NASA Contract Report 159095

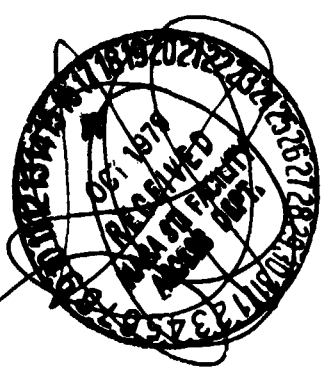
Integrated Testing and Verification System for Research Flight Software

Design Document

Richard N. Taylor
Randall L. Merilatt
Leon J. Osterweil

Boeing Computer Services Co.
Seattle, Washington 98124

Contract No. NAS1-15253
July 31, 1979



National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665
AC 804 827-3966

Page intentionally left blank

INTEGRATED TESTING AND VERIFICATION SYSTEM FOR RESEARCH FLIGHT SOFTWARE

Design Document

By

Richard N. Taylor
Randall L. Merilatt
Leon J. Osterweil

July 31, 1979

Prepared Under Contract NAS1-15253

Boeing Computer Services Company
Space and Military Applications Division
Seattle, Washington 98124

For

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

ACKNOWLEDGEMENT

Pages 27 to 56 (Section 3.4.2.3) of this document are contained in "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," a paper submitted for publication to the IEEE. Copyrights for that paper have been assigned to the IEEE. The work is reused here by permission.

CONTENTS

CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

GLOSSARY OF TERMS

anomaly	Deviation from the common rule; irregularity; includes both errors and "suspect" program operations.
assertion	A statement made to indicate the intent and nature of a program.
avail	A set of variable related information attached to each node of a flowgraph; used during static data flow analysis.
commensurate	Two units are commensurate if there is a declared relationship between them, such as inches = 12 feet.
concurrency	A state in which two or more processes (programs or tasks) can execute simultaneously.
data flow analysis	Analysis of the characteristics of the flow of data within a program whereby various classes of anomalies are detected, such as uninitialized variables, variables which are assigned values before the value from a previous assignment has been referenced, etc.
dynamic analysis	Program verification analysis which is performed during program execution.
flowgraph	A representation of the control flow of a program by a directed graph.
global	Information and/or actions which are known everywhere in a program.
HALMAT	HAL/S intermedidate code created by the HAL/S front-end processor and input to the HAL/S code generator.
instrumentation	The suite of automatically generated code which is inserted into a program to perform dynamic analysis.
invariant	Used with assertions to specify expressions whose values do not vary within the scope of the assertion.
keep	A statement which is used, at execution time, to save an intermediate value of a variable or expression.
live	Similar to avail.
local	Information and/or actions which are known only with a limited scope in a program.

monitor	A statement or sequence of statements which are automatically inserted into a program by analysis tools to perform a specific dynamic analysis function.
probe	Same as monitor.
process augmented flowgraph (PAF)	A flowgraph which has been altered to represent concurrent process control flow in addition to normal program control flow.
program call graph	A graphical representation of the procedures/functions which call and are called by other procedures/functions.
software development life cycle	The entire spectrum of software development which is decomposed into phases; phases commonly identified are requirements analysis, preliminary design, detailed design, and code.
static analysis	Program verification analysis which is performed on the program text and which does not require program execution.
symbolic execution	Execution of a program where variables are assigned symbolic values; used to construct formulas which represent the computations performed by the program.
testing	The actual or simulated execution of a given phase within the software development cycle.
validation	The process of verifying any given phase within the software development life cycle to the user requirements.
verification	The process of demonstrating the internal consistency of any given phase within the software development life cycle and checking to insure that it successfully captures and develops the intent of its predecessor phase.

INTEGRATED TESTING AND VERIFICATION SYSTEM

FOR RESEARCH FLIGHT SOFTWARE

Richard N. Taylor
Randall L. Merilatt
Leon J. Osterweil[1]

1.1 Summary. NASA Langley Research Center is developing the MUST (Multipurpose User-oriented Software Technology) program to cut the cost of producing research flight software through a system of software support tools. Boeing Computer Services Company (BCS) has designed an integrated verification and testing capability as part of MUST. Documentation, verification and test options are provided with special attention on real-time, multiprocessing issues. The needs of the entire software production cycle have been considered, with effective management and reduced lifecycle costs as foremost goals.

Previous verification systems generally have utilized a single technique, such as static or dynamic analysis. However, thorough examination of any one program requires the use of several techniques. Besides providing a comprehensive set of analytical techniques, the integrated capability BCS has designed takes advantage of the complementary abilities of the different schemes in a synergistic manner. A "one-tool-does-it-all" concept has not emerged though. The need for a distributed set of tools became clear as the various usage modes present in the MUST environment were modeled. No single sequence of testing and analysis activities is optimally suited to all MUST requirements. Rather, for detecting specific classes of errors under specific operating constraints, a specific combination of analysis techniques is chosen.

The concern with multiprocessing issues is motivated by the increasing sophistication of flight hardware and software, which present difficulties such as protecting shared data. New research was conducted into the problem of statically detecting such errors with encouraging results. Consequently, capabilities have been included in the design for static detection of data flow anomalies involving communicating concurrent processes. Some types of ill-formed process synchronization and deadlock also are detected statically.

Although the HAL/S language is the primary subject of this design, the algorithms developed are readily applicable to other languages. Full implementation of the designed capabilities will provide the MUST user with extremely powerful program development tools. Such programming environments offer a very desirable and profitable alternative to the way software is typically produced.

1.2 Document Organization. - The bulk of the design is represented by SAMM diagrams, attached as Appendix D of this document. In discussion of this

[1]Associate Professor of Computer Science, University of Colorado

design a synopsis of the design activities is presented in Section 2, followed by a discussion of key features in Section 3, where a rationale for the design decisions is also presented. Section 4 indicates how the design satisfies the relevant items in the requirements document, and explores some items marked in the requirements document as requiring further examination. The requirements document is found in Appendix E. Concluding remarks are presented in Section 5.

Appendix A provides an introduction to the SAMM methodology, showing how the diagrams are interpreted. Appendix B contains a presentation of the data base envisioned as associated with the software development environment provided by MUST. Appendix C contains a description of the built-in functions which will be available to the user of the interactive testing system--one of the design's tools.

1.3 Introduction. - Considered from the user's viewpoint the development of software may be conviently decomposed into several phases, as indicated in Figure 1.2-1. The end user determines his needs; those needs are translated into a more formal specification and are analyzed. Preliminary design work produces the basis of a solution to the problem. The solution is further refined at the detailed design level. Lastly, actual code is produced to implement the solution.

Figure 1.2-1 Phased Approach to Software Development

Notice that "testing" has not been included as a separate phase in this overview of the software lifecycle. Rather, it must be stressed that testing and verification are pervasive activities taking place throughout the development cycle. Such activities are indicated by the diagram of Figure 1.2-2. Each phase

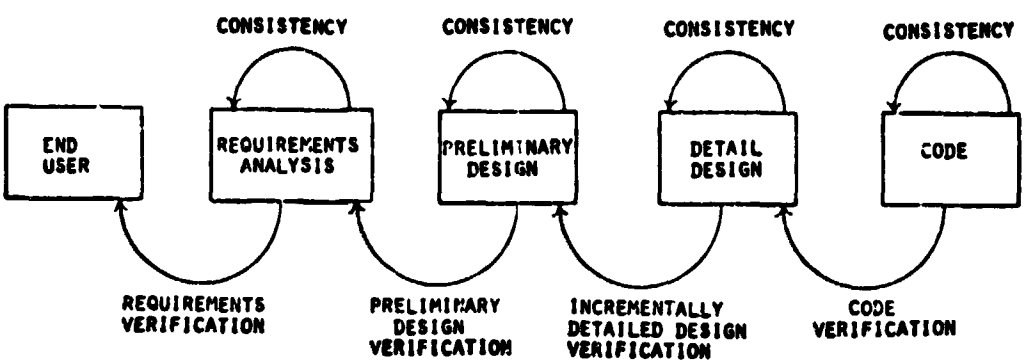


Figure 1.2-2 Lifecycle Verification

must be verified for internal consistency, as well as checked to ensure that it, as a refinement, successfully captures and develops the intent of its predecessor. The process of verifying any given level back to the user requirements is termed validation. Thus verification is not something which is "done" after a piece of code is written; on the contrary, all the tasks associated with the creation and maintenance of software are interwoven with various verification activities.

Figure 1.2-3 presents this view of the program development cycle in the specific context of the MUST system. It is this overview which provides the framework for the design of the individual verification and testing tools. Note that management activities to control and guide the development of the software are highlighted, with management providing direction at each phase. The basis for effective management is total visibility into the developing system, and is obtained through use of the system database, where each phase in the cycle uses and contributes information to it. This database is the repository for all information related to a software system. Note the correspondence between Figure 1.2-3 and the root of the SAMM model titled "System Development."

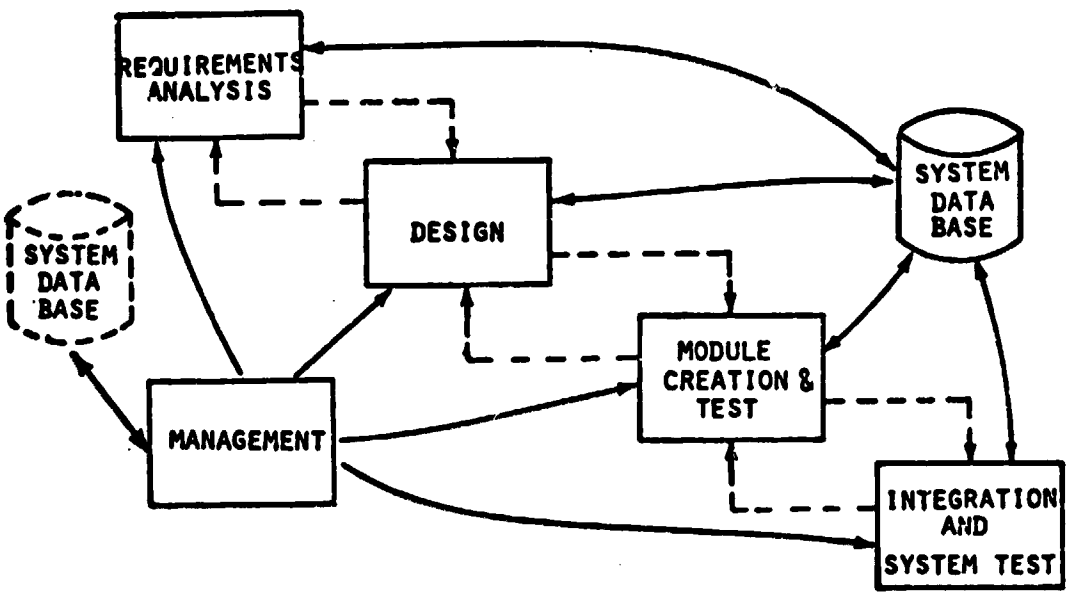


Figure 1.2-3 System Overview - Management of the Software Lifecycle and Data Flows

As the primary purpose of this contract is to design tools which specifically address the verification of HAL/S code, consider Figure 1.2-4. This figure illustrates many of the activities which are associated with verifying a module of source code. Internal verification, to be expanded upon shortly, is performed first, detecting as many errors as possible. Next, the intermediate representation of the program is targeted to the specific computer (or simulator) on which execution is to take place. Test data is created to validate that the acceptance criteria are met, then the program is executed. After execution, output values are examined as well as several aspects of the program's performance. Analysis may reveal the need for additional testing. If so, additional data is generated and the cycle repeats. (Figure 1.2-4 is related to SAMM node CC.)

Several options are available to the user concerning the type and amount of internal verification to be performed. Chapter 2 elaborates on the rationale behind providing a variety of ways in which the verification and testing tools can be combined. For the moment however, Figure 1.2-5 (a combination of aspects of SAMM nodes CBC, CCB, and CBCC) presents an overview of the facilities available to the user. (As alluded to earlier, the verification tools operate on an intermediate representation of HAL/S, produced by the compiler, known as HALMAT.)

Several tools may be implemented to provide the facilities noted by each box. Briefly we note that box A is not the full HAL/S compiler, but only the front half which checks the syntax, parses the program, and generates the HALMAT. Box B processes program assertions (statements made to indicate the intent and nature of the program) having program-wide significance. /*ASSERT GLOBAL X<=0 */; would be an assertion in this category. Box B would insert the necessary monitors to check that this requirement will be met throughout the program. If at any time it is violated, an informative message will be produced. Non-data flow static analysis may involve the use of several tools to perform its tasks, such as creation of helpful cross reference maps, checking for mismatches of units among program variables (such as adding feet to meters), and ensuring shared procedures are reentrant. Data flow analysis checks for errors including uninitialized variables and ill-coordinated procedures. Symbolic execution determines the functional effect of a specified program path. Functional testing allows interactive, interpretive execution of specified program segments. Lastly, if any program instrumentation is called for, it is inserted in the HALMAT at box F. Such instrumentation is the executable code required to perform verification tasks during program execution.

The following sections explain the design more fully, and indicate the hierarchical structure of the facilities.

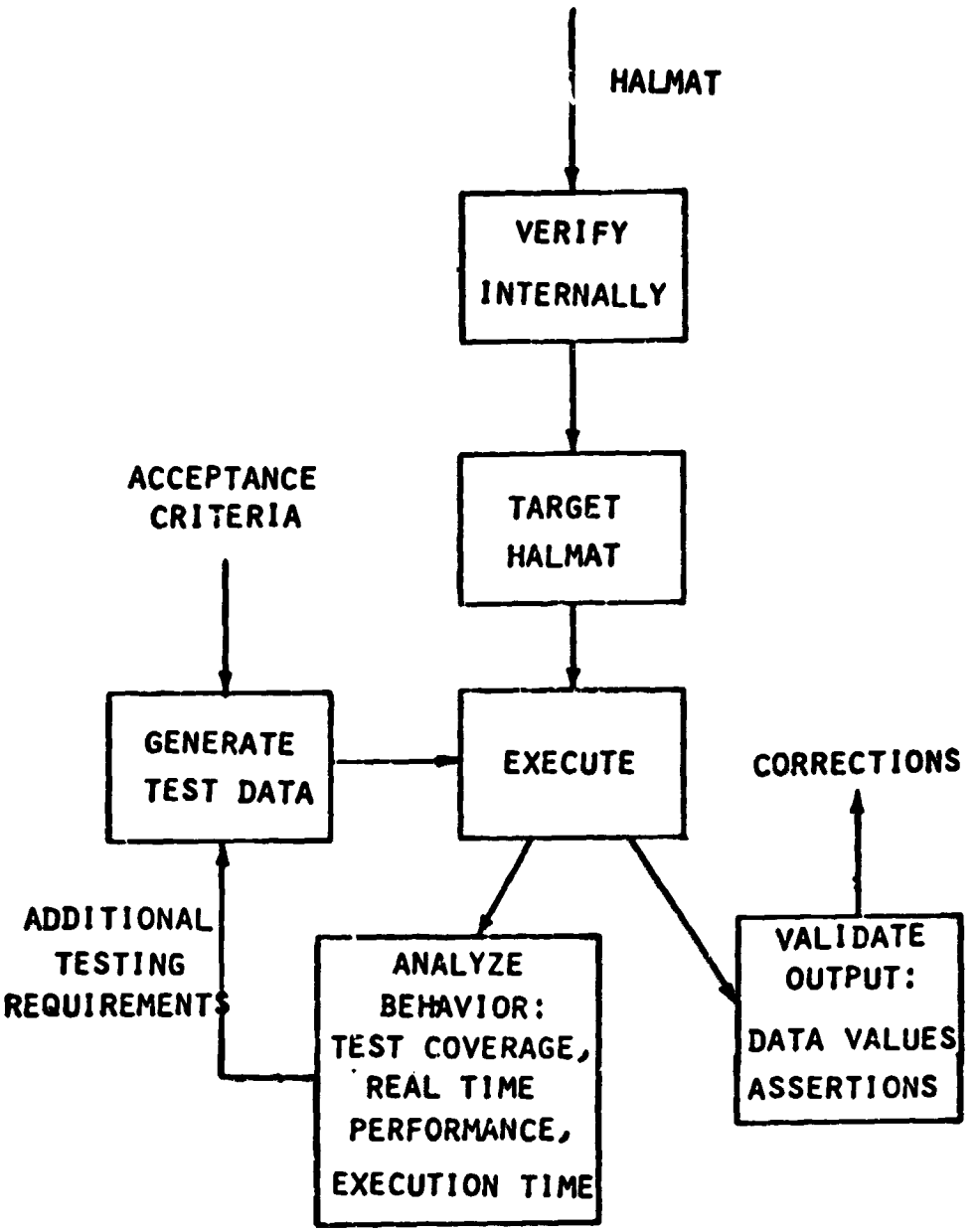


Figure 1.2-4 Source Code Verification and Testing

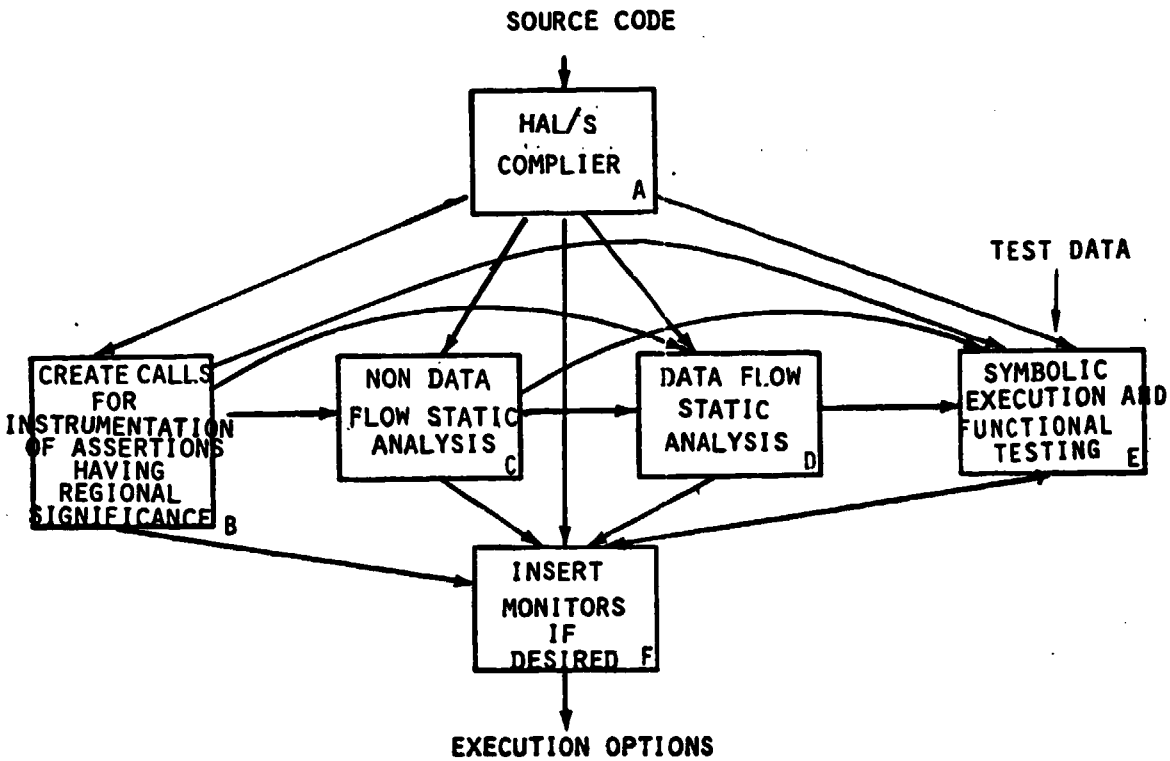


Figure 1.2-5 Module Verification Options

SECTION 2.0

Synopsis of Design Activities

As explained more fully in Appendix A, BCS has developed the Systematic Activity Modeling Methodology (SAMM) to aid in requirements analysis and the formalization of preliminary design. This formalism was chosen as the vehicle for expressing the preliminary design of the MUST verification and testing capability. In so doing hierarchical relationships among activities are clarified, data flows and dependencies are indicated, and critical functions are identified.

A SAMM model presents a hierarchical breakdown of an activity. Initial difficulty in using SAMM in the design of the verification and testing capability was laid to inadequate consideration of the actual user modes which would be present in the MUST environment. Once specific user tasks were identified preliminary design proceeded smoothly. An outfall of this was a deepening of our conception of how verification and testing tools should be integrated. Reference 1 [Osterweil, 1977] presents a scheme in which the techniques of static analysis, symbolic execution, and dynamic analysis may be combined so as to provide a single, comprehensive analysis tool.

Figure 2.1 presents the basis for the integration methodology proposed by the paper. Static analysis begins by detecting several classes of errors. Unfortunately some "errors" may be reported which in fact do not exist, since the "error" lies on a path which is not executable. In addition, the static analyzer may note statements at which an error might occur. This information can be passed along to the symbolic executor for further analysis. Symbolic execution may be able to determine whether or not a particular path is executable, and indeed may show that a suspicious construct is definitely erroneous. In addition, the symbolic executor could generate test data which would force program execution down the erroneous (or any other requested) path. Thus a link exists to the next phase: dynamic analysis. Using the generated test data, the program may be executed. While execution is proceeding, information can be gathered indicating the steps taken in the progression to the error, as well as reporting conditions prevelant at the time of error.

Figure 2.1 Basis for Integrated Verification Methodology

To summarize the paper, the three techniques complement each other and may be used in tandem. The generality and usability of the techniques vary widely however, as does their execution cost. It was consideration of these differences and the usage modes present in the MUST environment that led to our revised concept of how the tools should be integrated.

It now appears that when specific tasks in the creation and maintenance of a program are identified, different analysis modes are required. Each mode is subject to different constraints: goal, thoroughness required, available budget and time, degree of human interaction, and so forth. The synergistic combination of techniques is still called for and profitable, but not all the techniques will be required for any one analysis task. Rather, for each task an appropriate subset of the techniques will be combined which optimally addresses the problem.

This modeling activity has shown the need for small, modular facilities which may be combined in a variety of ways to accomplish many different tasks. Each combination would be configured to meet the constraints of differing goals and environmental (resource) requirements. Some of the modular facilities which have emerged are as follows: a facility to process "regional" assertions, a facility for local assertions, a tool for extracting internal documentation, one for answering simple questions about previously written code, several simple static analysis tools (an auditor, a units and scale checker, a cross reference map generator, and others), a data flow analysis tool, an execution time monitoring package, and a facility for inserting run time monitors. Each tool meets a particular need and, in conjunction with other tools, helps satisfy a global verification requirement.

Another dominant feature of the design is the pervasive use of a machine readable database of program related information. This database is begun with the requirements phase, and is updated and maintained throughout the entire software lifecycle. As a repository for the growing knowledge about the nature and solution of a given problem, the data base is a natural device for smoothing the transitions from requirements to design to coding to maintenance. It is this database which makes possible the verification and validation of each step in the development cycle. Such a database also provides a secure foundation upon which effective program management can be based.

The program data base concept was adopted early in the preliminary design process, and is an outgrowth of research into software lifecycle costs performed by members of the Space and Military Applications Division.

As verification of real time, concurrent process software is a poorly understood aspect of error detection called for in the requirements document, design effort was spent in basic research of the problem. It was believed that basic principles of error detection in this area must be understood before designing the entire verification and testing capability, to avoid any later requirement for restructuring, and so that an estimate could be obtained concerning the promise of analysis in this area. Significant results were obtained indicating the techniques and principles discovered are harmonious with the error detection techniques employed with single process programs.

In particular, it was discovered that the program flow graph for a system may be augmented with special edges indicating the concurrent processing constraints. If slightly modified data flow analysis is applied to this graph (called a process-augmented flowgraph or paf) data flow anomalies occurring between parallel processes can be detected. Importantly, this analysis can be performed concurrently with the detection of single process errors.

Auxiliary design activities included extensive literature surveys on various analysis techniques and further investigation into diverse topics, such as the University of Texas FAST system, the HAL/S compiler operation, the FSIM compiler capability, and several existent symbolic execution systems.

SECTION 3.0

Key Design Features

3.1 Tool Integration and Modularity. - The dominant characteristic of designs represented by the SAMM formalism is that they are purpose-oriented. Each task, or node, is present simply to fulfill the requirements of a higher level activity. No activity is present "for its own sake." The result is that all the tools included in the design function together for the purpose of creating better flight software with less total effort and cost.

The usage scenarios considered during the preliminary design were the following: creating a new software system, managing the development of a software system, adding a new capability to an existing system, performing "minor maintenance," documenting an existing system, module test, integration test, and the development of software by a team of programmers. Initially each scenario was examined separately, then jointly as similarities, dependencies, and interrelationships were discovered.

The examination of the various user modes envisioned has resulted in the isolation of several basic capabilities. In various combinations the capabilities represent the environment required for each user mode. Within a particular usage scenario, select capabilities may be side stepped in accordance with various constraints and desires. This integration and modularity of tools is particularly evident in the tools provided for the verification and testing of HAL/S code. For example, the instrumentation of global assertions within a module is separate from the instrumentation of local assertions; instrumentation of multi-module assertions is distinct as well. Static analysis and symbolic execution may both aid in determining the placement of monitors (adding and deleting them); non-data flow static analysis may be chosen apart from data flow static analysis. Several criteria may be involved in chosing a particular combination of tools. The type of verification desired, execution time, memory requirements, run-time overhead, and target machine capabilities may all affect the selection process. The following are a few representative combinations:

1. Isolation of a particular, relatively simple, "bug": dynamic analysis and functional testing with extensive assertion usage, placing most emphasis on the single suspect module.

2. Initial verification of a new piece of code: static analysis--both data flow and non-data flow.

3. Broad based verification, with few budget and time restrictions: static analysis, extensive symbolic execution and functional testing, and assurance of full test coverage through dynamic analysis.

4. Isolation of a difficult functional error (e.g., the program computes a slightly wrong value): symbolic and functional execution of appropriate paths, with dynamic analysis.

5. Verification of a collection of previously (internally) verified modules, now joined in a parallel processing environment: multi-process data flow analysis and static checking of integration requirements, followed by dynamic analysis of the concurrent process characteristics (such as process queue snapshots and monitoring for parallel processing errors).

Such a philosophy pervades the design. As automated tools are eventually required for requirements and design specification and analysis, such construction will be desirable and possible there as well. Indeed, the types of analysis required for such specifications will be very similar in nature to those required for actual code.

The most important model presented in the SAMM diagrams of Appendix D is that of creating a new software system. By extensively decomposing it, the scenarios of management, testing, and team development are included. Adding a new capability to a system may be modeled by emphasizing a particular path through the system creation model and making a few minor modifications.

The same is true for "minor maintenance." That activity implies a small change in the design (or requirements) of a module; coding changes are made, testing and integration is performed and the system is released. Thus in Appendix D, only two complete hierarchies are presented: system creation and documentation. Duplication and excessive detail are thus avoided.

3.2 System Database. - As introduced in chapter 2, the concept of a comprehensive machine readable database of program related information is inherent to the design presented. This database forms the basis for orderly program development and effective program management. All the information related to a particular program is present in this database. Documents, formal specifications, test data, program output, source code, and management reports are all included. Such inclusiveness allows the rapid determination of any needed program related information. The centrality of the information prevents wasted effort in consulting separate sources. More importantly, the database may be systematically monitored during program development to ensure that all the components are generated in a timely manner. This is essential as the progression from one phase of the software development cycle to the next is dependent upon full information being available from the previous phase.

Such considerations may be carried further with the immediate observations that communication among development team members is increased, visibility into the developing system is promoted, analysis may be performed and reviewed in a systematic manner, testing activities may be scrutinized for thoroughness, and documentation may be readily distributed and updated. Clearly management functions are enhanced and the efficiency of the development operation is increased.

A less obvious but critical outflow of the use of the system database is in the maintenance function. The term "maintenance" is used to describe a variety of activities, usually everything occurring after the initial release of a piece of software. Typically this includes alteration of requirements, followed by design, coding, and testing functions. The use of the system database allows such activities to proceed in an orderly manner as the information contained in the database provides a complete history of the development process. Thus the effect of small changes in the requirements may be readily traced on to the design, then to the code, and so forth. At each stage the historical information allows the "maintainer" to determine the impact of proposed changes. Proper development may then proceed.

A further discussion of these concepts in a general setting is found in reference 2 [Osterweil, Brown, and Stucki, 1978].

3.3 HALMAT. - An intermediate representation of the HAL/S language, called HALMAT, is used as the primary representation of the programs analyzed by the various tools. In so doing, the separate tools do not have to perform any parsing, thus saving much time and effort. Additionally, the tools are largely isolated from syntax changes to the language.

3.3.1 Relating Verification Error Messages to the Source Text. - All error messages which the verification facilities produce should be related to the source code, and phrased in a manner readily understood by the user. The output writer (described in Section 3.9) produces a source listing of the compiled program, marking each statement with a unique statement number. All error messages can be keyed to these numbers. If the source listing is not created until after the verification tools have been executed, the error messages from all the tools can be included directly in this single listing.

This all may be done by working directly with the HALMAT. There exists a one to one mapping from the HALMAT "paragraphs" to the source statements. Even HAL/S statements which do not generate any executable code (such as declarations) create a HALMAT paragraph. Each paragraph contains a field with the originating source statement number on it. The statement numbers also appear on the listing.

To form comprehensible error messages the symbol table is also required. From it (and the other tables) the symbolic variable names created by the user may be incorporated in the messages.

3.3.2 HALMAT Monitor File. - The design presented contains several tools which may request that monitors be inserted into the program under analysis. In addition, the integration philosophy employed allows the specification of monitors to be successively refined. A specialized capability may reveal that certain dynamic monitors are unnecessary, as the conditions prevailing at that point in the program are known a priori.

The medium upon which the analysis tools operate is HALMAT. The monitors need to be placed within the HALMAT, and must therefore eventually be HALMAT. To allow the flexibility needed as indicated above, it is therefore recommended that two files of information be kept in parallel. One file will be the HALMAT produced as a result of program compilation, the other will be an evolving file of monitors. When all analysis tasks are completed and the final set of monitors is decided upon, the two files may be merged into a single file of HALMAT. This file is then ready for code generation and execution.

One clear advantage of this scheme is that the internal pointers in the program's HALMAT only need to be modified once. Execution of a statement in the program may require the value of a previously computed expression. The HALMAT contains a pointer to the statement where the expression was computed. If a monitor is inserted between the expression evaluation and its use, the pointer must be appropriately altered. With the proposed scheme this alteration will only occur once: when the HALMAT and monitor files are merged. Any implementation restrictions concerning checksums or the number of paragraphs which may be stored in single record may be met at this time as well.

HALMAT's paragraph notion allows the mapping between the monitor file and the program file to be particularly simple. The SMRK instruction which delineates the HALMAT corresponding to a single source language statement contains the number of that statement. Thus when the compiler places ASSERT and KEEP statements on the monitor file, it may reference them to the HALMAT by simply including the appropriate statement number in the monitor file. Some monitors will definitely require mapping to specific HALMAT instructions, though. In this case a second level of mapping will be required: first, a pointer to the proper paragraph (SMRK), second, a pointer (offset) to the proper HALMAT operator within the paragraph.

The various records within the monitor file will evolve through several stages. At any time the file may contain monitors in various stages of development. The monitor file will first emerge from the compiler (node CBCAAB), and will contain a representation of the ASSERT and KEEP statements encountered by the compiler. Such records will have expressions parsed into HALMAT, but will not contain the logic necessary to implement the required monitor. Node CBCB(C) performs this development. Later, the static analyzers may insert monitors which are highly "developed" - checking for a very specific error. At a later point, these monitors may be removed, or "turned off." If a monitor is turned off, it does not have to be removed from the monitor file--a switch may be set. Some monitors may only be developed when system level testing is begun. In such a case they will remain unexpanded throughout module test, and will be skipped over during the merge phase between the HALMAT and monitor file.

Several factors enter into the specification of the structure of the HALMAT Monitor File (HMF). Foremost is that the HMF is an external data structure to several tools. From its initial generation to its ultimate merger with the HALMAT file (or other disposition) several distinct tools read and update the file.

Operating system utilities may be used to preserve copies of it, and a user interface will use the utilities to move the HMF from tool to tool. Next, the records on the HMF will be of several different logical types - some corresponding to assertions, others to keeps, and so on, as mentioned.

To permit such flexibility we recommend the following. For this presentation we shall assume that Pascal is the implementation language and that only sequential file structures are available. More sophisticated file facilities would simplify handling of the HMF in obvious places.

The HALMAT Monitor file may be defined as:
HALMAT_Monitor_File = **file of integer** ;
Several basic routines -utilities- are then provided for convenience in manipulating the file and to ensure integrity of the data structures.

The **file of integer** is logically broken down into records. Each record has a fixed format header, or "prologue". Each header contains a word marking the beginning of the record and a word indicating the length of the record (i.e. an offset to the start of the next record). Following this is a description of the record's contents.

The utilities, callable by any of the tools, would include procedures to 1) read a record's prologue (and thereby discern its type), 2) skip a record, 3) skip to the next record whose prologue met certain criteria, 4) read a record into a data structure of appropriate logical type, and 5) write a record of a given type. (We thus assume the ability to "translate" an integer word on the HMF into a Pascal data type. Such a feature is given in Pascal through the use of variant records which omit the tag field. This is akin to the PL/I unspec feature, or the HAL/S %copy.)

A Pascal description of the logical structure of the prologue follows.

```
const
    record_marker = maxint ; (* Note that the use of this number as the marker
    is not an infallible guide to determining record boundaries. This must be
    done by counting record sizes from the beginning of the file. *)

type
    monitor_level = ( system, program, task, procedure, block, local) ;

    monitor_types = (*The following list may be expanded as need arises *)
        (initial_assert, (*the first representation of an assertion*)
        initial_keep, (* the first representation of a keep *)
        assert, (* an algorithmic representation of an assertion *)
        keep, (* an algorithmic representation of a keep *)
        elem_units_decl, (* elementary units declaration *)
        units_relationship, (* a relationship between units *)
        units_spec, (* a units-variable binding *)
        subscript_monitor, (* check for subscript out of range *)
        zero_divide_monitor, (* check for division by zero *)
        division_ratio_monitor, (* check for bad divisor/dividend sizes *)
        error_monitor (* other monitors created by analysis tools *)
        ) ;

    string = packed array (1..10) of char ;
        (* used for tool identifiers *)

    development_status = (incomplete, complete) ; (* indicates whether the
    monitor is ready for insertion in the HALMAT file (or output on a listing) *)

    prologue =
        (* ordered *) record
```

```
        marker: integer ; (* = record_marker *)
        length: integer ; (* length of the record, counting the marker*)
        monitor_type: monitor_types ;
        active: record
                    switch: boolean ; (* is or is not currently active. (This
                    could be changed to a level indication, if needed) *)
                    determinor: string (* which tool last set the switch *)
                end ;
        origin: string ; (* which tool was responsible for the monitor's
        original generation *)
        SMRK: integer ; (* pointer to applicable SMRK in HALMAT file
        *)
        SMRK_offset: small_int; (* offset into interior of a HALMAT
        paragraph where monitor belongs *)
        level: monitor_level; (* highest program unit level at which this
        monitor can have effect *)
        status: development_status ;
end ;
```

Within the scheme of a sequential file structure, all the records on the HMF should be kept in an order corresponding to the order of the HALMAT file. Such a restriction will simplify processing of the HMF. All monitors corresponding to a single SMRK will thus be together. In order to maintain this order each tool which updates the HMF must in effect create a new HMF which is equivalent to the old HMF with the appropriate changes made.

3.3.3 Merging HALMAT and the HALMAT Monitor File. - Several tools modify the HALMAT Monitor File (HMF) in the course of various verification activities. Some tools contribute new monitors, others verify that certain previously created monitors are unnecessary. When all the modifications to the monitor file are complete, the monitors which have been selected (Section 3.6.7) to be inserted as inline code must be placed into the HALMAT file. The operation is simple in concept, but many details are involved.

The merge activity is found in the SAMM diagrams at node CBCD. It will be discovered there that two basic operations are involved. The first is the creation of HALMAT code to represent the algorithmic instrument required; the second is the insertion of this HALMAT into the existing file of HALMAT (which represents the user's program).

The HALMAT Monitor File is structured such that the mapping between the instruments and the user program is very clear. Assertions and keeps are mapped to their own statement number. Error monitors are mapped to the statement (or portion thereof) to which they apply. All monitors which pertain to a given statement number can easily be found when processing the monitor file.

For each type of instrument (i.e. for each algorithmic instrument) a more or less fixed pattern of HALMAT may be used. Each pattern may be formed by coding the algorithmic representation of the instrument in HAL/S, then putting it through the front-end compiler (which generates the HALMAT). The HALMAT may then be inspected and pointers to the symbol table (variable references) may be noted. For error monitors, these pointers are the only aspect of the HALMAT which will vary from one instance of an instrument to another. The bulk of the HALMAT which corresponds to ASSERT and KEEP instruments will represent the computation of the comparison or expression involved in the statement. The HALMAT to perform this computation will (largely) be created by the front-end compiler, at the point where it must recognize the assertion/keep statements. Formation of an instrument, therefore, consists of taking a HALMAT template which corresponds to the type of instrument being processed and filling in the pointer (symbol table) references appropriately. A virtual accumulator reference to the comparison or expression computation must also be appropriately made.

The actual merge operation is similar to the formation of the instruments. The HALMAT file may be processed sequentially from the start. If a SMRK is encountered which has a HMF pointer to it, the instrument is placed into the HALMAT file "at that point." Virtual accumulator references in other parts of the HALMAT file may require alteration as a result of the insertion, however. Once these changes have been made, processing may continue with the next SMRK.

3.4 Static Analysis.

3.4.1 Units/Scales Specifications and Algorithms. - The implementation of this facility will follow the recommendations of reference 3 [Karr and Loveman, 1978] very closely. The following items need to be considered.

1) Basic principles and options available to the user
2) Specification of elementary units
3) Specification of relationships among units
4) Declarations of variables having unit/scale mode
5) Algorithms for checking/enforcing adherence to unit and scale commensurateness or equality
6) Issues to be resolved

Subsequent correspondence in Communications of the ACM (October, 1978) supports the design chosen. Previous implementations have been successful and very helpful to a wide variety of users.

3.4.1.1. Basic Principles and Options Available to the User.

- Error detection will not inhibit code generation.

- There will be two basic operating modes, selected by a switch. In the default mode the facility will require "corresponding" expressions to have equal units. If equality cannot be verified, commensurateness will be checked.

Example:

```
DECLARE CONSTANT /* ELEMENTARY_UNIT*/ (1), feet, inches, volts,
                                watts, amps;
DECLARE  /* UNIT: feet */ f1, f2;
DECLARE  /* UNIT: inches */ i1, i2;
DECLARE  /* UNIT: volts*/ v;
DECLARE  /* UNIT: amps*/ a;
DECLARE  /* UNIT: watts*/ w;
/* RELATIONSHIPS: Inches = 12 feet; watt = volt amp; */ ;
```

```
(1)     f1 = 4 feet;
(2)     i1 = inches;
(3)     f2 = f1 + i1/12;
(4)     f2 = f1 + i1/3;
(5)     a = 0 amps;
(6)     v = 5 volts;
(7)     w = v a
(8)     w = 16 v a
```

In statements (1), (2), (5), and (6) the units of the right side of the expression exactly match the units of the left side: no error or message is generated.

In statement (3) the units of the expression i1/12 might be feet, considering the relation inches = 12 feet, but, as seen in statement (4) with expression i1/3, this is only an assumption. Does i1/3 represent 4 times i1 converted to feet? Or is it a logic error? In statements (7) and (8) the units of both expressions are clearly watts - no ambiguity arises even though the factor 16 is involved.

Therefore, we restate our principle as follows: If, when manipulating the units of two expressions for comparison, the application of a units relation involving a constant is required, only commensurateness will be assured, not equality.

Inches is commensurate with feet, but not equal. Watts are equal (and thus obviously commensurate) with volt-amps.

In any message indicating two expressions are commensurate but not equal, the system will indicate what (unit-less) factor must be applied to guarantee equality. In so doing the programmer may visually assure himself that such a factor has or has not been applied.

We note again that this is the default mode. In optional mode it is assumed that the programmer will not insert any conversion factors. The system will determine what factors, if any, are required and insert them in the code automatically. A notation will be provided indicating what factors have been applied.

3.4.1.2. Specification of Elementary Units and Scales.

3.4.1.2.1 Units. Two objectives are accomplished by the scheme for declaring elementary units described below:

1) The domain of units to be used in the program is defined.

2) A device for manipulating units is provided: variables having a units attribute may be safely initialized, and the units attribute may be "stripped off" a value when required.

Scheme: Declarations of the following form must be included for each elementary unit to be employed:

DECLARE CONSTANT /*ELEMENTARY_UNIT*/
(identity value for the type of the unit)
type declaration, list of elementary unit names;

It is anticipated that the only types to be employed will be integer and scalar, and the identity value will therefore be one.

Example:

DECLARE CONSTANT /*ELEMENTARY_UNIT*/ (1)INTEGER, apples, oranges;
DECLARE CONSTANT /*ELEMENTARY_UNIT*/ (1.0) feet, meters;

In illustration of item 2), variables possessing these unit attributes may be assigned values in the following (safe) manner:

f = 4 feet;
m = 6.25 meters;
x = 6 apples;

3.4.1.2.2 Scale. Elementary scale factors do not require declaration, as is the case with elementary units. The following scales are automatically declared: the integers 2,4,8,16,32,64.

An integer variable declared to possess elementary scale 4 is to be interpreted as possessing the value: (integer value)/4. In other words, there is an implied binary point 2 bits from the right end of the integer word.

Note: If Language Change Request #147 (FIXED type) is adopted and implemented in the Langley HAL/S compiler there will be no need for this facility.

3.4.1.3 Specification of Relationships Between Units and Scales. After all units to be employed in a program have been declared, relationships among them may be set forth. Such relationships are indicated by the following statement:

/* RELATIONSHIPS: list of relations */;

Example: /* RELATIONSHIPS: feet = 12 inches, watts = volt amps */;

Note that a relationships specification is a (null) HAL/S statement. As such it must follow the declare section of a program or procedure. Only simple arithmetic relationships may be declared, involving only multiplication, division, and exponentiation. (Relations such as a=b+c do not normally have much utility in engineering/scientific applications, with the possible exception of conversion from ^{o}C to ^{o}F. For such conversions a sequence of relationships may be defined, which together allow complete checking.)

The utility of constant values in relationships is subject to the considerations of Section 3.4.1.1, namely if, when manipulating the units of two expressions for comparison, the application of a units relation involving a constant is required, only commensurateness will be assured, not equality. Relationships between scales do not have this restriction.

/* RELATIONSHIPS: 8=4 2 */; defines a valid, useful relationship. At the user's request, default relationships such as this could be automatically defined.

3.4.1.4 Declaration of Variables Having Units/Scale Attributes.

- Variables may have both units and scale attributes.
- All units must be declared before the variable is declared.
- No variable may have more than one unit attribute, or more than one scale attribute.
- Declaration of variables having these attributes is accomplished by inserting the special comments described below in with other attributes of a declaration.

Syntax: /*UNIT: arithmetic expression involving previously declared unit(s)*/

/*SCALE: integer scale value */

These declarations may be contained in a single comment if both scale and unit attributes are requested.

Concerning the implementation of these features, two vectors (in the sense of the reference) will be associated with each variable: one containing units information, the other containing scale specifications.

3.4.1.5 Algorithms. The algorithms employed in the analysis task will be those of the reference. No changes are anticipated.

For the default situation described in Section 3.4.1.1, the analysis algorithm acts within the following framework.

```
check expression commensurateness, ignoring any numeric factors;

if commensurate
    then
        if computed factor ≠ 1
            then
                issue "factor required" message;
                print factor needed
            else
                no factor needed
        fi

    else
        print error message
fi
```

3.4.1.6 Issues to be Resolved.

1) The scope of declarations of elementary scale/units and relationships. Is the scope global?

   Yes: The information is, in a sense, global knowledge;
   Implementation would be simpler
   Other possible mode attributes such as INTEGER, SCALAR are global.

   No: Variables are not global. Should their definable attributes be?

   It may be desirable to override "global knowledge."
   "Yes" is contrary to the principle of information hiding-incompatible code could result from 2 different programmers.

