than frustration in the user community. These considerations cover many of the construction concepts previously presented as well as some attention to practical problems faced by the designer. Some problems may be

peripheral to the theoretical system operations such as providing meaningful diagnostics, allowing user defined defaults, providing understandable and predictable actions when description faults are discovered, easing system initialization, providing a runnable description when errors are encountered, and providing a convenient symbolic environment.

Since the user is usually not the software developer of the library components, some attention is directed toward checking the usage of functional modules in the described environment to insure that software design limits are not abused. This reduces the necessity for voluminous documentation to coordinate interface with the user community.

Functional Library

It is not desirable to force all library components to an arbitrary level of detail. Rather, the collective components of the library should cover the constructions required for various target systems, but functional overlap and duplication of effort are not expressly discouraged. Rather, components should embody constructions which are primitive enough to be fully comprehended functionally. The internal workings of each functional module should exploit any machine specific features which enhance the operational performance of the routine. Primary interface with user application is explanation of module inputs and outputs including any information coding used by the routine module to dynamically interpret the operations. (Note: the coding pattern should be user adaptable to a specific application.)

Functional modules in the SVC library are characterized by the following properties:

1. Module input and output data are partitioned in the formal parameter list.
2. Activation conditions (if any) for individual modules will be contained in the internal procedure.
3. When activated, the module is responsible for reporting any output line which potentially changes. (Note: the module does not directly place the new value on the output line, but rather passes this information to the supervisory change monitoring routine.)

Describing the Target Process

The user describes the target system by local interconnection of functional modules. Although the process is most convenient with an interactive terminal, the interconnection can be accomplished symbolically by labeling the functional blocks and information lines and linking them through reference techniques similar to programming languages. The symbolic presentation requires labeling and translation labor in which transcription errors are likely; these considerations increase the examination burdens of the SVC support systems to detect these errors.

When the user specifies a target system component, his specification is checked for the following:

1. The subroutine to perform the component function exists in the library.

2. The input lines specified are outputs from other target components.
3. The names of output lines are not in conflict with other component outputs.
4. The inputs and outputs of the specified module are compatible with the requirements of the implementing subroutine.
    a. Input (output) information of the target component is connected to subroutine input (output) parameters.
    b. The information line size of the target component does not violate requirements of the implementing subroutine.
    c. The number of inputs and outputs is equal to the number of parameters in the implementing subroutine.

If this checking indicates a deficiency in the user description, warning messages are issued and predictable defaults are inserted to produce a runnable description.

The user controls the target system by specifying delays on the functional computations of his components. The delay figure can be defaulted to a prescribed value. For the default value, the most natural selections are:

1. Unit delay
2. Zero delay.

After weighing the merits of these two alternatives, the zero delay figure was selected for the demonstration system. The reasons for this selection are:

1. In organizing the target description with delay times, the user frequently identifies an information flow path through the description and associates a time with this path. The aggregate time is then distributed among the components of this path. If changes are required to correct or modify the description, more components may be inserted in the paths. With zero delay default, these changes can be inserted without affecting the coordination of timing among the paths. With the unity delay default, all affected paths must be rebalanced with each insertion. Constant readjustment is both irritating and time consuming.

2. Some functional modules must be included in the description which resolve target/host media mismatches. These processes actually take zero time in the target machine (e.g., selecting a subfield of an information pattern). The selection of zero delay permits minimum distraction to the user when these incorporations are necessary. It also permits the supervisor system to treat these special functions in the same fashion as other modules.

Other defaults to resolve are:

1) Information line sizes - defaulted to the smallest allocation of host machine resource (usually one word).

2) Information coding convention - defaulted to host machine property unless otherwise specified.

3) Initial value - defaulted to zero.

Constant Provisions

Although the information lines discussed have implied varying values, there is the necessity for providing constants within the system for use as inputs to the functional modules. For example, an adder functional module can be tailored to perform incrementation by attaching a constant to one of the addend inputs.

To provide constants in a compatible fashion, a dummy component is attached to the target description. The implementing subroutine for this component never modifies any of its "outputs". These information lines are used as recepticles for the constant values. When outputs for the dummy routine are specified in the description, the initial values are declared and maintained for the execution duration. By connecting these information lines to other module inputs, the constant values are distributed to other system components.

Initialization

To facilitate the initialization process, several techniques are required. Those information lines which require values other than the

default number are initialized in the target machine description. These values are inserted in the allocated host space before simulation execution commences.

For larger storage units, for example, main memory, there are two alternatives:

1) Permit the target machine description to simulate the the actual bootstrap operation.

2) Prestore the information before simulation begins.

Although the first alternative might be instructive for a few runs, the expense is clearly prohibitive for all simulation runs. Furthermore, the bootstrap approach is infeasible for a partial simulation (i.e., where only a portion of the target is described).

Thus the prestoring facility is required to bring the SVC system to a worthwhile level. The only decision is whether to treat initialization as a special case or to accomplish initialization through the target system description (either by user specification or automatic modification to the description).

By definition, initialization procedures are performed only at the onset of the processing. Incorporating initialization procedures through expanded target machine description produces description components which become inactive after initialization has been performed. These inactive units may impose a dynamic processing load on the execution to ascertain whether information is to be routed from the functional components or the initialization components. To the

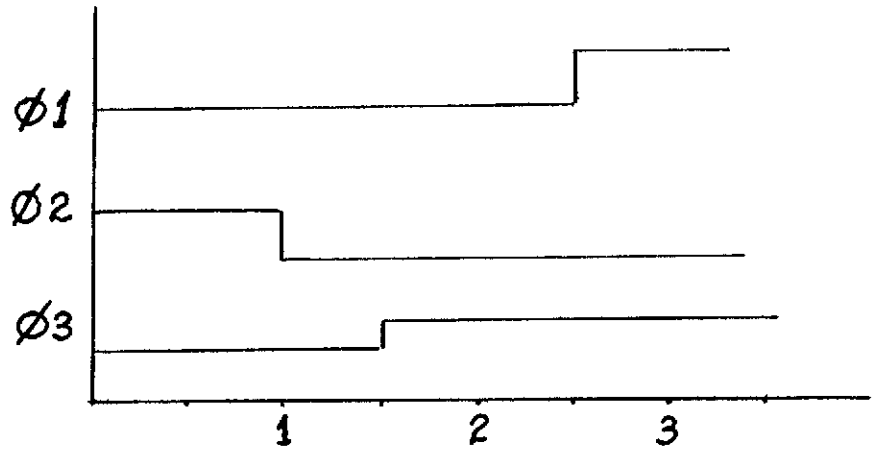
user community, the description expansion approach may produce unjustifiable overhead.

For this reason, ad hoc techniques were adopted for functional procedures involving mass loading prior to execution. Secondary entry points were introduced into the operational subroutines for linking externally prepared load files. It should be noted, however, that this technique does not exclude target process bootstrapping or description augmentation techniques if the user desires these initializing methods.

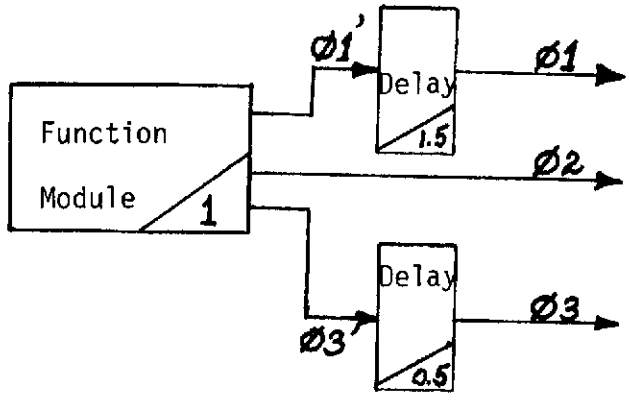
Delay Characteristics

To control the execution, delays are inserted in target machine components. The question is whether to associate this time with the module itself or the output information signals. For modules with single outputs, the approaches are equivalent. If, however, the module produces several outputs, associating the delay with the module implies all outputs occur simultaneously. In some hardware modules, for example, the outputs may vary in availability time; this variation is significant only if some of the outputs are required for use before the other values appear. If delay values are associated with each line, significantly more information is required to express delay characteristics of the target process. Even if the outputs occur simultaneously (i.e., the same delay value for each output), multiple locations must be used.

It was determined that usually either the outputs of a module appeared simultaneously or, if the timing differed, it was not



Required Output Phasing



SVC Model

FIGURE 7

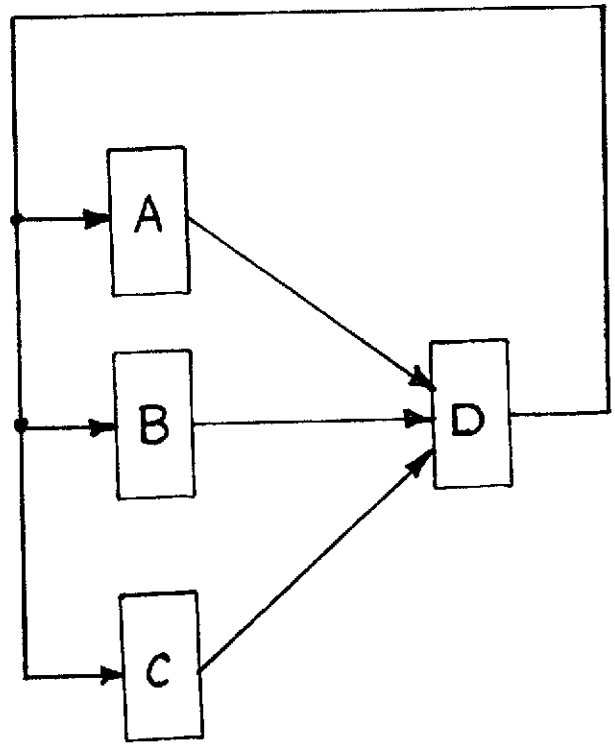
significant in the target process. Thus, the delay time was attached to the module directly. If a module required different timing, the minimal time of all outputs is used for the modules and delay modules are inserted in series with these lines to achieve additional delay times required to phase the arrival outputs of the module (see Figure 7).