2) Should there be a facility for making "enforced remarks about expressions" in the sense of Section 6.2 of reference 3 [Karr and Loveman, 1978]?

3.4.2 Static Data Flow Analysis. - The data flow analysis techniques described in this design are due primarily to the work of Fosdick and Osterweil of the University of Colorado. Most of their work, directed at the detection of errors in FORTRAN programs, is directly applicable to HAL/S code. The construction of the DAVE system to analyze FORTRAN programs has provided a test bed for evaluation of the techniques and their effectiveness in detecting anomalous data flow. The experience with DAVE allows the design of a capability for HAL/S to be approached from a knowledgeable position.

Several items may be noted about DAVE. First, the system detected an interesting class of errors which was of definite benefit in verifying a program. Often the errors detected were very "simple" - yet examination revealed that they resulted from deeper problems in the program's construction.

Secondly, DAVE was constructed as an experimental program before some important analysis algorithms were recognized. This revealed itself in the speed and size of the system - it was big and slow. Students wrote much of the code, and it evolved over a period of time. As a result, it is hard to modify to improve its characteristics.

Thirdly, many of the error messages produced by DAVE referred to phenomena which occurred only along unexecutable paths. The analyst was thus faced with the chore of separating the true errors from the spurious. Often this was simple, yet it represents an undesirable characteristic.

Lastly, DAVE has proved to be unwieldy in many production environments simply because it requires the source input to be ANS FORTRAN (1966). No language extensions are allowed.

In designing the static analyzer for HAL/S, we have taken cognizance of these characteristics, as well as recent advances made in the area. We may therefore describe aspects of the design as follows.

1. The static analyzer for HAL/S relies on the compiler to do all the parsing required. The analyzer thus begins its chore with the creation of the program flowgraphs, and the annotation of the program nodes with bit vectors conveying information about the activities which transpire at the nodes (as required by the analysis algorithm). Any language extensions or syntax changes will thus have minimal impact upon the analyzer.

2. The most important part of the static analyzer is the algorithm employed to detect the errors. The HAL/S analyzer will employ the so-called "parallel-bit" algorithms developed by Allen, Cocke, Hecht, Ullman, and others. These algorithms and references to them may be found in reference 4 [Fosdick and Osterweil, 1976]. As a result, the time for analysis of a program should be on the order of its compilation time.

3. The expressive power of HAL/S is much greater than Fortran, so the analysis techniques have required extension. The two major additions to the language (as far as static analysis is concerned) are the real-time, concurrent

processing statements and the NAME, or pointer variable, capability. Of the two, the concurrent processing features has presented the greatest challenge. The NAME facility is just another aspect of the aliasing problem.

In response to this, considerable effort was devoted to the concurrent processing problems, resulting in an initial paper describing the results (reference 5) [Taylor and Osterweil, 1978]. The initial results have been further pursued, resulting in several firm algorithms for detecting important anomalous conditions. The anomalies which are detected include both data flow and control flow/synchronization errors. The design presented incorporates these results. Further research in this area is called for, though, as only a few concurrent control structures have been considered to date. The latter part of this section is a full presentation of the results obtained.

4. Since HAL/S is not a recursive programming language, the same processing scheme may be taken as for FORTRAN programs: a "leaves-up" approach. Thus the unresolved problem of applying static data flow analysis techniques to recursive programs did not have to be addressed.

5. To prevent the generation of spurious error messages representing phenomena occurring along unexecutable paths, the techniques of reference 6 [Osterweil, 1977b] will be employed. These techniques use the parallel-bit algorithms in the basic analysis tasks, but a new post process is added. A substantial improvement in the quality of error messages produced is anticipated.

More specifically, the techniques described in reference 6 have been refined into specific algorithms which may be directly incorporated in the HAL/S data flow analyzer. At the heart of the techniques are the same LIVE and AVAIL algorithms which are employed in the anomaly detection process. These techniques have been developed by Lee Bollacker of the University of Colorado who has described them in a University of Colorado Technical Report (reference 29). During the course of the design effort this general problem has been examined and it is felt that this technical report adequately details the procedures to be employed. It will not, therefore, be considered further here.

6. In order to generate the most helpful error messages and to provide analysis paths for the interactive testing system (which includes a symbolic executor), a post processor will be used to generate all messages. The parallel-bit algorithm detects errors at nodes only. To relate those errors to the paths along which they occur requires another technique: depth first traversal. Though this procedure is slower than the parallel-bit algorithms, the time penalty is only incurred when an error is discovered. The overhead penalty should therefore be relatively small. Interest has been shown in reducing this penalty however, and an indication as to the approach to take may be found in reference 30 [Gallucci, 1978]. For a detailed discussion of the depth-first algorithms, the reader is referred to references 4, 21, and 31.

In summary, the early analysis techniques have been improved during the last few years and these improvements have been incorporated in the design.

3.4.2.1 Database Required For Static Analyzer. - The static analyser's data base contains all the local information related to its operation. This database would include items such as:

the flowgraphs (and pafs)

live, avail, gen and kill sets

parameter list information

program call graph.

These are internal in nature. In addition, the HALMAT and symbol tables are required for generating this information and producing the error messages. The error messages themselves must be saved for later (possibly automatic) perusal. These data objects are external in nature and will be contained in the ISIS, or system, database.

The efficiency of the static analyzer and its overall capabilities depend to some extent on the speed of accessing items stored in the internal database. Since ISIS is not necessarily involved, it should be possible to optimize this information's format and its retrieval. The ramifications of multi-level static analysis (i.e. static analysis on the module level, then on the program level, then on the multi-process level) needs to be explored, as regards the internal database.

3.4.2.2 Static Verification Of Output Assertions. - The assertion facility presented in the design contains a construct having the following syntax:

/* ASSERT expression list OUTPUT */;

This specification gives a complete list of the expressions, usually variables, which are "produced" or modified by a section of code. It is therefore implied that only those expressions, and no others occurring in the current scope, will occur in reference contexts following the OUTPUT assertion.

Such an assertion can easily be checked using static data flow analysis. The "reference sets" associated with each node in the program flowgraph indicate which variables are used in each statement. Following an OUTPUT assertion these sets may be checked to verify that no variables are referenced which have been determined to be "dead" - by their absence from the OUTPUT list.

Such an assertion also provides a basis for strengthening the other anomaly analyses performed by the static analyzer. More specifically, one of the anomalies the static analyzer checks for is variables which are defined but not subsequently referenced. In other words, useless computation is detected. Such a situation cannot normally be classified as an error. It is only "suspect". The presence of an OUTPUT assertion increases the number of places such anomalies may be detected: without assertions the anomaly is detected upon exit from the static scope of the variable in question. With the assertions the anomalies may be detected at each OUTPUT specification.

In a similar manner static data flow analysis can be used to verify the correctness of INVARIANT assertions. Static analysis can be used to verify such assertions even in the case where the protected (invariant) region is executing in parallel with another process. This analysis is performed by examining the definition sets associated with the nodes in the program flowgraph. Where multiple processes are active, all nodes which occur in the parallel sections are examined.

3.4.2.3 Static Data Flow Analysis of Concurrent Process Software.

3.4.2.3.1 Introduction. - Data flow analysis has been shown to be a useful tool in demonstrating the presence or absence of certain significant classes of programming errors (reference 21). It is an important software verification technique, as it is inexpensive and dependably detects a well defined and useful class of anomalies. Work to this point has been directed at the analysis of single process programs. Data flow analysis of concurrent programs has not been investigated. Concurrency causes difficulty in the detection of most errors which occur in single process programs; it also creates the possibility of new classes of errors.

One of the simplest errors which can occur in both categories of programs is referencing an undefined variable. Another programming anomaly which may occur in both categories is a dead variable definition. This occurs when a variable is defined twice without an intervening reference, or if a variable is defined yet never subsequently referenced. In concurrent software these types of anomalies and errors can occur in more subtle ways than in single process programs. For example, within a system of concurrent processes, one process may reference a shared variable while a parallel process may be redefining it. It is clearly desirable that such errors and anomalies be analytically detected or shown to be absent from programs.

In this section, we show that data flow analysis can reliably demonstrate the presence or absence of these and other programming anomalies for both single process and concurrent programs. While the anomalies are of interest in themselves, they are particularly important because experience has shown that consideration of why they arose in the program's construction often leads to the detection of significant design errors.

3.4.2.3.2 Example and Basic Definitions.

3.4.2.3.2.1 Programming Language Description. - In order to clarify the types of errors we are addressing, several examples are needed. For purposes of this presentation we shall use a slightly modified subset of HAL/S. The changes we make (mainly to the wait statement) are minor. These changes have been made for two reasons: the examples are more readily understandable and the prohibition of mixing logical and's, or's, and not's in wait statements simplifies the analysis algorithms. Note that while the analysis techniques were developed specifically for HAL/S they may be applied to other languages supporting concurrency as well.

3.4.2.3.2.1.1 - Subset Language Statements.

1. Assignment statement. This statement is of the form
   variable = expression ;

In executing this statement, the expression is evaluated and the result is then assigned to the variable.

2. Process declaration statements (**program, task,** and **close**). The declaration of each process begins with a declaration statement. The main program begins with a **program** declaration statement. Other processes begin with a **task** declaration statement. The end of a process declaration is marked with a **close** declaration statement.

3. Schedule statement. The execution of any process except for the main program is enabled through execution of a **schedule** statement. Execution of a **schedule** does not guarantee that the specified process will begin immediately, it merely indicates that the process is ready for execution. The actual time of initiation of a process is determined by the system scheduler. Any number of processes may be enabled for concurrent execution, but a process may not be scheduled to execute in parallel with itself. The schedule statement explicitly names the process or processes to be started; run-time determination of processes to be scheduled is not allowed.

4. Wait statement. This statement causes the executing process to wait for another process (or processes) to terminate before continuing with its own execution. A process has terminated when it has completed its execution and no longer resides in the system scheduler's "ready" queue. As with the **schedule** statement, the process(es) waited for is (are) named explicitly in the declaration; run-time determination is not allowed. The statement may be formulated two ways:

wait for process_name$_1$ **and** process_name$_2$ ...

or **wait for** process_name$_1$ **or** process_name$_2$ ...

When the process names are joined through logical disjunction, the wait is interpreted as **wait**-for-any. As soon as one of the named processes has terminated, the waiting process may proceed. When the process names are joined by logical conjunction all of the named processes must terminate before the waiting process may proceed. We shall refer to this as a **wait**-for-all statement.

5. Shared variables. Program variables have associated with them Algol-like scoping rules. This scoping exists at the program level, meaning that two processes may both access the same variable. We assume that no protection mechanism exists.

6. Transput. Input to a program is accomplished through a **read** statement. Values are output via a **write** statement.

3.4.2.3.2.2 Example. - Using the above constructs we now present an example program (Figure 3.4.2.3.2.2-1 which contains several anomalies.

A few of the anomalies are listed below.

1. An uninitialized variable (x) may be referenced at line 5, as task T1 may execute to completion before task T2 begins.

```
1    Main: program;

2        declare integer x,y;
         /* x,y are global variables known throughout the main program and all
         tasks */
3        declare boolean flag ;

4        T1: task;
5            write x ;
6            wait for T3 ;
7        close T1 ;

8        T2: task;
9            x = 5 ;
10           y = 6 ;
11       close T2 ;

12       T3: task;
13           read x ;
14       close T3 ;

         /* end of declarations */

15       schedule T1 ; /* first executable statement of Main*/
16       schedule T2 ;
17       read flag ;
18       if flag then x = 8 ;
19       write x ;
20       y = 9 ;
21       wait for T2 ;
22       if flag then y = 10 ;
23       write y ;
24       wait for T2 ;
25       schedule T1 ;

26   close Main ;
```

Figure 3.4.2.3.2.2-1 Example Program With Several Data Flow and Synchronization Anomalies

2. The definitions of y as found in task T2 (line 10) and the main program (line 20) may be "useless", since y may be redefined at line 22, before y is ever referenced.

3. y is defined by two processes which act in parallel - thus the reference at line 23 may be to an "indeterminate" value.

4. Variable x is assigned a value by task T2 (line 9) while simultaneously being referenced by the main program at line 19.

5. There is a possibility that task T1 will be scheduled in parallel with itself at line 25 since there is no guarantee that T1 terminated after its initial scheduling.

6. The **wait** at line 24 is unnecessary as T2 was guaranteed to have terminated at line 21, and it has not subsequently been rescheduled.

7. The **wait** at line 6 will never be satisfied as T3 was never scheduled.

3.4.2.3.2.3 - Event Expressions. - Clearly many of these error phenomena are interrelated. Hence a more precise categorization and definition system is desirable. We shall modify some notions employed in reference 4 to gain this precision. In reference 4 errors were described in terms of anomalous or illegal sequences of events occurring along a path through a program.

For instance, the events, "reference", "define", and "undefine" are the significant ones in the detection of undefined variable references and dead variable definitions. Thus in determining the presence or absence of these errors in a given program, the execution of the program is modelled as the set of all potential execution sequences of these three events happening to each of the program variables. In a single process program any path traceable through the program's flowgraph is taken to represent a potential execution. Now denote the events "reference", "define", and "undefine" by r, d, and u, respectively. Then clearly an undefined variable reference can occur within a program if and only if there is a path subsequence of the Form "ur" for some variable and some potential execution. Similarly a dead variable definition is indicated by either a "dd" or "du" path subsequence.

In a concurrent program it is more difficult to determine the potential executions and hence the potential sequences of events. Different processes may be executing simultaneously on different CPU's, or in some non-determinable interleaved order on a single CPU. If these processes operate on shared data, then the sequence of events happening to that data cannot be predicted, even though the code for each process is known. All that can be safely assumed is that every interleaving of the statements of all processes which can act concurrently must be considered a potential execution. Hence the set of execution sequences for a given concurrent program is the set of all possible sequence of events which could result from a potential execution of the program.

Thus, for example, in Figure 3.4.2.3.2.2-1 noting that all variables are initially undefined and that a write is a reference, variable x may have the sequence "urd", "udr", "ud", "ur", or "u" by the time line 17 is reached. "urd" corresponds to task T1 acting first, then T2; "udr" corresponds to T2 actually executing before T1 (there is nothing in the program prohibiting this); "u" corresponds to tasks T1 and T2 both being ready to execute, but not actually having done so.

3.4.2.3.2.4 - Error Categorization and Definitions. - Using the notation developed above we may now formulate definitions for the errors in which we are interested. The following are anomalies which we wish to detect in all programs. Their detection is more complicated in programs using concurrency constructs.

1. Referencing an uninitialized variable. An execution during which this error occurs will have an event sequence of the form "purp"' for some program variable, where p and p' are arbitrary event sequences.

2. A dead definition of a variable. An execution during which this anomaly occurs will have an event sequence the form "pddp"' for some variable.

The following are errors and anomalies which we wish to detect in concurrent code. In the following the schedule event will be denoted by an "s", the wait by a "w". All processes will be assumed to be in state "u", unscheduled, when not scheduled.

3. Waiting for an unscheduled process. This anomaly is represented by the event expression "pusp"'.

4. Scheduling a process in parallel with itself. This anomaly is represented by the event expression "pssp"'.

5. Waiting for a process guaranteed to have previously terminated. The expression "pwwp"' is symptomatic of this condition.

6. Referencing a variable which is being defined by a parallel process. There exists a schedule, s_0, such that for some variable both the event sequence "ps_0rdp"' and the event sequence "ps_0drp"' are possible.

7. Referencing a variable whose value is indeterminate. There exists a wait, w_0, and two separate definition points for a given variable, d_1 and d_2, such that both the event expressions "$pd_1d_2w_0r$" and "$pd_2d_1w_0r$" are possible.

For each of the above errors we will be interested in determining whether they exist in the event expression at a statement (i.e. the event expressions consisting of the preceding events concatenated with the current event) or in the event expression which represents the transformations undergone after leaving a statement. In addition we will wish to distinguish between errors which are guaranteed to occur and those which might occur.

3.4.2.3.2.5 - Program Representation. - At the heart of data flow analysis are algorithms which operate on an annotated graphical representation of a program. Single process programs may be represented by a flowgraph. As introduced in reference 8 communicating concurrent process programs may be represented by a process augmented flowgraph, or paf. A paf is formed by connecting the flowgraphs representing the individual processes with special edges indicating all synchronization constraints. In our example language an edge must be created for each ordered pair of nodes of the type (schedule p_name, task p_name) and (close p_name, wait for p_name).

Figure 3.4.2.3.2.5-1 is a paf for the example program of Figure 3.4.2.3.2.2-1 The creation of the paf for programs in our language is quite straightforward. It is important to note however that most actual languages incorporate synchronization constructs which greatly complicate the construction of the paf. In fact, it is impossible to create a fixed static procedure capable of constructing the paf of any program written in a language which allows run-time determination of tasks to be scheduled and waited for. These issues will be discussed later in this paper.

3.4.2.3.2.6 Data Flow Analysis Algorithms. - Data flow analysis algorithms arose out of work in global program optimization (references 22 and 23). Our usage of them has a different objective, however. The algorithms are described in detail in references 4 and 24. The purpose of these algorithms is to infer global program variable usage information from local program variable usage information, and then to infer verification and error detection results from the global usage results. The local variable usage is represented by attaching two sets of variables, gen and kill, to each program flow graph node. The global data usage is represented by attaching two sets, live and avail, to each node. The algorithms presented in the references cited assure that, when they terminate: 1) a variable v is in the live set for node n, if and only if there exists a path, p, from n to another node n' such that v is in the gen set at n', but that v is not in the kill set of any node along path p; 2) a variable v is in the avail set for node n, if and only if, for every path, p, leading up to n there exists a node n' on p such that v is in the gen set at n', but v is not in the kill set for any node between n' and n along p.

The implications and usage of these algorithms, and the modifications required to them as a result of concurrency considerations, will become apparent from considering some examples.

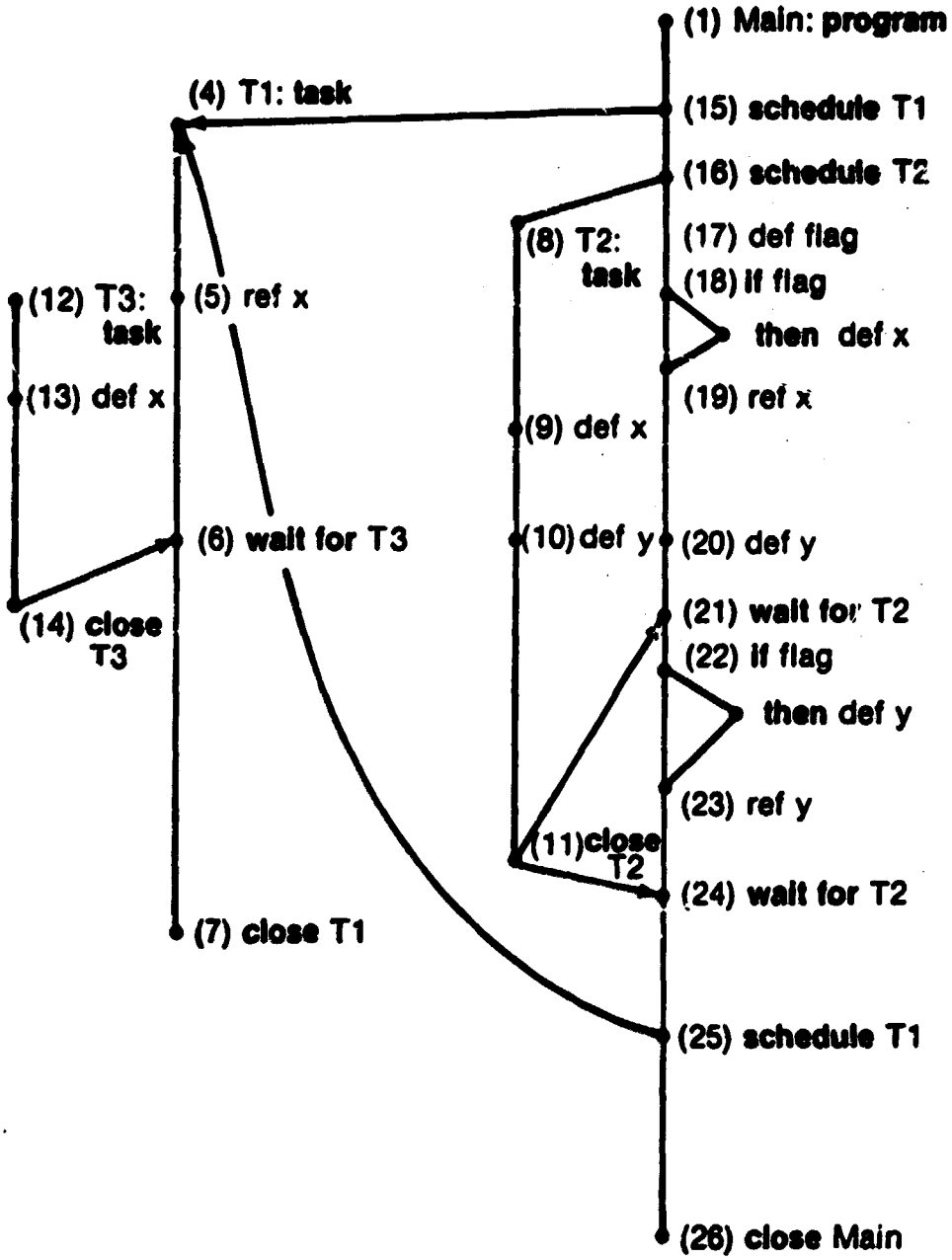


Figure 3.4.2.3.2.5-1 Process-Augmented Flowgraph for the Program of Figure 3.4.2.3.2.2-1

3.4.2.3.3.0 Detection of Uninitialization Errors. - Before we examine the large example given in Section 3.4.2.3.2.2, consider the following simple example:

```
1  Main: program;

2       declare integer x ;
3       declare boolean flag ;

4       T1: task;
5             write x ;
6       close T1 ;

7       T2: task;
8             write x ;
9       close T2 ;

10      schedule T1 ;
11      read flag ;
12      wait for T1 ;
13      if flag then  read x ;
14      schedule T2 ;

15 close Main ;
```

The paf for this program is given in Figure 3.4.2.3.3.0-1. All the nodes are annotated corresponding to the program statements.

Let us now consider the uninitialization errors which are present and how they may be detected.

Two uninitialization errors are present in the program. When task T1 is executed the write statement will reference uninitialized variable x. There is no possibility for x to have been initialized, even by the main program which is operating in parallel with the task. When task T2 is executed, there exists a possibility for referencing x as uninitialized. If "flag" has the value true, x will have been initialized and no error will occur. If, however, flag is false, x will still be uninitialized. Thus we have an instance of an error which "must" occur and an instance of an error which "might" occur. In addition, we may detect each of these anomalies at two different places: the point of variable reference, or at the start node. Thus we see that there are four different subcategories of the uninitialized variable reference error.

The balance of this paper will be devoted to specifying algorithms for detecting the various subcategories of this error and a variety of other errors and phenomena of interest in the analysis of concurrent software. These algorithms will, in general, involve the use of the LIVE and AVAIL procedures described in Section 3.4.2.3.2.6 of this paper. It will be shown that a diversity of diagnostic algorithms can be fashioned by using a variety of criteria for marking the nodes of the program flowgraph with gen and kill notations, and choosing suitable criteria for interpreting the output of the LIVE and AVAIL procedures.

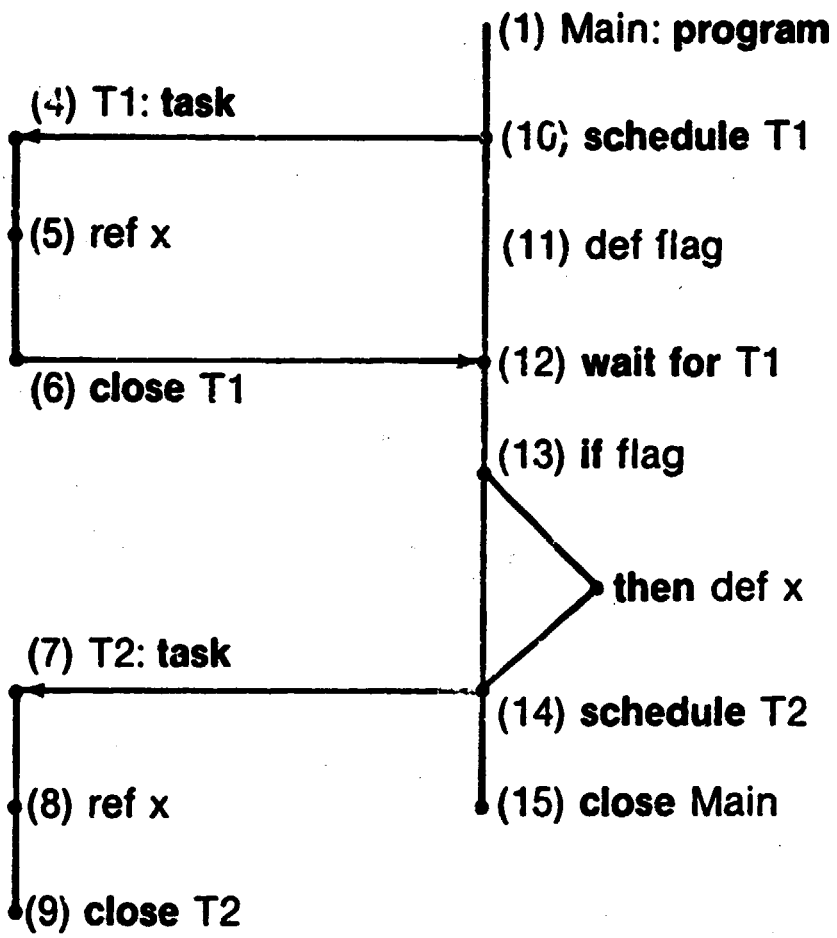


Figure 3.4.2.3.3.0-1 Paf for Program With Two Uninitialization Anomalies

For these reasons, it should be apparent that the algorithms presented here are much involved with placing gen and kill annotations on flowgraph nodes and interpreting live and avail annotations that subsequently appear on flowgraph nodes. This annotation information will be represented by means of bit vectors, denoted in the following way.

If an annotation criterion dictates that a particular variable, say x, is "gen'ed" at a node n, this will be indicated by seting the value of the function gen(n,x) to 1. Otherwise the value of gen(n,x) is 0. The function kill(n,x) is defined similarly. We shall assume that the program unit being analyzed has v variables, and that a one-to-one function, f, has been defined mapping the variables of the program unit onto the integers (1,...,V). Hence a bit vector is defined by the values (gen(n,x_1), ... , gen(n,x_i), ... , gen(n,x_V)) where x_i is used to denote the variable x for which f(x)=i. We use this bit vector as the definition of the function GEN(n). KILL(n) is defined similarly.

We shall also assume that there exists algorithmic procedures, LIVE and AVAIL, which operate upon a flowgraph containing N+2 nodes, and annotation functions GEN and KILL defined on the N+2 nodes. We shall always assume that node 0 represents an initialization action immediately preceding the first executable statement of a program unit. Node N+1 represents a termination action which immediately follows all statements which end execution of the program unit (i.e. the close of the main program) or end execution of any process which is not waited for (i.e. all process close nodes which are not joined to any wait nodes.) LIVE and AVAIL, when executed, compute annotation functions LIVE(n) and AVAIL(n), respectively, defined on the N+2 nodes. The values of LIVE(n) and AVAIL(n) are V-bit vectors for n between 0 and N+1. The bits of LIVE(n) and AVAIL(n) are defined by live(n,x_i) and avail(n,x_i) respectively, where x_i is the variable x for which f(x)=i. The functional dependencies of live(n,x) and avail(n,x) upon gen(n,x) and kill(n,x) are as described in Section 3.4.2.3.2.6 of this paper.

If live(n,x)=1 then we shall say "variable x is live at node n." If avail(n,x)=1 then we shall say "variable x is avail at node n."

We now begin by presenting an algorithm for detecting all statements at which an uninitialized variable reference "must" occur. Referring to the example in Figure 3.4.2.3.3.0-1 we see that this algorithm is designed to detect that the reference to x at statement 5 is a "must" uninitialized reference error.

For this and subsequent algorithms we will need to define functions REF(n) and DEF(n) on the nodes of the flowgraph. REF(n) is a V-bit vector whose i-th component is defined by ref(n,x_i). ref(n,x_i) is 1 if and only if the statement represented by node n involves a reference to the variable x which is mapped by f onto index value i. Otherwise ref(n,x_i) is 0. DEF(n) is defined similarly. def(n,x_i) is 1 if and only if the statement represented by node n defines the variable x which is mapped by f onto the index value i. Otherwise def(n,x_i) is 0. We also define 0 to be a V-bit vector, all of whose components are 0. 1 is a V-bit vector all of whose components are 1.

ALGORITHM 3.1:

```
for n := 1 to N+1 do
      GEN(n) := 0 ;
      KILL(n) := DEF(n) ;
od ;
GEN(0) := 1 ;
KILL(0) := 0
call AVAIL ;
for n := 1 to N do
      for i := 1 to V do
            if ref(n,i) = 1 and avail(n,i) = 1
                  then print("an uninitialized reference to", f^-1(i),
                     " must occur at node ", n ) ;
            fi ;
      od ;
od ;
```

It is important to observe that algorithm 3.1 is designed to assure that the error message will only be generated when a particular variable cannot possibly be initialized by any execution sequence leading up to the reference at the node to which the message pertains. In particular it is important for the reader to verify that this algorithm correctly analyzes the program in Figure 3.4.2.3.3.0-1. Figure 3.4.2.3.3.0-2 shows the contents of each set (gen, kill, etc.) at each node upon termination of algorithm 3.1. Note that variable x is in the avail set at the **write** node in task T1. Also note that x is not in the avail set at the **write** in task T2. Thus we are assured that an error message will be produced for the reference to x at statement 5, but not for the reference at statement 8.

We now present an algorithm for detecting "may" uninitialized variable reference errors at a node. This algorithm is designed to detect a variable reference occurring at a statement for which there exists an execution sequence which leads up to the statement and which does not initialize the variable. Referring to Figure 3.4.2.3.3.0-1 again, clearly such an error occurs at statement 5, but of more interest there is also such an error at statement 8. Algorithm 3.1 does not detect the error at statement 8, but algorithm 3.2 will.

Before presenting algorithm 3.2, we must first discuss a necessary modification to the AVAIL algorithm. The AVAIL algorithm is devised to assure that at termination a variable, x, will be avail at n if and only if for every possible execution of the program leading up to n, there is a previous gen of x without an

NODE	REF	DEF	GEN	KILL	AVAIL
0			x,flag		x,flag
1					x,flag
2	---	---	---	---	---
3	---	---	---	---	---
4					x,flag
5	x				x,flag
6					x,flag
7					
8	x				
9					
10					x,flag
11		flag		flag	x,flag
12					x
13	flag	x		x	x
14					
15					
16					

Figure 3.4.2.3.3.0-2 Contents of the Data Flow Analysis Sets for the Paf of Figure 3.4.2.3.3.0-1

intervening kill of x. For single process programs, AVAIL(n) is computed correctly at every flowgraph node n provided that the following equality is achieved at termination of the AVAIL algorithm.

$$\text{AVAIL}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{predecessors} \\ \text{of } n}}{\text{intersect}} \left(\text{GEN}(n_i) \text{ union } (\text{AVAIL}(n_i) \text{ intersect not KILL}(n_i))\right)$$

For concurrent process programs it is helpful to define a somewhat different equilibrium condition under certain circumstances. We proceed as follows.

Suppose n_w is a flowgraph node which represents a wait statement. In the paf, G, of the program containing n_w,n_w will be the tail of some edges which are usual flow of control edges, and the tail of at least one edge whose head represents the termination activity for a concurrent task. Suppose now that $(f_i)_{i=1}^{F}$ represents the set of heads of usual flow of control edges whose tails are n_w, and that $(p_i)_{i=1}^{P}$ represents the set of concurrent task termination nodes which are the heads of edges have n_w as their tails. Now create a new graph node n'_w delete the edges ((f_1,n_w), ... , (f_F,n_w)) and replace them by the edges ((f_1,n'_w), ... , (f_F,n'_w), (n'_w,n_w)). Suppose this is done for every wait node in G. Denote the resulting graph by G'. Now compute AVAIL(n) as usual, except use the following equilibrium condition at the wait-for-any nodes of G' only.

$$(*)\ \text{AVAIL}(n_w) = \underset{(p_i)_{i=1}^{P}}{\text{intersect}} \text{AVAIL}(p_i) \text{ union AVAIL}(n'_w)$$

A different equilibirum condition is required at wait-for-all nodes.

$$(*)\ \text{AVAIL}(n_w) = \underset{(p_i)_{i=1}^{P}}{\text{union}} \text{AVAIL}(p_i) \text{ union AVAIL}(n'_w)$$

The resulting AVAIL(n) bit vectors will be quite useful to us. Thus let us denote by AVAIL* the algorithm which employs the starred formulas as the equilibrium conditions for all of the wait nodes of G'. In all the algorithms which follow we assume that graph G' has been created and that the analysis takes place on that graph.

We can now state algorithm 3.2.

ALGORITHM 3.2:

```
for n := 1 to N+1 do
    KILL(n) := 0 ;
    GEN(n)  := DEF(n) ;
od ;
KILL(0) := 1 ;
GEN(0) := 0 ;
call AVAIL* ;
for n := 1 to N do
    for i := 1 to V do
        if ref(n,i) = 1 and avail(n,i) = 0
            then print("an uninitialized reference to",f^-1(i),
                "may occur at node", n);
        fi ;
    od ;
od ;
```

Using a different algorithm we may indicate to the programmer the event sequence associated with this anomaly. Unfortunately many such event sequences are unexecutable. This problem and partial remedies to it are discussed elsewhere (reference 6). In our example here, variable x is not in the avail set at either write statement in task T1 or T2. Thus the potential for error is reported at both nodes. In this case the associated event sequences are clearly executable.

We now present an algorithm for detecting at the start node all the "must" uninitialization errors. In the example of Figure 3.4.2.3.3.0-1 we are again interested in detecting the error which occurs at the reference to x in statement 5, except in this case the point of detection (and error message generation) will be the start node of the program.

Analogous to the presentation of Algorithm 3.2 we must discuss a necessary modification to the LIVE algorithm. The LIVE algorithm is devised to assure that at termination a variable, x, will be live at n if and only if there exists an

execution sequence beginning at n such that there is a gen of x before there is a kill of x. For single process programs, LIVE(n) is computed correctly at every flowgraph node n provided that the following equality is achieved at termination of the LIVE algorithm.

$$\text{LIVE}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{successors} \\ \text{of } n}}{\textbf{union}} (\text{GEN}(n_i) \textbf{ union } (\text{LIVE}(n_i) \textbf{ intersect not } \text{KILL}(n_i)))$$

For concurrent programs it is useful to define a different equilibrium condition which is applied only at **schedule** nodes.

$$(*) \quad \text{LIVE}(n) = \underset{\substack{\text{all } n_i, \\ \text{immediate} \\ \text{successors} \\ \text{of } n}}{\textbf{intersect}} (\text{GEN}(n_i) \textbf{ union } (\text{LIVE}(n_i) \textbf{ intersect not } \text{KILL}(n_i)))$$

We shall denote by LIVE* the algorithm which creates the live sets, employing (*) at all **schedule** nodes of G. (A graph G' is not required in this case as a schedule node only has a single control flow edge leaving it. All others lead to a **task** initialization node.)

ALGORITHM 3.3:

```
for n := 0 to N do
        GEN(n) := DEF(n) ;
        KILL(n) := REF(n) ;
od ;
GEN(N+1) := 1 ;
KILL(N+1) := 0 ;
call LIVE* ;
for i := 1 to V do
        if live(0,i) = 0
                then print( " an uninitialized reference to ", f^-1(i),
                        " will occur ");
        fi ;
od ;
```

In the example of Figure 3.4.2.3.3.0-1 variable x will be missing from the live set at the start, due to the kill present at line 5. (The live set at the wait node does contain x, however, as the error in task T2 is dependent on the execution sequence taken.)

The detection of possible errors is achieved through the following algorithm.

ALGORITHM 3.4:

```
for n := 0 to N+1 do
        GEN(n) := REF(n) ;
        KILL(n) := DEF(n) ;
od ;
call LIVE ;
for i := 1 to V do
        if live(0,i) = 1
                then print ( " an uninitialized reference to ", f^-1(i),
                        " may occur ");
        fi ;
od ;
```

In our now tired example variable x is in the live set at the start because of the references in both tasks. (Now note that the **wait** node has x in its live set - indicating that there is an execution sequence following which encounters a reference before any initialization. An error in that execution sequence would depend on x not being initialized before the **wait**, which of course it is not.)

To summarize briefly, two basic algorithms are involved. One computes live sets, the other avail sets. With suitably created gen and kill sets attached to the paf and special rules applied at **wait** nodes during the computation of avail and at **schedule** nodes during the computation of live, a comprehensive set of programming anomalies may be detected in concurrent process programs.

This is not the end of the problem, however.

3.4.2.3.4.0 Parcelling of Analysis Activities. - Let us now return to the example of Section 3.4.2.3.3, and modify the program slightly. In that example task T2 performed the same actions as task T1. There was no need to declare two tasks, except that it made our analysis simpler, as we shall see. Below we show the program written with only a single task declaration.

```
1  Main: program;

2       declare integer x ;
3       declare boolean flag ;

4       T1: task;
5             write x ;
6       close T1 ;

7       schedule T1 ;
8       read flag ;
9       wait for T1 ;
10      if flag then read x ;
11      schedule T1 ;

12 close Main ;
```

The paf for this program is given in Figure 3.4.2.3.4.0-1. As before, the nodes are numbered and annotated with the corresponding statements.

Note that the paf has been drawn with two edges entering the task's start node. Suppose now that we wish to look for "must" uninitialization errors, and detect them at the point of reference. We are therefore concerned with computing the avail sets as described in algorithm 3.1. Using this algorithm on the graph as shown will result in x not being in the avail set at the reference (at line 5). Thus we cannot say that whenever this node is executed an uninitialization error will result. Indeed this is a correct statement as the second time the task is scheduled there is only the possibility for an error at this line. This is somewhat unsatisfactory, though, as it is clear that the first time T1 is scheduled an error will occur, regardless of the execution sequence. We may improve the

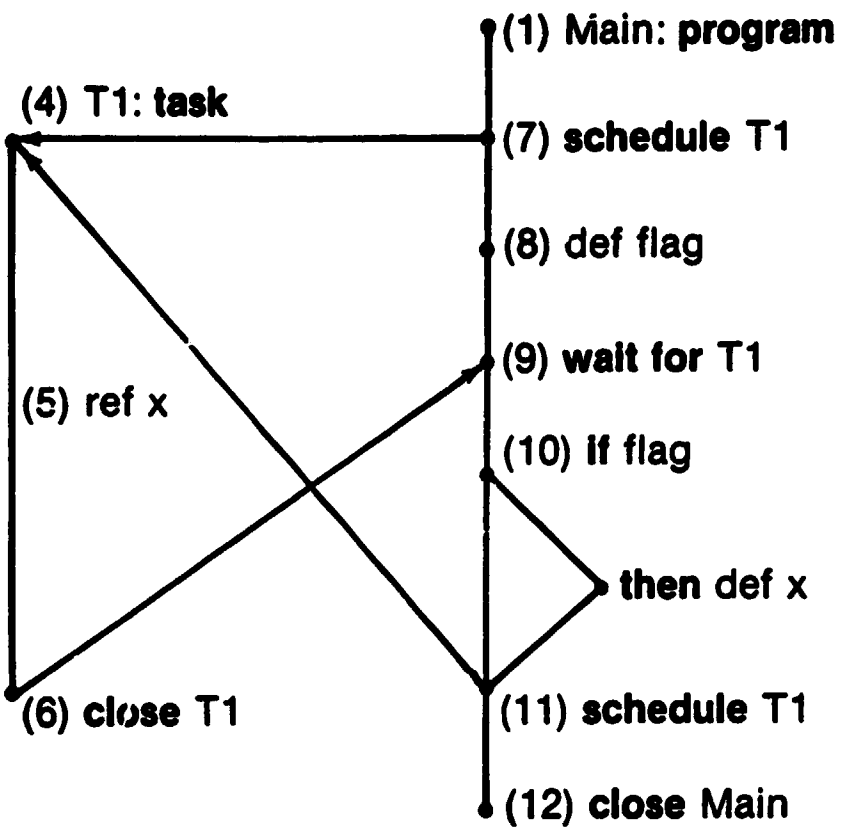


Figure 3.4.2.3.4.0-1 Paf for Program With Two Uninitialization Errors With a Single Task

strength of our analysis in this regard by parceling the paf and detecting the error not at the reference, but at the point where the task is scheduled. One cost of doing this is that we will not be able to point directly to the statement in the task at which the error occurs.

Our method for doing this is based on the technique presented by Fosdick and Osterweil for handling external procedures when performing data flow analysis on single process programs (reference 4). Their technique abstracts the data flow in each procedure using the LIVE and AVAIL algorithms, and attaches this abstracted information to all invoking nodes for each procedure. Data flow analysis can then be performed on the invoking procedure. We here adapt this technique to the analysis of tasks for anomalous event sequences. The data usage patterns within each task are determined using LIVE and AVAIL. These abstractions are attached to all **schedule** and **wait** nodes referring to each task. Analysis of this remarked ("trimmed") graph then proceeds as described previously.

For the example of Figure 3.4.2.3.4.0-1 the analysis will proceed roughly as follows. The algorithms 3.1, 3.2, 3.3, and 3.4 would be run for the local variables in task T1 first. (Since there are no local variable this step is omitted.) Next T1 is annotated as described in algorithm 3.3 for global variables (in this case x). The LIVE algorithm is run, giving the result that x is live at the **task** node, 4. We consequently label nodes 7 and 11 with ref (n,x)=1, indicating that execution of node 7 or node 11 always results in a subsequent reference to x. We can now run algorithms 3.1, 3.2, 3.3, and 3.4 for the variables local to Main. Algorithm 3.1 will show that x is avail at 7 indicating an uninitialized reference to x will always occur as a consequence of executing node 7. Ideally the resulting error message will indicate that the error actually occurs "somewhere" in the scheduled task. Determination of the error's precise location would be relegated to a separate (depth-first) scan of the task.

Clearly we could continue passing up data usage abstractions through an arbitrary number of levels of task scheduling. The restriction we have to impose on the program in order to adopt this technique is that the process invocation graph be acyclic. In the single process program situation there is an analogous restriction that the subroutine call graph be acyclic: recursion is prohibited. This prohibition exists for multiprocess programs as well, but the process invocation graph is also required to be acyclic. This is a stronger restriction as it is possible to have a cyclic process invocation graph which does not involve any recursion, either on the process or the subroutine level. For the moment we are satisfied that a significant class of programs is nevertheless being addressed, but further investigation is clearly called for here.

3.4.2.3.5.0 Additional Reference/Definition Anomalies.

3.4.2.3.5.1 Referencing a Variable While Defining It in a Parallel Process.- Let us now reconsider the example of Section 3.4.2.3.2.2. At line 5, in task T1, we have a reference to variable x which, in absence of a "fortunate" sequencing of events, will be uninitialized when the task is first scheduled. If algorithm 3.1 is run on the paf corresponding to this program (Figure 3.4.2.3.2.5-1), an "always" uninitialization error will be detected at the reference. (We are assuming that the analysis is carried out in the parcelled manner described in the preceding

section, as the second time the task is scheduled the possibility exists that x will have been defined.) Would it be proper to report this as an always error? What is termed a "fortunate" sequence really makes this an anomaly. It is conceivable that known operating environment conditions guarantee that the initialization performed by task T2 transpires before the reference in task T1. A "sometimes" error message is unsatisfactory, though, as such "guarantees" are outside the domain of the program. This confusion is due to the referencing and defining of a variable by two processes which may be executing in parallel. This construction, besides impairing our other analyses in the manner described, seems inherently dangerous and should be reported as an anomaly in its own right.

This anomaly may be detected in a rather naive manner given that we can determine which sections of the program may be operating concurrently. Let us assume that the paf is parcelled into S subgraphs, G_i. Each section corresponds to a task or a portion of a task. Briefly, section boundary nodes are program, task, close, and wait nodes. (Our notion of a section is roughly equivalent to that of a task which contains no wait statements.)Let us further assume that we have at our disposal a boolean function, PARALLEL, which determines which sections can execute in parallel. That is, PARALLEL defines a function of two variables, i and j, such that PARALLEL(i,j) is true if and only if G_i and G_j represent sections which might execute in parallel. It is important to note that the algorithm for determining this is not trivial. Indications of how such an algorithm can be constructed may be found in references 25 and 26. Based on these assumptions we can now state an algorithm for detecting the possibility of referencing and defining a variable from parallel tasks or sections. Suppose the nodes of graph G_i are numbered from $n_{i,0}$, the logical predecessor to the sections start node, to $n_{i,j}$, the logical successor to the sections final node. For clarity, also assume as before that f maps all the variables of the program onto the integers 1-V.

Algorithm 5.1:

```
for i = 1 to S do;
    for j = 1 to l_i do
        REF(n_{i,0}) := REF(n_{i,0}) union REF(n_{i,j});
        DEF(n_{i,0}) := DEF(n_{i,0}) union DEF(n_{i,j});
    od;
od;
for i := 1 to S do
    for j := 1 to S do
        if i ≠ j
            then if PARALLEL(i,j)
                then if REF(n_{i,0}) intersect DEF(n_{i,0}) = 1
                    then print (" the following may be referenced ",
                    "and defined in parallel by sections", i, "and",j);
                        for v := 1 to V do
                            if ref(n_{i,0},v) = 1 and
                               def(n_{i,0},v) = 1
                                then print( f^{-1}(v));
                            fi;
                        od;
                    fi;
                fi;
        fi;
    od;
od;
```

This algorithm detects the possibility of references and definitions occurring in parallel. We can also construct an algorithm that determines when this error must occur, regardless of the execution paths within the processes. The only change required to algorithm 5.1 is in the creation of the REF and DEF sets at the nodes $n_{i,0}$. The REF sets at $n_{i,0}$ would be computed by an algorithm similar to 3.3, and the DEF sets by an algorithm similar to 3.1.

3.4.2.3.5.2 Unused Variable Definitions. - A programming anomaly not truly erroneous but which often indicates the presence of a design error is that of unused variable definitions. The example of Section 3.4.2.3.2.2 has such an anomaly in it. Variable y is defined both by task T2 and by the main program (at line 20). Y is then possibly redefined at line 22, before ever being referenced. (We will examine the anomalous situation which occurs at the reference to y (line 23) in the next section.) This anomaly may be detected by techniques very similar to those presented in Section 3.4.2.3.3. Here, as with uninitialization errors, there are four cases to examine: detecting errors which always occur through examining all possible event sequences which follow a node, detecting the possibility of such errors, detecting errors which always occur through examining all event sequences preceding a node, and detecting possible errors by examining the preceding event sequences. We present here only the algorithm for determining the anomalous situation where a variable v is defined at node n, yet on all paths leading to n, v has been previously defined without any intervening reference. Algorithms for the other three related anomalous situations should be derivable by analogy. Not that the presence of reference-definition in parallel anomalies may impair the quality of the analysis here, like as was described previously.

The procedure given here assumes that the program graph has been parcelled into task subgraphs. We assume that the process invocation graph is acyclic and that the labelling used in algorithm 5.1 is used here as well. This particular error also requires that we define a new equilibrium condition to be applied during the computation of the AVAIL sets at wait nodes. The new condition is as follows:

$$(**)\ \mathrm{AVAIL}(n_w) = \underset{(p_i)_{i=1}^{P}}{\text{intersect}}\ \mathrm{AVAIL}(p_i)\ \text{union}\ \mathrm{AVAIL}(n_w') - \underset{(p_i)_{i=1}^{P}}{\text{intersect}}\ \mathrm{REFED}(p_i)$$

This condition applies at wait-for-anys. At wait-for-alls:

$$(**)\ \mathrm{AVAIL}(n_w) = \underset{(p_i)_{i=1}^{P}}{\text{union}}\ \mathrm{AVAIL}(p_i)\ \text{union}\ \mathrm{AVAIL}(n_w') - \underset{(p_i)_{i=1}^{P}}{\text{union}}\ \mathrm{REFED}(p_i)$$

We shall denote by AVAIL** the algorithm which employs the double-starred formulas as the equilibrium conditions for all the wait nodes of G'. REFED(n) is a V-bit vector defined during the computation of AVAIL** which is used to save the value of some intermediate AVAIL sets.

ALGORITHM 5.2:

```
declare bit vector PROCESSED (1:S);
PROCESSED := 0;
while PROCESSED ≠ 1 do
    for i := 1 to S do
        if processed_i = 0 and processed_t = 1 for all tasks, t,
            for which G_i waits
        then
            processed_i := 1;
            for j = 1 to l_i do
                GEN(n_i,j) := REF(n_i,j);
                KILL(n_i,j) := 0;
            od;
            KILL(n_i,0) := 1;
            call AVAIL*;
            REFED(n_i,l_i) := AVAIL(n_i,l_i);
            for j := 1 to l_i do
                GEN(n_i,j) := DEF(n_i,j);
                KILL(n_i,j) := REF(n_i,j);
            od;
            KILL(n_i,0) := 1;
            call AVAIL**;
            for j := 1 to l_i do;
                if DEF(n_i,j) intersect AVAIL(n_i,j) ≠ 0
                then print(" the definition(s) at node ",n_i,j,
                        "is always immediately preceded by another",
                        " definition. The variable(s) is (are): ");
                        for k := 1 to V do
                            if def(n_i,j,k)=1 and avail(n_i,j,k)=1 then print( f^-1(k));
                            fi;
                        od;
                fi;
            od;
        fi;
    od;
od;
```

3.4.2.3.5.3 Referencing a Variable of Indeterminate Value. - In the above presentation we deferred discussion of the anomalous data flow situation existing at the reference to variable y occurring at line 23 of the example program in Section 3.4.2.3.2.2. Y is defined by task T2 at line 10, by the main program at line 20, and possibly again in the main program at line 22. If for the moment we ignore the definition at line 22, then it is definitely indeterminate whether the definition from the task or from the main program is referenced at line 23. If we acknowledge the presence of the definition at line 22, depending on the event sequence (namely whether variable flag is **true**) the reference at line 23 may be to an indeterminate value.

The algorithm we now present is designed to detect indeterminate reference anomalies which will occur regardless of execution sequence. The anomalies will be detected at the point of indeterminate reference.

Algorithm 5.3:

declare bit vector PROCESSED (1:S) ;

PROCESSED := 0 ;

while PROCESSED ≠ 1 **do**

for i := 1 **to** S **do**

if $processed_i$ = 0 **and** $processed_t$ = 1 for all tasks, t,

for which G_i waits

then

$processed_i$:= 1 ;

for j = 1 **to** l_i **do**

$GEN(n_{i,j})$:= $DEF(n_{i,j})$;

$KILL(n_{i,j})$:= 0 ;

od ;

$KILL(n_{i,0})$:= 1 ;

call AVAIL* ;

$DEFED(n_{i,l_i})$:= $AVAIL(n_{i,l_i})$;

for j:= 1 **to** l_i **do**

$GEN(n_{i,j})$:= 0 ; $KILL(n_{i,j})$:= $DEF(n_{i,j})$;

od ;

for all $w_{i,a}$, **wait** nodes in G_i **do**

COUNT := 0 ;

for all predecessor nodes, $p_{i,a,b}$, of $w_{i,a}$ **do**

COUNT := COUNT **vector-add** $AVAIL(p_{i,a,G})$;

od ;

```
                    GEN(w_{i,a}) := 0;
                    for v := 1 to V do
                        if COUNT_v greater than 1
                        then GEN(w_{i,a},v) := 1;
                        fi;
                    od;
                od;
                call AVAIL;
                for j := 1 to l_i do
                    if AVAIL(n_{i,j}) intersect REF(n_{i,j}) ≠
                    then print("indeterminate reference at", n_{i,j});
                    fi;
                od;
                AVAIL (n_{i,l_i}) := DEFED (n_{i,l_i});
            fi;
        od;
    od;
```

3.4.2.3.6.0 Process Synchronization Anomalies. - As an outgrowth of our investigation into the detection of data flow anomalies in concurrent process software it became clear that some forms of synchronization errors could be detected in essentially the same manner. We have alluded to the nature of these errors in the introduction. They will now be considered in detail. Note that in form the synchronization anomalies are analagous to data flow anomalies. In addition, as with data flow, many of the anomalies are not strictly errors, but they are conditions which may be interpreted as erroneous in the sense of indicating deeper problems. At the very least they represent conditions which should be clearly documented.

3.4.2.3.6.1 Waiting for an Unscheduled Process. - This anomaly is perhaps the most apparent, and is closest in form to the data flow anomalies already discussed. The example of Section 3.4.2.3.2.2 contains such an error at line 6 in task T1. Task T3 is never scheduled, yet T1 waits for it. The analogy is to detection of uninitialized variables. As such we will present algorithm 3.1 rewritten to detect this anomaly. Thus we are interested in detecting anomalies which must occur, and the anomaly is to be detected at **wait** nodes. Our notation requires that we introduce functions SCH(n), WAIT_ALL(n), and WAIT_ANY(n). All function values are T-bit vectors. We shall assume that the program unit being analyzed has T processes, and that a one-to-one function, g, has been defined mapping the process names onto the integers (1,...,T). The i-th component of SCH(n) is defined by sch(n,t_i). sch(n,t_i) is 1 if and only if the statement

represented by node n schedules the task t which is mapped by the function g onto index value i. WAIT_ALL and WAIT_ANY are similarly defined, for the two types of wait statements in our language.

Algorithm 6.1:

```
for n := 1 to N+1 do
    GEN(n) := 0 ;
    KILL(n) := SCH(n) ;
od ;
GEN(0) := 1 ;
KILL(0) := 0 ;
call AVAIL ;
for n := 1 to N do
    for i := 1 to T do
        if ( wait all(n,i) = 1 or wait any(n,i) = 1) and avail(n,i)=1
            then print (" the reference to process ", g^-1(i),
              " at node ", n, " is to a process which has not been scheduled.");
        fi ;
    od ;
od ;
```

In Figure 3.4.2.3.2.5 task T3 will be in the avail set at the node corresponding to line 6, thus the error will be detected.

As may be expected, there is also an analogue to the reference-definition in parallel condition here. The following program presents such a condition.

```
1     Main: program;

2           T1: task;
3                 schedule T2 ;
4           close T1 ;

5           T2: task;
6                 /* do something */
7           close T2 ;

8           schedule T1 ;
9           /* do something */
10          wait for T2 ;

11    close Main ;
```

In this program there is the possibility that task T2 will be scheduled before the **wait** at line 10 is encountered. Our analysis described above will cause an "always" message to be generated. Thus we need to perform "schedule/wait in parallel" analysis to give a complete description of the situation. This would be performed in a manner analogous to that of reference/definition in parallel analysis.

3.4.2.3.6.2 Waiting for a Process Guaranteed to Have Already Terminated. - The example of Section 3.4.2.3.2.2 still has additional errors to consider. At line 24 the main program waits for task T2 to complete. Yet the task was already assured to have terminated at line 21. The second **wait** is thus superfluous and possibly misleading. Since our language syntax allows us to specify **wait**-for-all and **wait**-for-any we must be careful to distinguish the errors which we will detect and the algorithms which apply in each case. To indicate the nature of our technique we will just consider a single case: looking for constructs which, regardless of event sequence, assure us that at least one of the processes named in a **wait**-for-all has in fact already terminated at a previous **wait**.

Algorithm 6.2:

```
for n := 1 to N+1 do
    KILL(n) := SCH(n) ;
    GEN(n) := WAIT_ALL(n) ;
    if the statement represented by node n is a task statement for t_i
        then gen(n,t_i) := 1 ;
    fi ;
od;
GEN(0) := 1 ;
KILL(0) := 0 ;
call AVAIL ;
for n := 1 to N do
    for i := 1 to T do
        if avail(n,i) = 1 and wait all(n,i) = 1
            then print (" termination has already been ensured for task",
              g^-1(i), " at node ", n);
        fi ;
    od ;
od ;
```

In Figure 3.4.2.3.2.5-1 task T2 is in the avail sets of all predecessor nodes of the node corresponding to the first wait (line 21), but is in only one of the avail sets of the predecessor nodes of the wait at line 24. Thus the first wait is correct, while the second is anomalous.

The algorithm we have presented may be easily modified to detect the possibility of anomalies. To detect anomalies occurring at wait-for-anys we must develop new procedures to account for situations such as:

```
wait for T1 or T2 ;
      .
      .
      .
wait for T1 or T2 ;
```

In the absence of other synchronization statements the second wait is spurious; satisfaction of the first wait guarantees immediate satisfaction of the second.

3.4.2.3.6.3 Scheduling a Process in Parallel with Itself. - The last synchronization anomaly which we shall examine is that of scheduling a process to execute in parallel with an already active incarnation of the same process. In the example of Section 3.4.2.3.2.2 there is an instance of this error at line 25, where task T1 is scheduled for the second time (the first being at line 15). At no point in any process, let alone before the second **schedule**, has T1 been guaranteed to have terminated.

We will present the algorithm for detecting situations where, regardless of event sequence, termination has not been assured by the time a **schedule** is reached.

Algorithm 6.3:

```
for n := 1 to N+1 do
    GEN(n) := SCH(n) ;
    KILL(n) := WAIT_ALL(n) union WAIT_ANY(n) ;
od ;
KILL(0) := 1 ;
GEN(0) := 0 ;
call AVAIL* ;
for n := 1 to N do
    for i := 1 to T do
        if sch(n,i) = 1 and avail(n,i) = 1
            then print (" termination of process ", g^-1(i),
              " has never been ensured before the schedule at node ", n) ;
        fi ;
    od ;
od ;
```

If this algorithm is applied to our example an error will be detected at line 25. As may be expected, if a **schedule** may be performed in parallel with a **wait** (on the same process) the quality of our analysis is impaired. In particular, if such a condition exists algorithm 6.3 will detect a "for sure" error, where in fact there is an event sequence where termination takes place.

3.4.2.3.7.0 Conclusion

3.4.2.3.7.1 Summary. - In this section we have presented several algorithms useful in the detection of data flow and synchronization anomalies in programs involving concurrent processes. Data flow is analyzed on an interprocess and interprocedural basis. The basis of the technique is analysis of a process augmented flowgraph, a graph representation of a system of communicating

concurrent processes. The algorithms have excellent efficiency characteristics, and utilize basic algorithms which are present in many optimizing compilers. A procedure is outlined which allows analysis to proceed on "parcels" of the subject program. Only the most basic synchronization constructs have been considered, however.

3.4.2.4 Creation of the Program Flowgraph and Paf. - The program flowgraph may be constructed directly from the HALMAT representation of the program. The creation of the flowgraph may logically take place in two phases. The first phase reads the HALMAT file and emits a stream of (node, node) pairs, along with a map: (node, SMRK, operator). Each element of the map indicates the correspondence between a node number in the flowgraph and an instruction in the HALMAT. (The node is thus mapped to the source statements as well.) Each node-node pair indicates an edge in the program flowgraph. The edge exits the first named node and enters the second. The flowgraph is thus a directed graph. The second phase of flowgraph construction consists of forming the graph representation of the flowgraph from the sequence of edges emitted by the first phase. The algorithm for this operation, as well as a description of the graph representation scheme to use, may be found in Hopcroft and Tarjan, 1973 (reference 32). The representation scheme described is known as an edge list representation.

The algorithm for the first phase will be quite simple. Several primitive operations, such as create_new_node, join_node_to_predecessor, get_branch_target_node_number, join_node_to_branch_target, and enable_joining_of_node_to_successor will be required, as well as tables indicating, for example, the correspondence between statement labels and node numbers, and between internal flow numbers and node numbers. The overall algorithm will then function by sequentially processing the HALMAT file, using a **case** statement to direct the processing of each HALMAT operator. As each operator is processed and nodes are created, the bit vectors which indicate the operations which transpire at each node may be created as well.

In order to determine which primitive operations to apply at each operator it must be known what HALMAT operators are generated by the compiler for each control structure (**do while**, **if**, **case**, and so forth). If by no other means this may be determined by writing a few small test programs which exercise each of the structures, compiling them, and examining the resulting HALMAT. Analysis of the HALMAT will indicate which operators require the introduction of edges and which operators may be ignored (if any).

The advantages of employing an edge-list representation of the flowgraph for the data flow analysis activities are discussed in references 4, 31, and 32. Recall that the data flow analysis algorithms employed originated in consideration of global flow optimization. A considerable body of literature exists on this topic which may be consulted in matters such as these. A text has recently appeared detailing much of this work Hecht, 1977 (reference 33).

3.4.2.5 Tracing of the Effects of Inputs and the Origins of Outputs. - Flight software is a particularly good class of software to which this form of verification applies. The basis of the verification technique is the knowledge that each output from a program is (intended to be) the function of a certain, understood, set of inputs. Conversely, each input item should affect the computation of a certain, understood, subset of the output values of the program. The verification technique, therefore, is to provide a "map" indicating for each input variable all the output variables which are affected by it, and for each output value which input variables are involved in its calculation. This "map" could, for example, be represented by a matrix. The rows and columns would be labelled with every external (input or output) variable of a program. A check mark at an intersection in the program would then indicate that the two variables are involved. If rows were labelled with input variables and columns with the output variables, scanning across a row would reveal all those output variables which are affected by that particular input variable. Scanning down a column would indicate for that output variable which inputs are involved in its computation.

As in most data flow analysis anomaly detection problems there are two varieties of the question of "involvement." One may be interested in knowing what variables may possibly be involved in the computation of another variable, and what variables are always involved in the computation of another variable.

One means of creating this matrix is to employ symbolic execution, represented in this design through the interactive testing system. Using that tool questions such as, "For a given path, what inputs are involved in the computation of output x?", may be answered. The procedure would be to symbolically execute the path and examine the output formula associated with variable x. That formula would contain all the inputs which are involved in x's computation. Similarly the question, "For a given path, what outputs are affected by input variable y?" may be answered using this tool. The path is executed and all output formulas are examined. All formulas which involve y correspond to the output values desired.

Note that we have been careful to state that such checking is pathwise dependent. To expand the answers to a larger portion of the program requires that more paths may be examined. With some effort this may be done using the interactive testing tool; the effort involved may be very acceptable if the number of decision-decision paths is relatively small. (The number of times a loop is executed is less important, as more variables are not necessarily involved the longer the loop is executed.) Further, the path conditions formed throughout the testing of the program may lend themselves to "easy" analysis. For example, the presence of a particular input may preclude execution of a segment of the program.

Though we may use the interactive testing tool to generate (at least most) of the results desired, that solution is unattractive for some obvious reasons. First, the path dependencies require significant extra steps to guarantee that complete answers have been obtained, second, the interactive tool requires user interaction, and third, the execution time penalty may be unacceptable. An

alternative is thus desirable. Examination of the problem indicates that the task is basically one of data flow analysis and the capability may be created within the data flow analysis tool. This approach is considered in more detail below.

Data flow analysis as described before examines data activities along all paths within a program. Both "possible" and "always" data flows are considered, as has been noticed in the formulation of the various data flow anomalies. The new aspect of the problem which arises here is that the algorithms presented thus far only examine the data flow of individual data items, not the cumulative effect of several data items. Consider the following sequence of code:

```
d = e ;
c = d ;
b = c ;
a = b ;
```

Clearly the value of a is dependent on the value of e. Transitivity of assignment provides us with this result. Data flow analysis, on the other hand, would indicate that e is not used following the first statement.

The data flow analysis algorithms presented thus far (LIVE, AVAIL, AVAIL*, etc.) may be utilized, however, to take account of transitivity. The basic algorithm for tracing the effect of a specified input would proceed as follows. The point of input would be marked by a gen on the program flowgraph. All nodes which (re-) define the input variable would be marked with a kill. AVAIL would be run. Next the flowgraph would be remarked. For all variables v and for all nodes n, gen(n,v) = def(n,v) **and** there exists j such that ref(n,v_j) **and** avail(n,v_j). kill(n,v) = def(n,v) **and not** there exists j such that ref(n,v_j) **and** avail(n,v_j). AVAIL would then be run again. This procedure of running AVAIL and remarking the graph would continue until no changes were made when the graph was remarked. All points of output would then be examined. If the output variable were avail, then that variable is for some paths a function of the original input variable being examined. This algorithm is described in detail in a Ph.D thesis from Cornell (reference 34). The reader is referred to it for addtional discussion of the precise technique. The important thing to note is that the basic tools for constructing such a capability are already included in the design of the static data flow analyzer.

3.4.3 Unresolved Design Issues. - Several matters discussed in this presentation clearly warrant further investigation. The most pressing need now is a consideration of additional synchronization and communication constructs. These will introduce new classes of errors and may require that changes be made to the algorithms presented here.

One issue not addressed here is the creation of correct process-augmented flowgraphs. In our subset language this was a relatively trivial task, but as additional (real) synchronization constructs are added significant problems are anticipated. It is not clear at this point if "correct" pafs can always be generated. The analysis schemes may require alteration to accommodate such a situation.

Dynamic determination of synchronization paths has not been considered at all here, but work has been done in this area (reference 27). Likewise recusive procedures and processes have been precluded. Work has been done in data flow analysis of recursive routines (reference 28), but it appears inadequate for the analysis performed here.

3.5 Interactive Testing System.

3.5.1 Design Philosophy. - The testing of computer programs has always been a time consuming, tedious task. This has been due, in part, to the fact that it has never really been determined how much testing is required before we can be reasonably assured that a program reliably satisfies its requirements. Various testing methodologies have been devised to add structure and control to the testing process but, as yet, no single testing strategy has emerged that encompasses the whole spectrum of testing activities. Rather, choosing a combination of testing methods based upon the nature of the software being tested seems to be the most effective approach. This principle is, of course, essentially the same as that taken with regard to the entire spectrum of software verification activities.

Traditionally a program has been tested by selecting test data, executing the program using that test data as input, and then checking the validity of the resulting output. Most of the automatic tools which have been developed to assist program testing either aid in selecting test data or aid in checking the validity of a program's output. Tools which aid in controlling program execution during a test seem to be limited only to interactive (or worse yet, batch) debugging systems. These are usually very cumbersome to use and tend to generate a large amount of worthless output and, therefore, are used as a "last resort". Perhaps their worst characteristic is that there is no methodology associated with them to guide the testing process. A haphazard approach is more often the rule.

Functional testing (reference 10) [Howden,1978c] has, historically, been one of the most used (and abused) testing methods. It is a strategy whereby a program is considered as a "black box" or, in the mathematical sense, a function which relates input values to corresponding output values. The set of input values selected for a test are constructed based on the functional properties of the program. However, the sub-functions which comprise the program's design are ignored during the test. Howden has altered the conventional approach to functional testing so that those sub-functions are identified and individually tested themselves. To evaluate this approach, he applied this technique in an effort to detect errors in a commercially available set of standard mathematical programs. Very good results were achieved which so far seem to indicate that this method is at least as reliable as any other in its ability to show the presence of program errors.

The problem with Howden's functional testing is that a thorough knowledge of a program is required in order to isolate the functions comprising a program and to intelligently select test values for them. The method can also be very cumbersome in that it is often difficult to execute only a portion of a program and, as a result, much of the functional testing must be done by hand. The first problem, inherent to the testing process in general, can be addressed by having the author of the program document the program's internal functions while they are being written. Thus a comprehensive test plan may be created in a systematic manner. The second problem can be dealt with mechanically and is the primary focus of the design which will be presented here. Specifically, the design presented is an effort to create a single tool which will aid in the three testing activities mentioned above: selecting test data, executing the program, and checking the output.

The Interactive Testing System (ITS) has been designed to provide the capabilities required to isolate and individually test those sub-functions which make up the logical design of a program. It has also been designed to support any of a variety of testing methods that the user may choose. To direct our efforts, however, the methodology of functional testing has been considered a primary technique. The ITS is essentially an interactive debugger which has been enhanced to conveniently allow the execution of any part or parts of a HAL/S program complex and to perform symbolic execution. Three existing systems were carefully studied and the design presented here is basically an amalgam of them.

Johnson's RAIDE (Run-Time Analysis and Interactive Debug Environment) system (reference 20)[Johnson,1978] has supplied many of the ideas pertaining to the interactive user interface and the supporting command language. Clarke's ATTEST and AID systems (references 7,37) [Clarke,1976] [Winters, Ogden, Clarke, 1978] and Howden's DISSECT symbolic execution system (reference 8) [Howden,1978a] have been the impetus behind the design of the symbolic execution facility of the ITS.

3.5.2 System overview - The ITS, as its name implies, is an interactive based system for testing HAL/S computer programs. The system is designed so that a programmer can sit at an interactive terminal and test all or part of a HAL/S program complex. The ITS will allow a programmer to execute, for example, only a single procedure or even a single statement if he so desires. The capability to execute an entire program is present as well. Complete user control over the computation state gives the user the capability to inspect or change the value of any variables at any point during execution.

A symbolic execution facility is also provided. Symbolic values may be assigned to variables as can actual values. When the ITS encounters an expression in which one or more of the variables referenced in the expression contain symbolic values then symbolic evaluation of the expression is performed with the result being a symbolic expression. The user maintains control over the program execution paths which are followed by the ITS during symbolic execution.

The next section gives the syntax and description of the Interactive Command Language which defines the ITS user interface. The syntax follows the same notation used by the assertion and statistics gathering language found in Section 3.6. Some of the non-terminals are not specified but their use should be clear from their name; others can be found in the syntax found in Section 3.6.

3.5.3 Interactive Command Language (ICL)

3.5.3.1 ICL Syntax

ICL command ::= simple command
    DO; (simple command)+ END;
simple command ::= after
    assert
    assume
    before
    break
    call
    cancel
    close
    display
    do for
    execute
    if
    install
    proc
    quit
    restore
    save
    scope
    select
    set
    skip
    when
after ::= AFTER interrupt def ICL command
interrupt def ::= stmt des (, stmt des)*
    stmt-type list STATEMENTS
stmt des ::= proc ENTRY
    proc EXIT
    stmt def
stmt def ::= stmt no
    comp unit name.stmt no

```
proc ::= comp unit name.proc name
         proc name
assert ::= ASSERT assert comparison ;
assert comparison ::= comparison
         quantifier comparison
quantifier ::= FORALL quantifier completion ( , quantifier completion )*
         EXISTS quantifier completion ( , quantifier completion )*
quantifier completion ::= variable ( arith exp TO arith exp )
assume ::= ASSUME comparison ;
before ::= BEFORE interrupt def ( , interrupt def )* ICL command
break ::= BREAK ( 'character string' )
call ::= CALL test proc name (( arg list )) ;
arg list ::= parameter ( , parameter )*
cancel ::= CANCEL ;
close ::= CLOSE test proc name ;
display ::= DISPLAY (( number )) disp action ( , disp action )* ;
disp action ::= variable
         LINE ( number )
         PAGE ( number )
         COLUMN ( number )
         TAB ( number )
         PATH
         CONDITION
         SOLUTION
         'character string'
         intrinsic function ( arg list )
do for ::= DO FOR temp var = arith exp do for type ;
temp var ::= identifier
do for type ::= ( , arith exp )*
         TO arith exp ( BY arith exp )
execute ::= EXECUTE ;
if ::= IF comparison THEN ICL command ( ELSE ICL command )
install ::= INSTALL file name ;
proc ::= test proc name : PROCEDURE (( formal arg list )) ;
formal arg list ::= formal parameter ( , formal parameter )*
```

formal parameter ::= identifier

quit ::= QUIT ;

restore ::= RESTORE file name ;

save ::= SAVE file name ;

scope ::= SCOPE scope def (, scope def)* ;

scope def ::= stmt def TO stmt def

    proc

select ::= SELECT stmt def select clause ;

select clause ::= LOOP (loop list)

    CONDITION (cond list)

    CASE (case list)

loop list ::= loop cnt (, loop cnt)*

loop cnt ::= number

    number#number

cond list ::= cond alt (, cond alt)*

cond alt ::= number#T

    number#F

    T

    F

case list ::= case alt (, case alt)*

case alt ::= number

    number#number

set ::= SET variable (, variable)* set clause ;

set clause ::= = set expression

    INITIAL (initial list)

set expression ::= expression

    "symbolic value"

skip ::= SKIP skip clause ;

skip clause ::= number

    TO stmt def

when ::= WHEN comparison ICL command

3.5.3.2 ICL Description. - The Interactive Command Language has been designed to provide a simple and concise method of specifying commands to the Interactive Testing System (ITS). The syntax has been structured to reflect many of the properties of the HAL/S language. The specific commands have been selected to furnish the user of the ITS with the flexibilty to control the testing of any or all parts of a HAL/S software system. Each of the ICL commands will be described in the following paragraphs along with some examples illustrating their use.

3.5.3.2.1 Execution Scope. - The **SCOPE** command is used to identify those portions of the HAL/S program complex which are to be involved in the testing. The command consists of a list of block names and/or ranges of statement numbers within the compilation units making up the program complex. The list can be specified in any order. The first executable statement of the first entry in the list, however, identifies the point at which execution is to begin. Execution stops when a statement is executed which causes a transfer of control outside of the defined execution scope.

Examples.

SCOPE 25 **TO** 50;

This command specifies that only statements between statements 25 and 50 (inclusive) of the compilation unit residing on the HALMAT file will be executed. Note that when the compilation unit name is not included in the statement number definitions it is assumed that only one compilation unit is available for execution by the ITS.

SCOPE NAV.25 **TO** NAV.50, COURSE_CORRECTION;

This defines the execution scope to be statements 25 to 50 of 'NAV' and the block labeled 'COURSE_CORRECTION'. The block 'COURSE_CORRECTION' may or may not be defined in the compilation unit identified by 'NAV'. If it is not defined in 'NAV' then it must be a unique block name throughout the entire program complex residing on the HALMAT file. The first statement which will be executed in this example will be statement 25 (or, if not executable, the first executable statement following it).

3.5.3.2.2 Interrupt control commands - The ICL contains seven commands which allow the user to interact and control the ITS during execution.

The **AFTER/BEFORE** commands are used to allow ICL commands to be processed after/before execution of the specified statement or block within the currently executing program.

The **WHEN** command allows ICL commands to be processed as soon as the specified condition becomes true.

The **BREAK** command is used to interrupt the ITS during program execution to allow the user to stop execution and enter commands.

The **CANCEL** command will cancel all active interrupts set by **AFTER**, **BEFORE**, and **WHEN** commands.

The **EXECUTE** command causes execution to resume at the interrupted execution point (or at the first executable statement, if just beginning).

The **QUIT** command is used to terminate an ITS session.

Examples.

AFTER prog1.check **ENTRY BREAK** 'procedure check entered';

Execution will halt with the message "procedure check entered" printed at the user's terminal after entry into the block labeled 'check' in compilation unit 'prog1'.

WHEN x = 0 & y = 0 & z = 0 **BREAK**;
Execution will halt whenever x,y, and z all become equal to zero.

BEFORE prog.lookup **ENTRY DO**;
    SET x = "x";
    ASSUME x>0 ;
    SKIP 1;
    EXECUTE ;
END;

Prior to each entry into the block labeled 'lookup' in compilation unit 'prog' the following actions will be performed. The variable x will be assigned the symbolic value "x", x>0 will be conjuncted with the current path condition, the next executable statement will be skipped (the statement which transfers control to 'lookup'), and execution will resume.

3.5.3.2.3 State modification commands. - The ICL provides four commands which are used to modify the computation state of the currently executing program.

The **SET** command is used to assign actual or symbolic values to variables within the name scope in existence at the time the **SET** is processed. Only variables (simple, arrays, or structure terminals) of types integer, scalar, matrix, and vector can be assigned symbolic values. In addition, variables of type matrix/vector cannot have symbolic values assigned to their individual elements; the symbolic value is associated with the name of the matrix/vector (these restrictions will be discussed further in section 3.5.4). The expression part of the **SET** command has the same syntax and semantics as HAL/S expressions.

The **SKIP** command is used to reposition the statement pointer. The statement at which the pointer is set as a result of this command will be the next statement to be executed when execution resumes.

The **SAVE** command will save on the file identified by the file name the computation state of the currently executing program.

The **RESTORE** command will restore the computation state from the file identified by file name. The state must have been previously saved by the **SAVE** command.

Examples.

SET X = "A";

Processing of this command will result in X being assigned the symbolic value "A".

SET A **INITIAL**(20#0);

'A' is an integer array of size 20. This command will initialize 'A' to zero.

SET X = (- B + SQRT(B B - 4 A C))/ 2 A;

This will assign the value of the expression '(- B + SQRT(B B - 4 A C))/2 A' to the variable 'X'. If any of the variables in the expression have symbolic values then the result will be symbolic.

SKIP 3;

This commmand will cause the next 3 statements in the HAL/S program being tested to be skipped. The statements which are skipped are the next three statements in the program text, not the next three statements which would have been executed otherwise.

SKIP TO NAV.55;

Processing of this command will reset the statement pointer to statement number 55 in compilation unit 'NAV'. If 'NAV.55' is within the defined execution scope then it will be the next statement to be executed. Otherwise the command is in error.

SAVE MYFILE;

The computation state will be saved on file 'MYFILE'.

RESTORE MYFILE;

The computation state will be restored from file 'MYFILE'.

3.5.3.2.4 Display Command. - The **DISPLAY** command is used to output any information contained in the computation state of the currently executing program and in the ITS Itself. The syntax follows closely the same rules as the HAL/S write statement. In addition to displaying values of variables the command can be used to display a trace history of the path which has been executed, the current path condition, and, if possible, a solution to the current

path condition. The ITS also provides special built-in (intrinsic) functions which can be used to perform various **DISPLAY** functions such as displaying statement execution counts, the statement number of the current statement, and the name of the currently executing block (a list of suggested functions is provided in Appendix C).

Examples.

DISPLAY 'X = ',X ,LINE(1),'Y = ',Y;

Suppose that X was equal to 10 and Y had the symbolic value 'A' when the above command was executed. The following output would be printed at the terminal:

X = 10
Y = "A"

DISPLAY 'CURRENT PATH CONDITION = ',**CONDITION**;

This command will result in the current path condition being output to the users terminal which could look like:

CURRENT PATH CONDITION = X>0 & X<Y & Y = 10

If there has not been any symbolic execution performed then the following would be output:

CURRENT PATH CONDITION = TRUE

3.5.3.2.5 Path Selection. - If any variables have been assigned symbolic values during the execution of a program it may not always be possible to evaluate expressions associated with conditional branch/loop statements to actual values (e.g. **TRUE** or **FALSE**). In addition, it is usually desirable to specify the number of loop iterations to execute when a loop is encountered along a path during symbolic execution. Using the **SELECT** command the user can pre-select which branches to follow at specified conditional statements and also the number of loop iterations to perform at specified loop statements. Section 3.5.4 contains a detailed description of how path specification is controlled by the ITS.

Examples.

SELECT NAV.20 **CONDITION** (T);

Assuming that statement 20 in compilation unit 'NAV' is an IF, this command will inform the ITS that the **TRUE** branch is to be followed.

SELECT 18 **CONDITION** (5#T,F);
SELECT 18 **LOOP** (5);

If statement number 18 is a DO WHILE both of the above commands will inform the ITS that the loop should execute 5 times.

SELECT COURSE_CORRECTION.52 **CASE** (1,2,3,4);

Statement 52 in compilation unit 'COURSE_CORRECTION' must be a DO CASE statement. This command will indicate to the ITS that case 1 should be followed the first time the statement is executed, case 2 the second time, and so on.

SELECT 135 LOOP(2#3,1);

If statement 135 is a DO FOR then the first two times the loop is entered it will be iterated three times. The third time the loop is entered it will iterate only once.

3.5.3.2.6 Predicate commands. - Two commands are furnished by the ITS to allow the user to control the evolution of the path condition associated with the currently executing path.

The **ASSUME** command is used to logically conjunct predicates (e.g. initital conditions) to the current path condition. The predicate specified on the **ASSUME** command will be checked for consistency with the existing path condition. If it is found to be consistent then the predicate will be added to the current path condition. If, however, it is found to be inconsistent a message will be output to the user's terminal but the path condition will remain unchanged.

The **ASSERT** command is used to make assertions about the computation state of the currently executing program. If the predicate associated with the **ASSERT** command cannot be evaluated to false and is consistent with the existing path condition then no action is taken. Otherwise, an assertion violation message is issued to the user's terminal.

Examples.

ASSERT X< 0;

If X is not less than zero or X< 0 is inconsistent with the current path condition at the time this command is processed then an assertion violation message will be issued to the user.

ASSERT FORALL I(1 TO N) A$I = 0;

This assertion will check that the first N elements of array A are zero.

ASSERT EXISTS I(1 TO N) A$I = 0;

This assertion will check that there is at least one element of array A whose value is zero.

ASSUME X>0 & X<1;

This command will conjunct the predicate 'X>0 & X<1' to the current path condition if the predicate is consistent with it.

3.5.3.2.7 ICL Procedures. - The ICL allows for language extension by providing the means whereby the user can define testing procedures. The pertinent commands are the **PROCEDURE, CLOSE,** and **CALL** commands. The syntax associated with each of these commands closely resembles that of the corresponding HAL/S statement. The procedures can be directly entered into the ITS during a terminal session or may be entered into a system program file and then made available to the ITS by means of the **INSTALL** command. Unless previously entered on a system file, all user-defined procedures will disappear at the end of a session.

A rudimentary parameter passing mechanism is supplied by the ICL which is much like that found in many macro processors. The formal parameters in the argument list of the **PROCEDURE** command are specified as simple identifiers and are separated by commas. Parameters passed to ICL procedures are supplied in the argument list of the **CALL** command and must be listed in the same order as the corresponding formal parameters. ICL procedure invocation will create a new name scope with the executing program so that any formal parameter names which conflict with program variable names will have precedence. Choice of formal parameter names must be carefully made in order to be able to reference all desired program variables.

Example.

Assume the following procedure resides on file 'MYFILE'.

```
STUB: PROCEDURE(PROC,VALUE);
    AFTER PROC ENTRY DO;
        SET X = VALUE;
        ASSUME X = VALUE;
    END;
```

This procedure (along with any others defined on 'MYFILE') can be made available to the ITS by entering the following command:

```
INSTALL MYFILE;
```

'STUB' can be invoked with

```
CALL STUB(COURSE_CORRECTION,1.414);
```

When the block labeled 'course_correction' is entered, this will cause X to be assigned the value 1.414 and the predicate 'X = 1.414' to be added to the path condition.

3.5.3.2.8 ICL Control Statements. - There are two control structure mechanisms provided by the ICL. An **IF-THEN-ELSE** is used to support the conditional execution of ICL commands. The syntax and semantics associated with it are the same as the HAL/S IF-THEN-ELSE. Iterative and discrete **DO FOR** loops are also supported by the ITS. The syntax and semantics of the **DO FOR** command also

follows that of the corresponding HAL/S statement, except the conditional WHILE/UNTIL clause is not supported. The end of the body of the loop text is specified by the **END** command. Nested loops follow the same structuring rules as in HAL/S. The loop control variable is a temporary variable only and is only active during the execution of the loop. Program variables with the same name as the loop variable cannot be referenced within the body of the loop.

These statements are most useful within ICL test procedures and furnish a great deal of flexibilty in creating them.

Example.

```
DO FOR I = 1 TO 3;
    DO FOR J = 1 TO 3;
        IF I = J THEN
            SET A$(I:J) = 1;
        ELSE
            SET A$(I:J) = 0;
    END;
END;
```

In this example I and J are variables used by the ITS as loop control variables. 'A' is a matrix which will be set to an identity matrix when the above commands are processed. Notice that the flavor of these ICL commands is very much like that of the corresponding HAL/S statements.

3.5.4 ITS Operation.

3.5.4.1 Interpreter. - The ITS has been designed to interpret the HALMAT produced by phase one of the HAL/S compiler. A statement pointer points to the next statement to be executed. It is checked with the defined execution scope before execution of the statement to determine if execution is to continue. User interrupt control is handled by the ITS by maintaining an interrupt control list which is also checked before execution of each statement. In this way all actions specified by ICL commands are always performed between statement executions. Interrupt actions associated with the execution of particular statements (e.g. **AFTER/BEFORE** commands) are maintained by a list which is in statement number order. This will allow a rapid search of this list in order to reduce ITS overhead. The interrupt actions associated with the **WHEN** command are more costly. The conditional expression associated with this command will need to be evaluated each time any of the values of the variables referenced in the expression changes. Although this can be a very powerful command, its use will increase the overhead incurred in the interpretation cycle.

It is possible that during the interpretation of a program a variable will be referenced that has not been assigned a value. If the ITS is being operated interactively, the user will be notified of the situation and the system will stop execution. The user may then assign a value (symbolic or actual) to the variable

and resume execution. Other options may be selected as well. At all times the user is in complete control over whatever actions the ITS is to perform.

3.5.4.1.1 Arithmetic. - The HALMAT intermediate language defines a command for each of the different types of arithmetic operations (e.g. integer addition, scalar addition, vector subtraction). The ITS is designed to incorporate all arithmetic operations as individual functional modules. In addition, in order to simulate the arithmetic defined by the architecture of the target computer, the routines will be generalized to handle these different arithmetic properties. Knuth (reference 40) [Knuth,1969] supplies most of the algorithms necessary to facilitate this design. This also will allow easy modification if, later, more arithmetic operations are added to HAL/S (e.g. fixed point arithmetic). The target computer to be simulated will be specified by a parameter supplied to the ITS at the time the system is invoked. As additional target computers become necessary, the ITS will need to have the arithmetic modules modified to allow for the different architectures.

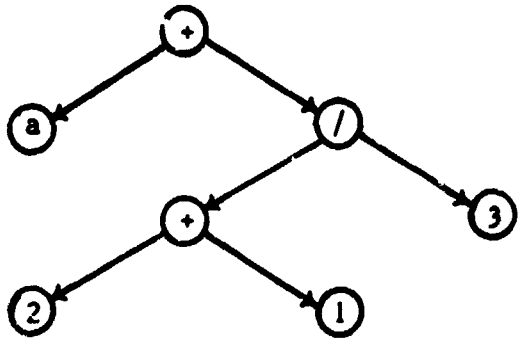
3.5.4.1.2 Real-Time. - Real-time execution can only be simulated by an interpreter and, as a result, there will be problems which will need to be solved. The main area of difficulty lies in the interpreter's maintenance of the information needed to control real-time execution in a manner which will not greatly increase the speed of interpretation. Some of the items which need to be maintained are the process queue and a "pseudo" real-time clock. Process queue control, in all likelihood, will not be any more costly to maintain than the one in the existing HAL/S monitor. Simulation of the real-time clock will require knowledge of the execution times for each HALMAT operation and, in addition, I/O operation timing will need to be performed (or estimated) with the clock being updated after each operation. In any event, the overhead incurred by the interpreter to allow it to handle real-time execution may make this feature prohibitively expensive but it seems reasonable to expect that the actual costs, given good design, can be kept to a minimum.

3.5.4.2 Symbolic Execution. - The symbolic execution facility is an extension of the ITS and, as such, it is actually an extension of the interpreter. A symbolic, rather than actual, arithmetic operation is performed whenever the interpreter encounters HALMAT arithmetic operations in which one or more of the associated operands have symbolic values. In order to identify which variables have symbolic values, it is necessary to attach an additional attribute to them. The attribute is simply a boolean flag which is **TRUE** if the value of the variable is symbolic.

3.5.4.2.1 Expression Evaluation. - Evaluation of assignment statements containing expressions involving symbolic values will, usually, require that both actual and symbolic operations be performed. The result of this evaluation will be an expression represented as a tree where the leaf nodes of the tree are symbolic and actual values. For example, suppose that the following assignment is to be executed where Y = "a", Z = 2, and N = 3:

(1) X = Y + (Z + 1)/N;

The expression tree prior to execution will look like:



After the expression has been evaluated the resulting expression tree will be:

and this will become the symbolic value of the variable X.