Execution Effort Reduction

While efficient execution is not the primary concern, the cost of running investigations will obviously impact the utility and application. Since some approaches will strongly influence the operational system cost, tradeoffs were examined to guide the initial decisions.

Subroutine/Macro Tradeoffs. The functional modules can be implemented as subroutines or macros; the impact of these alternatives will affect system size and execution speed. Guiding alternatives are the linkage overhead of a subroutine versus the increased system size through expanding macros. Furthermore, macro utilization will require text editing of the FORTRAN source code prior to compilation. Thus, the system will require recompilation if minor changes are made. Subroutines are directly supported in FORTRAN and more naturally fit in the system. The impact of changes will be moderated if the target description is modified.

The size reduction with subroutines will naturally depend on the ability to use common procedures. Common procedures require not only functional process equivalence, but also absence of compiled constants or residual execution information (i.e., OWN variables). If information



MULTIPLE ACTIVATION HAZARD

FIGURE 8

storage can be moved to the parameter interface, opportunities for sharing code are substantially enhanced. The use of subroutines was embraced for the experimental system for design flexibility rather than size efficiency.

Multiple Activation Suppression. Since each module independently activates its successors, the possibility exists that a successor module may be activated several times for the same logical time. Consider the example illustrated in Figure 8, where, at a particular time, modules A, B, and C are active and they all have the same completion time, say t_0. Then, if all three outputs are changed, each module will request the supervisor to schedule module D for execution at time t_0, and the new data values will be held pending this time. If no special action is taken, this will result in module D being executed three times to produce new results. Each time it executes, the new value will be identical; thus, if it differs from the current value, three requests will be made from D to activate successor modules. Thus, A, B, and C will all appear in the execution schedule three times. After each of these modules executes, D will be scheduled 9 times. Thus, the hazard of explosive proliferation exists for activation requests.

Notice that, if each module has no memory of its own (i.e., all memory is stored on external linkages), there is no danger of producing erroneous results. The proliferation of redundant executions, however, seriously degrades execution characteristics. To avoid this, the supervisory system must examine each activation request for the presence of the module requested in the same time slot. If it is present, the

request for activation has been satisfied by another module.

User Declared Activation Suppression. In addition to automatically suppressing free running events, the user may indicate some suppression within the target system. From semantic operation knowledge, the user may determine that a particular sequence of arrivals will occur. Interim calculations can be suppressed so the last arrival will trigger the computation.

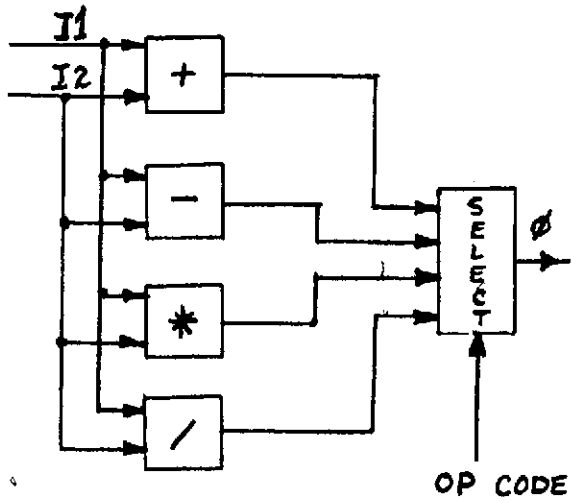
To accomplish this, provisions were incorporated in the description to distinguish two types of inputs to a functional module:

1) Activation inputs
2) Data inputs.

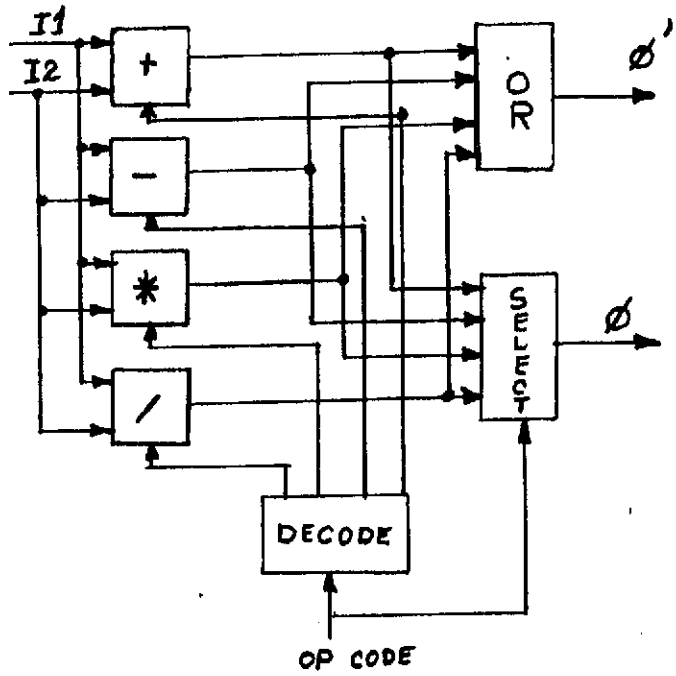
The functional module is evoked only if the changed information is connected to an activation input. A change on a "data" input will not cause the functional module to activate. The distinction is important if some input information is dominated by other inputs.

Example SVC Constructions. To illustrate SVC operations, several construction examples are presented to show both description alternatives and execution impact of selected techniques.

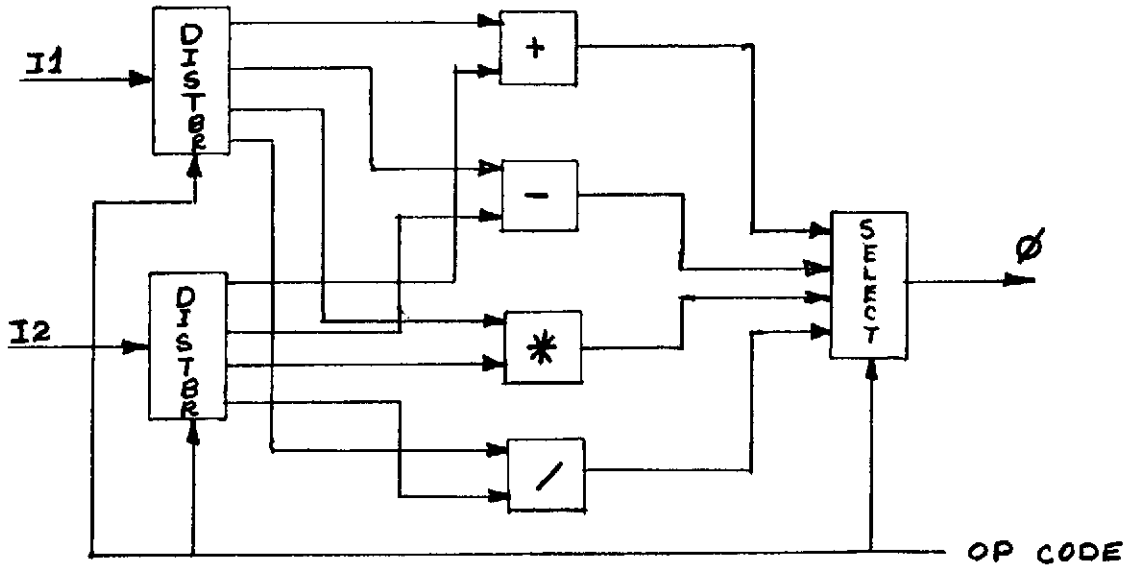
Figure 9 illustrates three possible configurations to construct a controlled arithmetic function generator. Two inputs (I1 and I2) are presented with a coded operation to perform (OP CODE) to produce an output (O). In Figure 9(a), the operation is very simple; the output is controlled directly by selecting one of the four computed results. Unfortunately, the useful operational activity of this


I1
I2
+
−
*
/
SELECT
ø
OP CODE

(a)


I1
I2
+
−
*
/
OR
ø'
SELECT
ø
DECODE
OP CODE

(b)


I1
I2
DISTR
DISTR
+
−
*
/
SELECT
ø
OP CODE

(c)