Symbolic execution is used to construct the formula which represents the computation associated with a given output variable along a specified program execution path. This formula can then be compared with the functional specification for that particular output variable in order to determine its correctness (or incorrectness). Visual inspection of the formulas resulting from symbolic execution, therefore, becomes an important factor and the manner in which symbolic formulas are output should be given careful consideration. Howden (reference 8) [Howden, 1978a] has shown that a two-dimensional format for symbolic expressions can show the presence of errors caused by the incorrect placing of parentheses. As an example of this format consider again the previous assignment statement (1) and this time let Y = "a", Z = "b", and N = "c". The output in a one-dimensional format resulting from the command **DISPLAY** X; would look like:

$$a + (b + 1)/c.$$

In two-dimensional format the output from the command would be:

$$a + \frac{b + 1}{c}$$

The ITS will utilize this method. Note that this is not the same as HAL/S multi-line format. The exponents and subscripts associated with symbolic values

should be output in single-line mode so that the output formulas will not be cluttered and also to simplify the display algorithm.

3.5.4.2.2 Path Selection. - Selection of the program execution path to be followed during symbolic execution can be made in two modes, static and dynamic. In static mode, path selection is made by means of the **SELECT** command. In dynamic mode the system will query the user for the value of loops and conditional expressions which cannot otherwise be determined. If the system is being used interactively both path selection modes can be utilized. When the system encounters a conditional branch/loop statement which cannot be evaluated to **TRUE** or **FALSE** a path specification list generated by the specified **SELECT** commands is checked. If there is a **SELECT** for the statement in question then the specified branch will be followed. If, however, there is no **SELECT** specified then the system switches into dynamic mode and the user is queried. When the system can evaluate the loop or predicate condition the path selection list is still checked for the corresponding **SELECT** command. If one is found then it is compared with the value of the conditional and if it does not agree a warning message is issued to the user.

If the system is being used in batch, the path must be completely specified. The ITS will terminate execution whenever a branch or loop is encountered which cannot be determined and no **SELECT** has been provided for it. A warning message will also be issued when a **SELECT** command does not agree with the value of its corresponding branch/loop condition.

When a branch's true path has been selected the predicate associated with the conditional statement is logically conjuncted with the current path condition. If the system can determine that a particular predicate is inconsistent with the path condition then a message is issued and execution of the path terminates. A predicate is inconsistent if it results in an unsolvable path condition (e.g. the predicate $X < Y$ is inconsistent with $X > 10$ & $Y < 3$).

The system will not be able to find solutions (and, therefore, check consistency) to predicates involving general non-linear constraints. This is due to the unsolvability of the problem of finding solutions to arbitrary systems of inequalities. The ITS will minimize this problem by solving the linear constraints contained within the path predicate leaving only the non-linear constraints which can then be **DISPLAY**'d by the ITS user (who might attempt to find the solution himself).

3.5.4.2.3 Symbolic Value Assignments. - The assignment of symbolic values is restricted to variables of the following types: integer, scalar, matrix, and vector. Structure terminals and array elements of the aforementioned types can also be assigned symbolic values. Variables of any other types (character, bit, or event) cannot be assigned symbolic values. The symbolic values assigned to variables of type matrix and vector are further restricted in that elements of matrices and vectors cannot be assigned symbolic values. The value assigned to a matrix or vector is associated with the entire matrix or vector. This restriction is necessary because of the matrix and vector operators (e.g. inner and outer

product) which are part of the HAL/S language. It is not at all clear how useful it would be to perform, say, an inner product on two matrices whose elements consist of both symbolic and actual values. It is clear, however, that performing these types of operations symbolically will be expensive. A better approach would be to build matrix algebra operations and simplification rules into the symbolic executor so that formulas resulting from symbolic execution involving matrix and vector variables would more closely resemble their mathematical counterparts. The extension of the symbolic executor to handle matrix and vector simplification is a new feature unique to this particular application. This extension, however, will be quite straightforward to implement and should not pose any difficult problems.

Symbolic execution is used to construct the formulas which represent the computations associated with program variables. It is not clear how useful formulas involving non-numeric variables would be in showing computational errors. In addition, there are currently (to the best of our knowledge) no symbolic execution systems in existence which specifically address themselves to non-numeric applications. And, in as much as real-time flight software is more numerical in nature, it is our contention that symbolic execution should be limited as stated above.

There are, however, some unpleasant ramifications resulting from the above restrictions. Symbolic execution is also used to detect infeasible program execution paths. This capability will be limited by the above restrictions. Again, however, there are currently no existing systems which have the capabilty of solving constraints involving non-numeric variables.

3.5.5 Discussion. - As mentioned earlier, the ITS is designed to provide a tool to aid in the three testing activities: selecting test data, executing the program, and checking the results.

The ITS helps the user to select test data in several ways. Test data can be generated by the symbolic execution facility which will cause a specified path to be executed. Use of the ITS to test an instrumented program (HALMAT monitor file merge performed before ITS execution) will make statement execution counts available to the user. This will allow the user to utilize branch testing methods which can guide him in selecting the test data required to execute any untested code.

The total user control over an executing program provided by the ITS is probably its most important feature. In order to have confidence that a program operates according to its specifications it is necessary to have confidence in the ability of the low-level functional parts to operate properly. The ability to test a program at the lowest level of abstraction can do a great deal to increase one's confidence in the reliability of a program.

The assertion mechanism furnished by the ITS can be used to verify that the computation state satisfies particular constraints at particular points during program execution. This can be used to check that outputs do agree with their

specifications. For example, the following assertion could be used after a sort program has sorted array A to show that the array has indeed been sorted:

ASSERT FORALL I(1 TO N-1) A$(I)<= A$(I+1); .

Assertions are also useful in verifying that input parameters meet the constraints required by their corresponding procedures. This can be useful particularly when a large system is being implemented where the modules comprising the system are being developed by different analysts and it is important to show that the module connections are correct.

The primary utility of symbolic execution is centered around the testing of the functional components of a program at the lowest level of abstraction (reference 38) [Howden,1978c]. A recent paper (reference 35) [Clarke and Ogden,1978] has, however, described a technique whereby symbolic execution can aid in the top-down testing of programs. In the ITS this technique involves setting a trap **AFTER** entry into, as yet, uncoded (or partially coded) procedure "stubs". The ITS **ASSUME** command would then be used to add the constraints which will normally apply to the procedure to the current path condition and the **SET** command would assign symbolic (or actual) values to the output variables. A rudimentary example of this method is found in section 3.5.3.2.7 and is implemented using an ICL procedure. The ITS can be a very useful tool in performing this type of testing.

Prior to embarking on the current ITS design, the existing HAL/S 360 debugger was studied and rejected for the following reasons. Although many of the 360 debugger's capabilities are available in the ITS, that system is only usable in batch and, as a result, it is not possible to specify additional user control over a program once that program's execution begins. In addition, it did not offer any of the controls necessary to perform functional testing, let alone symbolic execution. It also is little different from conventional interactive debuggers in that it is very good at generating a very large amount of output.

In that the ITS is a rather ambitious design, we feel that the system should be built in several stages. The first stage would include an interpreter capable of performing actual interpretation of real-time HALMAT code. The symbolic execution facility would not be included in the initial system but it should not be precluded by the detailed design. The ICL is not a trivial language to parse so that a good deal of attention will need to be given to this part of the system. In particular, the **WHEN, ASSERT, IF,** and **SET** commands all contain expressions which need to be parsed into an interpretable form (i.e. HALMAT). Some restrictions on the complexity or type of expressions which can be used may be appropriate as a result.

The second stage would add a basic symbolic executor to the initial system. It would contain expression simplification modules but it would not check for predicate inconsistencies nor would it attempt test data generation.

The third stage would include all of the features for symbolic execution, i.e., consistency checking and test data generation.

3.5.6 Conclusion. - The design of the Interactive Testing System which has been presented is an attempt to integrate into a single tool the features of an interactive debugger with those of a symbolic executor. The result, however, is a system which is more than a debugger and symbolic executor, it is a software testing system. It is based on the concept of functional testing and its emphasis is on testing and not on debugging. The design has been kept as simple as possible in order to be as user tractable as possible. The use of the symbolic execution facility, however, will still require a sophisticated user who has a thorough understanding of both the capabilities of the ITS and of the inner workings of the program being tested.

All in all, we feel that the Interactive Testing System will be a very valuable tool in the verification of research flight software.

3.5.7 Unresolved Design Issues

3.5.7.1 Real-Time - The problems associated with the interpretation of real-time code are discussed in Section 3.5.4.1.2. As the detailed design of the interpreter is carried out, solutions to these problems should make themselves manifest.

3.5.7.2 ICL Parser. - It was noted in Section 3.5.5 that some restricitons may need to be made on the complexity, or type, of expressions which can be used within particular ICL commands. It may be possible to utilize the routines which are used by the front-end HAL/S compiler to parse the expressions within the ASSERT and KEEP statements in the ICL parser. The resolution of this problem will also become apparent as further design is completed.

3.5.7.3 Ambiguous Array References. - An ambiguous array reference occurs when an array subscript contains a symbolic value. This situation poses problems during symbolic execution due to the fact that it is impossible to uniquely determine which element of the array is being referenced. Current methods of dealing with this problem include marking variables which contain ambiguous values, creating lists of the possible values a variable can have, or just simply ignoring them. Research into a tractable solution to this problem is continuing.

3.6 Dynamic Analysis.

3.6.1 Assertion Facility.

3.6.1.1 Design Principle. - The Assertion and Statistics Gathering Languages (statements) designed are general and powerful. Some of the features are totally new to such languages; to our knowledge a system of this scope has not been implemented anywhere. The decision to adopt such a broad design is based upon our principle that it is crucial to anticipate future needs and make appropriate provisions for them. Indeed, the syntax is incompletely defined - further consideration and experience with the provided features will dictate their completion.

Basic features will be implemented at first. As experience guides, additional features will be supported, with their implementation and integration being able to proceed smoothly. The syntax will require no revision and previously instrumented programs will not require any changes. In fact, programs may contain assertions which reference (hitherto) unsupported features. Such assertions serve as important documentation. When the support features they require are provided, they then assume their role as active monitors.

The basic ideas contained in the design were obtained from two primary sources: reference 11 [Stucki, 1976] and reference 12 [Chow, 1976]. Both contain excellent expositions of the utility of assertions and provide many examples. Chow provides several examples of specialized assertion functions which may be defined. Some of these would require special implementation, but their semantics are harmonious with the design presented. Thus they are candidates for future inclusion in the assertion language.

3.6.1.2 Implementation. - The support of the designed features will certainly take place in phases. Four major factors are involved.

First is the problem of determining a suitable instrumentation schema for any given facility. For most of the facilities described, this is straightforward. Difficulties arise though, for example, in consideration of the INVARIANT clause. If such a clause is used in a concurrent process program, guaranteeing the invariance of shared data may be very difficult. Perhaps more important is the problem of discovering efficient instrumentation schemas.

The instrumentation required to implement the histogram-type information could be provided in several ways. One alternative is to utilize the execution monitor, as opposed to inserting special probes directly in the code. The default compiler mode generates calls to the execution monitor following each HAL/S statement. The monitor performs any duties associated with the real time aspects of the program, among other things. Since the "hook" to each statement is thus automatically provided, the monitor could be modified to gather the histogram information. Such modification is not recommended, however. Unnecessary overhead would result, controlling the extent of histogram-gathering would be difficult, and dependencies would be placed on using the monitor - which

may be undesirable on many target computers. Much greater flexibility and economy is achieved through the direct insertion of probes.

Another decision governs the nature of the probes which are inserted. Inline code may be created, or a procedure call may be used. Inline code executes faster, but may incur a size penalty in the object program. Global declarations to support the instruments must be supplied by the tool, and some run time flexibility may be sacrificed. Subroutine calls are smaller, require less "declaration" effort, and greatly increase flexibility over in-line code. The execution time penalty associated with procedure invocation may be prohibitive in many cases, however.

In light of these considerations, it is recommended that in-line code be used predominantly, but that the ability to use procedure calls should not be precluded. Since the cost of procedure invocation may vary significantly from implementation to implementation (and language to language), it is also recommended that timing studies be undertaken to aid in determining the proper mixture of techniques. User control over the type of instrumentation to be utilized is an important option.

Regardless of the scheme used to implement the histogram-type facilities, two concerns must be kept in mind. It must be guaranteed that regardless of where program termination occurs statistics will still be captured and the interfaces with the file system must satisfy overall efficiency requirements.

All of these concerns have been addressed and are discussed at length beginning in Section 3.6.6.

Second in the list of major implementation factors is the problem of translating the assertions into the instrumentation required. Parsing of the assertion itself is a problem, as sophisticated expressions may be present. Clearly the use of compiler routines is mandated, and this should be readily accomplished as the routines to pull the assertion out of the comment brackets will be included within the compiler. This is discussed in Section 3.9.

Third, as noted in the description of the assertion syntax, the problem of "specification" or denotation arises. This is with regard to path specification. The facilities which are desired to present such a capability must be determined. This will be discussed later.

3.6.1.3 Restrictions and Capabilities. - Much of the generality provided by the assertion and statistics gathering statements arises from the ability to invoke a function, in the general sense, as a part of expression evaluation. Sophisticated, tailor-made functions may be provided to perform a variety of checking activities. These functions may be catalogued and saved for use on many different classes of software. For example, certain functions might be particularily useful when verifying real-time software. (Note that the real time clock may be referenced). The instrumentation of such functions would allow their execution to take place in "zero time" in a simulation environment. To fully simulate a real time program, though, the timing of external interrupts must be adjusted to compensate for the increase in actual execution time.

The caveat associated with this capability is that the functions which are called must not have any side-effects. A program must execute with instrumentation identically as without. Enforcement of this rule will necessitate restrictions on the composition of the functions.

A partial list of supporting capabilities follows. The most important are discussed at length later in this document.

1. The assertion processor accepts controlling input. This allows information such as selective instrumentation commands to override commands embedded in the source text.

2. The selective instrumentation capabilities allow, for example, only one module of a multi-module program to be instrumented. Even if execution halts in an uninstrumented module, the statistics from the instrumented module will still be gathered (assuming the instrumented module is executed at least once).

3. Standard functions may be provided to query aspects of the operating environment. These queries allow the program to assert that it is operating under the conditions for which it was designed. Such functions may concern physical (hardware) characterisitics or software support. These functions will necessarily be implementation dependent. (For example, an assertion may be made about the target machine's word size).

4. The post processor, which prepares reports containing the statistics gathered during execution, will allow information gathered from several test runs to be presented in a single report. Summaries of the statistics obtained from each run will be obtainable as well.

5. The facility which actually inserts the instrumentation provides an indication of which assertions/keeps actually generated monitors. In addition, for any given set of instruments the facilities should attempt to estimate at least the increase in program size caused by inserting the instruments into the program, if not a timing estimate too.

6. If relevant aspects of the output from test runs are retained in the system data base, a facility may be provided which will monitor the progress of the testing activities. Test coverage may be considered, as well as examination of the number of assertions violated per run. It would then be possible to use software reliability models to estimate the quality of the software. See reference 13 [Lloyd and Lipow, 1977], chapter 17, or reference 14 [Sukert, 1977] for a consideration of this.

3.6.1.4 Notation. - The grammar used to describe the assertion and statistics gathering languages is a variant of BNF, described below.

i) Nonterminals are underlined, e.g., assert statement

ii) Terminals composed of Latin letters are printed in upper case, e.g.,
ASSERT

iii) Terminals composed of special characters are printed in bold face, e.g., **( )**

iv) Items which are optional are enclosed in parentheses, e.g., (GLOBAL)

v) Items suffixed with an asterisk (*) may appear zero or more times

vi) Items suffixed with a plus sign (+) may appear one or more times

vii) multiple productions corresponding to a single non-terminal are listed on successive lines. The non-terminal and the ::= sign only appear on the first production.

e.g., value ::= comparison
path expression

3.6.2 Assertion Language.

3.6.2.1 Grammar.

assert statement ::= /* (special label) ASSERT (GLOBAL) ext-logical-exp-list */;
ext-logical-exp-list ::= ext-logical-exp (; ext-logical-exp)*
ext-logical-exp ::= value (relop value)*
    expression list INVARIANT (TO special label)
    expression list (NOT) IN range+
    expression list OUTPUT
value ::= comparison
    path expression
    quantifier comparison
expression list ::= expression (, expression)*
relop ::= conditional AND
    conditional OR
quantifier ::= FORALL quantifier completion
    EXISTS quantifier completion
quantifier completion ::= integer variable range (, integer variable range)*
range ::= (constant (TO constant))
path expression ::= PATH special label WAS path
    PATH path
path ::= to be determined
end ASSERT stmt ::= /* END ASSERT special label */ ;
invariant mark ::= /* END INVARIANT special label */ ;
statistics value ::= (name) @ (special label)

3.6.2.2 Context Sensitive Rules.

1. No two ASSERT statements may have the same special label.

2. No two invariant mark statements may have the same special label.

3. Multiple ASSERT GLOBAL/ END ASSERT statements are possible, and nesting is not required. As the system is used and feedback obtained, such a requirement may be added later.

4. The GLOBAL keyword may not be used in conjunction with the TO special label clause, nor the OUTPUT clause.

5. To aid in efficient implementation, the number of ext-logical-exps allowed in a single assertion may be limited to an implementation-defined maximum. The same is true for the number of expressions allowed in expression list and the number of ranges allowed in a single ext-logical-exp. Note that such restrictions do not affect the power of the facility--only the format of the special statements.

3.6.2.3 Semantics.

1. Any and all ASSERT statements may be labeled with a Dewey decimal number. Their instrumentation may be controlled by an external mechanism which references these numbers.

2. The ASSERT GLOBAL statement specifies a list of conditions which must continuously hold over a range of the program. This range is demarcated by the ASSERT statement and the END ASSERT statement whose special labels match. If no such END ASSERT statement exists or if the ASSERT statement is unlabeled, the assertion applies to all program text following the ASSERT statement in the current static scope (at the procedure, task, or program level). The expressions listed must not reference any variables whose scope is smaller than the range of the assertion. Note that if subscripted variables (or otherwise parameterized expressions) are used in the assertion, the entire expression is evaluated anew each time a check is required. Thus if the expression was A$(I) = 1, then the subscript I is evaluated anew at each check point.

3. A series of logical conditions may be expressed in a single ASSERT statement. If the GLOBAL keyword is present, all the logical conditions must hold throughout the range of the assertion.

4. Contradictory assertions may be specified for the same program region. By definition, if both are instrumented, an assertion violation will be reported whenever that region of code is executed.

5. Each extended logical expression which is checked may include conditional operands (tokens such as CAND and COR may be appropriate). In the conditional expression A conditional and B, B will be evaluated if and only if A is true. In the conditional expression A conditional or B, B will be evaluated if and only if A is false. Evaluation proceeds from left to right, with no parenthetical nesting. By using such expressions dynamic control over assertion evaluation is achieved. (Indeed, if the first part of the expression is evaluable at compile time, a more efficient form of instrumentation may be possible.)

6. "Threshold" control may be achieved in a similar manner. The special value VIOLATE (special label) may be used within any comparison in an assertion. Its value is the number of times the referenced assertion has been violated. If no reference (special label) is provided, the number of violations of the current assertion is taken.

7. Special values COUNT (special label) and statistics value may also be used within any extended logical expression. COUNT refers to the execution count of the referenced KEEP statement (if no reference is provided, the COUNT of a hypothetical KEEP COUNT statement which immediately precedes the ASSERT is used.) statistics value allows the value of any HAL/S expression which is saved in a KEEP to be referenced in an assertion. The optional name which precedes the label allows a particular value to be referenced out of several saved at the KEEP (there may have been a list of expressions to KEEP). The name supplied must be textually identical to one of the expressions listed in the KEEP.

8. Path expressions allow assertions to be made about execution paths previously taken and kept, or predictive assertions about what path will be followed after checking of the assertion. Further discussion of this facility may wait until a suitable notation for specifying a path is adopted. Such a mechanism must allow a convenient, useful path specification to be made before program compilation. Decisions about what is done when subroutines are called will have to be made.

9. Quantifiers on comparisons allow the formation of powerful assertions. The quantifier completions presented allow for looping constructs. When used with FORALL, the assertion must hold true as the integer variable assumes each of the values specified in the range. When multiple integer variables and ranges are specified the assertion must hold for every combination of integer variable and value. When used with EXISTS the assertion must hold true for at least one integer-variable and value (or combination thereof, if several integer variable ranges are specified).

10. The INVARIANT (TO special label) clause specifies that none of the expressions listed in the statement will vary in value as long as control remains within the scope of the invariant assertion. The invariant mark statement may be used to define the end of the region throughout which the value of the expressions must remain constant. The special label found on the TO clause and the invariant mark must be identical. If no such mark is found, or if the TO special label phrase is omitted, the end of the current local scope (at the procedure, task, or program level) is used. If control enters the scope of the invariant without "passing through" the ASSERT statement, the value of the expressions must be the same as (invariant to) the value of the expressions the last time the ASSERT was executed. If the ASSERT has never been executed an assertion violation will result. The INVARIANT (TO special label) clause allows assertions which may check for parallel processing errors. (The clause is also useful for indicating what variables are input-only to a routine. This allows protection of global variables used in internal scopes. Procedure parameters are already protected through the formal parameter/ASSIGN mechanism.) If several shared variables are being referenced in a supposedly critical region, they should not be updated concurrently. They must remain INVARIANT to the end of the critical region. Efficient implementation of this feature for such concurrent processing applications will require significant study if such checking is performed dynamically. Static verification is the preferred technique.

11. The IN range specification indicates that each value specified in the expression list must lie within one of the ranges provided. A range may consist of a single value. Initially, ranges may only be specified for integer and scalar valued expressions.

12. The expression list OUTPUT specification gives a complete list of the expressions, usually variables, which are "produced" or modified by a section of code. It is therefore implied that only those expressions, and no others to which a definition is given in the current scope, will occur in reference contexts in the same (static) scope following the OUTPUT assertion.

3.6.3 Statistics Gathering Language

statistics statement ::= /* (special label) KEEP GLOBAL function list */ ;

    /* (special label) KEEP svalue list (qualifier) */ ;

end keep statement ::= /* END KEEP special label */ ;

special label ::= integer (. integer)* (.)

function list ::= function (, function)*

function ::= COUNT ((stmt-type list))

    VARTRACE (varlist)

    PROCTRACE (proclist)

    SNAPSHOT

    normal function

svalue list ::= svalue (, svalue)*

svalue ::= expression

    COUNT

    SNAPSHOT

    PATH ((integer))

qualifier ::= IF comparison

stmt-type list ::= stmt-type (, stmt-type)*

stmt-type ::= ALL

    ASSIGN

    CALL

    CANCEL

    DOCASE

    DOLOOP

EXIT
FILE
GOTO
IF
ONERROR
OFFERROR
READ
RESET
RETURN
SCHEDULE
SENDERROR
SET
SIGNAL
TERMINATE
UPDATE
WAIT
WRITE

Context Sensitive Rules

1. No two KEEP statements may be labeled with the same number.

2. Multiple KEEP GLOBAL / END KEEP pairs are possible, and nesting is not required. As the system is used and feedback obtained, such a requirement may be added.

Semantics

1. All KEEP statements may be labeled with a dewey-decimal number. As such they are individually named and their instrumentation may be controlled in a sophisticated manner by an external mechanism (directives to the KEEP statement processor).

2. The KEEP GLOBAL statement specifies a list of functions which are to be called after every (applicable) statement within the textual scope defined by the KEEP GLOBAL statement and the END KEEP statement whose special labels match. If no matching END KEEP is found, such a statement is generated at the end of the current textual scope (at the procedure, task, or program level).

3. The functions which may be invoked at each (appropriate) statement are as follows:

COUNT - provides a count of the number of times each statement was executed. The stmt-type list qualifier allows the user to restrict the types of statements for which this information will be kept. The default is ALL statements.

VARTRACE - each variable listed in the function is traced throughout the range of the KEEP. At each point of change the variable's name and new value is printed, along with the statement number of the statement causing the change, and the current execution count (if kept).

PROTRACE - each procedure listed in the function is traced throughout the range of the KEEP. At each procedure invocation the procedure's name is printed, along with the statement number of the CALL, the value of the parameters, and the current execution count (if kept). Printing of the parameters may be subject to implementation restrictions.

SNAPSHOT - whenever a statement in the range of the KEEP directly causes a change in the status of any process, a snapshot will be taken of all the system monitor's queues. Thus an indication is given of which processes are ready, waiting, and inactive. Depending on the implementation of the monitor, an indication may also be given of what events (or event variables) the waiting processes are blocked on. The snapshot will also give the statement number of the statement which caused the snapshot to be generated.

normal function - this is a general HAL/S function which will be called after the execution of each statement. This provision is in keeping with the overall criterion of providing a general syntax. Implementation restrictions, as previously mentioned, are almost certain. The function name may be followed by a list of parameters. The scope of the parameters must be consistent with the scope of the keep. statistics value (described in Section 3.6.2) is a legal parameter.

4. If GLOBAL is not specified, the KEEP statement refers only to the program state defined at the point of the KEEP.

5. svalue may be any computable expression (including HAL/S normal functions) and is subject to the rules provided for functions in rule 3 above. statistics value is a legal part of an expression. COUNT has the same meaning as noted above, but may not be qualified. Thus it refers only to the number of times control passed through the KEEP statement. SNAPSHOT causes a snapshot to be taken of the monitors queues as they exist at the time of execution of the KEEP.

6. Specification of PATH will cause a record to be kept of the execution path taken from the KEEP statement until an END KEEP statement is encountered

which has a matching special label. If the PATH is qualified with an integer n, the path record will be limited to a maximum of n statements. Only the first n statements encountered will be retained.

7. If a KEEP statement has a qualifier phrase, the information requested will be kept only if the condition is met. Evaluation of the condition is subject to the extensions and restrictions applied to normal functions in rule 3 above.

3.6.4 Rationale. - There are two main motivations for the provision of the KEEP statements. The first is to allow assertions to reference previous values of variables. The second motivation is to allow the user to control to some extent the information which will be produced as a "histogram" of the programs execution. This histogram normally contains execution counts, but may include other items as well.

The keeping of voluminous amounts of detail concerning a programs execution history is most closely associated with debugging systems. Such systems have a decidely different flavor than the dynamic analysis system considered here. As the preliminary design includes an interactive test system (SAMM node CCB), facilities for production of such information are not included in this specification. Necessarily the line drawn between the two is somewhat arbitrary, but we believe the distinction drawn is a useful one.

3.6.5 Sample Usages of the Assertion and Statistics Gathering Facility.

1) /* ASSERT A=B+C; D>6; F(X)¬=0 */;
Three simple arithmetic relations which must be true at the point of assertion placement.

2) /* ASSERT A>5 CAND F(X) = F(Z) */;
Two arithmetic relationships. The second relationship is checked (causing evaluation of the functions) if and only if A>5.

3) /* ASSERT A>5 CAND B<0 COR C=0 */;
Three arithmetic relationships. B < 0 is evaluated if A > 5. C=0 will be evaluated if the value of the entire expression to the left of the COR is false. The chart below indicates all possible evaluation/value combinations.

A>5	B<0	C = 0	Assertion value
T	T	unevaluated	T
T	F	T	T
T	F	F	F
F	unevaluated	T	T
F	unevaluated	F	F

4) /* ASSERT VIOLATE <5 CAND F(X)=0 */;
F(X) will only be compared with zero if this assertion has not been violated more than 4 times.

5) /* ASSERT GLOBAL X>0 */;
X must remain positive from the assertion through the end of the current scope (either procedure, task, or program end).

6) /* 1 ASSERT GLOBAL X>0 */;
: X must remain positive throughout this region
/* END ASSERT 1 */;

7) /* ASSERT A,B,C INVARIANT TO 3.1 */;
: A,B,C must remain unchanged in this region
/* END INVARIANT 3.1 */;

8) /* ASSERT X+Y INVARIANT */;
The value of the expression X+Y must remain constant until the end of the current procedure, task, or program.

9) /* ASSERT X IN (1 TO 6)(12) */;
The condition $1 \leq X \leq 6$ or X=12 must be satisfied.

10) /* ASSERT X,Y OUTPUT */;
Only variables X and Y will occur in reference contexts below this point in the current textual scope (procedure, program, or task).

11) /* 1.1 KEEP X */;
The current value of X is retained for later use in an assertion.

12) /* ASSERT X@(1.1) = X */;
Asserts that the last value of X stored at KEEP 1.1 is equal to the current value of X.

13) /* ASSERT @(1.1) = X */;
Same as example 12). This syntax is valid if KEEP 1.1 only retained variable X.

14) /* KEEP GLOBAL COUNT */;

An execution frequency count is kept for all statements occurring after the KEEP until the end of the current scope (procedure, program, or task).

15) /* KEEP GLOBAL COUNT (READ, WRITE, FILE) */;

An execution frequency count is kept on all input-output statements occuring after the KEEP until the end of the current scope.

16) /* KEEP COUNT IF FLAG */;

A selective execution count will be kept for this statement. The count will be incremented only when variable FLAG has the value TRUE.

17) /* KEEP X IF F(X)>5 */;

The value of X will be retained only if F(X)>5.

18) /* ASSERT X<0 COR SPECIAL_ERROR_HANDLER(X)*/;

This example illustrates how special processing may be performed on assertion violation. If X is not less than zero then (presumably) something has gone awry in the program. In order to gather as much information as possible a user-supplied function is called which may, for example, print out a helpful message.

19) /* ASSERT FORALL I(1 TO 10), J(1 TO 5) A$(I)<B$(J) */;

This assertion is equivalent to the logical conjunction of the following assertions:

A$(1)<B$(1)
A$(1)<B$(2)
.
.
.

A$(1) < B$(5)
A$(2) < B$(1)
A$(2) < B$(2)
•
•
•
A$(10) < B$(5)

In other words, each element of A must be less than every element of B.

20) /* ASSERT FORALL I(1 TO 10) A$(I) < = A$(I+1) */; This asserts that the first 11 elements of A are sorted in ascending order.

21) /* ASSERT EXISTS N (4 TO 100) A**N = B**N+C**N*/;
This assertion declares that there exists at least 1 value of N between 4 and 100 inclusive such that $A^N = B^N + C^N$, for values A, B, and C. (This assertion will fail, of course, if A, B, and C are integers and (A)(B)(C) ≠ 0)

22) /* 22. KEEP MAX(X, @(22.)) */;
Assuming that function MAX returns the larger of its two arguments, this KEEP will retain the maximum value of X which occurs at this statement. (MAX must also be able to detect that @(22.) is undefined on the first call of the function. This ability is implementation dependent).

3.6.6 Instrumentation Schema - The assertion/keep preprocessor (SAMM node CBCAAB) will generate the following information describing the ASSERT/KEEP statements present in the program. This information may then be translated into algorithms to implement the statements. The algorithms to be used are presented following the information description. Assuming Pascal as the implementation language, it is used as the description means.

3.6.6.1 ASSERT Information

```
const

    max_dewey_length = 10 ; (* maximum nesting level for special labels *)
    max_exp_length   = 50 ; (* maximum number of integer words for HALMAT
        computation of an expression *)
    max_num_expressions = 1 ; (* maximum number of expressions allowed in an
        expression list *)
    max_num_ranges = 10 ; (* maximum number of ranges which can be given in
        a single range assertion *)
    max_num_conditionals = 5 ; (* maximum number of relops allowed in an
        ext-logical-exp *)
    max_num_values = max_num_conditionals + 1 ; (* maximum number of
        values allowed in an ext-logical-exp *)
    max_ext_logical_exps = 1 ; (* maximum number of ext-logical-exps in an
        ext-logical-exp-list *)
    max_quantifiers = 6 ; (* maximum number of integer variable - range pairs
        allowed in a single quantifier completion *)

type

    small_int = 0..100; (* small integers *)
    dewey_decimal = array (1.. max_dewey_length) of small_int; (* dewey
        decimal numbers used in special labels *)
    assertion_type = (value, invariant, range, output); (* the basic types of
        assertions allowed *)
```

```
relop = (cand, cor) ; (*  the internal tokens we shall use for conditional
      and/conditional or *)
constant_type = (int, scalar); (*  the only constant types allowed in range
      specifications *)
quantifier_type = (forall, exists); (* the kinds of quantifiers possible*)
Halmat = integer ; (*  this defines what a single HALMAT word is.  This
      could be defined more explicity (and possibly usefully) as a variant
      record without a tag field.  Each possible HALMAT word format could
      then be described.*)
expression = array (1..max_exp_length) of Halmat ; (* the sequence of
      HALMAT words corresponding to the computation of the expression.
      Special values COUNT and VIOLATE are accepted by the preprocessor
      and generate special pointers into the symbol table. *)
expression_list =
      record
            exps: array (1..max_num_expressions) of expression;
            num_exps: integer ; (* number of expressions actually present in
                  the list *)
      end ; (* expression list record definition *)
constant =
      record
            case con_type: constant_type of
                  scalar: (float: real );
                  int: (fixed: integer );
      end ; (* constant record *)
range =
      record
            first: constant;
            second: constant;
      end ;
```

```
quantifier_completion =
     record
          quant_var: integer ; (* a pointer into the symbol table indicating
               the variable used in the quantifier.*)
          quant_range: range; (* the range specified *)
     end ; (* quantifier completion record *)
value =
     record
          quantifier: quantifier_type; (* the nature of the quanifier pres-
               ent, if any *)
          num_quantifier_completions: small_int; (* the number of quanti-
               fier completions present*)
          quant_completions: array (1..max_num_quantifiers) of
                quantifier_completion; (* ordered list of the quantifier
               completions specified *)
          value_exp: expression; (* the comparison which is quantified*)
     end ; (* value record *)
ext_logical_exp =
     record
          case assert_type: assertion_types of
               value:
                    (num_conditional_ops: small_int; (* number of condi-
                         tional operands in this ext-logical-exp*)
                     conditional_ops : array (1..max_num_conditionals) of
                         relop; (* an ordered list of the operands
                         found *)
                     values: array (1..max_num_values) of value; (* a list
                         of the separate values *)
                     );
               invariant:
                    (limit: dewey_decimal; (* the special label of the END
                         INVARIANT statement (if specified) *)
```

```
                    invar_exps: expression_list; (* the expressions which
                        must remain invariant *)
                    );
                range:
                    (num_ranges: small_int; (* the number of ranges speci-
                        fied*)
                    ranges: array (1..max_num_ranges) of range ; (* the
                        ranges that were specified *)
                    range_exps: expression_list; (* the expressions to
                        which the ranges apply *)
                    );
                output: (output_exps: expression_list)
end ; (* end of ext-logical-exp record definition *)
initial assert =
    record
        SMRK_pointer: integer ; (* the SMRK of the ASSERT state-
        ment*)
        special_label: dewey_decimal ; (* the special label on the
            ASSERT*)
        global: boolean ; (* whether or not the GLOBAL keyword is
            present *)
        count_flag: boolean ; (* whether or not the assertion requires the
            existance of a count-type KEEP statement before the
            assertion*)
        num_ext_logical_exps : small_int; (*the number of ext-logical-
            exps present in this assertion*)
        ext_logical_exps : array (1..max_ext_logical_exps) of
            ext_logical_exp; (* here is the body of the assertion *)
    end (* initial assert record *)
```

3.6.6.2 Instrumentation. - The instrumentation to be derived from this information is as follows. The basis for all the instrumentation is the simple rule:
if not assertion_condition **then call** assertion_violation ;
The assertion_violation procedure is largely independent of the type of assertion. The code to perform the necessary actions can be contained in a parameterized procedure. The following HAL/S procedure (Figure 3.6.6.2-1) is an example of such a procedure. The utility of its parameters should be self-evident. They are fully described in the section entitled "Control of Instrumentation." Additional parameters could be allowed which would, for example, indicate the value of the expressions which did not satisfy the condition in the assertion.

It should be clear that with this instrumentation schema (and the control provided in Section 3.6.7) the user has full control of actions taken on assertion violation. If instruments are left in the code executing on the flight computer the same flexibility exists. If program termination is desired, that is possible subject to the capabilities of the system monitor (if any). Likewise a continuing record of violations may be kept, as long as there is a channel on which the messages may be preserved.

3.6.6.2.1 Global parameter - The presence of the GLOBAL keyword will require the addition of the following information to the above data structure.

```
num_important_vars: small_int;
important_vars: array (1..max_num_important_vars) of integer ;
    (* where each integer is an index into the symbol table. Each variable
    involved in the computation which checks the assertion is contained in
    the list *)
```

The implication is that whenever any of the variables listed appears in an assignment context in the range of the assertion, the assertion must be checked after the assignment has been made. Such assertions must always be checked after each procedure/function invocation as well, to ensure that the variables involved were not altered in an illegal manner due to procedure/function side effects. (If the absence of side-effects is guaranteed such checking may be eliminated. The checking for side effects could be performed by the static analyzer. The existence of such checking would be reported to the user. He would then be obligated to remove the unnecessary checks through directives to the HMF/HALMAT merge operation.)

3.6.6.2.2 Conditional operands. - The instrumentation of a sequence of comparisons (values) separated by conditional operands requires the use of a simple device. Suppose the following sequence is to be instrumented:

$$c_1\ r_1\ c_2\ r_2\ c_3$$

where c_1, c_2, c_3 are comparisons and r_1 and r_2 are relops (here we shall use

```
 1 !                                                                                !
 1 ! ASSERTION_VIOLATION: PROCEDURE (MESSAGE, CHANNEL, STMTNUMBER,                  ! ASSERTION_VIOLATION
 1 !                       COUNT_AVAIL, COUNT, HALT)                                ! ASSERTION_VIOLATION
 1 !                       ASSIGN (VIOLATE) REENTRANT ;                             ! ASSERTION_VIOLATION
 1 !                                                                                ! ASSERTION_VIOLATION
 2 ! DECLARE BOOLEAN, MESSAGE ; /* WHETHER OR NOT TO WRITE A MESSAGE */             ! ASSERTION_VIOLATION
 3 ! DECLARE BOOLEAN, COUNT_AVAIL ; /* WHETHER OR NOT THE EXECUTION COUNT IS */     ! ASSERTION_VIOLATION
 3 !                    /* AVAILABLE */                                             ! ASSERTION_VIOLATION
 4 ! DECLARE BOOLEAN, HALT ; /* WHETHER OR NOT TO STOP EXECUTION */                 ! ASSERTION_VIOLATION
 4 !                                                                                ! ASSERTION_VIOLATION
 5 ! DECLARE INTEGER, CHANNEL ; /* CHANNEL TO WRITE MESSAGES ON */                  ! ASSERTION_VIOLATION
 5 !  /* THIS PARAMETER IS CURRENTLY UNUSED AS THE CHANNEL NUMBERS IN */            ! ASSERTION_VIOLATION
 5 !  /* WRITE STATEMENTS CANNOT BE PARAMETERIZED */                               ! ASSERTION_VIOLATION
 6 ! DECLARE INTEGER, STMTNUMBER ; /* STATEMENT NUMBER OF THE ASSERTION */          ! ASSERTION_VIOLATION
 7 ! DECLARE INTEGER, COUNT ;  /* CURRENT EXECUTION COUNT */                        ! ASSERTION_VIOLATION
 8 ! DECLARE INTEGER, VIOLATE ; /* VIOLATION COUNT FOR THIS ASSERTION */            ! ASSERTION_VIOLATION
 8 !                                                                                ! ASSERTION_VIOLATION
 9 ! REPLACE STOP BY ' INFINITE_LOOP: SEND ERROR $(4:14); GO TO INFINITE_LOOP;';   ! ASSERTION_VIOLATION
 9 ! /*  THIS WILL HAVE THE EFFECT OF ABNORMALLY TERMINATING THE PROGRAM */        ! ASSERTION_VIOLATION
 9 ! /*  ACCORDING TO THE HAL/S-370 MANUAL (VERSION IR-58-15     ) NO LOOP IS */   ! ASSERTION_VIOLATION
10 ! /*  REQUIRED, BUT  THE MANUAL DOES NOT CORRESPOND TO THE SEMANTICS OF */;     ! ASSERTION_VIOLATION
10 ! /*  THE SYSTEM.  ACTUALLY, BECAUSE HAL/S CONTAINS NO HALT STATEMENT */       ! ASSERTION_VIOLATION
10 ! /*  THE IMPLEMENTATION OF SUCH A CONSTRUCT WILL ALWAYS BE DEPENDENT  */      ! ASSERTION_VIOLATION
10 ! /*  ON THE ACTIONS OF THE SYSTEM MONITOR  */                                  ! ASSERTION_VIOLATION
10 !                                                                               ! ASSERTION_VIOLATION
10 !                                                                               ! ASSERTION_VIOLATION
11 ! VIOLATE =  VIOLATE + 1 ; /* THIS IS THE MINIMAL ACTION ON ASSERTION VIOLATION*/! ASSERTION_VIOLATION
11 !                                                                               ! ASSERTION_VIOLATION
12 ! IF MESSAGE THEN                                                              ! ASSERTION_VIOLATION
13 !   DO;                                                                         ! ASSERTION_VIOLATION
14 !      WRITE (6) ' ***** ASSERTION VIOLATION ***** AT STATEMENT NUMBER ',       ! ASSERTION_VIOLATION
14 !          STMTNUMBER ;                                                        ! ASSERTION_VIOLATION
15 !      IF COUNT_AVAIL THEN                                                     ! ASSERTION_VIOLATION
16 !          WRITE (6) COLUMN(8), ' CURRENT EXECUTION COUNT = ',COUNT ;          ! ASSERTION_VIOLATION
17 !   END;                                                                        ! ASSERTION_VIOLATION
17 !                                                                               ! ASSERTION_VIOLATION
18 ! IF HALT THEN                                                                 ! ASSERTION_VIOLATION
19 !   DO;                                                                         ! ASSERTION_VIOLATION
20 !      IF MESSAGE THEN                                                         ! ASSERTION_VIOLATION
21 !         WRITE (6) ' EXECUTION TERMINATED ABNORMALLY DUE TO ASSERTION',       ! ASSERTION_VIOLATION
21 !            ' VIOLATION ';                                                   ! ASSERTION_VIOLATION
24 !      STOP ;                                                                  ! ASSERTION_VIOLATION
25 !   END;                                                                        ! ASSERTION_VIOLATION
25 !                                                                               ! ASSERTION_VIOLATION
26 ! CLOSE ASSERTION_VIOLATION ;                                                  ! ASSERTION_VIOLATION
26 !                                                                               ! ASSERTION_VIOLATION
```

Figure 3.6.6.2-1 Assertion Violation Procedure

CAND and COR). The instrumentation requires the introduction of a temporary boolean variable p.

The instrumentation is as follows:

```
p = c1;
if ((r1 = CAND) and (p = true)) or
     ((r1 = COR) and (p = false))
     then p = c2 ;
if ((r2 = CAND) and (p = true)) or
     ((r2 = COR) and (p = false))
     then p = c3;
if not p then call assertion_violation ;
```

The comparisons r_i = CAND and r_i = COR can be evaluated at preprocessing time. If r_1 = CAND and r_2 = COR the following instrument sequence is generated:

```
p = c1;
if p then p = c2;
if not p then p = c3;
if not p then call assertion_violation ;
```

3.6.6.2.3 Invariant Assertions. - Invariant assertions will best be checked by a static data flow analyzer. Checking by instrumentation is possible only in a limited sense.

One instrumentation scheme involves the establishment of a "filter" between all active processes and memory. If one process declares an expression invariant, then the filter does not allow any changes to any of the variables involved in the computation of the expression, so long as the invariant clause is in effect. (It could allow "changes" to the variables, as long as the new value was equal to the previous value.) The inefficiencies and difficulties associated with this scheme are apparent.

Another candidate is as follows. When the invariant assertion is encountered the values of all variables involved in the computation of the expressions are stored in temporary variables which are local to the process in which the assertion appears. Throughout the range of the invariant assertion all references to the involved variables are directed to the temporary variables.

(Definition contexts in the process containing the INVARIANT clause are illegal, of course, and are statically detected.) One problem here is if procedure or function invocations are included in either the expressions themselves or within the range of the assertion. In the former case it may not be clear what variables must remain constant, nor could such a restriction be enforced. In the latter case references to invoked variables within the procedure/function could not be directed to the temporaries. Still another clear drawback is that we have forced the expressions to be invariant, rather than checked them to be so. As such the program will behave differently with the instrumentation than without--a violation of a basic principle. We must conclude this candidate is unacceptable.

A third solution is to enclose the statements comprising the range of the assertion in a LOCK group. This is not desirable, however, as the semantics of the program are different when the assertion is instrumented, as opposed to when it is not, as in the store-values-in-temporary-variables candidate above.