FIGURE 9

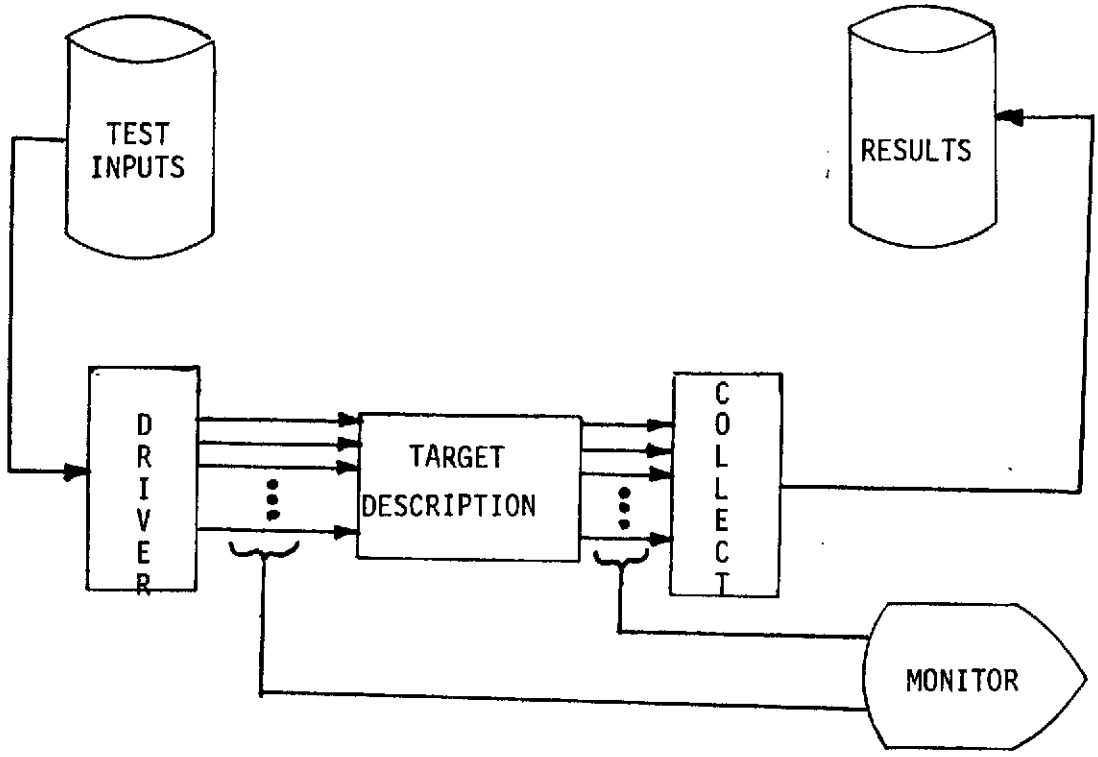
construction is very poor. All four possible functions are computed for each input combination, but only one result is selected for output.

Execution efficiency is improved in Figure 9(b). Here control lines are provided to each of the four functions. The output is not computed unless the control line is "true". Control line values are produced by the DECODE module from the OPCODE.

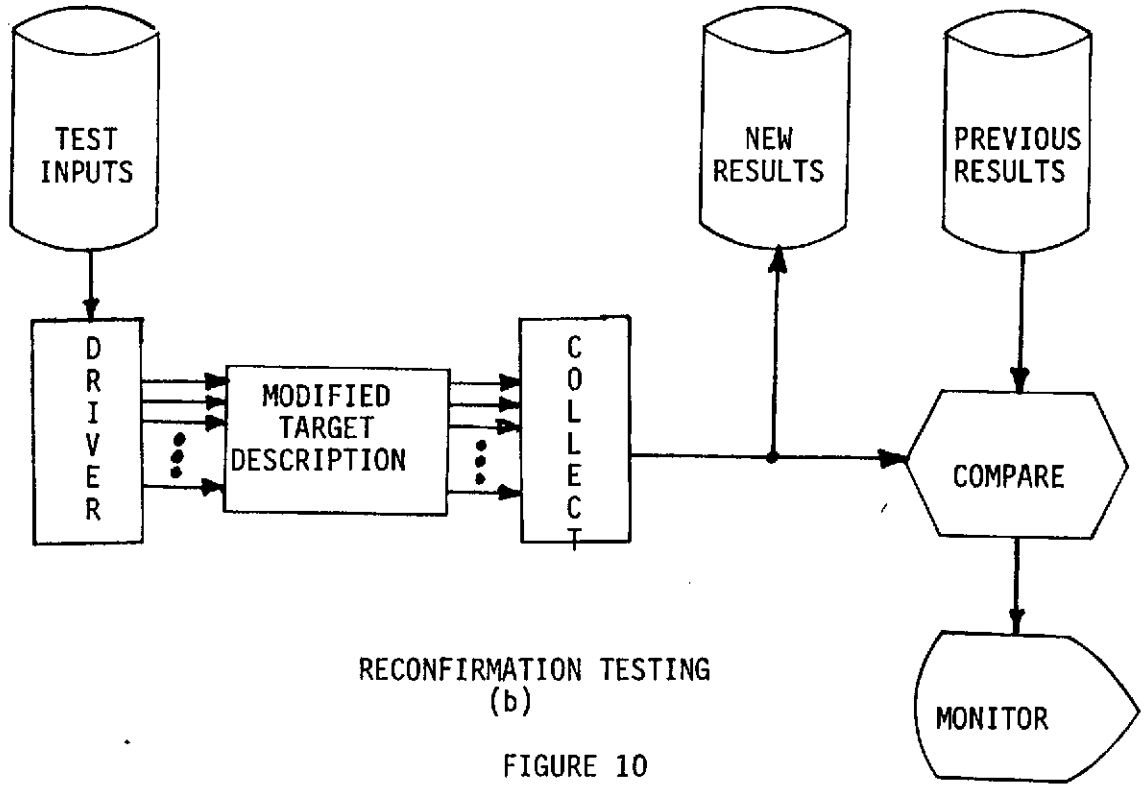
The arithmetic module computational requirements are dependent upon the method used to select the appropriate output. If the SELECT function is used, the computation module needs only to compute the arithmetic function if the control line is "true". If the OR network is used, the computation module will also have to set its output to zero if activated with a "false" control line value. With the network construction of Figure 9(b), each input value change will cause all four arithmetic function modules to activate. Only one, however, will evaluate the arithmetic function since exactly one control line will be true. Therefore, the number of activities has not been substantially reduced, but the duration of each may be shorter.

A further improvement is presented in Figure 9(c). Inputs are directed by the DISTRIBUTION module toward the arithmetic function module which has been selected. The correct output (also controlled by the OPCODE signal) is produced as the output of the selection module.

It is noteworthy that execution efficiency has been enhanced by increased usage of dominant control information (i.e., the OPCODE) in the construction. With the most efficient execution construction,



INITIAL TESTING
(a)



RECONFIRMATION TESTING
(b)

FIGURE 10

changes in the OPCODE value interpretation will cause more substantial modification than the inefficient construction of Figure 9(a). In effect, we have been improving the simulation performance by increasing semantic assumptions among modules and increasing system cooperation. All three forms, however, produce the same numerical results.

As major subsections of the target system description are developed, each may be individually tested as illustrated in Figure 10. Test inputs are provided from external storage and the results are both monitored for hard copy inspection and stored for future reference. If the subsystem is modified, the same input file can be used to generate results for comparison with previous results (Figure 10(b)). This technique provides confirmation of the modification. Both during and after the testing, the network subsystem description remains unchanged, even if testing probes are inserted in the interior. Thus the operational system is identical to the tested description.

IV. Experimental System

Objectives

The experimental system was constructed to test the validity and convenience of the principles presented in previous sections. This exercise was to provide direct feedback to the designer and indicate areas requiring additonal refinement or improvement. The experimental system also provided direct experience to identfy beneficial host hardware characteristics and estimate the construction cost of an operational system.

Scope of Effort

While few frills were incorporated in the experimental system, the intent was to replicate user activities in describing an actual machine.

For these purposes, an implementation of SUMC was selected for simulation on the UNIVAC 1108 using a FORTRAN V based system. The SUMC simulation was performed at the microprogram level using the implementation described in MSFC document S&E-ASTR-C-004. A subset of the hardware description was selected to reduce the effort required for machine simulation and increase attention to the SVC techniques required. For this reason, some elaborate operations of the SUMC description were not incorporated (e.g., square root, floating point operations, interrupts, etc.).

The UNIVAC 1108 operating under EXEC 2 was used to host the experimental system. The software system was coded in FORTRAN V

supported on this system.

The software consisted of the following components:

1) Handcoded primitive modules for hardware simulation which could be tailored using PARAMETER statements and adjusting formal parameters.

2) Library preprocessor to accept the input/output and usage restriction description of functional modules.

3) Target machine preprocessor to convert symbolic target machine descriptions into tables and execution code compatible with the supervisory system.

4) Runtime supervisory routine to manage the dynamic execution of the system using the principles presented in Sections II and III.

In general, description automation and diagnostic capabilities of the experimental system were much more modest than requirements of a fully operational system but served as experience vehicles for the design staff.

To obtain a runnable SUMC model for simulation, the following activities were required:

1. Partition and model the hardware.
2. Code an example program in assembly language.
3. Code microprogram routines.

Of these, partitioning the hardware required the most effort. Extracting the necessary information from the documentation required

collecting not only details of individual components, but discovering their cooperation and interfacing requirements. Although the documentation was quite good in comparison to similar descriptions of other machines, some modeling work required incorporating assumptions and definitions which could not be gleaned from available information. This omission was most serious with respect to timing information of the example machine. Since only a few timing relationships could be ascertained from further investigations, the missing sequencing information was estimated for the model.

Once the operational characteristics were well understood, the conversion process to the functional module description required only naming all component modules and signal lines. These were then transcribed to the card input required by the preprocessing system. The conversion to SVC compatible form required approximately 3 man-days and two descriptive preprocessor runs to purge keypunch errors.

To exercise the example simulation, a sample routine was coded in assembly language and converted by hand to numerical machine code. This routine performed a bubble sort on a list of positive integers. The instructions used by this routine indicated the microprograms required to complete the exercise.

Microprograms were hand coded from the documentation flowcharts for instruction fetch and bubble sort instruction set. Producing the microprograms for the instruction fetch and 7 instructions required approximately 2 man-days. A great portion of this time was consumed in the hand conversion of microoperations to numerical values which could be directly loaded into micromemory.

After the description was complete, the example system was tested on the developed support software. This testing was intended to point out coding/logic errors in the SVC experimental system, description errors in the target process, and possible faults in the operational philosophy. Testing continued until only errors identified with the target process operation description were found. At this point, experimental runs were discontinued since the purpose of the project was clearly not the development of an operational SUMC simulator. Attention was then directed toward analysis of the experiences and possible extension consequences for SVC.

Experiment Analysis

From the experimental system, insights were gained into philosophy provoked, error sources, technique effectiveness, user problems with the description method, and execution inefficiencies. These observations result in the recommendations for SVC extension in Section V.

System Deficiencies

The most obvious shortcoming of the experimental system was execution speed. Although it was recognized that the raw translation would not produce an efficient simulation, the extreme slowness was surprising. Execution on the example system approximated one microinstruction per CPU second; clearly, this would not be sufficient for an operational system. The sources of this degradation were:

1. Excessive Overhead. Every functional module subroutine was analyzed for output change, scheduling of interconnected modules, data holding, and time scheduling. Furthermore, on a

microinstruction level, the execution time for primitive operations was extremely short. These factors result in a large percentage of computation efforts to center on supervisory functions rather than target process functions.

2. Excess Change Monitoring. Many constructions appeared in the example machine description for which exhaustive change monitoring was redundant or ineffective. In general, a module input change will usually result in an output change. The probability of output change increases as the output information path width increases (i.e., number of bits).

3. Excess Target Information Movement. The general assumption that all modules may be simultaneously active causes computed information to move from a local area in the functional subroutine to a holding supervisor queue and finally to the allocated space for the target information path. In a great many cases, the information could have been directly placed on the information path without going through the supervisor.

Some of these problems were anticipated initially; their degradation potential, however, was vastly underestimated. In Section V, methods are indicated to reduce these problems through automatic adjustments of the target description before execution commences.

Error Sources

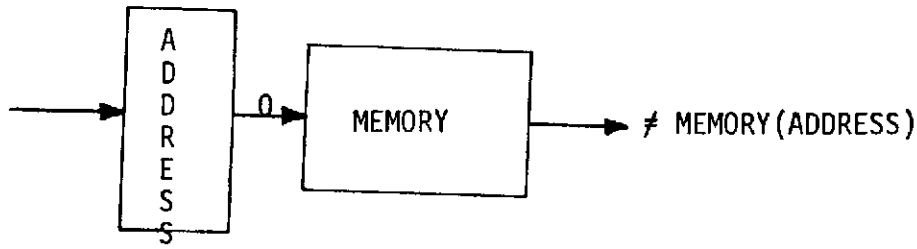
The testing process revealed several problems which were not discussed in Section II. Among the problem sources were the following:

1. Inconsistent Initial Conditions. The discussion of Section II and III deal with system dynamics after the execution has begun. It was assumed that the system was in a correct operating state and the methodology used would not create errors. It was not anticipated, however, that initial conditions could be specified such that outputs of functional target components would be inconsistent with initial inputs and operational functions such that the system would start incorrectly and never recover.

   For example (Figure 11(a)), in one test run, the address to memory was initially zero (through default). The memory was properly loaded, but the first address presented to memory during execution was zero (i.e., no change in address). This resulted in the memory not being activated to extract the correct memory location and the simulation stalled. The problem is that the memory output did not represent the value of location addressed.

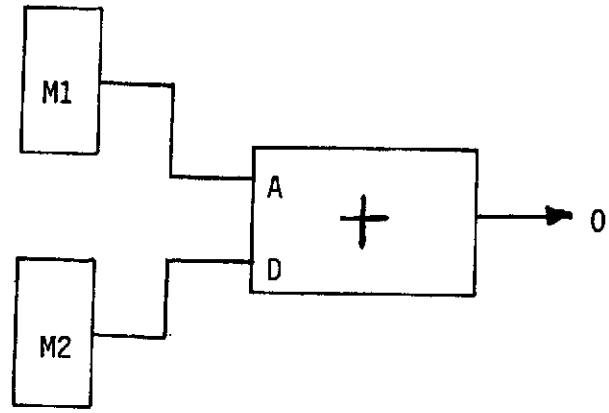
2. Activation Hazard. In Section III, the distinction was introduced between activation and data inputs for functional modules to permit user suppression of excess activity. Recall that a changed value on the data input will not cause the module to activate. There is a hazard with this approach which was not anticipated.

   Consider the example of Figure 11(b) with activation input a and data input b. This construction results from the user knowing



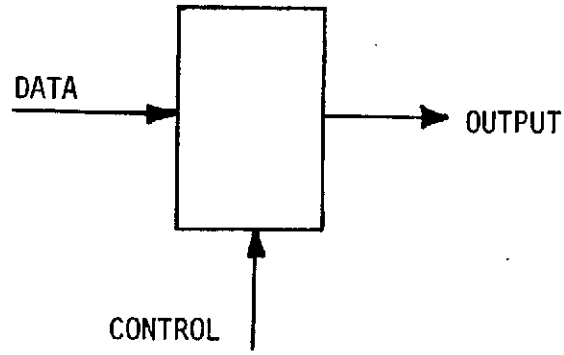
INITIAL CONDITION HAZARD

(a)



DATA INPUT HAZARD

(b)



(c)

FIGURE 11

target process semantics. From the target process operation, M2 completes before M1 and the value of O is not significant until M1 is complete. Notice, however, that if the b value changes and M1 produces no change for a, the functional module for addition will not be executed.

Reinterpretation is required for the "data" input. Rather than completely suppressing activation, this input should produce a subordinate scheduling of the module which occurs in the event an activation input does not change value. This has the effect of postponing the execution until all significant inputs have arrived.

3. Incomplete Functional Design. As illustrated in the construction examples of Section III, semantic assumptions among modules can cause different results to occur. This is frequently the case where the specification of the module's function is only partially given.

Consider the network of Figure 11(c). Suppose we specify the operation as,

> "When the control line is 'true', the output is set to the value of the input. When the control line is 'false', the input value is ignored."

This statement is incomplete in two respects. First, does the transfer from input to the output occur at the transition of the control line from 'false' to 'true', or is the input copied to the output anytime the control line is 'true'?

Second, what is the value of the output if the control line is 'false'? Two possible alternatives are that the output remains the same as the last output value (software approach) or is set to some predetermined constant such as zero (hardware approach).

Methods Analysis

In light of the experimental system experiences, the methods utilized were reviewed for applicability and convenience. In particular, their objectives are compared to the results and side effects they produce.

1) Event Activation. Event activation to guide the execution sequence was intended to eliminate overt expression of both sequencing and information flow information. In large part, this technique did not allow easy integration of components in the system. From the constructions used in the example system, however, the sequence was frequently apparent since the information flow patterns indicated few successors. In many cases, it appears possible to coalesce substantial portions of the description by static analysis of the interconnection pattern. This would eliminate some dynamic searching during execution to extract successor relationships.

2) Change Monitoring. Change monitoring to control event fronts caused substantially more problems than anticipated. As previously discussed, change monitoring produced excessive activity, directly contributed to activation hazards, and

provoked excessive overhead. It is obvious that substantially more attention is needed to reduce the number of change monitors and bring execution characteristics within acceptable limits.

One of the major advantages of change monitoring is to suppress execution of downstream components which will produce the same results. This prevents computation of target features which treat only "special cases" unless the special cases actually occur. Some tradeoff analysis is required to balance the effort of change monitoring to the cost of unfruitful downstream computations.

3) Description Control Using Delay Time. The use of delay times to control execution in the target description execution sequences was a mixed blessing. Delay times greatly simplified scheduling portions of the supervisor. The convenience to the user, however, was less than successful. Designers with hardware backgrounds had little difficulty in assigning delays to control the system. Software personnel had substantially more trouble with this description since their past experience provided no training for an orderly approach to the problem. It was found, however, after a few test experiences with the experimental description, they began to appreciate the requirements in this format. Time did not permit a second trial for software background personnel to assess how fully they comprehended the reqduirements.

It is recommended that the delay time control method might be retained for organizing internal sequencing of parallel operations, but that the numbers assigned to the modules be automated. The user might indicate the required sequences in more familiar terms which could then be converted into the required numbers for the system's use.

Objectives Review

The objectives presented in Section I are compared with experimental system experiences to determine which have been attained, are potentially possible, or appear unlikely.

Design Ease

In general, target process design using descriptive mechanisms of interconnection and delay time was more difficult than an equivalent design with traditional programming methods. Much of the additional effort resulted from the requirement for more exhaustive operation analysis before the system could begin executing. With inputs and outputs enumerated for library modules, designers must at least consider how and where to connect information paths. Thus, while more time is required to complete the initial design, the initial design resulted from more complete and mechanical analysis, making it less error prone.

As previously indicated, software oriented personnel found controlling the execution sequence with delay times an obtuse technique. The synchronizing points in the system were not as

transparent as with the "fork" and "join" software constructions. The information flow path description, however, greatly eased ignoring parallel activities that were computationally disjoint operations. With cooperating parallel processes, the points of interaction and information exchange are easily ascertained with the information flow diagram.

Operational incomplete designs

A system may be incomplete in one of two ways: 1) Only a portion of the whole system is available, or 2) Only a subset of the system operations have been developed. Awareness of target process operations and construction of the objective system are not instantaneous or disjoint activities. The objective is to prevent ambiguities or pending developments from halting system development progress.

Although independent testing of subsystems was not used in the experimental system, some hindsight observations can be made. Clearly, the SVC principles will support a partial description and permit exercise of this unit without the totality of system description. With subsystems, however, the necessary test cases which insure completeness may be difficult to derive. It appears that the best partitioning of subsystems should maximize the independence of input terminals: values on one input do not imply information concerning other input values.

When only a subset of the system capabilities are realized, the major obstacle to meaningful testing is deciding what should

be done in the event umimplemented components are required to execute by some test cases. Clearly, the output of these test cases will not be significant, however, the potential for negating succeeding testing exists. Thus, some "neutral" operation must be substituted for the missing feature. Whether this operation should not produce any outputs or should simply forward the input information will depend upon what function is missing.