3.6.6.2.4 Range Assertions. - Range assertions are handled as follows: for each expression in expression list and each range in range+, the check

```
if not(expression ≥ range.first and expression ≤ range.second)
    then call assertion_violation;
```

must be generated. If the GLOBAL keyword is present the check must be made any time an involved variable is defined within the scope of the assertion.

3.6.6.2.5 Output Assertions. - Output assertions, like invariant assertions, should be checked statically.

3.6.6.2.6 Quantified comparisons. - Comparisons which are quantified with FORALL or EXISTS clauses may be instrumented in the following way. Each integer variable - range pair may be translated into a loop. If several such loops are required they must be nested. integer variable ($constant_1$ TO $constant_2$) is translated into the following loop:

```
do for integer variable = constant1 to constant2 ;
```

If $constant_2$ is omitted then $constant_1$ is used in its place.

The difference between FORALL and EXISTS is in the contents of the loop. In the FORALL situation the comparison must be true for each execution of the loop. In the EXISTS situation the comparison need only be true for one execution of the loop. An auxillary variable may be used to keep track of the comparison "history".

Example: /* ASSERT FORALL i (1 TO 100) $A_i \not< B_i$ */;

```
do for i = 1 to 100 ;
    assert = assert and (A(i) ¬< B(i)) ;
if not assert then call assertion_violation ;
```

Note that if the assertion is false, and such falsehood is established for a small value of i, the entire loop need not be executed, and could be exited. Since assertion violation is (presumably) an infrequent occurrence this does not seem to be necessary. In the case of EXISTS quantifiers, however, a partial execution of the loop may firmly establish the validity of the assertion. Thus exiting the loop may substantially aid in the efficiency of the instrumented program. Since functional side effects (as pertaining to the program's execution) must not occur as the result of checking assertions, that is not a concern here.

3.6.6.3 KEEP Information.

```
const

    max_vartrace = 10 ; (* maximum number of variables which may be
        specified for tracing in a single KEEP statement*)
    max_proctrace = 10 ; (* maximum number of procedures which may be
        specified for tracing in a single KEEP statement*)
    max_num_svalues = 1 ; (* maximum number of svalues which may be
        specified in a single KEEP statement*)
    max_num_functions = 1 ;(* maximum number of functions (to be applied
        globally) which may be specified in a single KEEP statement*)
    max_num_parameters = 10 ;(* maximum number of acutal parameters which
        may be specified in a function which will be called globally *)

type

    stmt_types = (assign, call, cancel, docase, doloop, exit, file, goto, if,
        onerror, offerror, read, reset, return, schedule, senderror, set, signal,
        terminate, update, wait, write); (* all the various statement types
        which the user may specify to be COUNTed*)
    funcdescriptor = (* description of a function call, with actual parameters*)
        record
            funcname: integer; (* pointer into the symbol table to the
                function name being called*)
            funcparams: array (1..max_num_parameters) of integer;
```

```
                (* parameters to the function called (pointers into the
                symbol table) (This could be extended to allow expressions
                as parameters) *)
        end ; (*funcdescriptor record*)
initial_keep =
        record
            SMRK_pointer: integer ; (* the SMRK of the KEEP statement*)
            special_label: dewey_decimal ; (* the special label on the KEEP
                statement (all zeroes if none) *)
            case global: boolean (* whether or not the GLOBAL keyword is
                present*) of
            true:
                (num_functions: small_int; (* the number of functions listed
                    to be applied throughout the range*)
                 countlist: set of stmt_types; (* which statement types
                    should be COUNTed*)
                 vartrace: array (1..max_vartrace) of integer; (* integer
                    indices into the symbol of variables to be traced*)
                 proctrace: array (1..max_proctrace) of integer; (*integer
                    indices into the symbol table of procedure invoca-
                    tions to trace*)
                 funcs: array (1..max_num_functions) of funcdescriptor; (*
                    descriptions of functions and parameters which are
                    applied globally *)
                );
            false:
                (num_svalues: small_int; (* number of svalues to keep
                    here*)
                 COUNT: boolean ; (* whether or not to keep count of this
                    statements executions*)
                 PATH: integer ; (*keep path for specified number of
                    statements*)
```

```
            SNAPSHOT: boolean ; (*whether or not to take a snapshot
                here*)
            exps: array (1..max_num_svalues) of expression ; (* svalue
                expressions to keep (the HALMAT representation of
                the expressions*)
            qualifier: boolean ; (* whether or not a qualifier is
                present*)
            comparison: expression; (* the qualifier specified*)
        );
end ;(*end initial keep record definition *)
```

3.6.6.4 KEEP Instrumentation. - The instrumentation to be derived from this information is as follows.

1. Execution frequency counts. Execution frequency counts ("histograms") are required if, for any KEEP, global = true, or global = false and COUNT = true.

The latter case is simplest so it shall be examined first. Whenever the statement is executed a global counter of execution frequency for the statement must be updated. The instrumentation processor is responsible for the creation of this global counter. Since several counters will likely be needed due to KEEPs throughout the program an array of counters is appropriate. The instrumentation processor (node CBCD) may create this array by modifying the symbol table to include the array. The array should have global scope. Its size may be determined dynamically: as the instruments are generated a count is kept indicating how large the array must be. When the instrumentation processor is complete the array is added to the symbol table and all references to it may be established. (A back-chain of references may be appropriate to facilitate this.) Alternatively, a "fixed" array size may be determined before instrumentation, by input to the instrumentation processor. References to the array may thus be established at time of instrument creation.

If execution frequency counts are not kept "in-core" by means of an array, a channel may be employed to write out the count information for later retrieval and processing. Each time an instrumented statement is executed, the instrument will write the statement number on the count channel. These numbers may be later tabulated to obtain the frequency counts.

A second table is required in addition to that containing the execution counts. This table indicates which entry in the table corresponds to which KEEP COUNT statement. This is required so that upon post-processing the execution count appropriate to the statement may be displayed along side the statement.

Recall the discussion of inline instruments versus procedure calls. (Section 3.6.1) It is in instrumentation of execution counts that inline code shows the most superiority. The scheme outlined here is appropriate to either technique, however. In the case of using a procedure to perform the array update the array index is passed as a parameter, as well as the statement number.

Insertion of instruments throughout a region of code (global parameter = true) is accomplished in only a slightly more complex manner. Each individual instrument is as described above; the placement of the instruments is the only additional work required. The most sophisticated form of instrumentation required in this vein is if a count is desired for ALL statements in a given range.

The important principle to be observed here is that not every statement requires an instrument. Simple minded schemes which, in effect, instrument every statement suffer from unreasonable overhead. Such a scheme (which is definitely not recommended) could be implemented by using the system monitor to perform the count updates, and requiring that the monitor be called after execution of each statement. The small amount of extra work required to perform

intelligent instrumentation is well worth it. (Additionally, the simple minded scheme referred to would not support selective instrumentation as required by the KEEP language.)

Only so-called basic blocks of code require an instrument. A basic block of code is a sequence of statements through which control must always pass sequentially. Simple analysis of the program reveals those places where instruments are required. Simple heuristics may be applied. The mapping required by the post-process phase is analogous to that described above. It must be able to determine which counter applies to the current statement.

In summary, the following data structures are required.

```
const

    max_num_counters = 500 ; (* the maximum number of actual counters which
        are required to instrument a given program*)

type

    counter_table = array (1..max_num_counters) of integer ; (*the table where
        the execution frequency counts will be kept*)
    counter-stmt_number_table = array (1..max_num_counters) o
        record
            first: integer; (*first SMRK to which the counter applies*)
            last: integer; (*last SMRK to which the counter applies*)
            index: integer; (*index to the relevent counter in the
                counter_table*)
        end;
```

2. Variable tracing. All non-zero entries in array vartrace represent variables whose evolution is to be traced throughout execution of a given region of code. The code required is, again, simple. Whenever the variables listed occur in assignment contexts HALMAT must be inserted following the assignment (or return from a procedure or function call) to write the updated value of the variable on a specified channel. The channel number may be specified by appropriate input commands (see section 3.6.7). The statement number of the statement which caused the change in value should also be written, in addition to the name of the variable and its updated value. If the current statement execution count is available, it should also be printed.

3. Procedure tracing. Procedure tracing is analogous to variable tracing. Whenever a procedure pointed to from array proctrace is referenced within the

range of the keep, a message is written indicating: 1) the statement number where the call is being made, 2) the name of the procedure being called, (optionally) 3) the value of the parameters passed, and 4) the current execution count, if available. Channel control is available, as in the case of variable tracing. Implementation restrictions on the type of parameters which may be written out may be appropriate.

4. Snapshot generation. The instrumentation required here is a call to the system monitor, requesting it to dump a copy of the contents of its processing queues. This, of course, will require implementation dependent modifications to the system monitor. The form of the call to the monitor may be implementation dependent as well.

5. Globally applied functions. The instrumentation required here is simply that which is used to invoke a function and preserve its value as described in 7) below. The function to be called, or its template, must be compiled with the program whose KEEP statements reference the function. The function's value will be saved after execution of every statement in the scope of the keep.

6. Path tracing. Instrumentation to be determined.

7. svalue expressions. Local keeps may specify expressions to be kept during execution. The code required to implement this is 1) that which is required to compute the expression and 2) that which creates a "special variable" and stores the result of the expression computation in it. This special variable is like any other HAL/S variable, with the exception that it may only be referenced indirectly--either through its implicit creation at a KEEP statement or through a special value reference in a KEEP or ASSERT statement. Several such variables are likely to be needed throughout the program, but unlike the counter variables used to support the keeping of execution frequency counts, these variables may all be of different type. Each of these variables could be allocated separately (i.e., added to the symbol table separately). To support references to the kept expressions an internal table must be kept relating the KEEP to the special variable. This simple scheme would be alright, except that the user may wish to see the final values kept displayed on an annotated source listing, in the manner that execution counts are kept. Such ability requires that the final value kept be written on an external channel to be picked up by the listing generator (as the counts are). The counts are all easily written out since two tables contain all the necessary information. It is therefore recommended that a table be formed for the kept variables, indicating their names. When program termination is reached this table will be used inside a loop, where the body of the loop prints the special variables. (The matter could be simplified by restricting the type of the expressions which may be displayed on the listing to, say, integer and scalar. If this is done, then the technique applied to execution counts may be used directly.)

8. Qualified keeps. If expressions are qualified, the portion of the KEEP which computes and saves the expression values must be preceded by an IF check on the comparison listed in the KEEP. Phrased in terms of HAL/S:

```
if comparison then
do;
    compute expression;
    store expression value in special variable;
end;
```

As alluded to during the discussion of execution frequency counts and kept expressions, several tables need to be written out after the user's program has finished. These tables are used as input to the annotated listing generator, allowing the execution counts and final kept values to be displayed next to the appropriate statement. The code to write this information out should be placed just before every point in the program where execution may terminate. Thus even though only a portion of the program may be instrumented, each termination point requires this code. Fortunately, in HAL/S such places are few in number. A program may only terminate two ways: an error condition may not have a recovery, or the program **close** may be reached. Instrumentation of the latter case is easy. Location of the former cases is implementation dependent, as the handling of all error conditions is implementation dependent. When the set of such errors is determined, their error handler may be modified to include the code to write the tables before terminating abnormally.

3.6.7 Control Of Instrumentation - The use of special labels on the assertion and keep statements allows their instrumentation to be controlled externally. The mechanism which accomplishes this is the input to SAMM node CBCD, which, among other things, merges the HALMAT Monitor File (HMF) and the HALMAT file. Following is a description of the input to CBCD relevant to this task. The syntax is described first, in the same notation as employed for the assertion and keep statements. This specification is provided down to the separator level, as this input is not defined within the context of the HAL/S language (as the assert and keep statements were). The semantics follows, along with an indication of how the input is processed and used.

Syntax

controlling input::= separator* KEEP indicator keep options separator* terminator

separator* (ASSERT) indicator assert options separator* terminator

separator ::= ,
space
:

terminator ::= ;

indicator ::= separator* (PREFIX) separator+ dewey decimal with stars separator+

dewey decimal with stars ::= number thing (. number thing)* (.)

number thing ::= integer
*

keep options ::= instrument generation control
instrument type control
channel control
parameter control

assert options ::= separator* assert option (separator* , separator* assert option)*

assert option ::= termination control
instrument generation control
instrument type control
message control
channel control

termination control ::= TERMINATE
NOTERMINATE

instrument generation control ::= ON
OFF

instrument type control ::= INLINE (separator+ PROCS) (separator + simulation time control)

SPECIAL
EXTERNAL

simulation time control ::= TIME
NOTIME

message control ::= MESSAGE
NOMESSAGE

channel control ::= CHANNEL separator* (separator* integer separator*)

parameter control ::= PARAMETERS
NOPARAMETERS

integer ::= digit (digit)*

digit ::= 0
1
2
3
4
5
6
7
8
9

Semantics

The most important aspect of this input scheme is that it allows very flexible selection of the various assert and keep statements to which the controlling parameters apply. The facility is perhaps best explained through the use of examples. If indicator is 1.2 then the controlling parameters listed apply to the statement with the special label 1.2. If the indicator is PREFIX 1.2 then the controlling parameters apply to all statements whose special labels begin with 1.2. Thus 1.2, 1.2.0, 1.2.1, are all implied. Using a star (*) within a dewey decimal is a shorthand notation for any digit. Thus 1.2.* is equivalent to listing 1.2.0, 1.2.1, ..., 1.2.9. Similarly 1.*.4 is equivalent to the list 1.0.4, 1.1.4, ... , 1.9.4. The star notation and the prefix notation may be combined. Since it may be desirable to specify parameters for a class of special labels, yet invoke special parameters for a subset of that class, a hierarchy of precedence is necessary. PREFIX notation has lowest precedence, star notation has next higher precedence, and explicit notation (no stars or PREFIX) has highest precedence.

Separate control is provided for the instrumentation of assertions and keeps. Control of KEEP instrumentation is simpler. The KEEP keyword must be present, and the following options are provided: instrument generation control, instrument type control, channel control, and parameter control. Regarding instrument generation control, selection of ON causes the keep to be processed. Selection of OFF causes the applicable keep statement(s) to be ignored in further processing. instrument type control is as described below. channel control directs output messages from variable and procedure tracing to the desired channel. parameter control specifies whether, during procedure tracing, the values of the parameters of the procedures being traced should be written.

simulation time control determines whether or not the execution of the inline instrumentation will affect the simulated real time clock. TIME will cause the clock to be affected.

If the KEEP keyword is not present it is assumed that the controlling input statement refers to ASSERT statements. The ASSERT keyword makes this selection explicit. Several options pertain to the processing of assertions.

instrument generation control determines how the selected asserts are to be processed. If ON is selected further action is subordinate to instrument type control. If OFF is selected the assertion is not in any way inserted into the code, nor will it appear in the list of instruments to be placed in the code by an external means.

instrument type control determines how the instruments which are to be included are processed. INLINE requires that the algorithm which implements the assertion or keep be translated to HALMAT and inserted at the proper point into the HALMAT Monitor File. If the PROCS parameter is present the inline instruments will be procedure calls where appropriate, as opposed to straight-line code. Recall the discussion of Sections 3.6.1 and 3.6.6.4. EXTERNAL requires that the algorithm be printed out on a special listing so that the user can "insert" it at a later point. Most often this will be used to direct the processing of assertions or keeps which can be checked through the use of an instrumentable interpretive computer simulator (ICS). SPECIAL requires that the algorithm (or an indication of the algorithm) be inserted in the HALMAT through use of special HALMAT paragraphs. Proper interpretation and processing of these special HALMAT statements is left to the code generators/interpreters which follow in the processing sequence.

termination control allows the user to determine what happens when assertions are violated. If TERMINATE is selected then the users program will be forced to terminate (abnormally) at the point of assertion violation. Control will return to the execution (or system) monitor at that point, if such a monitor is present. Semantics of TERMINATE on small, dedicated computers operating without a run-time monitor is implementation dependent.

message control determines whether or not an informative message is generated upon assertion violation. MESSAGE indicates that one is to be generated, NOMESSAGE implies the converse. Such selection ability is desirable in the event that the user utilizes his own assertion violation handler (such as is done in example 18, in Section 3.6.5, entitled "Sample Usages of the Assertion and Statistics Gathering Facility)." If NOMESSAGE is selected only message generation is inhibited. Special value VIOLATE associated with the assertion is incremented as usual. (If other tasks are associated with assertion violation they would be performed as well.)

channel control allows user selection of the channel number to be used when writing out assertion violation messages.

As may be expected default values are available. They are as follows:

KEEP

ON

CHANNEL (6)

NOPARAMETERS

ASSERT

ON

INLINE

NOTERMINATE

MESSAGE

CHANNEL (6)

Implementation

As the above described input to node CBCD is processed a tree structure may be built which contains the parameters specified. The hierarchy of defaults, PREFIX, star notation, and explicit directives may thus be maintained. When the HMF is processed the parameter tree may be searched according to the special label on the statements. Separate trees could be kept for keeps and asserts, but better would be a single tree with dual annotations, one set for assertions, the other for keeps.

Control of Error Monitors

Instruments may exist in the HALMAT Monitor File which do not correspond to a labelled assertion or keep statement. These instruments are created by separate tools, and check for various errors, the most notable of which are subscripts out of bounds and division by zero. The user may wish to selectively control the insertion of these instruments as well as the assertion and keep instruments. In order to enable this a "pre-pass" mode should be supplied on the file merger tool. This pass will simply print a numbered list of all the instruments which exist in the HMF, along with a pointer to the HAL/S statement to which they apply. The user may examine this list and note the numbers of instruments which should, or should not, be inserted. These numbers are then supplied to the merge processor, the HMF is "rewound" and the merge operation takes place in the usual manner. The INLINE and SPECIAL modes may be desired as well. We therefore include the following in our control syntax:

controlling input::=

separator* INCLUDEONLY separator+ instrument type control separator+number list

separator* EXCLUDE separator+ number list

number list ::= separator* integer (separator* , separator* integer)*

3.6.8 Unresolved Design Issues - The instrumentation required to handle path tracing and path assertions has not yet been determined. The problem essentially involves designing a mechanism which can efficiently maintain the path history in such a way as to minimize the amount of data stored (an execution path can be quite long) and still be able to check path assertions without a great deal of overhead. As is the case with all of the dynamic analysis instrumentation, the resource constraints of various target computers (e.g., limited memory size and I/O channel availability for the execution statistics) may influence the implementation decisions.

3.7 Documentation. - Virtually all of the tools presented in the design have, as part of their duties, the production of different aspects of documentation. The most obvious facilities in this area are the non-data flow static analyzers (cross reference generator, call graph, code auditor, etc.), the comment extractor, and the assertion facilities. The system data base is the common repository of all documentation produced. A powerful user interface to it allows such documentation, generated by diverse tools, to be accessed in an efficient manner.

In particular, it is recommended that a output writer program be created which will access and present the items in the data base. Such an output writer should have a comprehensive scope, generating everything from the listing which is currently produced by the compiler to the annotated source listing which contains the execution frequency counts and KEEP values. There are two motivations for taking this approach. First, the listing generated by the LaRC front-end compiler is judged unacceptable in its present form. Multiline output (subscripts and superscripts) is not supported, error messages are nigh unto useless, and the type notation for each variable is not provided as is done in the Intermetrics 360 compiler. This is not to say that the solution is to copy the Intermetrics version. Its output is deemed counterproductive in the way that inline comments are handled. The only reasonable algorithm for printing such comments is to transfer them directly to the output, without massaging them in any way. Users often line up comments for special annotation purposes; the 360 compiler moves comments around in such a way as to often obliterate any legibility that may be present in the original text. The problem is greatly compounded when ASSERT and KEEP statements are included in the text. They are inserted in the text as null statements. The 360 compiler moves the special statement to the right of the listing. Long special statements are rearranged.

The second motivation for providing the output writer is that many tools contained in this design add information to "the compiler listing." This includes additional cross reference maps, error messages generated by the data flow analyzer, execution frequency counts, standards violations, and units conversions applied. Each of these tools could generate its own separate report, or post-process an existing listing, adding new information to it. Both these approaches have disadvantages. Most obviously is that the number of listings or documents produced will be approximately equal to the number of tools executed. Inefficiencies will abound.

To alleviate the inadequacies of the compiler and avoid the inefficiencies associated with each tool generating its own document the output writer is proposed. The basic operating scheme would be as follows. Run as many or as few analysis tools as are desired. Each tool deposits its "raw" information in the system database. When all the tools have completed, the output writer is called. Parameters to it will control which items in the database will appear on the listing. One source listing then is generated, followed by cross-reference maps, or whatever else has been selected. All tables, maps, errors, and so forth will be keyed to the (one) pretty-printed source listing. The following list indicates which data items are possible inputs (and outputs) of the writer tool.

SAMM Diagram Where Input Item Generated	Sub-Activity Designation	Data Number	Description
CBA	A	2	Source Code
CBCAA	A	3	HALMAT
		6	Compilation Error Messages
		6	ASSERT/KEEP Errors
CBCCA	A	3	Units/Scale Messages
		5	Conversion factors Applied
	D	13	Programming Standards Violations
CBCCAB	A	2	Lock Group Membership Map
		3	Event Variable Map
	B	5	Shared Data Map
CBCCAE	B	3	Loop Condition Alteration Notation
	C	4	Type Coercions Performed
	A	7	Recursion, Unused Procedure Messages
	D	8	Miscellaneous Error Messages
CBCCAF	A	2	Reentrancy Notation
	B	3	Routine Dependency Notation
	C	4	Dependency/Termination Effects Messages
CBCCB	D	7	Data Flow Analysis Error Messages
CCD	B,C	8	Raw Execution Frequency Counts, KEEP Values
Root (Document Existing System)	B	3	Internal Program Documentation

Processing dependencies (which tools have to be utilized in order to generate the above listed information) may be determined from the SAMM diagrams. When the tool is written and documented these dependencies should be easily ascertained by the user. For instance, in order to extract internal documentation no tool need be

executed, only the source code need be available. On the other hand in order to display execution frequency counts for a run (or a series of runs) the program must be compiled, instrumented, and executed.

A final note regarding this tool is in order here. The tool, as alluded to above and in the discussion on execution frequency counts, must be able to display the counts from several separate runs. Thus input parameters must provide the tool with the tables containing the results of the runs.

Though not strictly in the realm of providing documentation tools, a comment is in order concerning the quality of the HAL/S language documentation. During the course of this design frequent reference was made to the language manuals, and a sophisticated prototype tool was developed using HAL/S as the implementation language. This experience has shown that the manuals utilized (HAL/S Language Specification Version IR-61-8 (June 16, 1976); HAL/S Programmer's Guide Version IR-63-4 (June 11, 1976); HAL/S 360 User's Manual Version IR-58-15 (June 15, 1977)) were unsuitable as reference manuals for the language. The manual organization does not lend itself to answering questions concerning the language. A complete, concise description of the syntax is not even available, let alone a description of the semantics. (The "Working Grammar" of Appendix G, HAL/S Language Specification, is incomplete and in an apparently random order). It is recommended that a small reference guide be created, with a target audience of experienced HAL/S programmers. An appropriate model for such a document would be the Pascal Report.

3.8 Error Class/Detection Technique Chart. - Table I contains a chart having on the vertical axis a list of errors commonly occuring during the development of large software systems. The horizontal axis contains a list of automated tools useful in the detection of such errors. At the intersection of each error and tool, an indication is provided as to how well the tool is suited to detecting the particular error. An empty intersection indicates the tool is not likely to directly aid in the detection of the particular error. Along each row of the chart (which corresponds to a single error) the tools which are appropriate for the error detection are ranked as to their ability. One tool is often more powerful (in a loose sense) than others, and will detect a higher percentage of the particular error in a given system.

This chart is useful for several purposes.

1. It is a guide to choosing the best strategy for detecting a particular class of errors.

2. It is a guide to choosing an implementation strategy. By scanning the columns of the chart, each tool can be examined as to how many error classes it is suitable for detecting. If the errors are weighted as to importance, and the efficacy of the tool is taken into account, an assessment of the "value" of the tool may be made. This value may be used in determining which tools are the most important to implement.

3. The chart gives an indication as to which errors are particularly difficult to detect. For some errors very few tools are appropriate, and those tools which are appropriate may not be very effective. Areas for further research in the development of tools are therefore highlighted.

Though these utilities are not to be overly deprecated, several considerations must be kept in mind when using the chart.

1. The error classification scheme used on the vertical axis is not universally accepted, nor does it necessarily reflect the major categories which exist on any given project. The scheme used is based mostly on a study performed by TRW for RADC (reference 15) [Thayer, et. al, 1976]. It is the culmination of examination of five large software development projects in the DOD environment. As such it probably is relevant to flight software projects, though there are clearly several exceptions. The classifications have been modified slightly to reflect the additional characteristics of flight software.

2. The list of tools which are rated is not exhaustive. A single tool may also require several programs for an implementation. Good and bad implementations exist for each tool as well. It is assumed here that all the implementations are "good" ones.

3. Any tool acting in a stand-alone capacity is not nearly as effective as a tool embedded in a verification environment. The power of an environment is greater than the "sum" of the powers of the components, due to the effect of working together. The chart attempts to rate the tools largely independently.

4. The ratings given in the chart are very subjective. In addition, some of the tools described have never been implemented in anything more than prototype form (e.g., design simulation). The ratings therefore represent educated estimates, considering both confirmed results from existing tools, and aniticipated results from planned tools.

5. Several of the tools require intelligent use, and such use is assumed in the ratings. As an example, program assertions are potentially very powerful, but the programmer must employ much thought and care when creating them in order to realize their benefit.

6. The chart does not provide an effective guide to the use of tools during program development. Specifically, detection of errors during requirements analysis is substantially more cost effective than detecting them during design. Detection of errors during design is substantially more cost effective than detecting them during coding. The same is true when comparing coding to the traditional concept of testing. Thus using "effective" tools at coding time is no substitute for proper analysis of requirements or design. Lastly, the class a particular error falls into is not normally known until after it has been detected.

The names of the tools given along the horizontal axis are intended to be descriptive of the tools' functions. Those which are perhaps unclear are described here. "Standards checker" is equivalent to SAMM node CBCCAD, checking for adherence to programming (coding) conventions. "Termination conditions" is equivalent to node CBCCAEL, checking that the body of all loops alter at least one variable involved in the computations of the termination condition associated with that loop. "Coercion analysis" (CBCCAEC) indicates all type coercions performed. "Query system" (CBAC) is the FAST-like source program question-and-answer system. "Monitors" dynamically check for violations of the "rules" of the programming language--division by zero, for example, or subscripts out of range. "Test coverage analysis" is examination of the statement execution frequency counts obtained from one or more program executions. "Performance analysis" examines the real time behavior of the system in its embedded environment or in a simulation environment.

		Requirements and Design Analysis Tools				Static Analysis Tools										Symbolic Execution	Dynamic Analysis Tools				
		Requirements	Design	Design to Req. Verifier	Design-Simulation	Code to Design Verifier	Parser & Syntax Checker	Cross-reference Maps	Units/Scale Checking	Standards Checker	Termination Conditions	Coersion Analysis	Data Flow Analysis	Query System	Interface Checker/Call Graphs	Miscellaneous	Symbolic Executor	Monitors	Assertions	Test Coverage Analysis	Performance Analysis	Interactive Debugger
A	**COMPUTATIONAL ERRORS**																				
A-1	Incorrect operand in equation		✓		1	✓		✓	3			✓	2	✓			1		✓		✓
A-2	Incorrect use of parenthesis				✓				2								1		✓		
A-3	Sign convention error				✓	✓											✓		1		
A-4	Units or data conversion error		2			✓			1			3					✓		✓			
A-5	Computation produces an over/under flow				2													1	✓			
A-6	Incorrect/inaccurate equation used			✓	2	✓											1		1			
A-7	Precision loss due to mixed mode		2				✓					1										✓
A-8	Missing computation	2	✓	✓	1	✓											1		2			✓
A-9	Rounding or truncation error								1			2							✓			
B	**LOGIC ERRORS**																				
B-1	Incorrect operand in logical expression		✓	✓	1	✓		✓	3				2				✓		3			
B-2	Logic activites out of sequence		✓	✓	2	✓							1				3			✓		2
B-3	Wrong variable being checked		✓	✓	✓	✓		✓	2		✓		1				3					
B-4	Missing logic or condition tests		✓	✓	1	✓							3				1		2	✓		✓
B-5	Too many/few statements in loop		✓		1	✓					2		3				1		3			✓
B-6	Loop iterated incorrect number of times (including endless loop)			✓	✓	✓	✓				1						1		2	✓		
B-7	Duplicate logic			✓	1	✓							2	2			1					
C	**DATA INPUT ERRORS**																				
C-1	Invalid input read from correct data file																	✓	✓			
C-2	Input read from incorrect data file	✓		✓		✓								✓								
C-3	Incorrect input format																	✓				
C-4	Incorrect format statement referenced			✓		✓	✓	✓												✓		
C-5	End of file encountered prematurely																	✓	✓			
C-6	End of file missing																	✓	✓			

Table 1 Error Class/Detection Technique Chart

		Requirements and Design Analysis Tools				Static Analysis Tools											Symbolic Execution	Dynamic Analysis Tools				
		Requirements	Design	Design to Req. Verifier	Design Simulation	Code to Design Verifier	Parser & Syntax Checker	Cross-reference Maps	Units/Scale Checking	Standards Checker	Termination Conditions	Coersion Analysis	Data Flow Analysis	Query System	Interface Checker/Call Graphs	Miscellaneous	Symbolic Executor	Monitors	Assertions	Test Coverage Analysis	Performance Analysis	Interactive Debugger
D	**DATA HANDLING ERRORS**																					
D-0	Data file not rewound before reading		2										1				✓		✓			2
D-1	Data initialization not done	✓	2		✓		2	✓					1	✓			3	2	✓			✓
D-2	Data initialization done improperly			✓		✓		✓					1				2		1			✓
D-3	Variable used as a flag or index not set properly		✓	✓	✓	✓							3				2		1			2
D-4	Variable referred to by the wrong name	✓	2			✓		✓					1	✓			✓					
D-5	Bit manipulation done incorrectly																		✓			✓
D-6	Incorrect variable type		1			✓	1					2			✓							
D-7	Data packing/unpacking error		1			3						2							✓			✓
D-8	Sort error				2												3		1			✓
D-9	Subscripting error		✓		2	✓							3				2	✓	1			
E	**DATA OUTPUT ERRORS**																					
E-1	Data written on wrong channel	1		✓	2	✓							✓				3					✓
E-2	Data written according to the wrong format	1		✓	2	✓																3
E-3	Data written in wrong format	1		✓	2	✓																3
E-4	Data written with wrong carriage control	✓		✓	1	✓																2
E-5	Incomplete or missing output	1		✓	✓	✓							3				2			✓		✓
E-6	Output field size too small	2		✓	✓	✓												1				
E-7	Line count or page eject problem	2		✓	1	✓																3
E-8	Output garbled or misleading	1		✓	2	✓													3			✓
F	**INTERFACE ERRORS**																					
F-1	Wrong subroutine called		1		✓	✓									1		2		✓	2		✓
F-2	Call to subroutine not made or made in wrong place		✓		1	✓									2		2			3		
F-3	Subroutine arguments not consistent in type, units, order, etc.		1	✓		✓	✓								1				✓			
F-4	Subroutine called is nonexistent	2	2	✓	✓	✓									1							
F-5	Software/data base interface error		2			✓									✓			2	✓			✓
F-6	Software user interface error	✓	1	✓	2	✓											3		✓			✓
F-7	Software/software interface error		2		3	✓									1				1			✓

Table 1 Error Class/Detection Technique Chart (Continued)

		Requirements and Design Analysis Tools				Static Analysis Tools										Symbolic Execution	Dynamic Analysis Tools				
		Requirements	Design	Design to Req. Verifier	Design-Simulation	Code to Design Verifier	Parser & Syntax Checker	Cross-reference Maps	Units/Scale Checking	Standards Checker	Termination Conditions	Coersion Analysis	Data Flow Analysis	Query System	Interface Checker/Call Graphs	Miscellaneous	Symbolic Executor	Monitors	Assertions	Test Coverage Analysis	Performance Analysis	Interactive Debugger
G	**DATA DEFINITION ERRORS**																				
G-1	Data not properly defined/dimensioned			✓	3	✓			2								✓		1		✓
G-2	Data referenced is out of proper range		✓		✓								2				✓	3	1		✓
G-3	Data being referenced at incorrect location			✓	✓	✓	✓	3					1	✓			2				✓
G-4	Data pointers not incremented properly				2						2		✓				1	✓	2		✓
H	**DATA BASE ERRORS**																				
H-1	Data not initialized in data base	1	2		3	✓	✓						2	✓			✓	1			✓
H-2	Data initialized to incorrect value				3									✓			2		1		✓
H-3	Data units are incorrect	2	3	✓					1										✓		✓
I	**OPERATION ERRORS**																				
I-1	Operating system error (vendor supplied)																		✓		
I-2	Hardware error																				
I-3	Operator error																				
I-4	Test execution error																				
I-5	User misunderstanding/error	1			2																
I-6	Configuration control error	1	2			3															
J	**OTHER**																				
J-1	Time limit exceeded				3							1			1					2	
J-2	Core storage limit exceeded				✓							1			1						
J-3	Output line limit exceeded				1															✓	
J-4	Compilation error				✓																
J-5	Code or design inefficient/not necessary				1							✓			✓		3			2	✓
J-6	Design nonresponsive to requirements			1																	
J-7	Software not compatible with project standards									1											

Table I Error Class/Detection Technique Chart (Continued)

	Requirements and Design Analysis Tools				Static Analysis Tools										Symbolic Execution	Dynamic Analysis				
	Requirements	Design	Design to Req. Verifier	Design-Simulation	Code to Design Verifier	Parser & Syntax Checker	Cross-reference Maps	Units/Scale Checking	Standards Checker	Termination Conditions	Coercion Analysis	Data Flow Analysis	Query System	Interface Checker/Call Graph	Miscellaneous	Symbolic Executor	Monitors	Assertions	Test Coverage Analysis	Performance Analysis	Interactive Debugger
K DOCUMENTATION ERRORS																				
K-1 User manual				✓																
K-2 Interface specification	1	2								3										
K-3 Design specification		✓		✓																
K-4 Requirements specification	✓		✓																	
K-5 Test documentation																				
L REAL-TIME (MONITOR INTERACTION) ERRORS																				
L-1 Illegal use of shared variables		1		✓			✓				✓	2		✓						
L-2 Synchronization errors		1		✓							✓	2		✓			✓	3		
L-3 Deadlock		1									✓	2		✓			✓	3		

Table 1 Error Class/Detection Technique Chart (Continued)

3.9 Modifications Required to the NASA-LaRC HAL/S Front End Compiler. - When utilized, units and scale specifications, assert statements, and keep statements all appear embedded in HAL/S source text. They occur within inline comments. As such, the HAL/S front end must be able to recognize them, preprocess them in the manner described in the preceding sections, and write them on the initial HALMAT Monitor File. Of these activities, the most involved activity is the parsing of HAL/S expressions occuring within the assert and keep statements. The expressions must be translated into HALMAT operations, with these operations being written as components of the HMF.

A brief study of the HAL/S front end compiler was conducted to determine the impact of the design decisions associated with the above mentioned statements. The Pascal source of the LRC compiler was utilized, along with Intermetrics document IR-182-2, "HAL/S-FC & HAL/S-360 Compiler System Program Description", dated March 31, 1977. Several items were observed which indicate the nature of the required work.

The current processing of inline comments is quite direct. While a statement is being processed any inline comments encountered are accumulated in the string buffer. A maximum of 256 characters are stored in the buffer, and no overstrikes are allowed. (Thus with the existing system an implementation restriction is imposed: only single line format may be used within assertions and keeps, and the length of the statement is restricted.)

Since the comments are retained in the buffer, they are easily accessible for the preprocessing function. Within each special statement (units, scale, assert, keep) the number of different formats and statement types is fairly limited. Thus no sophisticated recognition scheme is required at the top level. Substantial complexity can be encountered, however, within the expressions occurring in the special statements. Scanning, screening, parsing, and code generation activities are all required at this point. Of course the existing compiler contains routines for doing this, but they are very tightly bound to the overall compiler structure; it appears impossible to use them directly for accomplishing this auxillary task.

On the other hand, the compiler seems very well structured (at least in the light of the complexity of the HAL/S language) and documented. Good formal techniques are employed, and the levels of abstraction seem well chosen. It should be feasible, therefore, to create new routines, not bound to the current compilation task, which perform the necessary operations. These routines will be near copies of the current routines. In some cases the existing routines may be modified in a simple way to perform both tasks. Such a modification may be on the order of checking a flag to determine the file on which a piece of HALMAT should be written.

An interesting item was observed as a result of this study. The listing generated by the LRC compiler is the "primitive" listing available on the 360 compiler. Thus when considering the development of an output writer, it's not so much that one is being thrown away, as it is writing one to start with. This primitive listing generator operates essentially in parallel with the scanning operation. See the discussion of the output writer in section 3.7. (Note that this

problem is related to the quality of error messages produced by the LRC compiler: currently only the most cryptic notation is used.)

Another item noted during the study was that the LRC compiler does not currently build external templates describing procedures, tasks, etc. which may be shared among several compilation units. Such a facility will be required to retain the degree of checking required for such usage.

The overall impression left by investigating the structure of the compiler is that several changes will be required to implement the new features, but that such implementation will not be conceptually difficult. Several changes to the compiler are required independently of those to support the special statements, however, before the compiler can be considered suitable for release to the general user. It is fortunate that relatively little effort has been expended in this direction to date, as it will be advantageous to make all the required alterations together, in a non-conflicting manner.

SECTION 4.0

Verification To Requirements Document

To ensure that a preliminary design satisfies the requirements document, the two must be compared. As specification techniques and automated tools which address this level of specification come of age, such verification will become increasingly automated and precise. For the present however, an informal comparison must suffice and is thus presented below. The comparison is presented by referencing the section numbers of the functional requirements from the requirements document and the applicable nodes from the preliminary design, in conjunction with any discussion. Unless otherwise noted the node names are from the SAMM decomposition of "System Creation." Those requirements which relate directly to the detailed design are not discussed here.

4.1 Verification. - The following paragraphs begin with the requirements document paragraph number. Only the major functional requirements are considered, thus the paragraph numbers are not necessarily consecutive.

4.1.1 All the tools and usage modes will be callable through the ISIS user interface, with the exception of the interactive tools. They will likely require their own user interface. Adequate documentation and HELP messages will be provided, but without being burdensome.

4.1.2 Only node CCB, Interactive Test, is largely designed towards an interactive environment. Batch usable debugging and symbolic execution features will be present, however.

4.2.2.1 Addressed by node D, Integration of Modules into System.

4.3.2.1 The HAL/S environment has been specifically addressed (note the emphasis on the use of HALMAT). No language alterations to HAL/S have been proposed. The assertion, units, and statistics specifications are accomplished through the use of specially processed (and formatted) comments. An enhancement will therefore be required to the HAL/S front end (represented in the diagrams by node CBCAAB).

4.3.2.2 See Section 4.2.5 of this document. Note, however, that substantial success in attacking the aliasing problem may be made through the use of instrumentation. See reference 16 [Huang, 1978].

4.3.3.1 With the use of node CBCAA, the LRC-HAL/S compiler front end, all existing documentation features are retained and will not be duplicated.

4.3.3.2 HALMAT, and augmentation thereof, is used as a primary data object in the design. The bulk of verification activities work from the HALMAT directly. HALMAT has not been altered in any way. See Section 3.3.2 of this document.

4.3.4.2 Used in node C, of the Document Existing System model.

4.3.5 The targeting of HALMAT to a specific object machine is not specified in the preliminary design. With the exception of operating system interfaces (such as files needed in collecting run time statistics) verification activities are generally independent of a particular code generator. The interactive test system can perform arithmetic operations which emulate a number of target machines.

5.1.1 Maps will be produced at node CBCAAA, the compiler, and at node CBCCAB, generate cross-reference maps.

5.1.2 Node CBCCAEC, Annotate Type Coercions.

5.1.3 Node CBAC, answer questions about specified code segments, and node B, extract internal documentation, of the Documentation SAMM model.

5.1.4-6 Nodes CBCCAF, Document Real-Time Aspects, and CBCCAEC, generate cross-reference Maps.

5.1.7 Node CBAC, answer questions about specified code segments.

5.1.8 Node CBCCAFA, check shared routines for reentrancy.

5.2.1-5, 5.2.8 Node CBCC, perform internal verification, with additional requirements for runtime checks.

5.2.6,7,9 Node CBCCAA, check Units/Scale correctness, and node CBCCAEB, check termination conditions.

5.2.10 Nodes CCC, target HALMAT, and DA, check for recompilation requirements.

5.3.1 Node CBCD, instrumentation and levels on local assertions.

5.3.2 Monitors calls are created several places, but they are actually inserted at node CBCD. Some monitors would be required in the run time executive itself, which is not modeled in these SAMM diagrams. The HALMAT monitor file contains the set of monitor calls.

5.3.3 Node CBCD, instrumentation and levels on local assertions.

5.3.4 Node CBCCAC, generate timing estimate for specified paths.

5.4 Node CCB, Interactive Test.

4.2 Discussion of Investigations.

4.2.1 ISIS. - The relational database capabilities of ISIS referred to in the requirements document were discovered to be nominal, if existent at all. Consequently no assumption has been made in the design concerning such a feature. The multilevel file structure provided by ISIS will satisfy most requirements of the system database. Additional requirements can be met using data structures internal to the file structure.

Examination of ISIS's capabilities to invoke analysis tools was difficult, as little or no documentation was available. Indeed, it was discovered that the design of that capability was not complete, nor was its implementation. One of our original intentions was to create a prototype system, using stubs for the tools, to gain experience with the ISIS environment and evaluate the user-friendliness of the entire system (how convenient the required user interaction would be). This was impossible though, due to the state of implementation and documentation.

Probably the most disturbing discovery about ISIS was that it was designed to invoke batch tools alone. In order to invoke an interactive tool, either the ISIS environment will have to be exited, the tool environment entered, and then back to ISIS, or some other scheme used. Since interactive tools largely provide their own environment, this is not too severe. Simple things, however, like correcting mistyped input, may vary significantly. These are important from a human engineering standpoint. More importantly, the question of data and database manipulation arises. This is important considering the centrality of the system (ISIS) database. A clean interface may prove difficult to achieve, and the uniformity of a single interface will be lost. Once the ISIS implementation is completed, examination will be required to determine all the implications.

4.2.2 FSIM. - FSIM's capabilities were carefully examined and were discovered to be based on a single, simple, technique. In order to regulate concurrent and real time processes the compiler associates with each HAL/S statement an estimate of that statement's execution time. During compilation a call to the run time monitor is inserted after the code corresponding to each statement. The monitor, when called, adds the estimate of the just-executed statement to its current simulated clock time. That clock forms the basis for scheduling processes and all other activities associated with real time events. Though the main purpose of the clock is in real time control, clearly an estimate of the total execution time for another target machine is available by a suitable scaling of the estimates. Before any execution is performed an estimate of the execution time of any specified path could be formed by simply adding the estimates associated with the statements along that path. Such a capability is included in the design presented. Similarly, performance characteristics of a program in various run time environments may be obtained by simply changing the set of monitor routines; no alteration to the target program is required once the monitor calls have been inserted.

4.2.3 HALSTAT. - Several experiments were performed using Intermetric's HALSTAT tool. No surprising capabilities were observed. The tool seems strangely conceived as it provided both high and low level information side by side, viz. a code audit function listing the frequency of occurrence of each type of HAL/S statement along with a load map. Many of the features provided are specific to IBM architecture. The preliminary design attached contains the same functions, but separated into several tools and made available only under appropriate user selected environments.

4.2.4 FAST. - Due to a series of misunderstandings and complications, the University of Texas FAST ststem was evaluated only through reading the available literature (references 17 and 18) [Johnson, 1977] [Browne and Johnson, 1978]. Many of the basic capabilities of FAST are recognized as valuable and are contained in the preliminary design. Specifically, the ability to make language-oriented queries about a given program seems quite useful. Queries can be made, for example, about all the reference occurrences of a particular identifier. These types of queries are frequent while modifying existing pieces of code. SAMM node CBAC is the tool which performs these actions, and is designed to act in a role supportive of modification activities. The query-type abilities of FAST are considered the basic specifications of this tool. This tool is regarded as having a low implementation priority, and more detailed specifications for it may wait until such time as they are considered important.