Modification tolerance

The experimental system was quite successful with modification tolerance. Several substantial changes were made in the target machine description based upon early testing. The isolation of these effects was particularly encouraging when errors were accidentally introduced during modification. As hoped, the correctness of the change could be systematically confirmed by inductively showing some point in the machine description where downstream components would be unaffected.

Operational fidelity

Again, the results from these experiences were encouraging with the experimental system. Delay time control to introduce any degree of parallelism without disturbing existing system components was quite successful. No target machine functions were encountered which could not be expressed in the required execution format.

Automatic abstraction

The abstraction process was originally envisioned as an automatic process to reduce the process operation detail level. This was to be accomplished by considering larger subsections in the description and condensing the interior process. In large part, this is infeasible within current restriction on SVC operation presented.

While it is possible to establish larger subsections through arbitrary interfaces, the reduction of activities in these sections requires semantic analysis of the constituent components. That is, to combine and streamline activity sequences, the significance and utilization of dynamic activities must be analyzed.

One possible abstraction is to convert the dynamic interpretation of activities to a static sequence. This process is similar to compiling execution commands rather than interpreting them. The appropriate sequence could be gleaned by capturing the sequence of operations during test case exercise. The sequence could then be commanded by the supervisory system rather than driven from output change analysis. This technique requires,

1) The test cases used should exhaustively depict all possible sequences of the semantic operation in the target process.
2) The activity on the interface and conditions which cause subsection reaction should be identified.

To illustrate the problem, consider the effort required to convert the microprogram level simulation to an instruction level simula-

tion. The first problem is that logical entities of the instruction level model (i.e., individual instructions) use common physical machine parts in their execution (i.e., data paths, execution directing registers, functional units, etc.). Thus, the execution of two different instructions will possibly utilize some common physical resources on the microprogram level.

Furthermore, certain activities will require external interpretation to remove detail events. For example, information may be transferred through two cascaded adders by adding zero to a value. Thus, equivalence between adding zero and transfer of data must be displayed.

Some microprogram routines use iterative processes to determine the outcome of an instruction. It may be quite difficult to abstract this process since differences may exist among iterations. Compression requires mapping a sequence of microscopic activities to a single high level one.

Thus the automatic abstraction of the system appears to require more study to develop a recognition framework with sufficient generality for a host as flexible as SVC. Some aspects of the required semantic processing are presented in the optimization discussion of Section V.

V. Extension of Current Work

To move the Software Validation Complex from the current experimental stage to useful operation requires further attention to some areas. These activities include refinement of the target process description methods, performance enhancement in host execution, and development of a congenial support host hardware/software environment.

Modifications For Describing Target Process

The experimental system utilized an intentionally crude translation system for target process descriptions. This experience, however, permitted projecting desirable features and their relative importance in an operational environment.

The target description translator operated on a symbolic interconnect format. This required assigning unique names to every functional module and information signal. The usual aggravations of transcription errors and multiple definitions were observed. Furthermore, the user must assign names to entities which he does not care about. This problem is minimized by using an interactive description mode whereby blocks are interconnected and names assigned only if the user chooses to identify them symbolically.

Additionally, the user should be able to integrate previously developed subsystems simply interconnecting interface information paths. These connections could be automatically checked to assure

that design limits and assumptions across the interface are consistent.

The subsection storage and retrieval were not implemented on the experimental system, however, this effort is not expected to require substantial effort. Requirements for cataloguing available constructions could be treated in a very similar fashion to the library subroutines presently used. With little additional expense, a description editing support system could be included in this development.

The user communications interface should include ability to establish default values for bit numbering, initial values, information path sizes, binary value coding representation, etc. This information could be generated as a preamble to the description conversation and requires only the most simplistic processing.

As indicated in reviewing experimental system performance, the assumption that all components could be simultaneously active produces undesirable execution consequences. Thus, the assumption should be modified to permit the user to describe activities in sequence which could then be considered as a parallel component in the global execution. This feature could be added by selectively modifying some existing library subroutines to eliminate cooperation with the supervisor and insert the direct calling of the next sequential action without scheduling through the supervisor routine. Similarly, the data could be directly coupled among routines without going through the supervisory data holding

process. The appropriate delay could be accumulated through each routine as the sequential operation progressed. Thus, the same library routines could be used to generate sequential processes by textual source code modification to produce sequential static execution rather than interpretive dynamic operation. (Fig. 12)

To retain Fortran as the host language, a preprocessing system should be incorporated to massage the source code. This preprocessor should operate like a macro expander with output in Fortran text. This translation would accommodate a variable number of input parameters, assign unique subroutine names for different modifications of the basic routine, customize interface with the supervisor, and perform other modification functions to adapt the basic target process to a particular target environment.

Hardware/Software Simulation

To accomplish simulation of a hardware/software environment, there are two alternatives: 1) Describe the hardware host and present the software to be modeled as an exercise of the hardware, or 2) Model the hardware and software systems separately and define the interaction between them.

By operating target software support on target hardware, a complete duplication of the actual system is possible. This fidelity, however, is obtained at a substantial cost. Effectively, the system becomes a three level interpretation process and hence, extremely slow. Thus, while runs of the actual software support

system target code may be desirable to confirm its operation, the speed degradation will be so severe, other options must be investigated.

The other option is to partition the target machine into hardware and software portions and define the cooperation between them. This requires some overlap between the two descriptions such that the active operation of one will effectively prohibit operation of the other. For example, an external and separate model of the operating system could be provided along with the basic operation description of the target machine hardware. When the operating system portion is running, some hardware components will not be available (e.g., memory occupied by the operating system software, instruction register of the target machine, user registers of the target machine, etc.). Other target subsystems (e.g., disk transfers, I/O ports) may exhibit reduced performance. Interference can be modeled as conditional changes in delay characteristics. Additional exploration is required to refine the necessary principles to switch description level among different logical units to effect this more elaborate simulation with economies in time.

Performance Enhancements

From experiences with the experimental system, it is obvious that substantial efforts must be directed toward enhancing the operational cost of target executions. Acceptable vehicles for this advance are improvement of current operational methods, development of new techniques, and providing a more flexible

host environment.

Modification of current technique

Several sources of performance degradation were identified in the previous report section. Modifications are presented here to alleviate these problems by reducing operational generality once the target system is well defined.

In general, the experimental system suffered from an excess of supervision. While much of this resulted from the myopic view of functional units and simultaneous environment assumptions, once the target description is complete, generality can be reduced to the specific target machine case.

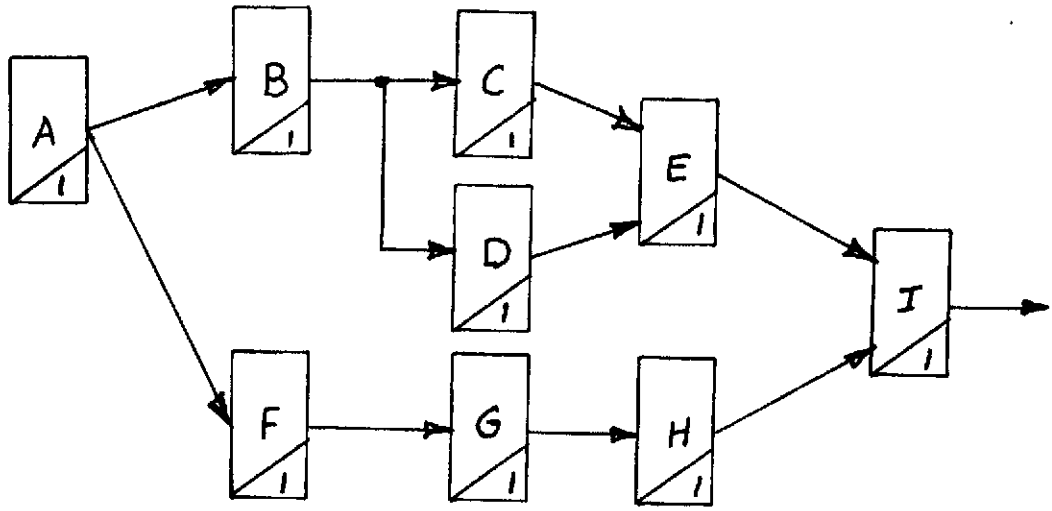
The supervisory system is responsible for the following activities:

1. Monitoring value change on information paths.
2. Scheduling modules for execution using delay and change information.
3. Holding resulting data back until the appropriate time.

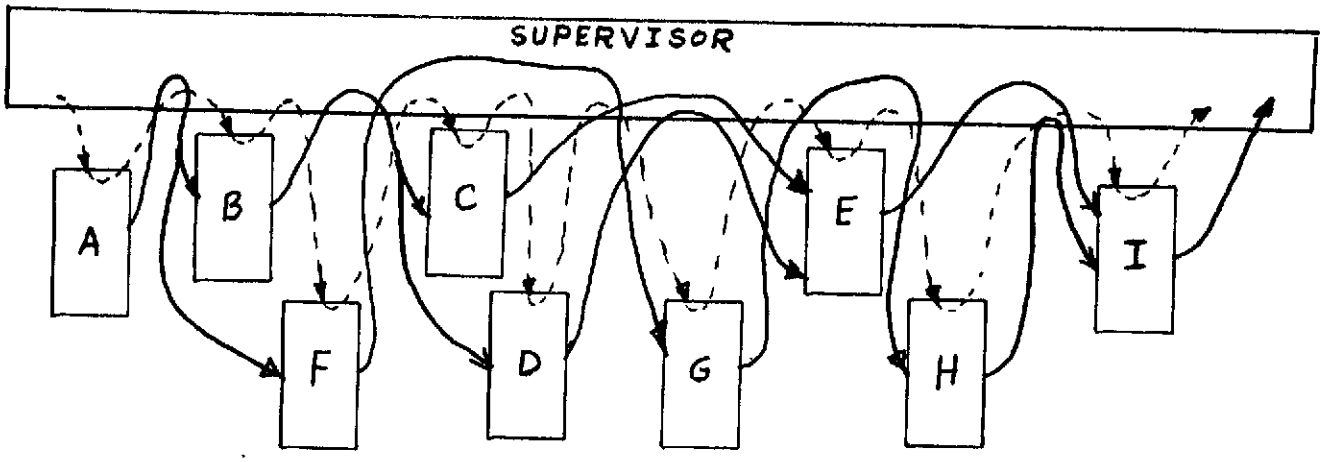
The change monitors accomplish two functions:

1. Terminate iterative procedures (feedback or phased information structures).
2. Indicate when further processing along information paths is no longer required.