One significant finding from the investigations conducted is that the analysis capabilities of the current implementation of FAST are not very impressive. FAST does not even attempt to detect initialization errors on an interprocedural basis because the current algorithm would be prohibitively slow. Indeed, intraprocedural detection of this error in noted in the documentation [Johnson, 1977] as being very inefficient. This finding strengthens our conviction that the functional capabilities of tools must be carefully chosen. Implementing sophisticated analysis tasks with inappropriate algorithms is foolish. (Extravagant claims about the implementation ease of particular tools must also be examined.)

4.2.5 HAL/S Problem Features. - Several features of HAL/S have been identified as presenting difficulties for the analysis tools which have emerged during the design process. These features are described below. It is important to note that only those features which present problems to the designed tools are presented, not features which may, for example, present difficulties to a particular coding methodology. Further, the designed tools operate on an intermediate representation (HALMAT) of the source programs. Therefore, problems which are strictly syntactical are precluded outright. If more analysis tools are designed later on, additional problem-causing constructs may be identified.

1. Real time, concurrent processing statements. These constructs pose a whole new class of problems for existing analysis techniques. Static analysis, symbolic execution, and dynamic analysis are all affected. The problems are by no means unsolvable, however. They simply require that existing techniques be

extended. Such extensions have begun as work supporting this design effort. In particular, significant extensions to static analysis techniques have emerged.

Within this general classification, the TERMINATE statement presents the greatest difficulty. Its use will significantly hinder analysis activities. It is recommended that the use of the statement be highly restricted, if not prohibited.

Cyclic scheduling of processes also presents some difficulties. Our research activities have temporarily ignored this feature until problems with the basic facilities have been resolved. We do not recommend this feature be deleted, however, as it appears quite useful. Rather, it should be noted as inhibiting analysis activities, until further research expands the capabilities of the tools.

2. Aliasing. Aliasing is the referencing of a single object by more than one name. Aliasing can hinder static data flow analysis under certain circumstances. For example, if an arrayed variable is indexed with a value which has been read in, analysis is hindered (reference 4) [Fosdick and Osterweil, 1976]. The forms of aliasing in HAL/S which present the greatest difficulties to static analysis are the NAME feature and global variables. The situation with global variables is similar to FORTRAN COMMON blocks. As such, this problem is well understood. FORTRAN does not have any analogue to the NAME feature, however, and it therefore represents a new difficulty which requires additional investigation. As Huang, (reference 16) indicates, aliasing presents little problem for dynamic analysis, so the complementary use of techniques seems an appropriate resolution of the problem.

3. Side effects. HAL/S functions may cause side effects when evaluated. Since functions may be evaluated as the result of processing ASSERT and KEEP statements, and since such statements must not cause any side effects, restrictions on function composition will be required in these contexts. Enforcement of these restrictions will require additional analysis.

Though this difficulty only exists as a result of the assertion language designed, it is regarded as a fundamental problem with HAL/S which should be evaluated in other lights. A general prohibition of side effects may be a wise rule.

4.2.6 RNF. - The University of Illinois text processor -RNF- was used during the period of the contract to produce interim reports and documents. This experience allowed a close look at its features. For our purposes, RNF was useful for producing medium size documents (40 pages or so). Larger documents would seem to be best handled by breaking them into smaller sections and processing each section separately (thus reducing CPU time through a course of several edits).

The documentation and command set are reasonable but not extravagant. Several errors exist in the implementation, however, and processing speed is not blinding. Remedies in these areas and extensions (e.g., superscripts and sub-

scripts) are in progress at the University of Illinois. When the enhanced version is distributed it should prove a very useful tool for maintaining and producing readily available documentation. The importance of providing documentation tools such as RNF must be stressed: timely, up-to-the-minute information is critical during software development, use, and modification.

4.2.7 Interpretive Computer Simulator. - It is envisioned that interpretive computer simulators (ICSs) will play a major role in the program development-test cycle as applied to flight software. The first time the actual target-object code is used is often in an ICS. The majority of the verification tools present in this design are independent of code generation and thus independent of ICSs. (This is desirable because target machine independence is supported. ICSs are typically standalone systems, each with its own vagaries.) The crucial verification/testing activity which is dependent to some extent on the target machine, however, is the use of instrumentation. Instruments, existing initially on the Halmat Monitor File (HMF), represent assertions, keeps, and error monitors.

Often, ICSs are instrumentable themselves, in the sense that a very limited amount of checking can be performed by the interpreter. Such checking is external to the program being interpreted. Tag fields are often used to indicate what checks are associated with given statements in the code.

In order to provide the greatest amount of flexibility in the use of ICSs the design presented allows instruments to be handled in several different ways. Instruments on the monitor file may optionally be translated directly into in-line code, be listed such that the user can translate the required instrument into an ICS monitor, be completely ignored, or marked as special in-line code. A description of how this selection is performed is contained in Section 3.6.7. The fact that the user must translate a HMF monitor into an ICS monitor is due to wide differences between ICSs. Capabilities, as well as formats, differ significantly. If a more uniform posture was assumed by ICS systems, such translation could be at least partially automated. Provision for this is made in the design by allowing for special in-line code. Such code could be picked up by the code generator and translated into the appropriate ICS commands.

Examination of three ICS systems (MCP-701, C-4000, NSSC-II) revealed that the above approach was best, as their capabilities were so diverse or primitive as to make identification of "special" instrument classes fruitless at this point.

SECTION 5.0

Conclusion

BCS believes the design presented adequately satisfies the requirements of the MUST environment. The design presented takes cognizance of problems associated with software production through its entire lifecycle. It is sufficiently flexible and well designed so that as additional capabilities are added, such as those supporting the automation and formalization of requirements and design activities, their integration may proceed smoothly. Careful choice of such tools should be made, however, to ensure that the progression from one phase to the next may be made naturally, with the ability to directly trace all design decisions between phases.

The proposed programming environment, when implemented, will provide features substantially more powerful than those found in almost any existing software development environment. Utilization of the tools will be natural, will increase productivity, improve software quality, and lower costs.

5.1 Listing of Programs and Implementation Recommendations. - The SAMM model of the system development and documentation processes contain many nodes which correspond to program units. Some of these nodes are decomposed below the program level (in the SAMM model) to indicate their internal structure. Below is a list of all the programs identified, followed by a brief description (usually just the title of the SAMM node). Listing the programs separately does not imply that all the tools must be invoked separately: many of the tools can be grouped and would be invoked as a system (such as those listed under the heading of "non-data flow static analysis"). Specifying programs allows an indication of implementation options. There are a few tools which are not found as specific nodes in the SAMM models, but which are discussed in the text of this document. They are described as well.

Clearly, some of the tools contained in the design are of greater importance than others. These tool-value relationships should be reflected in the order in which the tools are implemented. Our primary conviction is that implementation of the static and dynamic analysis tools should proceed immediately. These tools offer best benefit/cost ratio. Experience with prototype systems in this area (DAVE, PET) and studies by Howden, as mentioned earlier, have brought us to this conclusion. Those tools which are more specialized or less powerful should have a lower priority. (Also involved in an implementation would be provision of general support capabilities, such as a manager for the database described in Appendix B.)

A special note here concerns the implementation of the interactive testing system. Previously we had been skeptical about the utility of symbolic execution systems and even more so about interactive debuggers. As discussed in Section 3.5, this was largely due to not having a methodology to guide their use, as well as considering them separate tools. The ITS eliminates these objections, however, and it must be given a strong recommendation for implementation. ITS is of

lower priority than the static and dynamic test tools, though, and this lower priority is reflected in the priority numbers given below. (When considering implementation it should be kept in mind that it would be feasible and possibly desirable to implement the interactive test portion of ITS before implementing the supporting symbolic execution modules.)

In the following list of programs a priority number is attached to those verification tools which operate on HAL/S or HALMAT. Those with priority 1 are deemed most important. Requirements and design oriented tools are not ranked, nor are the utility non-verification tools (such as the compiler or loader).

SAMM Designator	Priority	Description
A		Check internal consistency of requirements specification
BB		Check internal consistency of system design
BC		Verify preliminary design to requirements
CAB		Perform internal verification of module design (if the same notation is used for system level design, this will be the same tool as BB).
CAC		Verify module design to requirements (which is the system design)
CADA		Check module design consistency with other modules
CADC		Simulate design
CBB		Verify module code to design
CBAC	3	Answer questions about specified code segments (a language intelligent text editor)
CBCB	1	Create monitor calls from assertions having regional significance
CBCD/DD	1	Instrument HALMAT for module/system tests
CBCAAA		Perform basic HAL/S to HALMAT translation
CBCAAB	1	Process (translate to monitors) local asser- and keep statements
CBCCB/DCB	1	Perform data flow analysis

SAMM Designator	Priority	Description
The next eleven programs belong in the group "Non data flow static analysis"		
CBCCAA	1	Check correct program use of units and scale
CBCCAB	1	Generate cross reference maps that are not produced by the compiler
CBCCAC	2	Generate pathwise estimate of execution time
CBCCAD	2	Check programming standards adherence/Warn of use of dangerous constructs
CBCCAEA	1	Generate program call graph
CBCCAEB	3	Check that all loops alter their termination conditions
CBCCAEC	1	Annotate listing with all type coercions performed
CBCCAEE	3	Generate program unit complexity measures
CBCCAFA	1	Check shared routines for reentrancy
CBCCAFB	1	Document the processes which are "dependent"
CBCCAFC	2	Check dependent processes for unforeseen effects when terminated
(End of non-data flow static analysis)		
CCB	2	Perform Interactive Testing
CCC/EC		Target HALMAT to executable/simulation code
CCDA		Load and produce load maps
CCDB		Monitor HAL/S execution (System monitor)
CCEBAA	1	Post-process Histogram/History File
DA	2	Check for recompilation requirements and merge modules into a single system

SAMM Designator	Priority	Description
DB	1	Expand calls for system level assertions/keeps
*B	2	Extract internal documentation
*C		Generate flowchart

* - Node belongs to the Document Existing System model

-	2	Test Harness - composed of nodes CCA, CCD, CCEBA, CCEA (Create test data, Execute, Check test coverage, Check output values). This operation may also require a file comparator.
-	2	Data Base monitor (Reports to management on which parts of the data base are empty, which tools have not been run, etc.)
-	1	Output Writer - generates annotated source listings

5.1.1 Interactive Tools - Only two of the code verification and testing tools require user interaction for effective usage. Most important in this category is the interactive test system; the other is the program query system (like FAST). Batch implementations of these tools would cripple their effectiveness. All the other tools may be effectively implemented as batch tools, given the provision that the output generated (such as the annotated source listing) may be conveniently examined from an interactive terminal. Thus it is recommended that all the tools be implemented to operate primarily in batch mode, with the exception of the two tools above. "Primarily in batch mode" is meant to imply that the user should be able to create the input/controlling information to a tool interactively, execute the tool on the input (without any user interaction during execution), and examine the output interactively upon job completion.

Note that no statement has been made concerning the recommended implementation of the requirements and design analysis tools. Only when their design is further developed may such recommendation be made.

INTRODUCTION TO THE SAMM METHODOLOGY

Appendix A

SAMM (reference 19) [Stephens and Tripp, 1978] is a BCS developed formalism whose purpose is to model a system through a layered structure of activities and data flow. A SAMM representation is primarily composed of a tree structure, which describes the context of a diagram in a system, and an activity diagram, describing the activity-data flow relationships of a system. The functional activities of a system are focussed upon, and these activities are hierarchically decomposed, resulting in the tree structure. Data values flow between boxes (called cells or tree nodes) which represent the activities.

SAMM diagrams indicate the tree structure (hierarchical decomposition) through the systematic use of node labels. Each node in the tree is uniquely labeled in such a way that the designation of each node indicates its parent node, as follows. Each individual node in the tree may only be decomposed into a maximum of six subnodes, indicated by the letters A-F. The subnodes of the root node are labeled by single letters (indicating the first level of decomposition). Thus they may be designated A, B, C, D, E, and F. If node A is further decomposed into seven nodes, their designators will be AA, AB, AC, and so on to AF. Two letters indicates the second level of decomposition. The designators of the ancestral nodes of a given activity are thus explicit. For example, node BCAD has as its immediate parent node BCA, whose parent is BC, whose parent is B, whose parent is the Root.

Data items in SAMM diagrams are indicated by a name and number. Only data numbers are used when indicating flow among activity cells; they are correlated to the data names in the Data Table (part 1 of the Activity Data Flow Diagram). Data items transput by an activity are of two categories: "forward"

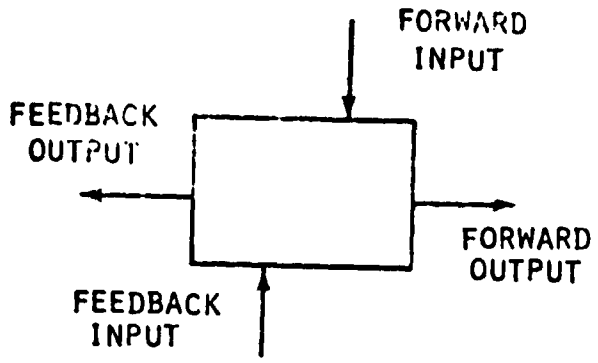


Figure A-1 SAMM Activity Cell With All Possible Inputs and Outputs

and "feedback." Forward output items exit the activity cell from the right, forward input items enter the cell from the top. Feedback output exits from the left, and feedback input enters from the bottom. See Figure A-1. Feedback items allow the modeler to depict data flow loops and mutual dependencies. With data flow paths connecting the nodes of the hierarchical breakdown, a directed graph is formed. Figure A-2 contains a sample SAMM diagram consisting of four activities and eight data items. Items 1 and 7 are external inputs; item 6 is an external output.

The formalism chosen is amenable to automated input, data management, and verification. BCS is currently creating tools to perform such tasks. One such tool is SIGS (SAMM Interactive Graphics System), which allows graphical entry and manipulation of SAMM diagrams. In addition to utilizing the easy entry and automatic checking facilities, the designer may sit at a graphics terminal and experiment with a design, considering several design alternatives. For each alternative the consequences and requirements associated with the changes are easily perceived. SIGS and the SAMM methodology are excellent tools for capturing requirements, and thus represent a potential candidate for inclusion in the MUST environment. Inclusion would substantially aid in the automation of node A, Analyze Requirements, of the attached model of "System Creation."

The SAMM methodology used in the accompanying forms is slightly modified from that described in the reference. As noted above, SAMM focusses on the hierarchical breakdown of activities. A breakdown of data objects is inherent in this as well, but the logical structure of the system data may not conveniently conform to the tree structure. At least it may be difficult to grasp all the data relationships present in the tree structure. Thus a data base model has been developed as well, and is presented in Appendix B. The non-standard notation arises when referencing this database. Feedback input items which appear "out of thin air" denote information being used from the database. Database inputs enter activity cells from the bottom; outputs (which are only entered in the database and do not immediately participate in the model) exit from the right. Such notation is only used where standard SAMM conventions would be awkward or unduly lengthy.

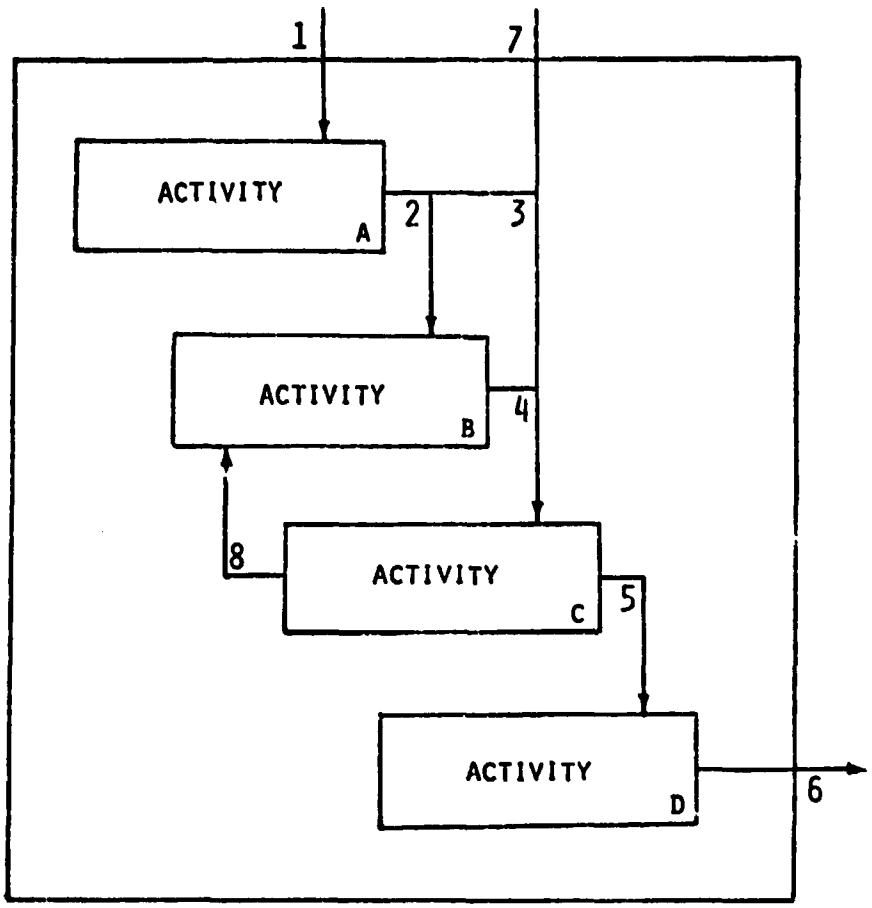


Figure A-2 Sample SAMM Diagram

SYSTEM DATABASE

Appendix B

The database envisioned as associated with the MUST programming environment is described below. The ad hoc notation used is for tutorial purposes, resembling a Pascal type definition. The keywords employed are taken from Pascal and have similar semantics. Comments are enclosed in braces ({}). Words in upper case are either reserved keywords, such as TYPE, RECORD, and END, or reference defined types, such as INTEGER, TEXT, and FOLDER. Types are not necessarily defined before they are used. Words in lower case are identifiers, and are used as field identifiers, type names, and variable names.

```
Examples:
    TYPE {keyword}
        mytype = {type being defined}
            RECORD  keyword
                code: {field identifier}
                        CODE; {type defined elsewhere}
                count: {field identifier}
                        INTEGER; {type defined elsewhere (in this
                                case by the system)}
                END; {keyword - end of definition of mytype}
        yourtype = {type being defined}
                MYTYPE; {type defined elsewhere, namely,
                        right above}
```

The database is described through the definition of type SYSTEM DATA BASE. Not all types referenced in its definition have been fully defined. The reader's intuition is relyed upon to provide an adequate definition for those types which fall in this category. Some types obviously have a fuller definition than others, such as CRITERIA, but to dwell on them would divert attention from the basic goal of this presentation.

Appendix B: System Database

```
TYPE system data base =
   RECORD
      system requirements: REQUIREMENTS; {see expansion below}
         design:
         RECORD
            document: DOCUMENT;
            formal statement:
               RECORD
                  number of modules: INTEGER ;
                  module: ARRAY [1..number of modules] OF
                     RECORD
                        requirements: REQUIREMENTS;
                        design: DESIGN; {see expansion below}
                     END {module record} ;
                  integration: IDAP; {how the modules are held
                                      together and interact.
                                      (overall design)}
               END {formal statement record} ;
            decisions: HISTORY;
            management: FOLDER;
            acceptance criteria: CRITERIA {which pertains to the
                                           design as a whole};
            simulation: SIMULATION {of the whole design} ;
         END {design record} ;
      modules: ARRAY [1..number of modules] OF
         RECORD
            documentation: CODE DESCRIPTION;
            code: CODE; {expanded below}
            test driver: CODE;
            results: ARRAY [1..number of test cases] OF
               RECORD
                  purpose: TEXT;
                  input: INPUT;
                  output: OUTPUT;
               END {results record}
         END {modules record}
      integration: CODE DESCRIPTION {same type of documentation
      as found in the modules record, but here at a
                  higher level and (possibly) with some
                  additional items};
      system test:
         RECORD
            management: FOLDER;
      number of test scenarios: INTEGER ;
      transput: ARRAY [1..number of test scenarios] OF
         RECORD
            purpose: TEXT;
            input: INPUT;
            output: OUTPUT;
```

```
                number of configurations: INTEGER ;
                   {A configuration is a collection of module intermediates
                    (possibly at different levels) which together form a
                    complete, coherent, system. The number of configurations
                    is a function of the number of intermediates per module,
                    where each intermediate corresponds to a different
                    level of instrumentation.}
                system performance: ARRAY [1..num of configurations]OF
                RECORD
                      configuration description:
                   ARRAY [1..number of modules] OF INTEGER ;
                   {Where the integer is the number of the
                       intermediate chosen}

                   monitor/performance data: OUTPUT {system level};
                   {module level performance stored at module
                   level in modules.code.lower levels.etc.}
                END {performance record};
          END {transput record};
       END {system test record};
    END {system data base record};

    code =
       RECORD
          source: HAL/S;
          first intermediate: HALMAT {output from the first half of
                                      the compiler};
          basic monitor file : MONITOR FILE;
          lower levels: TARGET CODE;
       END
    target code =
       RECORD
          number of intermediates: INTEGER {each corresponds to
             different levels of instrumentation which have been
             inserted};
          intermediates: ARRAY [1..number of intermediates]OF
             RECORD
                description: TEXT {indicating what instrumentation
                   has been inserted. Note that intermediate code
                   resulting from the expansion of assertions at the
                   integration step is stored here, as well as levels
                   expanded solely at the module level};
                   intermediate: HALMAT ;
                   target:
                      RECORD
                   number of targets: INTEGER {This structure level
                      reflects the MUST environment option of targeting
                      a single HAL/S program to several object machines.
```

```
                    This level of structure is optional and may well
                    be omitted};
                targets: ARRAY [1..number of targets] OF
                    RECORD
                        code: LOWL {an unspecified low level
                                    language};
                        perf: PERFORMANCE {this is the output
                            specific to a particular machine/OS/
                            instrumentation combination, and is
                            described below };
                        load info: LOADRELATEDOUTPUT; {such as
                        maps}
                    END {targets record} ;
            END {target record };
        END {intermediates record} ;
    END {targetcode record definition};

performance =
    RECORD
      {number of test cases: INTEGER ;
       data: ARRAY [1..number of test cases] OF
           MONITORING INFO;  depending on the program, such as if
              there are parallel or real time features, this could
              contain some stuff normally found in modules [].
              results ; }
    END {performance record } ;

code description =
    RECORD
        decisions: HISTORY;
        management: FOLDER;
        external: TEXT;
        internal: TEXT;
        flowchart: GRAPH;
        static analysis:
            RECORD
                non-data flow: TEXT and GRAPHs;
                data flow: TEXT;
            END {static analysis documents};
    END {documentation record } ;

requirements =
    RECORD
        document: DOCUMENT;
        formal statement: SAMM {or similar vehicle which must be
                                relatable to the design vehicle(s) };
        management: FOLDER;
        acceptance criteria: CRITERIA;
    END {requirements record } ;
```

Appendix B: System Database

```
design =
    RECORD
    document: DOCUMENT;
    formal statement: IDAP {or similar vehicle which must be
                            relatable to the requirements and code
                            vehicles};
    decisions: HISTORY;
    management: FOLDER;
    acceptance criteria: CRITERIA;
    simulation: SIMULATION OUTPUT
END {design record};

history = TEXT; {which indicates how the current level of specification
                 was arrived at from the previous level, including why
                 particular choices were made}

folder = TEXT; {all management related information governing development
                of this particular phase, such as reviews, status
                reports, action items, and the like}
```

ITS FUNCTIONS

Appendix C

FUNCTION NAME	ARGUMENTS	DESCRIPTION
NEXT_STMT	none	Returns the statement number of the next statement which will be executed.
CURRENT_BLOCK	none	Returns the name of the block (procedure, function, etc.) which is currently being executed.
CURRENT_UNIT	none	Returns the name of the currently executing compilation unit.
COUNT(N)	N: statement number	Returns the execution count for statement N when executing an instrumented program. (current comp. unit)
STMT_TEXT(N)	N: statement number	Returns the statement text for statement N of the current comp. unit.
DEFINED(var)	var: variable name	Returns TRUE if variable 'var' has a value.
SYMBOLIC(var)	var: variable name	Returns TRUE if variable 'var's value is symbolic.

In addition to these functions, all of the HAL/S built-in functions are also available to the user of the ITS.

SAMM DIAGRAMS

Appendix D

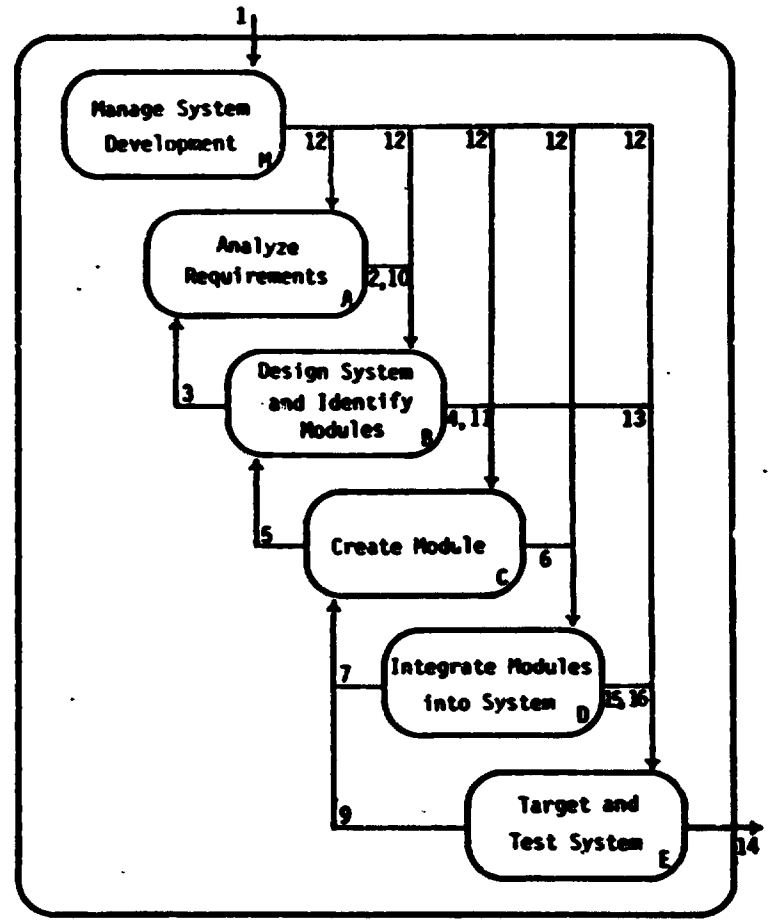
BGS

ACTIVITY - DATA FLOW DIAGRAM

TITLE: SYSTEM CREATION

ACTIVITY DESIGNATOR: ROOT

DATA ID	TRACE	DATA DESCRIPTION
1		Initial concept
2		Requirements specification
3		Requirements revision specification
4		Preliminary design and module requirements
5		Preliminary design revision requirements
6		Modules
7		Module revision requirements
8		System (HALMAT level)
9		"Results" based module revision requirements
10		System acceptance criteria
11		Module acceptance criteria
12		Managerial guidance and oversight, user input
13		Design acceptance criteria
14		Executable level of system
15		Suspect system path specifications
16		System flowgraph



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT NO(S) (SAMM)

CO 1000 1015 (ENG. 3/78)

DATA DESCRIPTIONS

NODE ROOT TITLE SYSTEM CREATION

DATA			RELATED ACTIVITIES	
ID	TYPE	DESCRIPTION	SOURCE	DESTINATION
		The acceptance criteria which are generated by several activities in this SAMM chart are very important. As a system is created many functions are written as a part of the implementation. Each layer of decomposition introduces a new set of functions, which together comprise the functions of the previous level. Each of these functions requires testing to guarantee that it produces the desired result. In order to enable this testing, as each function is defined, at every level of decomposition, a set of acceptance criteria needs to be defined for that function. The totality of these criteria enable effective, thorough testing when the code is produced. (They are useful for testing at higher levels, too, of course.)		

BCS BOEING COMPUTER SERVICES COMPANY
A Division of the Boeing Company

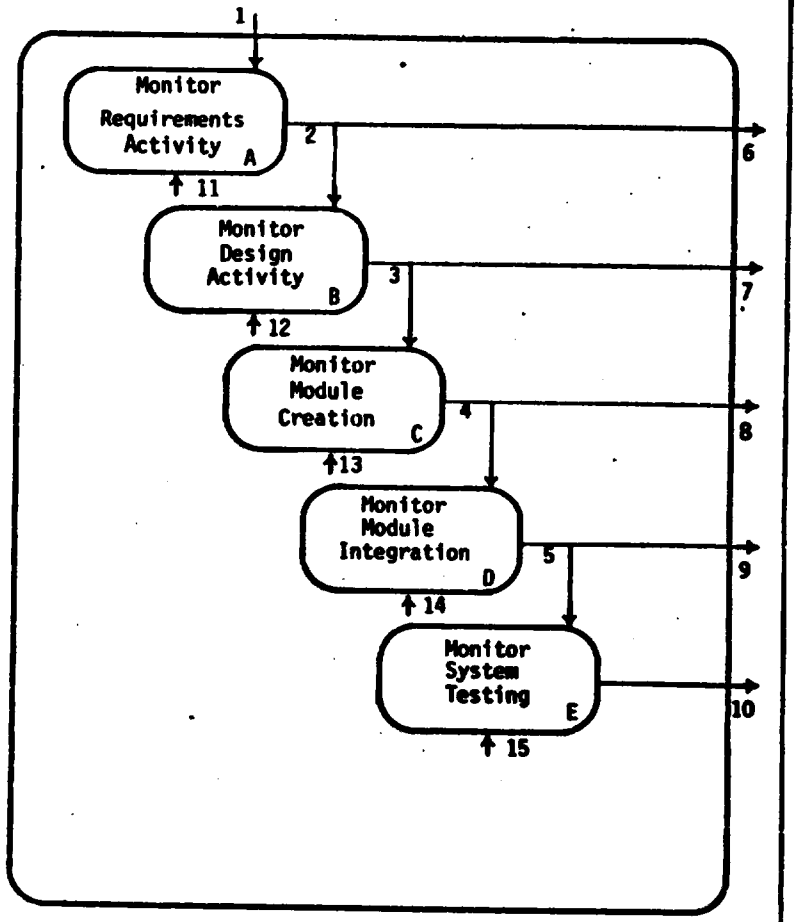
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: M

TITLE: MANAGE SYSTEM DEVELOPMENT

DATA ID	TRACE	DATA DESCRIPTION
1	1	Initial concept
2-5		Record of important decisions made (design issues) at each step - key spots to watch, etc.
6-10	12	Managerial approval/input into each process (controlling decisions)
11	db	System requirements
12	db	Design
13	db	Modules [] - documentation, performance information, and results
14	db	Integration
14	db	System test

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)
CO 1000 1016 ORIG. 3/76

BCS BOEING COMPUTER SERVICES COMPANY

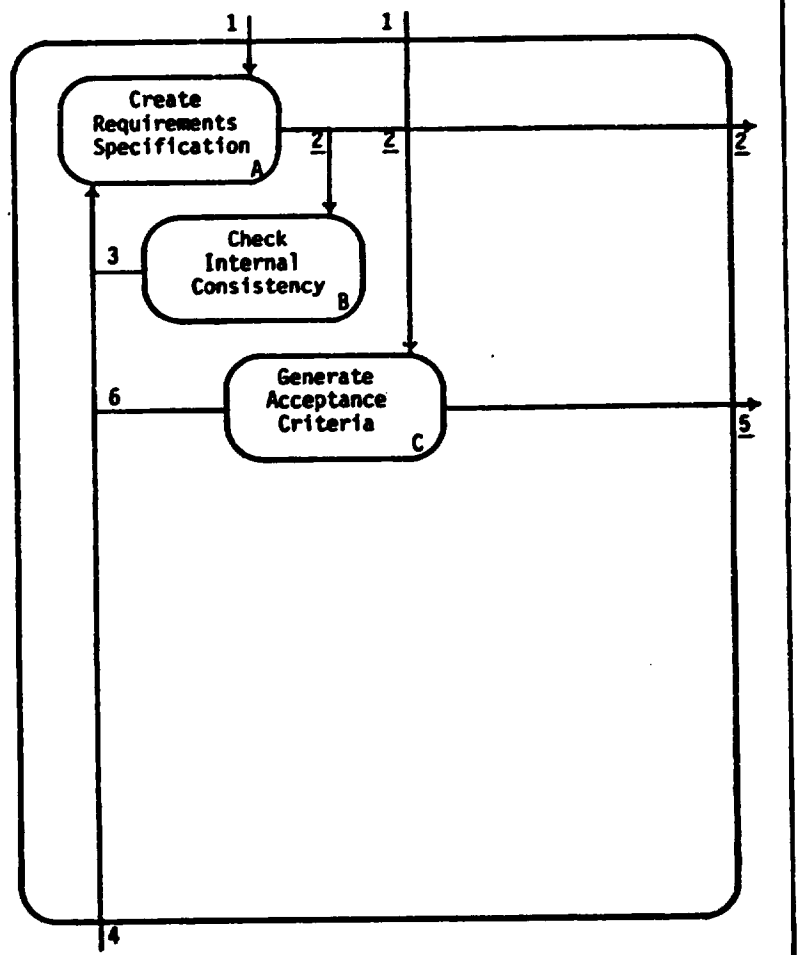
A Division of The Boeing Company

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: A

TITLE: ANALYZE REQUIREMENTS

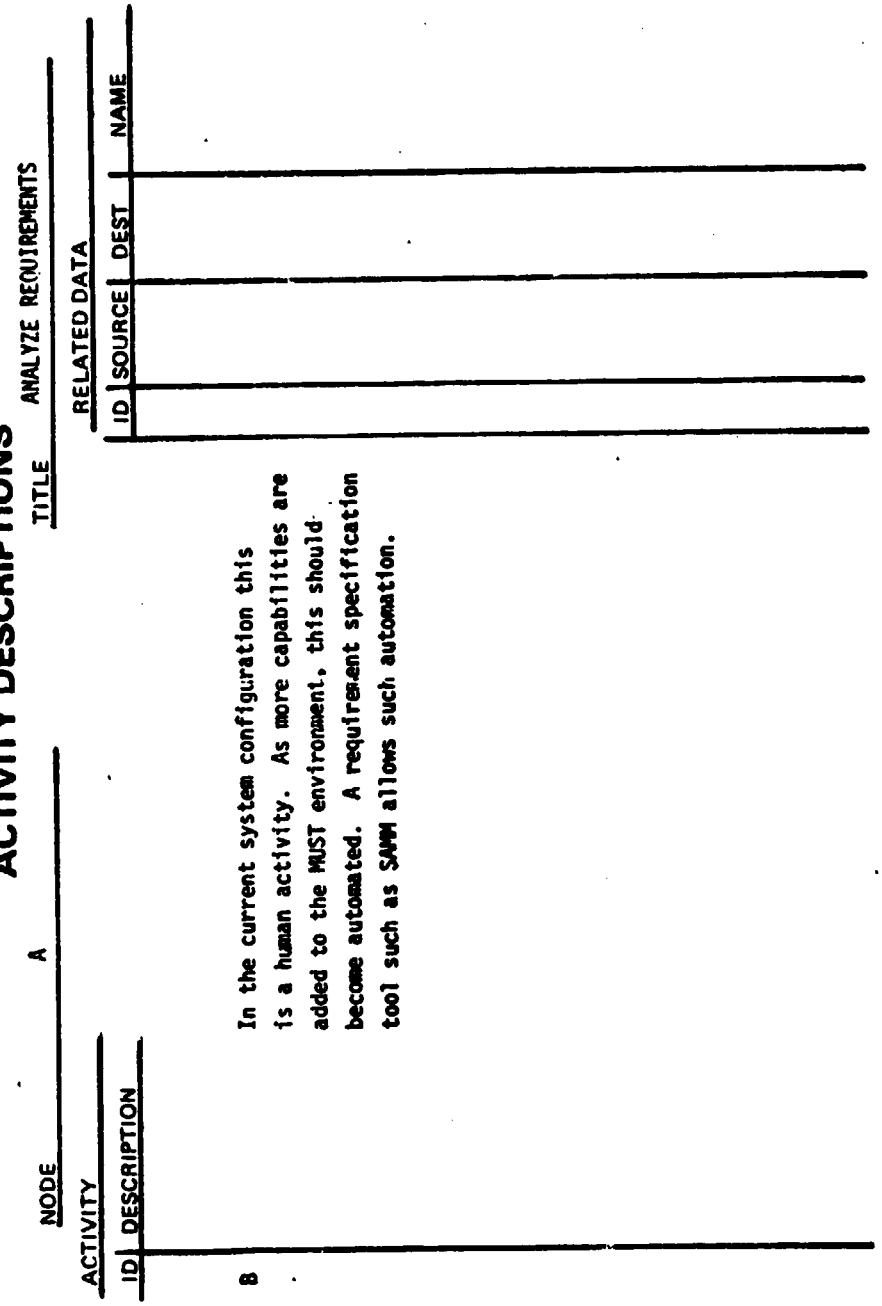
DATA ID	TRACE	DATA DESCRIPTION
1	12	Managerial guidance conveying initial concept and requirements
2	2	Requirements specification
3		Inconsistency - based revision requirements
4	3	Design based revision specifications
5	10	System acceptance criteria
6		Acceptance based revision specifications



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 30167 (SAMM)

CO 1000 1016 ORIG. 2/78


ACTIVITY DESCRIPTIONS
NODE A
TITLE ANALYZE REQUIREMENTS
ACTIVITY
ID DESCRIPTION
B
In the current system configuration this is a human activity. As more capabilities are added to the MUST environment, this should become automated. A requirement specification tool such as SAMM allows such automation.
RELATED DATA
ID SOURCE DEST NAME

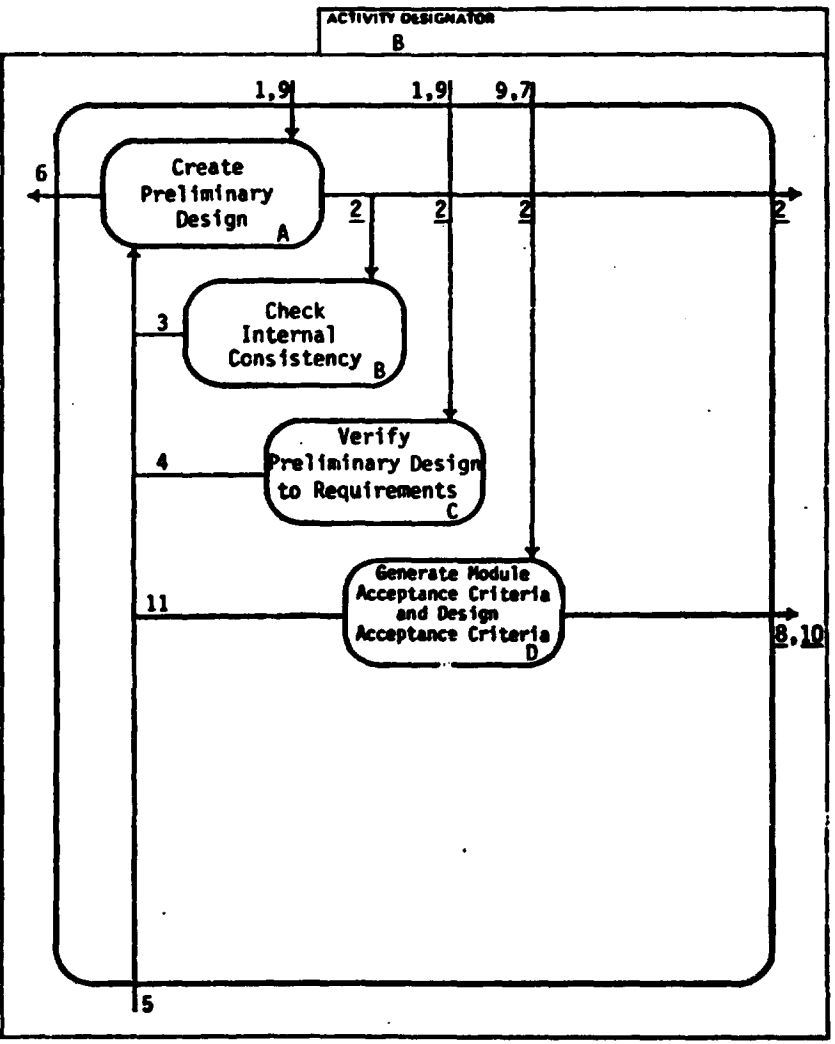
BGS

ACTIVITY - DATA FLOW DIAGRAM

TITLE: DESIGN SYSTEM AND IDENTIFY MODULES

DATA ID	TRACE	DATA DESCRIPTION
1	2	Requirements specification
2	4	Module requirements and integration specification (preliminary design)
3		Inconsistency based revisions
4		Verification based revisions
5	5	Module design based revisions
6	3	Requirements revision specifications
7	10	System acceptance criteria (requirements level)
8	11	Module acceptance criteria
9	12	Management oversight and review
10	13	Design acceptance criteria
11		Acceptance based revision requirements

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE B

TITLE DESIGN SYSTEM AND IDENTIFY MODULES

ACTIVITY ID	DESCRIPTION
B,C	As in the case of requirements analysis these activities are currently human performed. As capabilities are added, these should become automated.

RELATED DATA ID	SOURCE	DEST	NAME

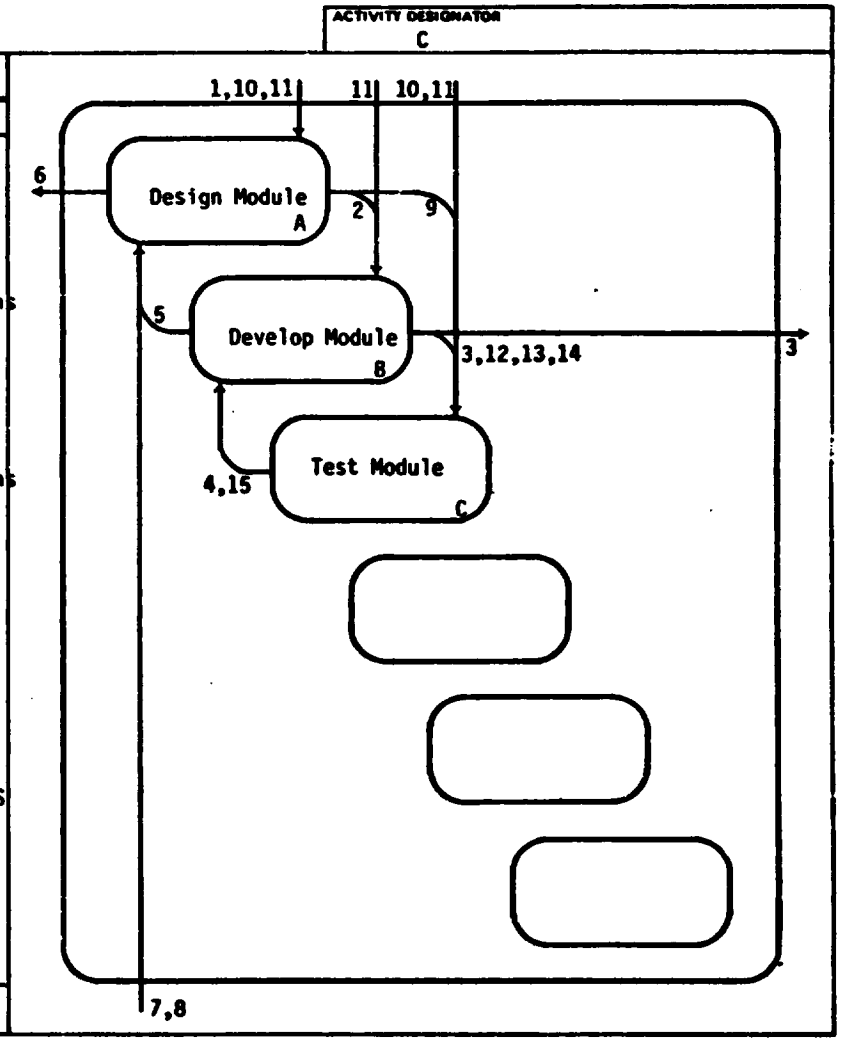
BCS BOEING COMPUTER SERVICES COMPANY
A Division of The Boeing Company

ACTIVITY - DATA FLOW DIAGRAM

TITLE: CREATE MODULE

DATA ID	TRACE	DATA DESCRIPTION
1	4	Preliminary design and module requirements
2		Module Design
3	6	Module code (HALMAT level)
4		"Test based" code revision specifications
5		Code development based design revision specifications
6	5	Module design based revision specifications
7	7	Integration based revision specifications
8	9	"Results" based revision specifications
9		Design level module acceptance criteria
10	11	Module acceptance criteria (requirements level)
11	12	Managerial input/controlling parameters
12		Path specification from static analyzers for symbolic execution
13		Annotated program flowgraph
14		List of instrumentation for use with ICS
15		Instrumentation revision specifications

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE



REF. DOCUMENT 10167 (SAMM)
CO 1000 1016 ORIG. 2/70

BCS BOEING COMPUTER SERVICES COMPANY

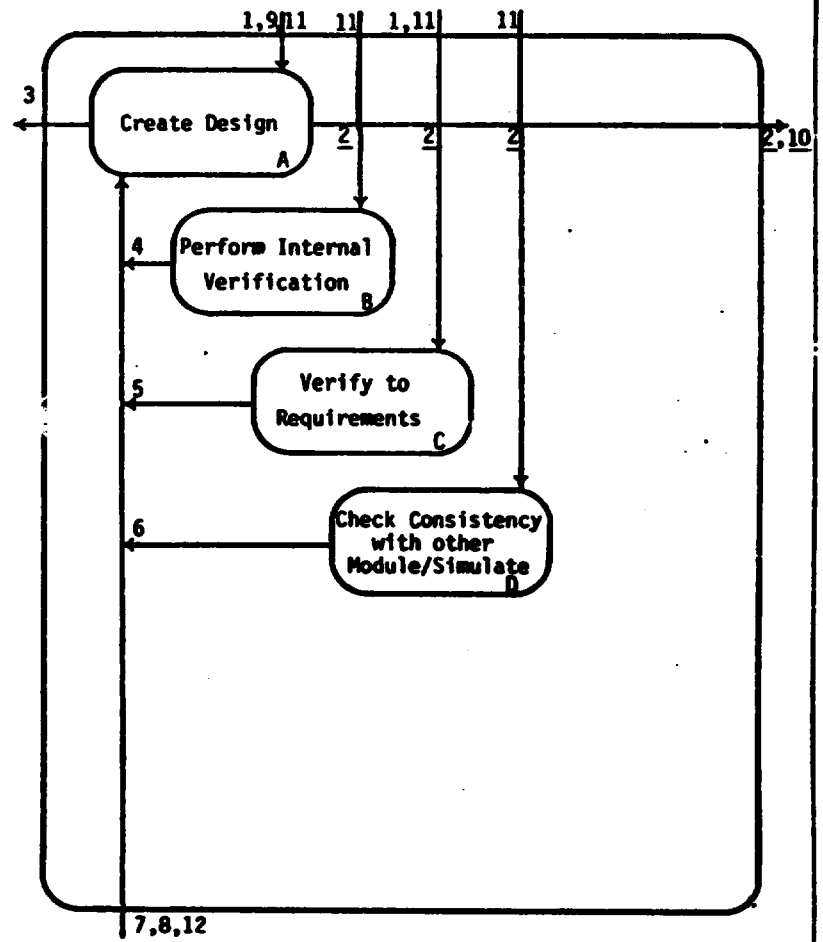
A Division of The Boeing Company

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CA

TITLE: DESIGN MODULE

DATA ID	TRACE	DATA DESCRIPTION
1	1	Module requirements
2	2	Module design specification
3	6	Module design based revision specifications
4		Revision requirements - internal
5		Revision requirements - verification to requirement
6		Revision requirements - verification to other modules
7	7	Revision requirements - integration
8	8	Revision requirements - results
9	10	Requirements level module acceptance criteria
10	9	Module design acceptance criteria
11	11	Managerial input
12	5	Code development based design revision specs



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 2/70

IBGS

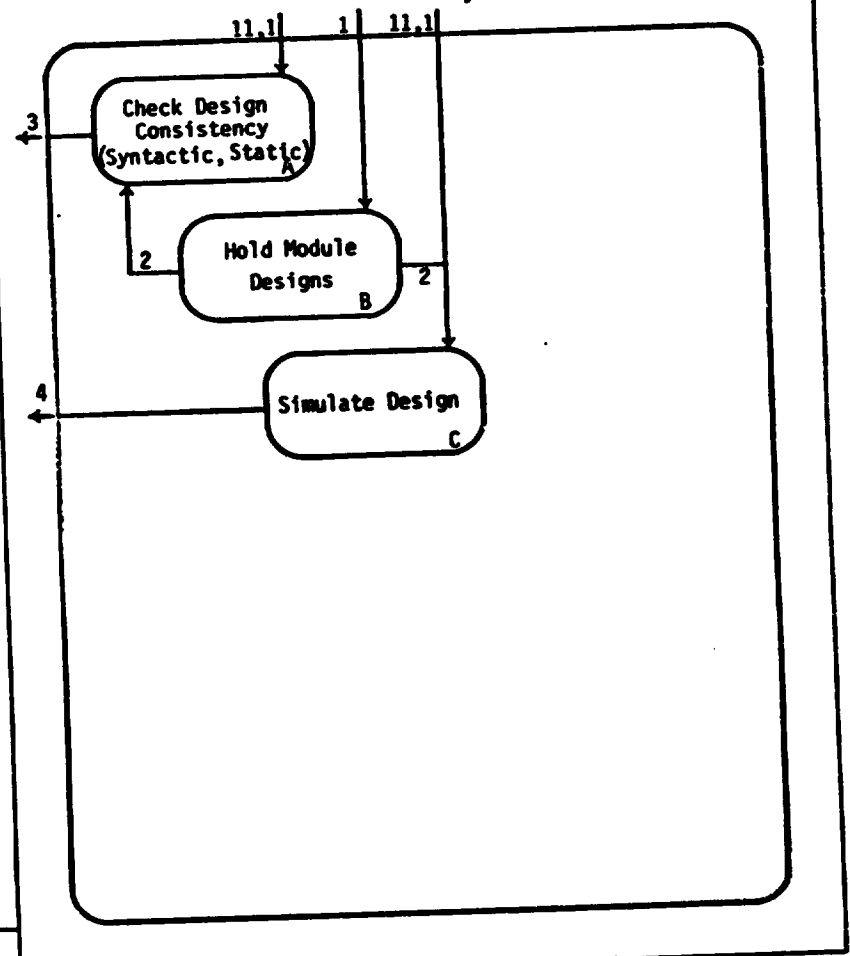
ACTIVITY - DATA FLOW DIAGRAM

TITLE: CHECK CONSISTENCY WITH OTHER MODULES/SIMULATE

DATA ID	TRACE	DATA DESCRIPTION
1	2	Subject module design
2		Additional module designs (previously created)
3	6	Consistency based revision requirements
4	6	Simulation based revision requirements
5	11	Managerial input

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

ACTIVITY DESIGNATOR: CAD



REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 3/78

ACTIVITY DESCRIPTIONS

NODE CAD

TITLE CHECK CONSISTENCY/SIMULATE

ACTIVITY

ID	DESCRIPTION
B	This activity explicitly models a function of the system database. Similar "activities" are modelled implicitly elsewhere in the design.

RELATED DATA

ID	SOURCE	DEST	NAME

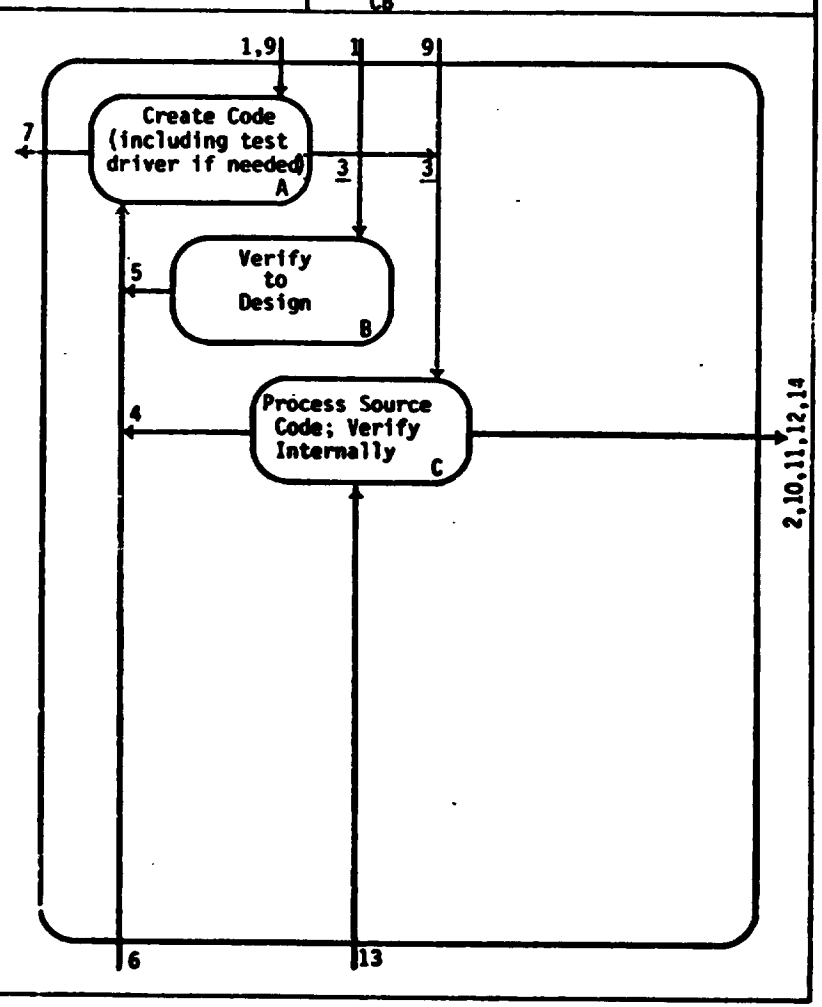
BGS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CB

TITLE: DEVELOP MODULE

DATA ID	TRACE	DATA DESCRIPTION
1	2	Module design specification
2	3	HALMAT
3		Source code (HAL/S)
4		Revision requirements - internal consistency based
5		Revision requirements - verification to design based
6	4	Revision requirements - test based
7	5	Design revision requirements
9	11	Managerial input/controlling parameters
10	12	Suspect path specifications
11	13	Annotated program flowgraph
12	14	List of instrumentation
13	15	Instrumentation revision specifications
14	3	MMF for system testing



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT NO(S) (SAMM)

CO 1000 1016 ORIG. 3/76

BCS

ACTIVITY - DATA FLOW DIAGRAM

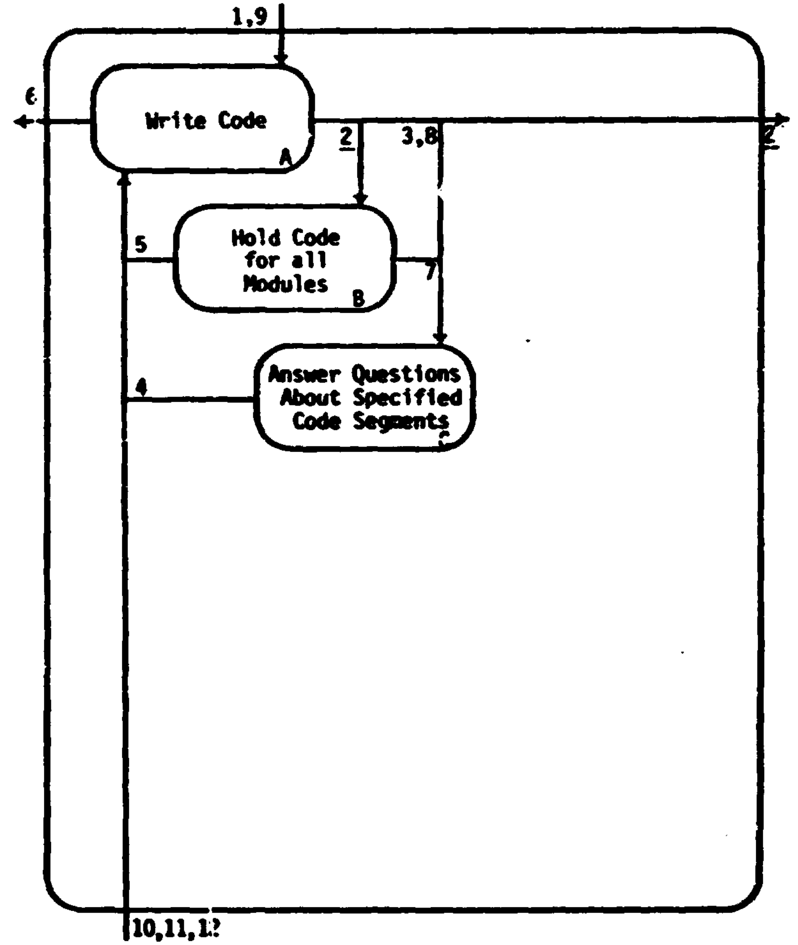
TITLE: CREATE CODE

DATA ID	TRACE	DATA DESCRIPTION
1	1	Design specification
2	3	Source code (HAL/S)
3		Queries about code usage
4		Internal documentation, answers to code usage questions
5		Old code to be used, library functions
6	7	Design revision requirements
7		All existent code
8		Requests for internal documentation
9	9	Managerial input
10	4	Internal consistency based revision requirements
11	5	Verification to design based revision requirements
12	6	Test based revision requirements

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10107 (SAMM)

ACTIVITY DESIGNATOR: CBA



ACTIVITY DESCRIPTIONS

NODE CBA

TITLE Create Code

ACTIVITY

ID	DESCRIPTION
B	The actions of the data base modelled explicitly.
C	This activity will provide functions like that of the University of Texas FAST system. Particular questions about the usage of variables, lables, and the like may be asked. In addition, internal documentations of routines (library functions) may be referenced. Note that this activity is only an intelligent text editor. Analysis capabilities are found in other tools, such as the static analysers and the symbolic executor.

RELATED DATA

ID	SOURCE	DEST	NAME

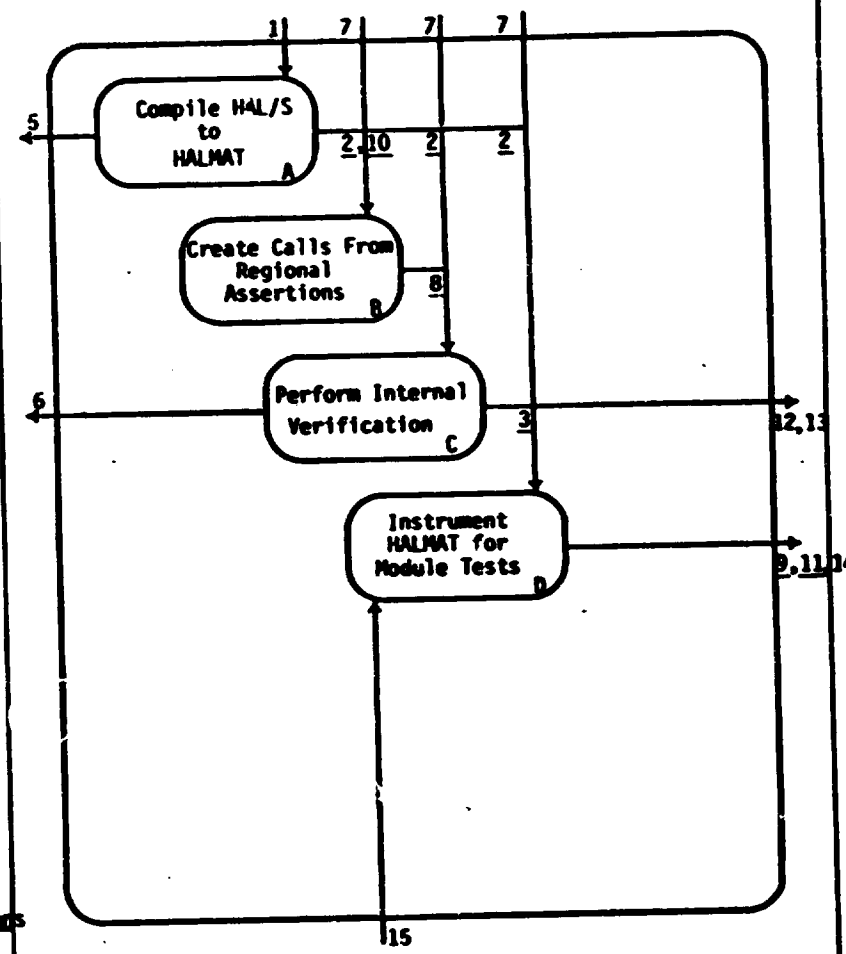
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBC

TITLE: PROCESS SOURCE CODE; VERIFY INTERNALLY

DATA ID	TRACE	DATA DESCRIPTION
1	3	Source code (HAL/S)
2		HALMAT
3		Further expanded HALMAT Monitor File
5	4	Compiler error messages
6	4	Verification based revision requirements
7	9	Managerial input (controlling parameters, such as which assertion levels to process, or which verification tools to use)
8		Expanded HALMAT Monitor File
9	2	Instrumented HALMAT
10		Initial HALMAT Monitor File (contains parsed assertions, units, and history statements)
11	14	HALMAT Monitor File containing monitor information to be expanded at system level
12	10	Specifications of suspect paths
13	11	Annotated program flowgraph
14	12	List of all instruments (including those which may be hand placed in an ICS)
15	13	Instrumentation revision specifications



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT NO(S) (SAMM)

CD 1000 1015 ORIG. 3/70

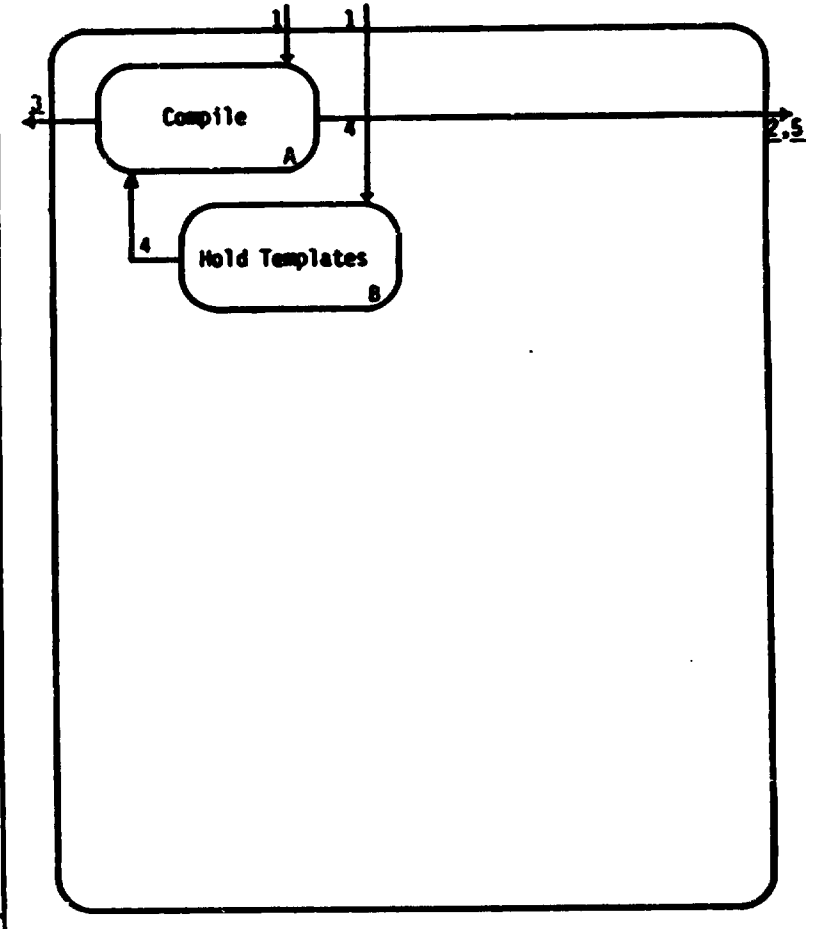
BGS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCA

TITLE: COMPILE HAL/S to HALMAT

DATA ID	TRACE	DATA DESCRIPTION
1	1	Source code (HAL/S)
2	2	HALMAT
3	5	Error messages
4		Procedure/Program/Task/Compool templates
5	10	Initial HALMAT Monitor File (parsed assertions, units, and history information)



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CD 1000 1010 ORIG. 3/78

ACTIVITY DESCRIPTIONS

NODE CBCA

TITLE COMPILE HAL/S

ACTIVITY

ID	DESCRIPTION
A	This "first half" compilation only generates HALMAT, the symbol tables, and module templates needed for later compilations. Further targeting of the code to the executable level is performed by later activities. Thus the traditional concept of a "compiler" is altered here.

RELATED DATA

ID	SOURCE	DEST	NAME

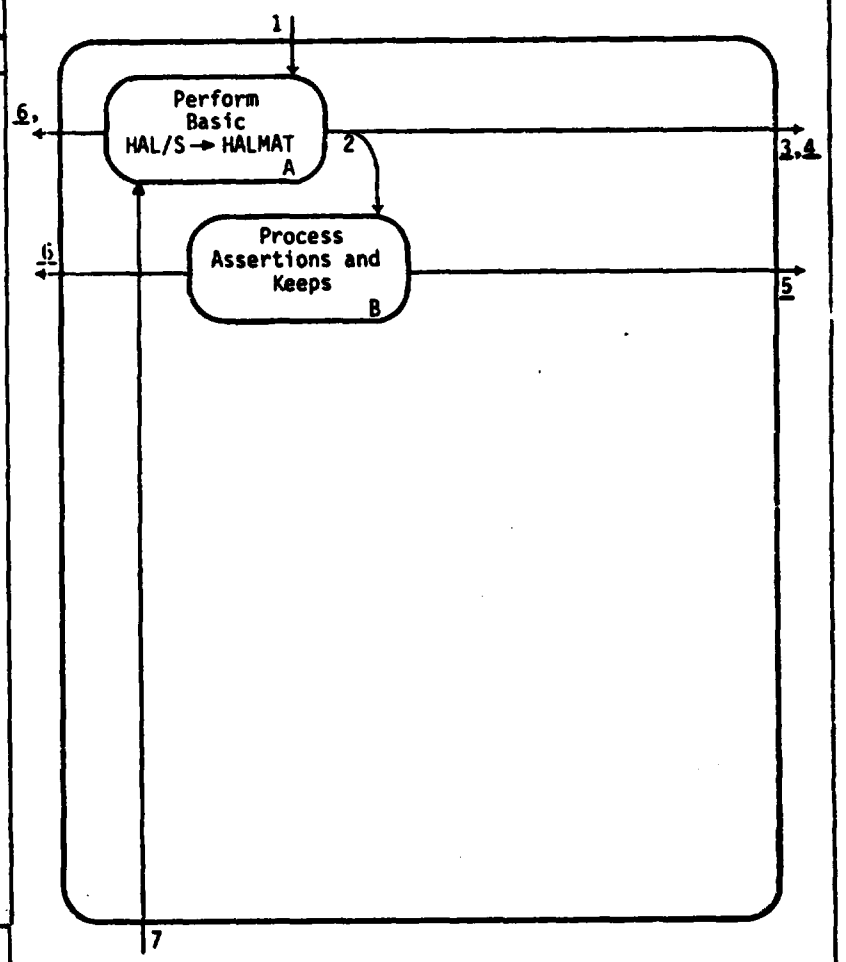
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCAA

TITLE: COMPILE (FIRST HALF)

DATA ID	TRACE	DATA DESCRIPTION
1	1	Source code: HAL/S including new features
2		New features (filtered out)
3	2	Stock HALMAT
4	4	Templates
5	5	Initial HALMAT
		Monitor File
6	3	Error messages
7	4	Pre-generated templates



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 9000 1015 ORIG. 2/76

ACTIVITY DESCRIPTIONS

NODE CBCAA

TITLE COMPILE (FIRST HALF)

ACTIVITY

ID	DESCRIPTION
A	Slightly modified Intermetrics/LRC compiler (filters out special comments for further processing).
B	Uses compiler procedures to crack special comments into a second file - The statements are not instrumented here, they are only broken down into a more manageable representation.

RELATED DATA

ID	SOURCE	DEST	NAME

DATA DESCRIPTIONS

NODE CBCAA

TITLE COMPILE (FIRST HALF)

DATA

ID	TYPE	DESCRIPTION
5		This file will contain HALMAT representing the expressions contained in the assert/keep statements, plus pointers/flags and so forth indicating the nature of the statement. See Section 3.3.2.

RELATED ACTIVITIES

SOURCE	DESTINATION

BCS BOEING COMPUTER SERVICES COMPANY

A division of the Boeing Company

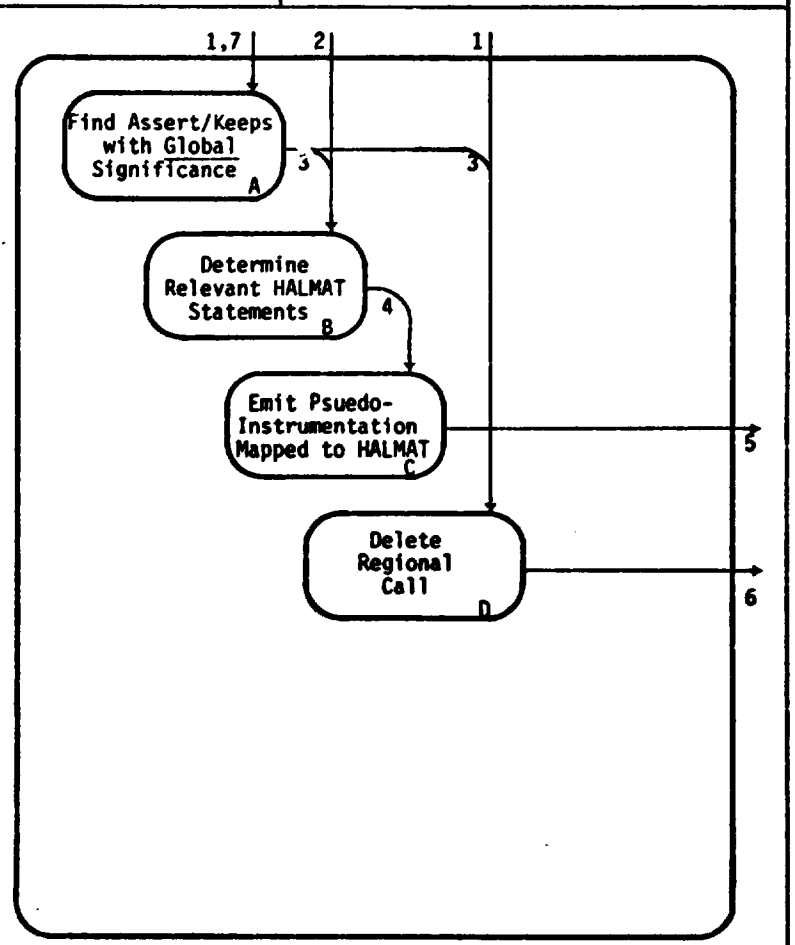
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCB

TITLE: CREATE CALLS FROM REGIONAL ASSERTIONS

DATA ID	TRACE	DATA DESCRIPTION
1	10	Preprocessed assert/keep statements
2	2	HALMAT
3		Assert/keep with regional significance
4		HALMAT statements (#'s) which require instrumentation
5	8	Instruments mapped to HALMAT
6	8	Preprocessed assert/keeps with regional ones "marked" as "done"
7	7	Controlling switch: perform expansion or not

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. LOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

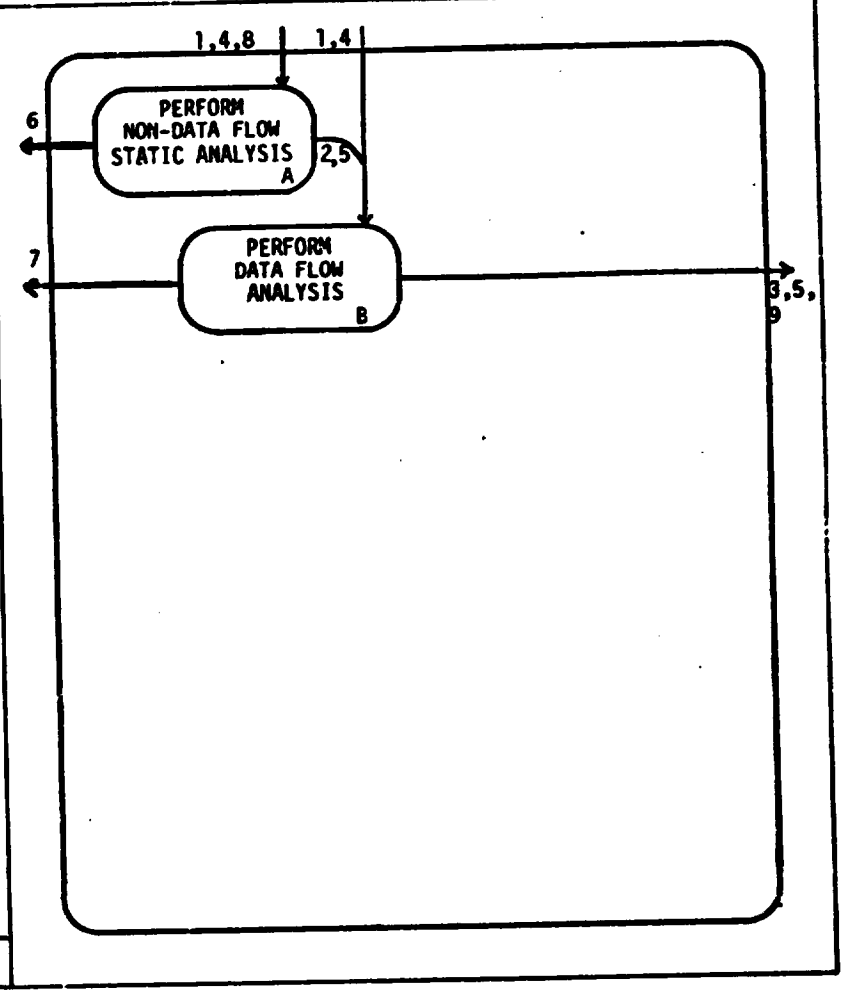
BCS Boeing Computer Services Company

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCC

TITLE: PEFORM INTERNAL VERIFICATION

DATA ID	TRACE	DATA DESCRIPTION
1	2	HALMAT
2		Call graph
3	12	Specifications of paths along which an error is suspected to exist.
4	7	Managerial input (controlling parameters).
5	3	Expanded HALMAT Monitor file (calls for instrumentation may be added or deleted as the analysis proceeds).
6	6	Error messages and documentation
7	6	Comprising revision requirements or code
8	8	HALMAT monitor file
9	13	Annotated program flowgraph



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 2/70

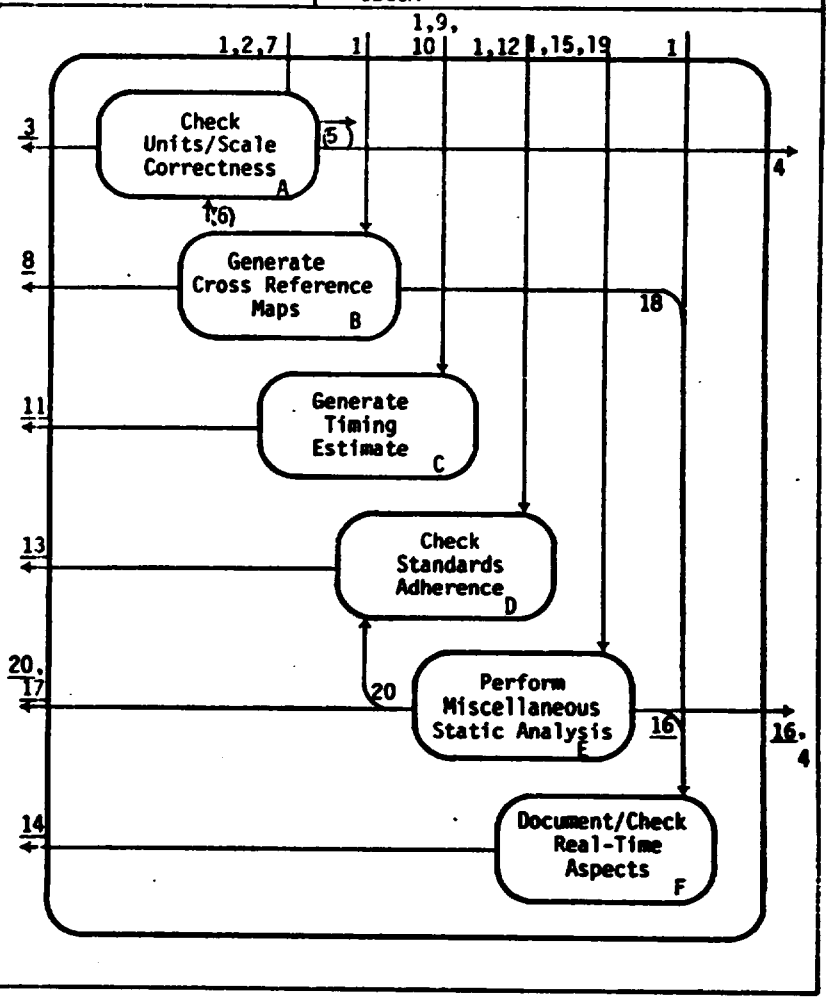
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCA

TITLE: PERFORM NON-DATA FLOW STATIC ANALYSIS

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	8	Processed units/scale declarations vectors associated with variables matrices of relationships
3	6	Error messages
4	5	Refined HMF
(5)	db	Annotated source listing
(6)	db	Source listing
7	4	Directives: coerce automatically or not
8	6	Cross reference maps
9	4	Machine instruction time specifications
10	4	Path specification
11	6	Timing estimate
12	4	Coding standards specifications
13	6	Violation messages, statistical summary
14	6	Documentation
15	4	Managerial input
16	2	Call graph
17	6	Error messages, documentation
18		Listing of shared variables
19	8	HALMAT monitor file
20	6	Program complexity measures



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CBCCA TITLE PERFORM NON-DATA FLOW STATIC ANALYSIS

ACTIVITY		RELATED DATA			
ID	DESCRIPTION	ID	SOURCE	DEST	NAME
D	This auditor should be able to check a variety of standards, including: presence of assertions (especially range-type assertions) comments control structure program size program complexity language constructs which have semantic ambiguities (as noted by Pratt) prohibited language constructs syntax conventions, such as certain declaration forms and complete expression parenthesization				

OUTPUT-CONDITIONS DESCRIPTIONS

NODE CBCCA

ACTIVITY	OUTPUT	INPUT
A	4	1,2,7

TITLE

CC	CONDITION CODE DEFINITIONS
1	1 User selected option. Programmer must not have included any conversion factors in his statements. All will be supplied by the system.

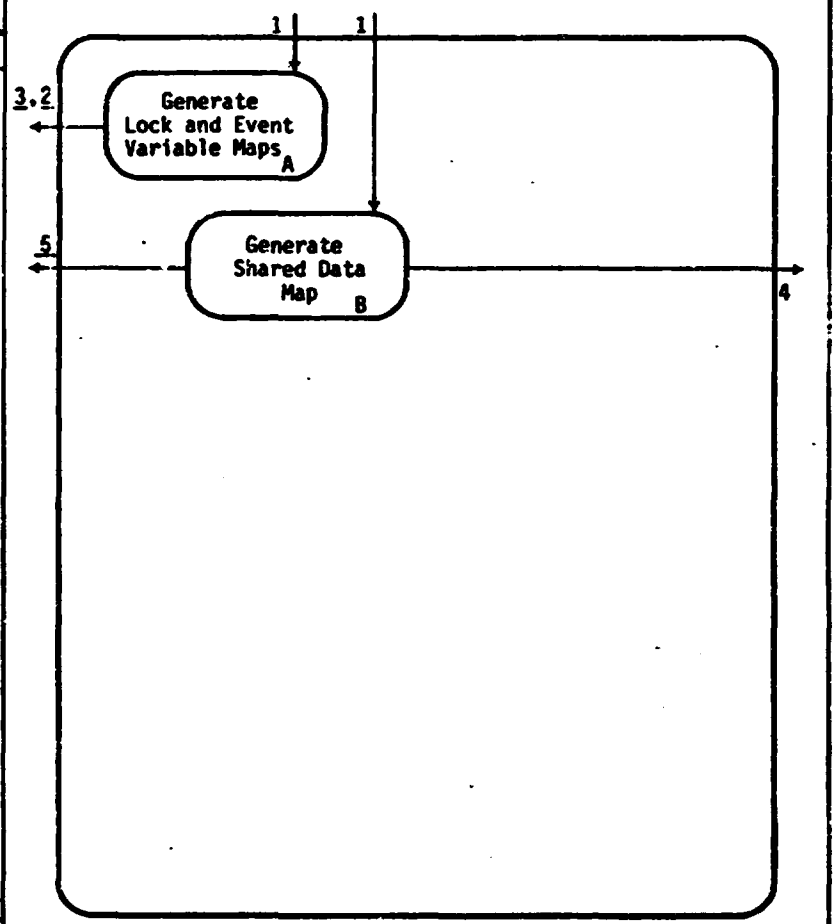
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCAB

TITLE: GENERATE CROSS REFERENCE MAPS

DATA ID	TRACE	DATA DESCRIPTION
1	1	Symbol table and HALMAT
2	8	Lock group membership map
3	8	Event variable map
4	18	List of shared variables
5	8	Shared data map



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10157 (SAMM)

CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CBCCAB

TITLE GENERATE CROSS REFERENCE MAPS

ACTIVITY

ID	DESCRIPTION
A	Lock group memberships and event variable maps are easy to generate and are an addition to the maps provided by the compiler. (An adequate map for COMPOOL variables is given by the compiler in terms of the declaration/templates, the block summaries, and the variable cross reference map.)
B	This map will indicate the global variables not belonging to LOCK groups which are used by processes (TASKs) which potentially operate in parallel.
	The Event Scheduling Statement Cross Reference is fairly well provided by the block summary created by the compiler. The block summary does not reference actual statement numbers, however. Two possibilities exist: a facility could be provided here to perform this and provide a map for the entire program, or the compiler could be slightly modified to augment the block summary.

RELATED DATA

ID	SOURCE	DEST	NAME

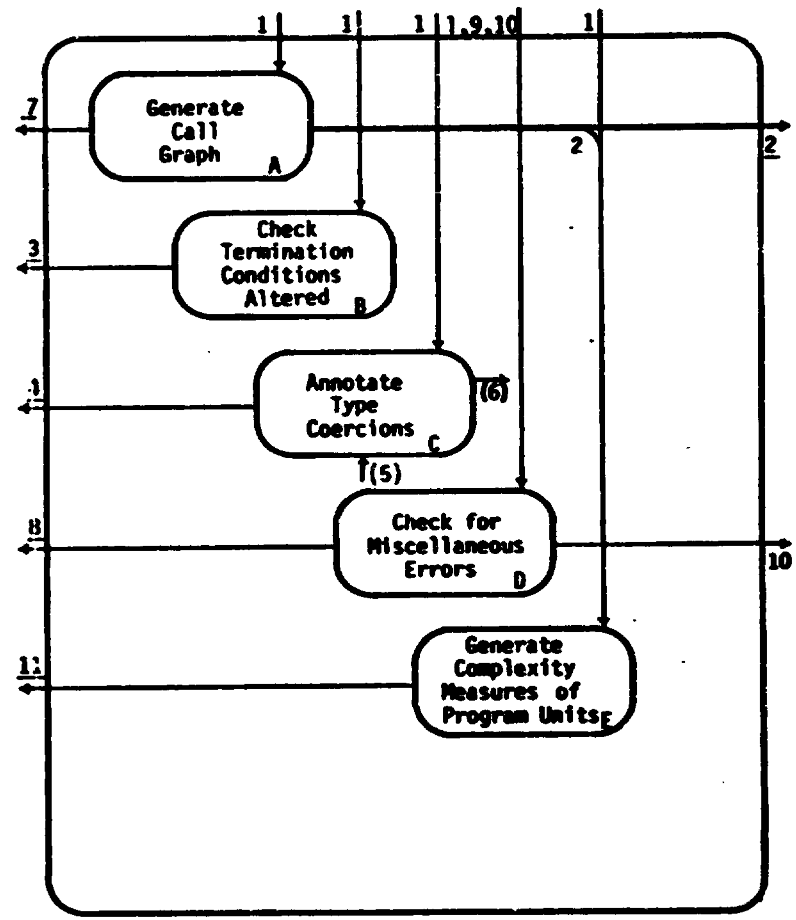
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCAE

TITLE: PERFORM MISCELLANEOUS STATIC ANALYSIS

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	16	Call graph
3	17	Indication (heuristic) as to whether loop termination conditions are altered in the range of the loop
4	17	Warnings as to coercions performed
(5)	db	Source listing
(6)	db	Source listing with annotations like 4
7	17	Error messages: recursion, unused procedures, etc.
8	17	Error messages
9	15	Managerial input: monitors desired
10	4	Monitors to apply to HALMAT
11	20	Program unit complexity measures (in the sense of "software science")



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 3/78

ACTIVITY DESCRIPTIONS

NODE CBCCAE

TITLE PERFORM MISCELLANEOUS STATIC ANALYSIS

ACTIVITY

ID	DESCRIPTION
A	Generate call graph. After graph generation, it should be analyzed for cycles (indicating recursion). Possible additional analysis could check for procedures not used, procedures not defined, etc. These errors are likely best detected elsewhere, however - the compiler, data flow analysis. The call graph could, alternatively to the scheme presented, be generated from analysis of the listing produced by the compiler: "combine" the contents of the compilation layout with the block summaries. This would be faster, though possibly inadequate for multiple compilation units.

RELATED DATA

ID	SOURCE	DEST	NAME

ACTIVITY DESCRIPTIONS

NODE CBCCAE

TITLE

ACTIVITY

ID	DESCRIPTION
D	This activity will check for (at least) the following errors/ error-prone conditions: 1. Paths through a function block which end on a close, instead of a return. 2. A variable used twice in the same subroutine call or function call. 3. Using an aligned minor structure with a DENSE BIT terminal as part of an ASSIGN parameter. 4. More than one unlatched event variable in a logical product of multiple event variables. 5. Scalars compared with an equality relation. Monitors generated (optionally): 1. Check relative size of numerator and denominator in divisions. 2. Check for overflow possibilities (machine dependent - may be better suited elsewhere). 3. Subscript out of range.