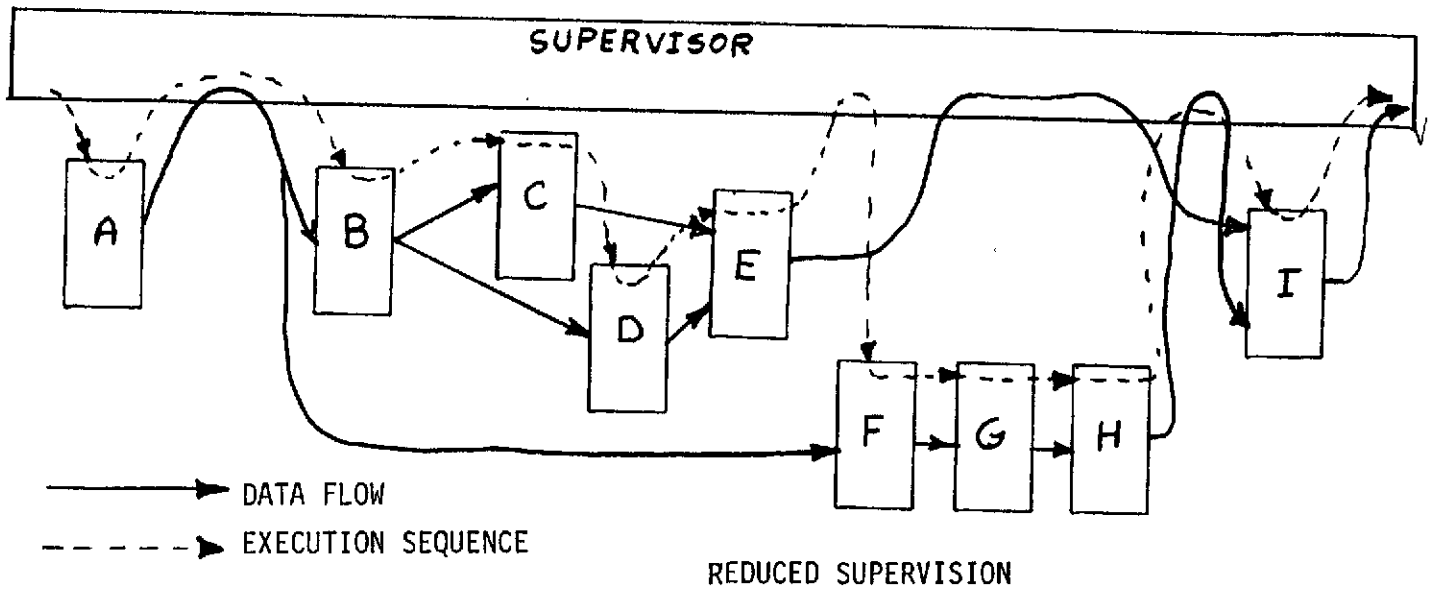
If we require that the functional procedures are coded in at least serially reusable fashion (i.e., they may also be reentrant), sufficient conditions can be defined to reduce the instances



TARGET PROCESS



FULLY SUPERVISED OPERATION



REDUCED SUPERVISION

FIGURE 12

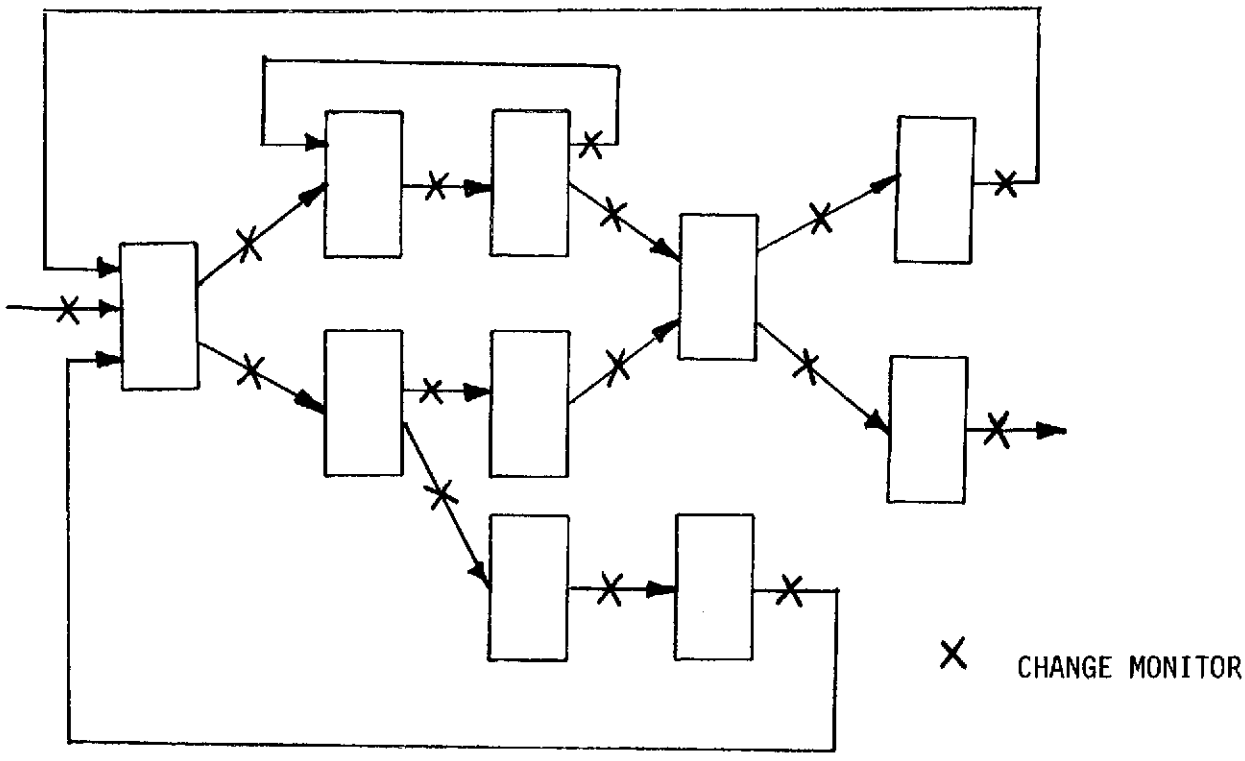
where change monitors are required. This set will insure that termination and phased relationships of the original target system are preserved. The serially reusable module property will insure that if a process is activated without an input change, incorrect results will not occur.

Having identified conditions to permit excess activation without error, it is possible to remove change monitors which are unproductive. As previously indicated, some monitoring locations are unproductive because they almost always indicate the value has changed. When this occurs, we can eliminate them and simply assume new values on these information paths are different from the current value. The question to be answered at each nonessential monitoring point is whether the value checking reduces execution time or would the system be faster by simply allowing target module execution.

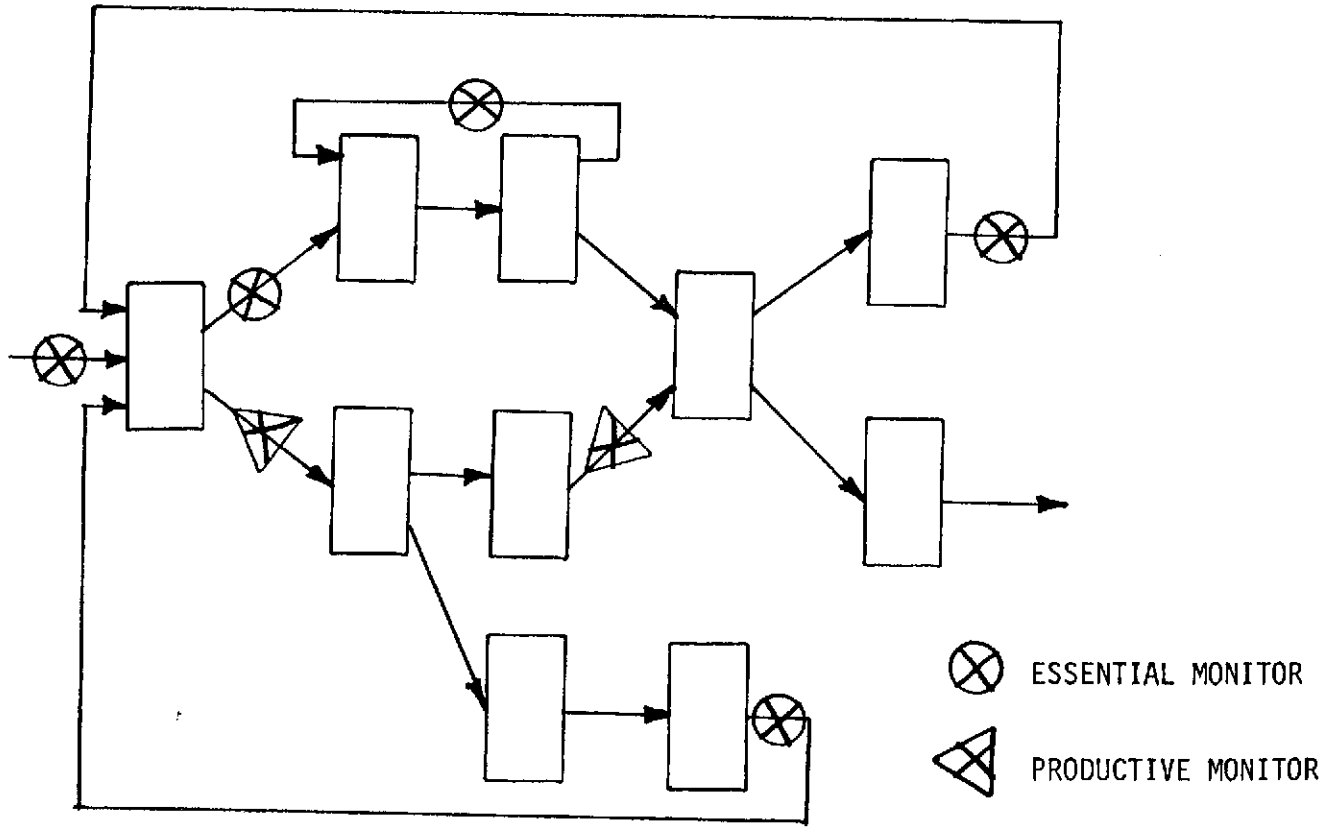
Essential monitoring points are found through graph analysis of the information flow structure. After a collection of target modules are found which form a loop, essential change monitors are required:

1) Sufficient to break all feedback loop structures.
2) Monitor all input changes from modules external to the looping structure.

Result data holding prevents the possible destruction of input information to parallel operating successor modules which have not been executed. The potential for this hazard can be ascertained from the delay times and information flow pattern.



FULL CHANGE MONITORING



REDUCED MONITORING

FIGURE 13

By showing that successor modules are not in simultaneous execution, results can be placed directly on the output information lines.

Execution sequences are dynamically determined by the supervisor since individual modules have no interconnection information. If however, nonfeedback paths are present in the target description, the paths can be executed without supervision. The final result and accumulated delay time is then presented to the supervisor. The supervisor must intervene when a module is encountered with inputs from more than one path.

The operational adjustments from change monitor reduction and direct data forwarding are illustrated in Fig. 13.

New techniques

In addition to the refinement of existing techniques, new methods may provide further simulation enhancement. As described in Section IV, one problem encountered was inconsistent outputs with the initial input conditions specified. This difficulty can be surmounted by introducing an autoinitialization procedure prior to commencing target execution. During this procedure, each functional unit in the target machine would be executed with the user supplied initial conditions. The resulting outputs would then indicate the required initial conditions for other information paths. The outputs produced would replace defaulted information or be compared with other initial values specified. If the output differs from the initial value specified on the information path, a diagnostic to the user would be issued.

Although the exact methodology has not yet been developed for autoinitialization, it is clear that some sequencing in the initialization execution will be required. For example, cyclic paths should not be followed to prevent instability in termination.

The current SVC operating principles ignore any analysis of individual target modules semantic properties. This restriction was imposed to ascertain how much could be doen without semantic information. Clearly, however, some improvement and analysis techniques are fundamentally dependent upon semantics. This information can be used to determine the potential sequential or simultaneous structures in the target machine and to modify SVC operations to streamline these events.

Semantics can be ascertained by two methods: 1) Externally defined operational characteristics, and 2) Dynamically ascertained behavior from runtime operation profiles. There are advantages and drawbacks to each of these approaches; a good approach will likely draw on both techniques.

With external definition of module behavior, any error in the description will likely provoke errors. Therefore, the behavior must be conservatively expressed even though some special cases rarely occur. Some semantic behavior in the target machine may arise from the combination of semantic patterns; extraction of these effects must be automated by the system. Since the individual describing the target process will usually not be the author of functional subroutine code, he is illequipped to define the semantic behavior. Thus, the externally supplied behavior

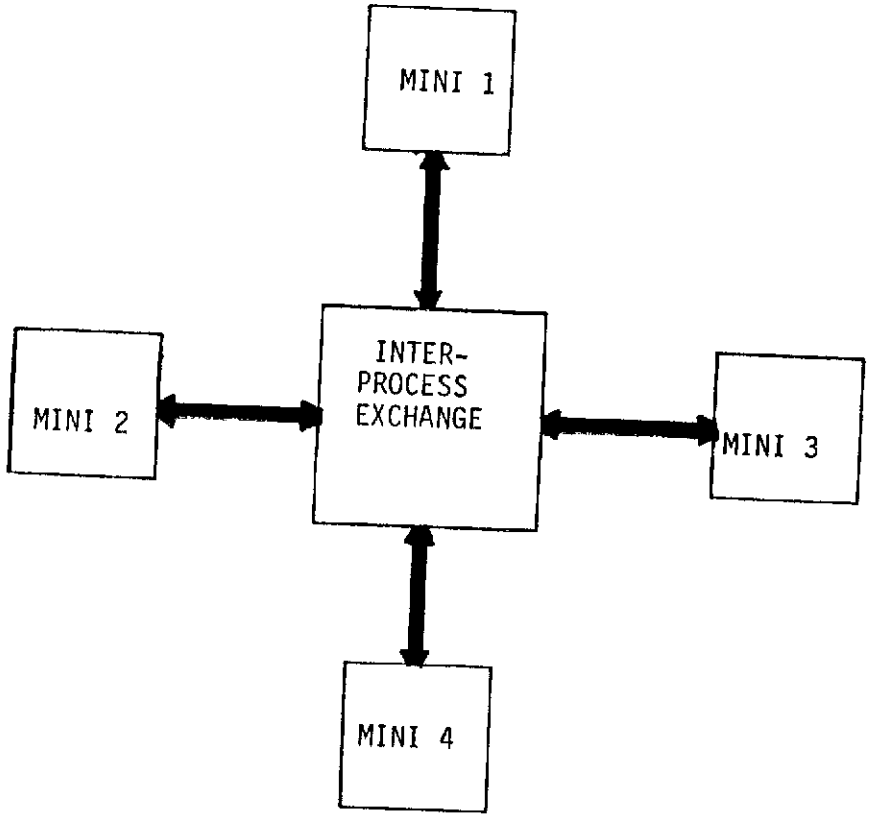
information should be attached to the library components and obtained indirectly by user application to these functions.

Dynamic behavioral information can be obtained by observing the dynamic transaction patterns in the target process during user test runs. Gathering this information involves observing the frequency of use and percentage of change in the target system. The objective is to ascertain what portions of the target description constitute the major processing sections to warp SVC service to favor these units. As will be indicated, this behavioral information will be extremely useful in distributing the work load on a flexible host.

The major problem with observed behavior is that the absence of characteristics does not insure they will not occur. That is, through incompleteness of the test cases, some target operations are not executed for observation. Thus techniques used in adapting the system with observed information will be limited to tuning the procedures rather than reducing them through elimination.

Host characteristics

Much of the process inefficiency on the experimental system resulted from the inherent restraint of executing a highly parallel target description on a serial host. The individual actions required by functional module subroutines do not require elaborate processing and in large part, the power of the 1108 was poorly utilized. Most code segments were so short that optimized code



SVC HOST CONFIGURATION

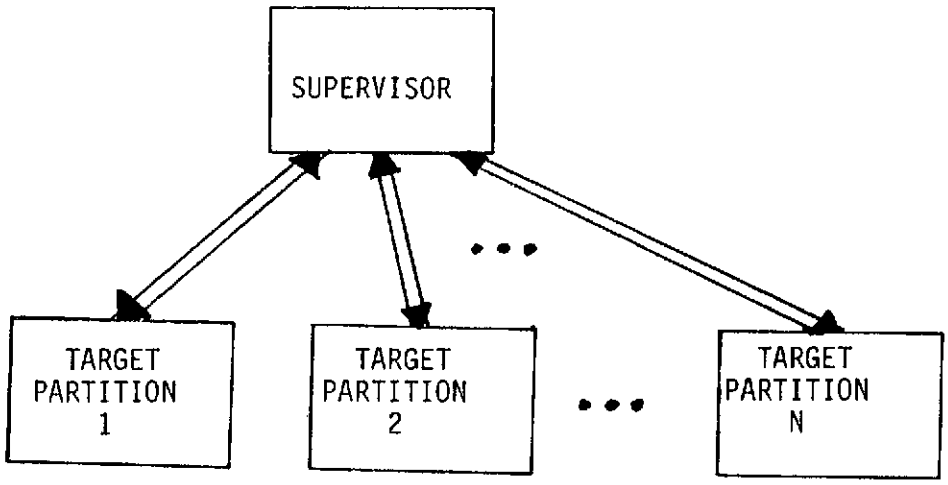
FIGURE 14

from the compiler would not be substantially faster than a raw translation. A much more modest and flexible machine could have easily performed the experimental tests with equal or increased execution speed.

It is therefore suggested that a more compatable host might be a collection of smaller machines (Fig. 14) rather than a single large machine. This type of host requires development of allocation policies and definition of the cooperative structure required to make the system operational.

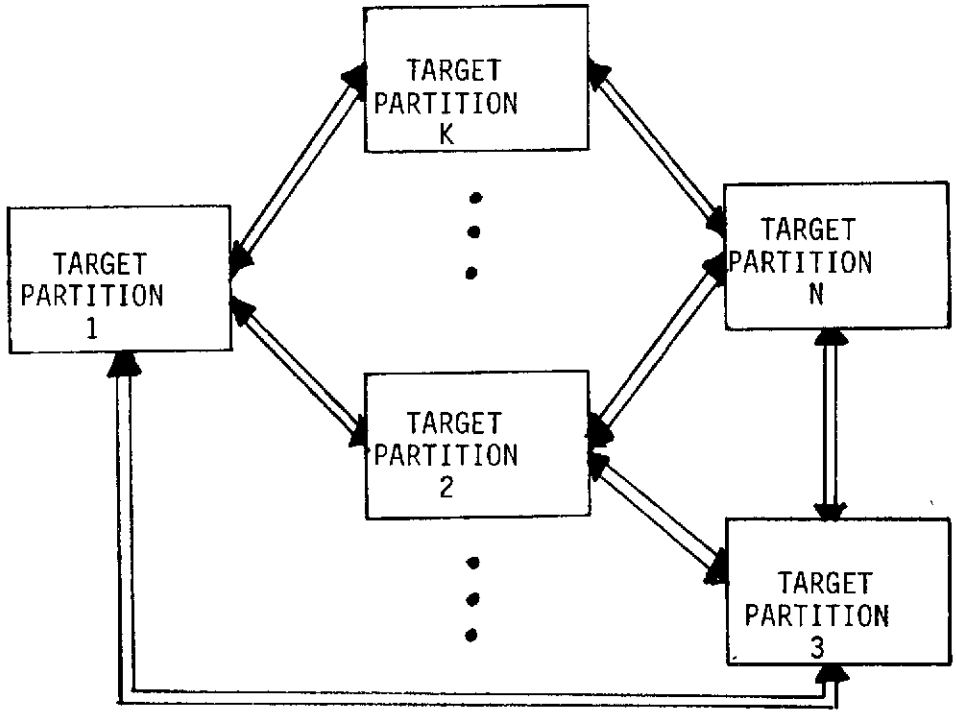
The two basic forms for instituting a federated system proposed are: 1) Hierarchy structure and 2) Distributed cooperative subunits. These forms are illustrated in Fig. 15. By simply extending the concepts of change initiated activation to permit external changes on information lines, the target process description does not restrict distribution of the process to host computers. Thus coordination and integration is simply a problem of how one member of the federated system identifies other members which must be notified concerning interface information changes.

The local process within one computer of the host system uses local activity information constrained by global status to manage the description subpart. It is necessary to prevent the processes in different computers from getting too far out of synchronization with other in time. The material effects of time spreads are that interface information between processors may be modified before its utility has expired. There is also



HIERARCHY CONTROL STRUCTURE

(a)



DISTRIBUTED CONTROL

(b)

FIGURE 15

the difficulty that one subsystem may "lap" another in the target process description. It is not necessary, however, to require all processors to operate in locked step so long as each can identify when to idle for the common good.

If the hierarchy structure is selected, control is centralized in higher levels. More global processes directly command lower level processors and indicate when to begin and end execution in the target description. For example, the supervisory machine will analyze global interconnection patterns to identify conditions (i.e., lines which change) which imply that coordination among computers is necessary; the subordinate process then executes in a free running fashion until the specified halt condition prevents further execution.

With the distributed control of the system, each target partition is responsible for cooperative action. Each partition is constrained to supply interface information to a common area (either globally common or shared between pairs of cooperating processes). In addition to the data value, the simulation time of the process producing the information must be supplied. Similarly, each processor is required to avoid overlaying information currently being used by other physical units. To accomplish this, P and V semaphores or some form of buffering scheme must be employed.

The decision to institute hierarchy or distributed control should be determined by merit analysis of each approach. Central

control is desirable when timing decisions are easily predicted and the supervision does not bottleneck the system. The amount of process supervision will probably decrease through centralization but concentrating it in one location may produce an overworked supervisory machine and substantially idle processing machines.

Decomposition into decentralized control will increase the number of supervision steps, but provide faster execution through parallel techniques. With decomposition, the potential for deadlock must be avoided to prevent simulation stalling where each description partition idles waiting for others to execute.

Allocation of target process. From the global target process description, processing portions are allocated to individual SVC host machines. This allocation is intended to speed execution through parallel techniques; several allocation guidelines contribute to this goal:

1. Match logical parallelism in target to physical parallelism in host.
2. Balance allocation to minimize idle time of each host processor.
3. Minimize information traffic among partitions to reduce overhead.
4. Maximize sharing of procedural routines.

The allocation is constrained by the physical characteristics of host computers, including the magnitude of resource availability (e.g. memory

size, processing speed, etc.) and configuration restraints (e.g. access to special peripherals, number of I/O ports, unique hardware capabilities, etc.).

By dynamic monitoring of the target process behavior during execution of user test cases, the allocation policy can be adapted to refine initial decisions. The interface data flow can be examined to determine the dynamic information flow among allocations and the location of "dynamic neighborhoods" in the target description. Using this information, partitions can be adjusted to balance the effort among physical host support computers.

Within each host processor, the machine resources can be assigned for the local target procedure to smooth interaction among federation members. For example, the partition might be initially implemented with linked subroutines. Through automatic monitoring of user test case executions, the primary information processing patterns would indicate those description components for which speedup techniques would most enhance system operation speed.

The most frequently used target description components and required SVC supervision functions would be converted to microprograms. Limitations are the number of opcodes unused by the target description subroutines and control storage space. By delegating these functions to microprograms, main memory space would be released.

The available free space in each host machine would be filled by converting subroutines to macros. This expansion would reduce

the execution and data linkage overhead of subroutine implementations.

SVC Host Design Estimates

To maximize the early completion of a usable system and minimize exploratory development costs, the initial SVC hardware and operation procedures should be simplified. It is assumed, therefore, that the initial SVC would be a dedicated system supporting a single investigation at any particular time.

The information interchange among minicomputers would be coordinated through a shared random access memory. This device would act as a buffer for information flowing between target partitions and a repository for infrequently required target processes. The local target information data and necessary execution routines would be permanently assigned to given minicomputers for the duration of target process execution.

Individual users would maintain physical control of their individual target systems, previously developed subsystems, and custom functional routines with magnetic tape and removable disk cartridges. At the session onset, the user would load the system with his development state and begin his investigation.

System support software, subroutine library, and special utility routines would be available from system disks and tapes. These would be available at the user's discretion. A card reader and line printer would provide initial offline development, documentation support, and post-mortem analysis for the investigation.

The main user interface during the session would be through the interactive CRT. Through this device, target subsystems would be constructed and exercised with requested results saved on the user's storage devices. At the end of the session, the developed target processors and investigation results would be dumped to his storage to clear SVC for the next user.

Host Hardware. The necessary equipment characteristics for an initial SVC host system and representative examples are:

1. Minicomputer- 16 bit machine, microprogrammable with writable control storage, 16K words of 1 usec memory, minimum of 8 I/O ports (Microdata MICRO 1600/21)
2. Communications core storage - 128K of 16 bit words, 750 nsec core storage (Standard Memories Model EF16K36CP5)
3. Disk system - 1.2M 16 bit words, 100KC word transfer rate, removable disk cartridge (Microdata Model 2852)
4. Magnetic tape transport - 800 BPI, 9 track, 25IPS (Microdata Model 2811)
5. Support peripherials
   - Line Printer - 250 lpm (Microdata Model 2732)
   - Card Reader - 300 cpm (Microdata Model 2720)
   - CRT Terminal - graphic display with keyboard and light pin interaction (DEC GT-40)

The specific equipment cited indicates available hardware with satisfactory performance specifications for use in the SVC. It is not suggested that these are optimal either in cost or performance but rather serve as examples

from which cost estimates can be developed (Fig. 16). With the possible exception of paper handling equipment, recent rapid improvements in hardware and drastic price reductions indicate that commitment to specific devices should be postponed as long as possible.

Software support system. To extend the present state of the SVC software system will require additional development effort. Some extensions will draw directly upon the experimental system groundwork software; others will require more research before full development is possible. Among the necessary SVC software components are:

1. Extension to current execution supervisor to accommodate external stimulation among partitioned target description elements.
2. Automatic analysis of the target process
   a. Adaptation of library subroutine code to perform data forwarding and serial execution linkage.
   b. Automatic activity monitoring to collect dynamic behavior information for input to the allocation policy.
   c. Distribution methods for distributing the target process among host computers using dynamic and static analysis information.
3. User interface software to process interactive target process descriptions and link previously catalogued descriptions. Symbolic reference support for user inquiries and demand monitoring.
4. Micro program translator to convert library process modules to microprograms and integrate them in the operational environment.

Estimates of required effort and costs for the necessary software development are presented in Figure 16.

From the analysis of Figure 16, developemnt of an operational SVC system will require a commitment of from $250K to $300K and development time of 1.5 to 2 years.

HARDWARE COST ESTIMATE

ITEM	QUANTITY	UNIT PRICE	COMPONENT COST
Minicomputer	4	$10,000	$40,000
Random Access Memory	1	30,000	30,000
Disk System	2	13,000	26,000
Magnetic Tape Transport	2	5,000	10,000
Line Printer	1	14,000	14,000
Card Reader	1	4,000	4,000
CRT Terminal	1	12,000	12,000
		Estimated Hardware Cost	$136,000

SOFTWARE COST ESTIMATE

SOFTWARE PACKAGE	EFFORT (MAN-MONTHS)
Supervisor	6
Target Process Adaptor	12
Performance monitors	3
Allocation software	6
User interface software	5
Microprogram translator	18
Estimated Effort	50 Man-Months
Cost @$2,500/man-month	$125,000

FIGURE 16

VI Conclusion

An operational methodology has been developed and examined for the construction of a Software Validation Complex. This test bed would substantially enhance current techniques of computer development and checkout for both proposed new systems and modifications to existing ones. Through a flexible host support system and automation of the support features the user is freed from many restrictions which impede his progress using current computer hosts. The SVC permits the host to adapt to the process under investigation rather than require contortions by the investigator to formulate the objective within host requirements.