RELATED DATA

ID	SOURCE	DEST	NAME

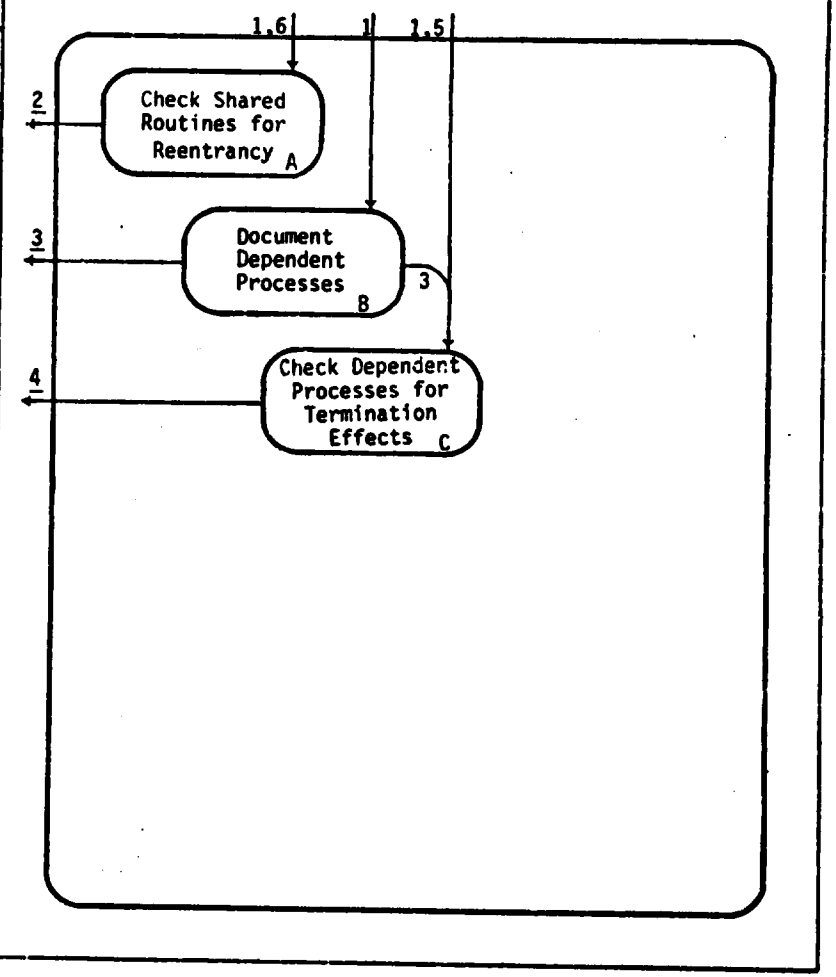
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCAF

TITLE: DOCUMENT REAL TIME ASPECTS

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	14	Indication of reentrancy for shared routines
3	14	Documentation of which routines are dependent
4	14	Documentation indicating effects of any terminates on dependent processes which use shared variables
5	18	List of what variables are shared
6	16	Call graph



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/76

ACTIVITY DESCRIPTIONS

NODE CBCCAF

TITLE DOCUMENT REAL TIME ASPECTS

ACTIVITY

ID	DESCRIPTION
A	Check shared routines for reentrancy Activities include: 1. Determine which routines should be reentrant 2. Check those routines for reentrancy - ensure they only call reentrant routines - ensure that all global data modified is locked - warn about statically declared variables - internal update blocks and initialize functions should declare no data

RELATED DATA

ID	SOURCE	DEST	NAME

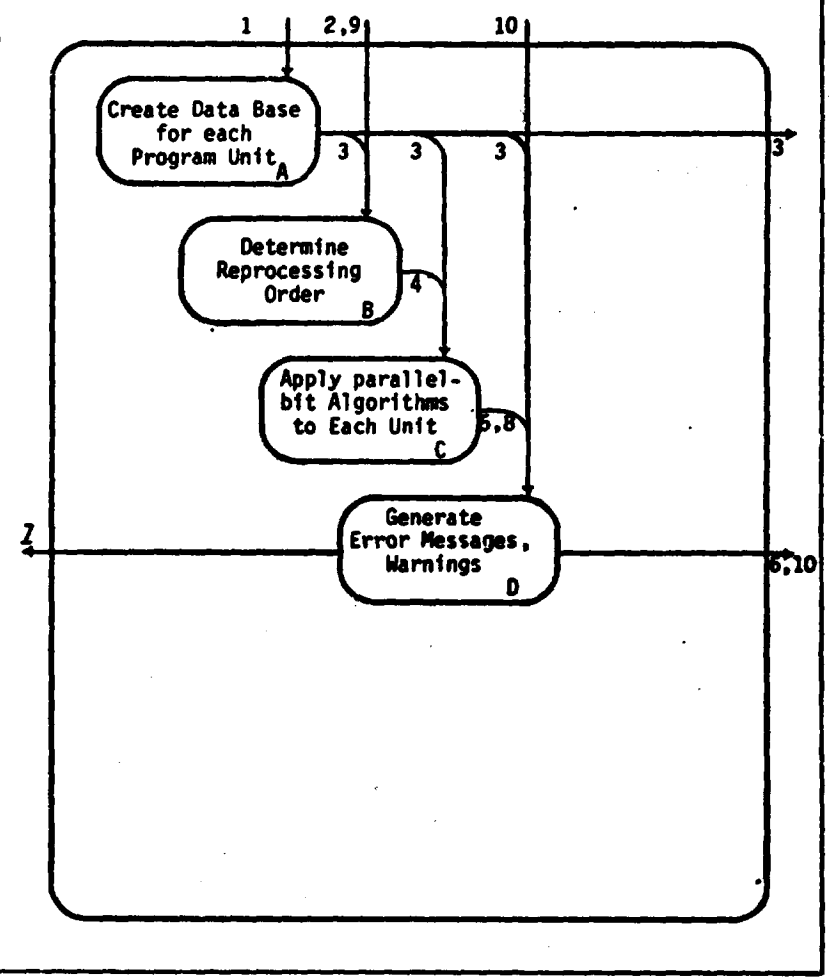
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR CBCCB

TITLE: PERFORM DATA FLOW ANALYSIS

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	2	Call graph (possibly non-existent)
3	9	Program unit database: flowgraph plus sets (gen, kill, null)
4		Processing order, control information
5		List of errors detected/node
6	3	Paths on which errors may lie
7	7	Error messages, documentation
8		Variable effects tracing information
9	4	Select quality of analysis: inter proc, intra proc, multi proc
10	5	HALMAT Monitor File (to check OUTPUT and INVARIANT assertions at least)



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 9167 (SAMM)

CO 1000 1016 ORIG. 2/78

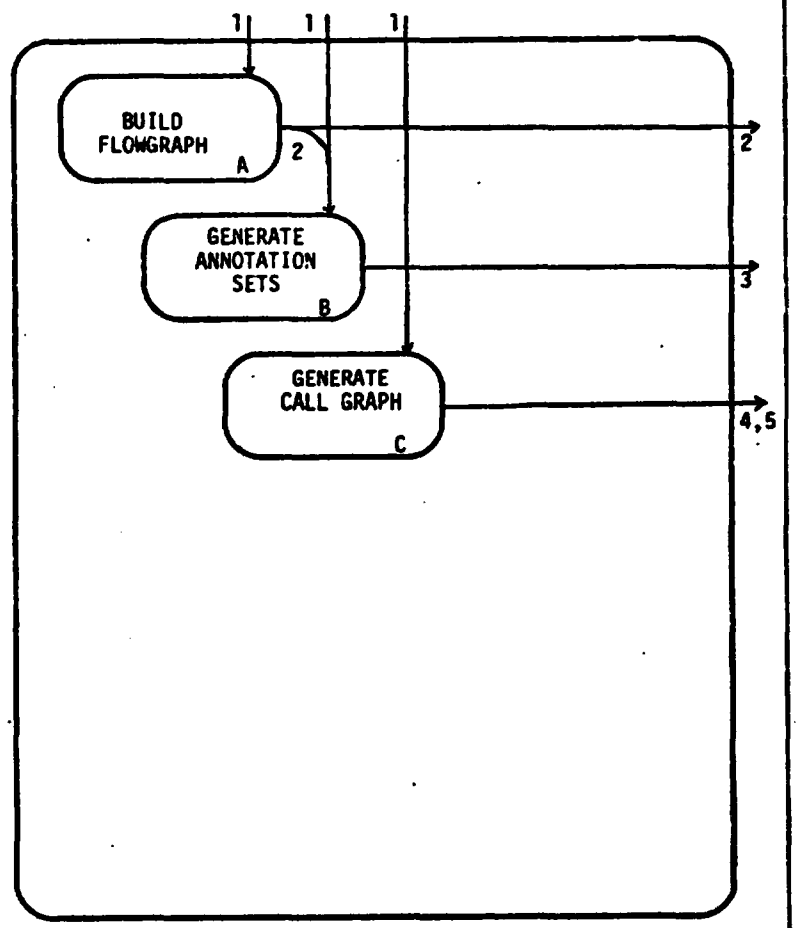
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCBA

TITLE: CREATE DATA BASE FOR EACH PROGRAM UNIT

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	3	Un-annotated flowgraph
3	3	Annotations for each flowgraph node
4	3	Call graph
5	3	Mapping of procedure headers to SMRK numbers (and thus to flowgraph nodes)



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

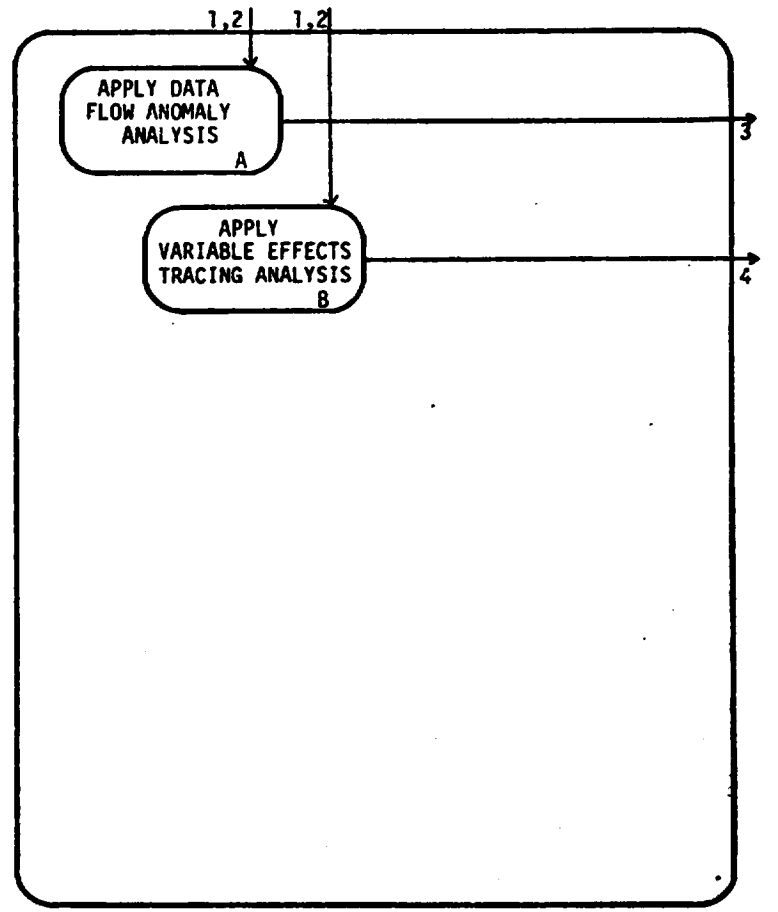
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CBCCBC

TITLE
APPLY PARALLEL-BIT ALGORITHMS TO EACH UNIT

DATA ID	TRACE	DATA DESCRIPTION
1	3	Program unit datbase
2	4	Processing order, control information
3	5	Anomaly messages (raw form)
4	8	Variable-effects tracing information



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)
CO 9000 1015 ORIG. 2/78

BCS

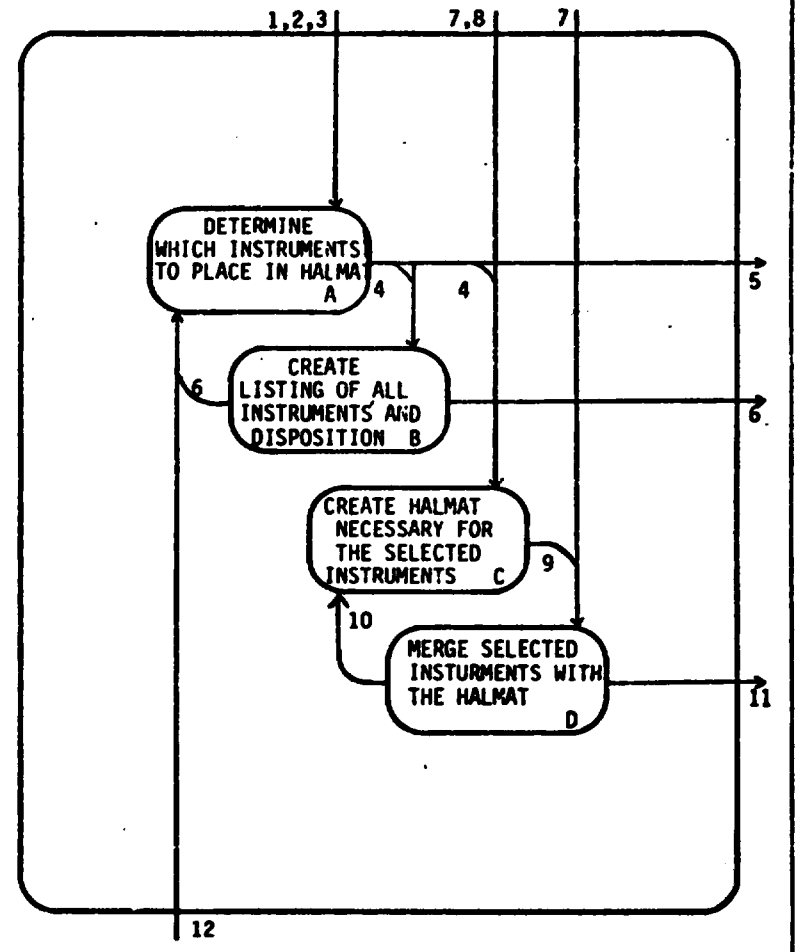
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CBCD

TITLE
INSTRUMENT HALMAT FOR MODULE TESTS

DATA ID	TRACE	DATA DESCRIPTION
1	3	HALMAT Monitor File
2	7	Directives: what level testing is to be performed at; what to do on violation; etc.
3	7	Directives: what HALMAT is to be targeted to - ICS, host, etc.
4		HMF with indication as to disposition of each monitor.
5	11	HMF to be expanded at system level.
6	14	Listing of all instruments and how each is to be handled.
7	2	HALMAT
8	7	Precompiled assertion procedures
9		HALMAT which represents an instrument.
10		Updated HALMAT (pointers changed)
11	9	Fully updated HALMAT
12	15	Instrumentation revision specifications resulting from interactive testing (pathwise verification).

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)
CO 1000 1016 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CRCD

TITLE INSTRUMENT HALMAT FOR MODULE TESTS

ACTIVITY

ID	DESCRIPTION
A	Given an indication of the level of testing (instrumentation) required and on what "device" the test is to take place, this activity should determine 1) which instruments are appropriate for the level, 2) which instruments should be directly translated into HALMAT, and 3) which instruments should be handled by a monitor external to the HALMAT. The implication is that if testing is to be performed on an instrumentable ICS, certain of the monitors should not be inserted in the HALMAT but rather relegated to the ICS instrumentation capability. This then requires that this activity be knowledgable of the capabilities of the various ICS's. A table may be appropriate here, with the rows listing the classes of instruments and the columns corresponding to different ICS's. Each table entry indicates if the given instrumention can be done by the particular ICS. As a new ICS is added to the environment the table has a new column added to it. See Section 3.6.7 for a full discussion of the mechanisms for controlling instrumentation.

RELATED DATA

ID	SOURCE	DEST	NAME

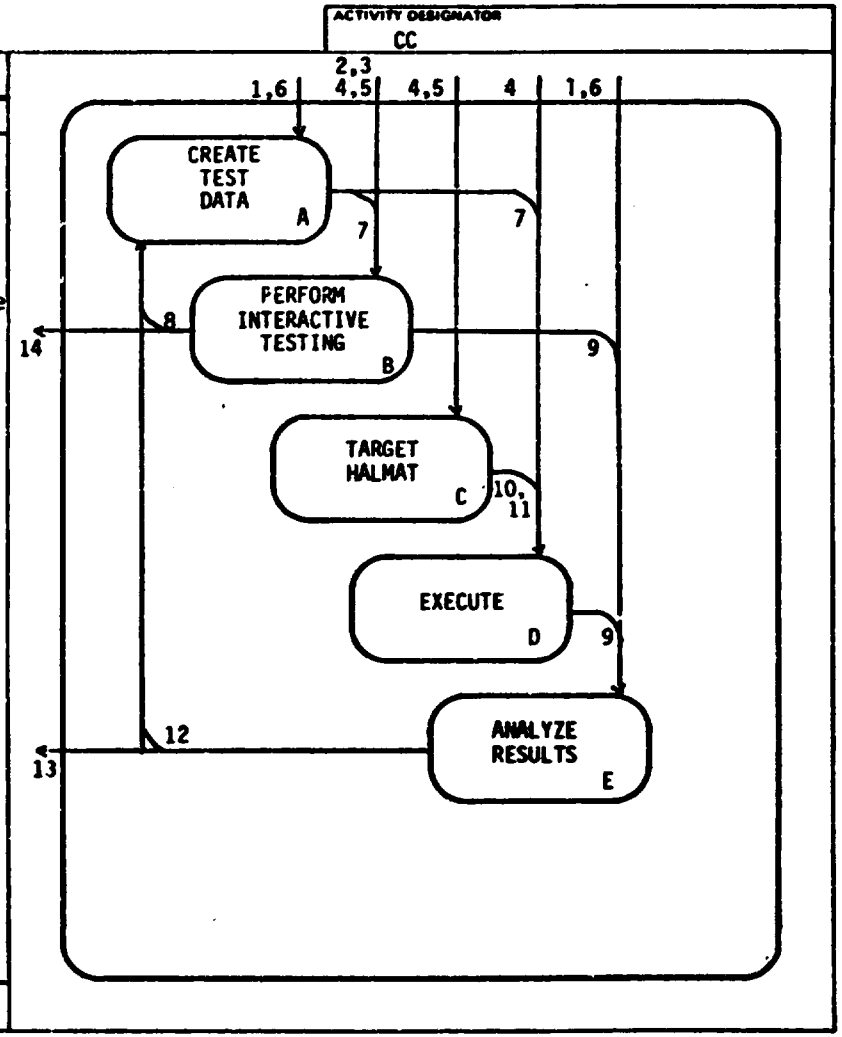
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

TITLE: TEST MODULE

DATA ID	TRACE	DATA DESCRIPTION
1	11	Managerial input
2	12	Suspect path specifications
3	13	Annotated program flowgraph
4	3	HALMAT (includes instrumentation)
5	14	List of instrumentation (including those instruments which should be handled by an ICS).
6	10	Module acceptance criteria
7		Test data
8		Generated test values
9		Output: Program output and monitoring data
10		Executable code
11		Mapping of variables to target machine "symbols". (Required, along with the symbol table contained in the HALMAT, for post-mortem dump generation)
12		Requirements for additional test data
13	4	Code revision specifications
14	15	Information indicating unncessary instrumentation.

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)
CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CC

TITLE Test Module

ACTIVITY

ID	DESCRIPTION
	Three aspects of this activity, nodes A, D, and E are the basic constituents of an automatic test harness. Several sets of test data/acceptance criteria may be supplied to it. Each case will be executed and the results automatically checked for correctiness. Such an apparatus is especially useful during retesting which is required as the result of program modifications.
C	Given HALMAT and a specification of the desired target machine this activity will generate executable code, produce a load map, and perform any static checking required at the target machine level. Thus many HALSTAT - type functions are included here.

RELATED DATA

ID	SOURCE	DEST	NAME

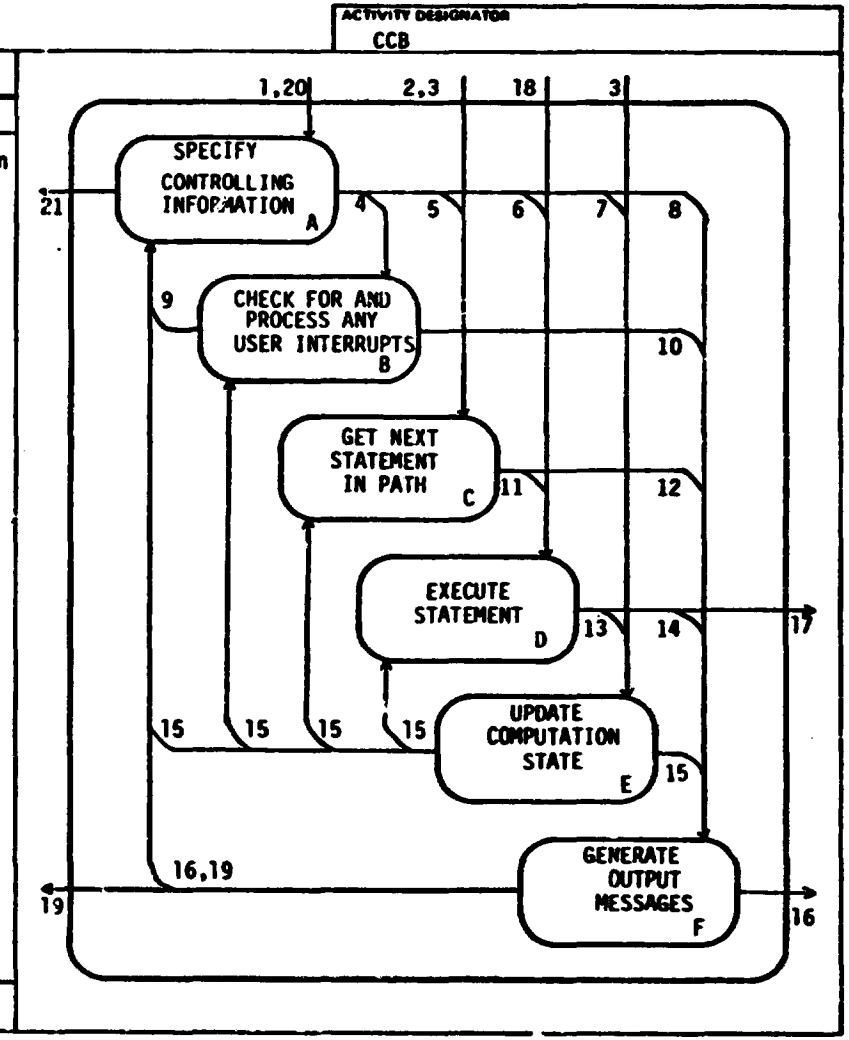
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

TITLE: PERFORM INTERACTIVE TESTING

DATA ID	TRACE	DATA DESCRIPTION
1	2	(possibly incomplete) path specification from static analyzer
2	3	Program flowgraph
3	4	HALMAT (incl. symbol table)
4		User interrupt control tables
5		Execution scope
6		Path specification list.
7		Variable and state initializations
8		Output/message control information.
9		Control commands
10		Interrupt messages (e.g. BREAK "message";)
11		Statement to execute.
12		Execution scope exit.
13		Statement pointer, variable, and path condition updates.
14		Path specification request or error/ data value request.
15		Computation state.
16	9	Output/error messages.
17	9	Program output.
18	7	Test data.
19	8	Generated test values.

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE



REF. DOCUMENT 30167 (SAMM)

CO 1000 1015 ORIG. 2/78

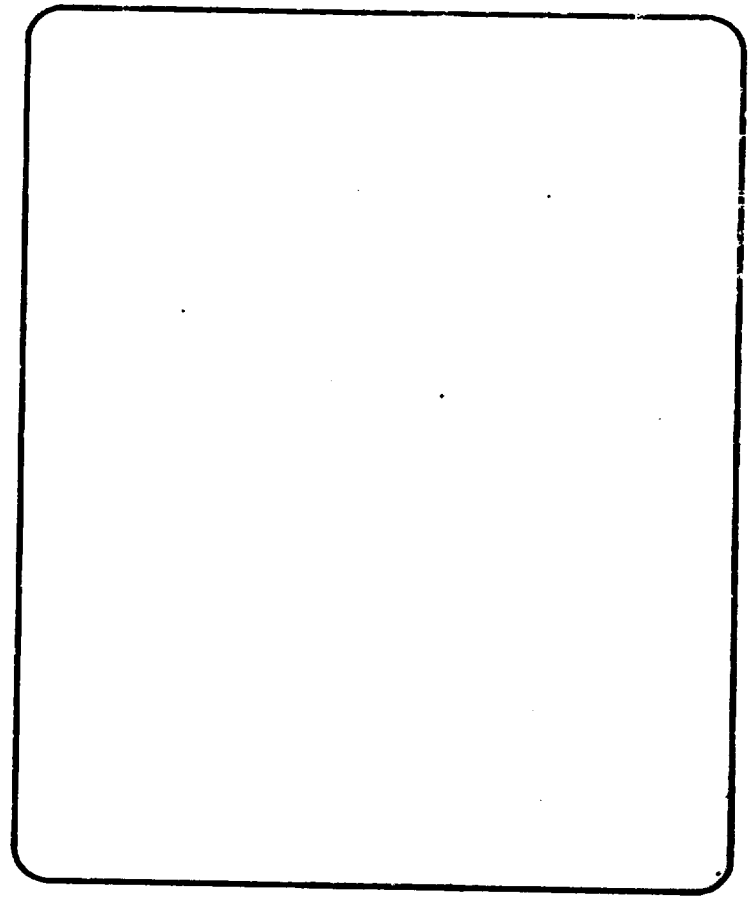
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCB (continued)

TITLE:

DATA ID	TRACE	DATA DESCRIPTION
20	5	List of instrumentation.
21	14	Information indicating unnecessary instrumentation.



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 80107 (SAMM)

CO 1000 1015 ORIG. 2/70

ACTIVITY DESCRIPTIONS

NODE CCB

TITLE PERFORM INTERACTIVE TESTING

ACTIVITY

ID	DESCRIPTION
B	This activity is entered before execution of every statement to check for any user interrupts which must be processed.
E	This activity will be composed of nothing more than the primitive operations which interface to the symbol table and the data space.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS

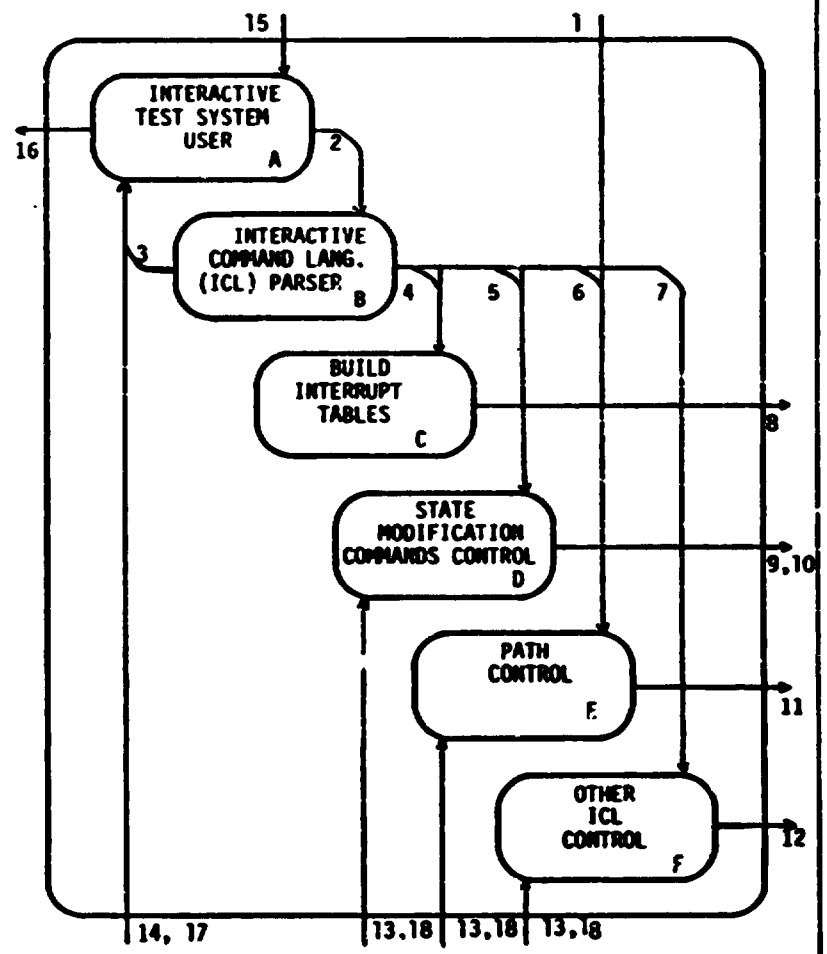
ACTIVITY - DATA FLOW DIAGRAM

TITLE: SPECIFY CONTROLLING INFORMATION

ACTIVITY DESIGNATOR: CCBA

DATA ID	TRACE	DATA DESCRIPTION
1	1	(possibly incomplete) path specification from static analyzers.
2		Interactive Command Language (ICL) statements.
3		ICL error messages.
4		Interrupt control commands (e.g. BREAK, AFTER, etc.)
5		State modification commands (e.g. SET, SKIP, etc.)
6		Path control commands (SELECT)
7		All other ICL commands.
8	4	Interrupt control tables.
9	5	Execution scope.
10	7	Computation state updates.
11	6	Path specification list.
12	8	Output message control.
13	15	Computation state
14	16	Output/error messages
15	20	List of instrumentation
16	21	List of instruments which are necessary.
17	19	Generated test values
18	9	Control commands

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/70

ACTIVITY DESCRIPTIONS

NODE CCBA

TITLE SPECIFY CONTROLLING INFORMATION

ACTIVITY ID	DESCRIPTION
F	Path constraint solutions will be performed within this activity as directed by the user (DISPLAY SOLUTION). Any test values that can be generated will be output. If there are solutions which cannot be determined then any test values which are found will be used to simplify the remaining constraints and the resulting path condition will also be output.

RELATED DATA ID	SOURCE	DEST	NAME

BGS BOEING COMPUTER SERVICES COMPANY

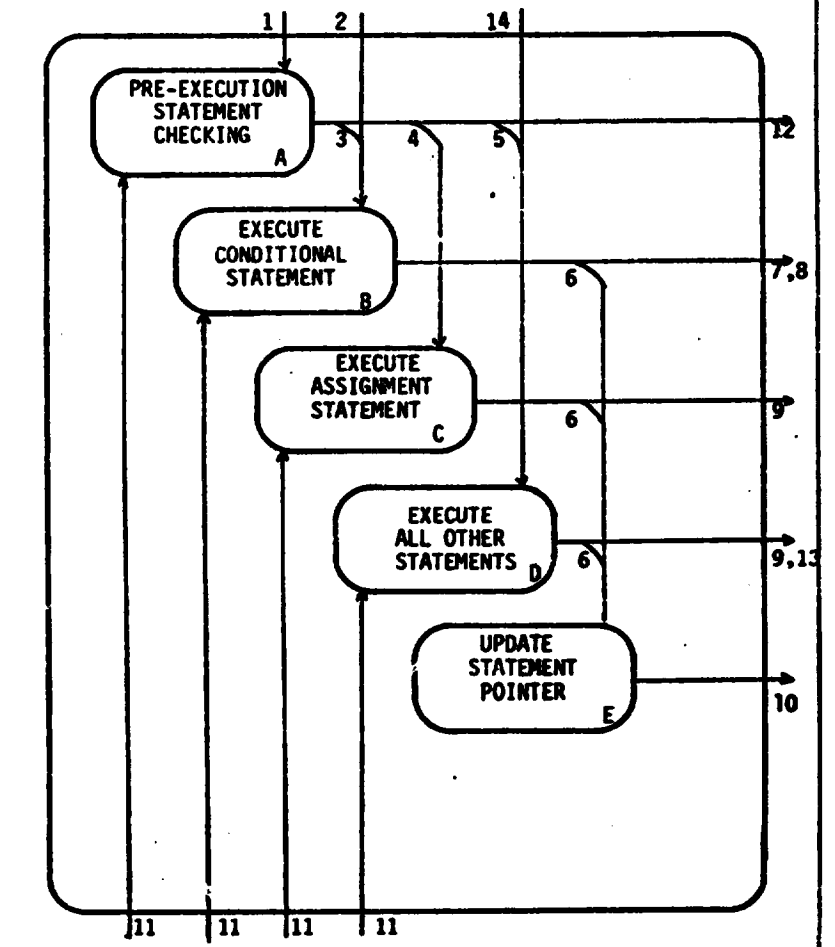
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCBD

TITLE: EXECUTE STATEMENT

DATA ID	TRACE	DATA DESCRIPTION
1	11	Statement to execute
2	6	Path Specification list
3		Conditional Statement (e.g. IF, DO WHILE, ect.)
4		Assignment statement
5		All other HAL/S statements (e.g. READ, CALL, etc.)
6		Statement pointer update
7	14	Path branch request/error
8	13	Modified path condition
9	13	Variable updates
10	13	Updated statement pointer
11	15	Computation state
12	14	Undefined variable assignment req.
13	17	Program output
14	18	Input test data

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 3/78

C-3

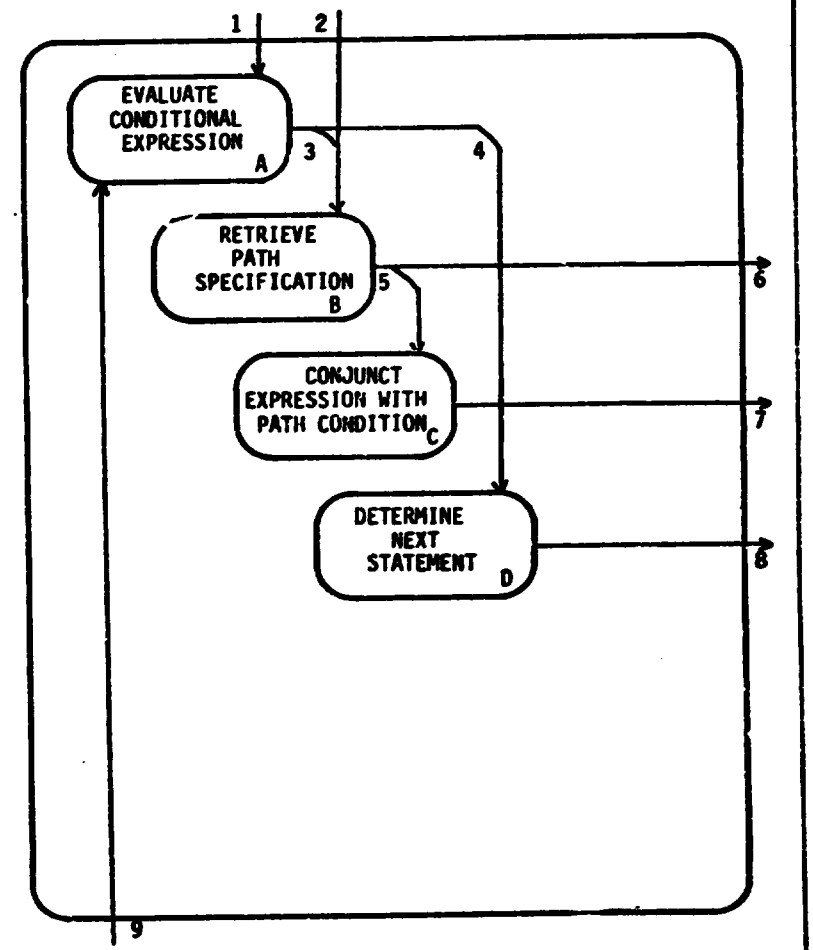
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CCBDB

TITLE
EXECUTE CONDITIONAL STATEMENT

DATA ID	TRACE	DATA DESCRIPTION
1	3	Conditional statement
2	2	Path specification list
3		Statement pointer, conditional expression
4		Statement type, expression evaluation result
5		Branch selection, symbolic conditional expression
6	7	Path selection request/error
7	8	Path condition
8	6	Statement pointer
9	11	Computation state



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)
CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CCBDB

TITLE EXECUTE CONDITIONAL STATEMENT

ACTIVITY

ID	DESCRIPTION
A	Expression evaluation will proceed normally when all values referenced in the expression are actual values. If the expression contains symbolic values then an attempt will be made to prove: path condition→expression. If this cannot be accomplished then activity B will be invoked.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS BOEING COMPUTER SERVICES COMPANY

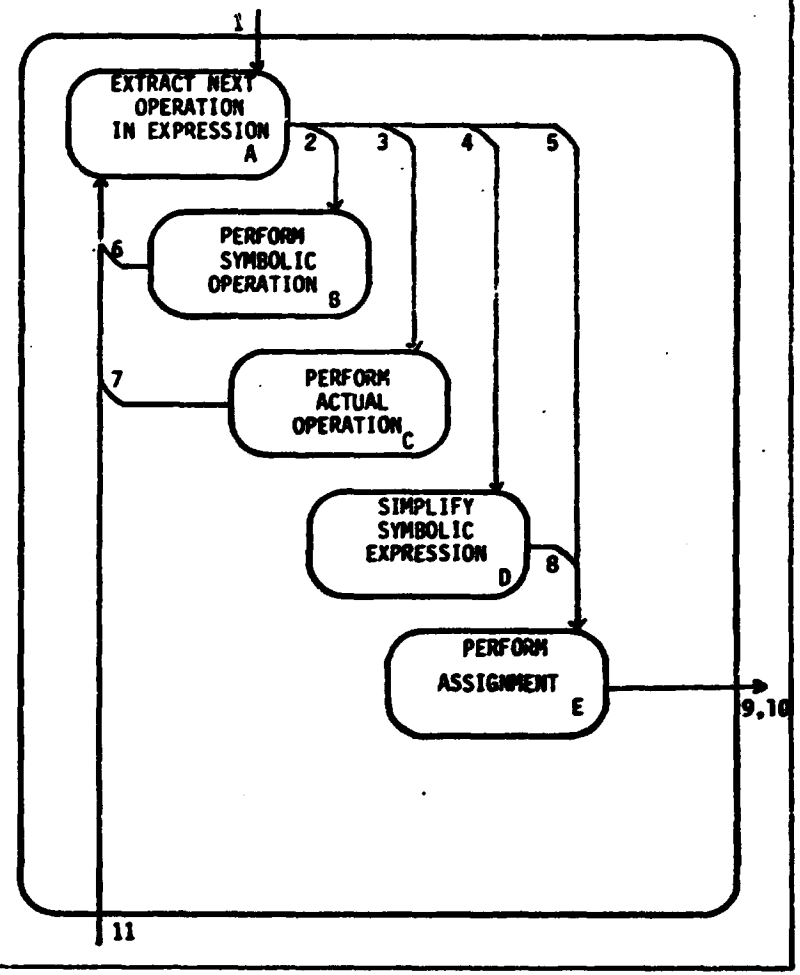
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCBDC

TITLE: EXECUTE ASSIGNMENT STATEMENT

DATA ID	TRACE	DATA DESCRIPTION
1	4	Assignment statement
2		Symbolic operands and operation
3		Actual operands and operation
4		Final (unsimplified) symbolic result
5		Final actual result
6		Intermediate symbolic result
7		Intermediate actual result
8		Simplified symbolic result
9	6	Statement pointer update
10	9	Variable update(s)
11	11	Computation state

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)
CO 1000 1016 ORIG. 2/76

ACTIVITY DESCRIPTIONS

NODE CCBDC

TITLE EXECUTE ASSIGNMENT STATEMENT

ACTIVITY

ID	DESCRIPTION
B	This activity will perform arithmetic operations involving one or more symbolic variables with the result being symbolic. Each type of operation will be a primitive function.
C	This activity will perform all operations involving variables none of which contain symbolic values. The type of operation to be performed will be determined by the attributes associated with the operands, the operator, and the target computer. For example: integer, scalar, fixed point (if available), matrix, and vector addition will all be separate primitive functions and, in addition, there may be more than one routine for each (to handle the different architectures of several target computers).

RELATED DATA

ID	SOURCE	DEST	NAME

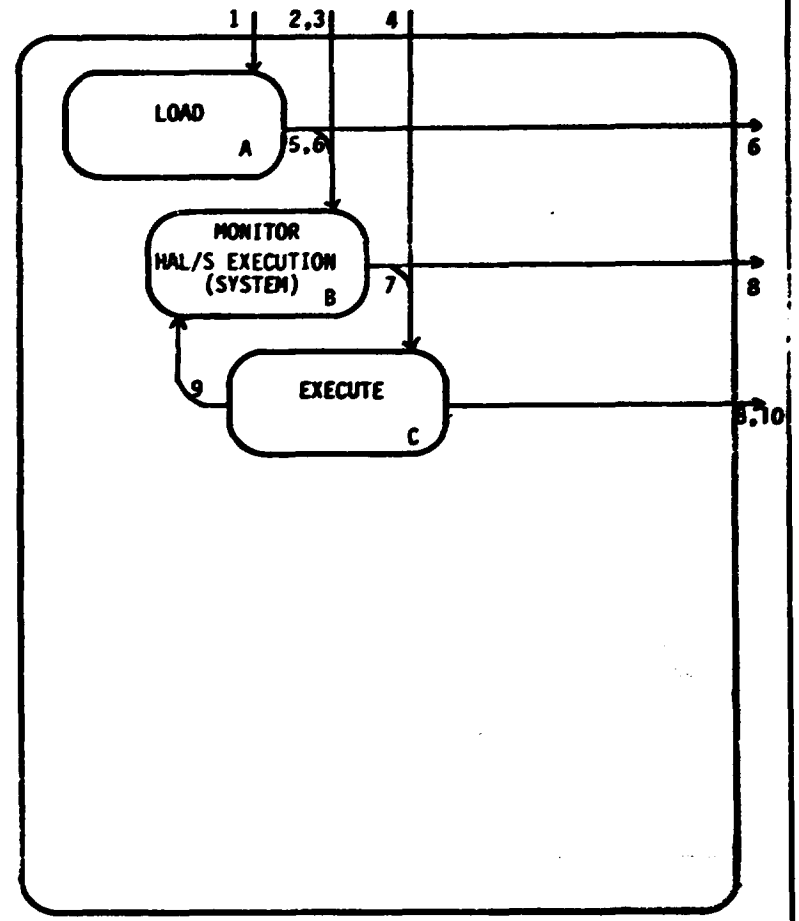
BGS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCD

TITLE: EXECUTE

DATA ID	TRACE	DATA DESCRIPTION
1	10	Executable code ("relocatable binary")
2	4	Symbol table (from HALMAT)
3	11	Mapping at symbol table to target machine symbols.
4	7	Test data
5		Load image
6	9	Load output - documentation
7		System control over executing program
8	9	Raw performance/histogram/history data/post-mortem dump
9		Calls to system monitor
10	9	Program output



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT NO(S) (SAMM)

CO 1000 1015 ORIG. 2/78

BGS — Boeing Computer Services Company

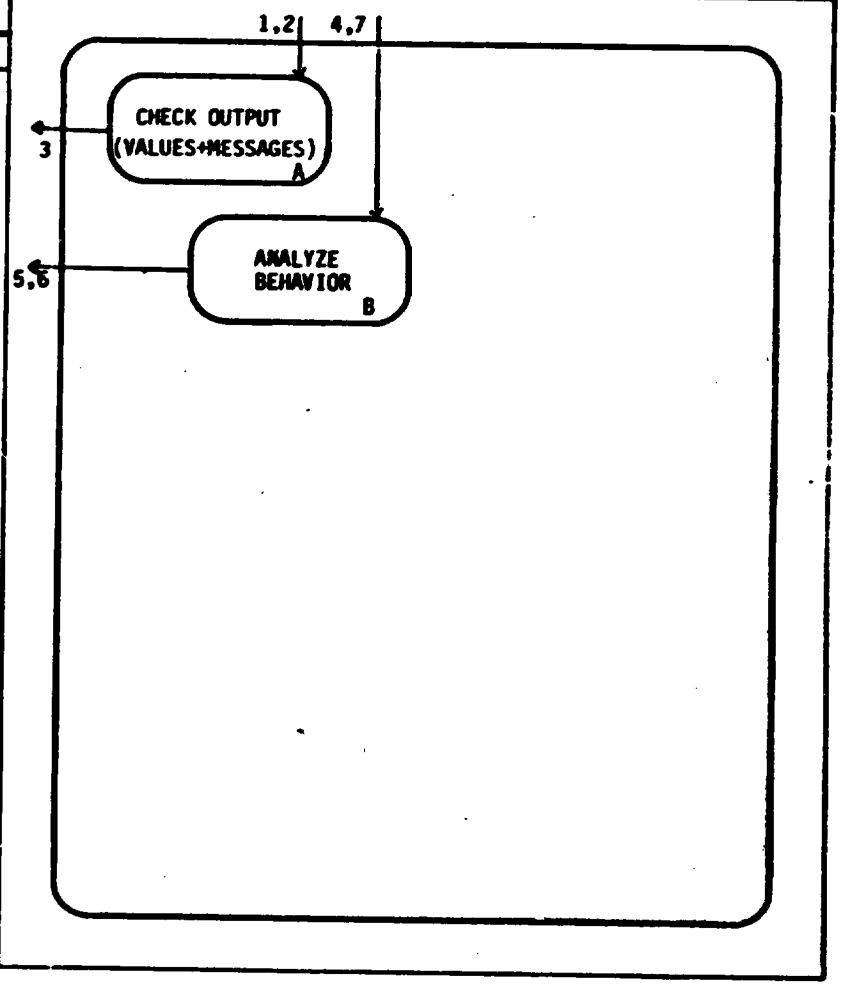
ACTIVITY - DATA FLOW DIAGRAM

TITLE: ANALYZE RESULTS

ACTIVITY DESIGNATOR: CCE

DATA ID	TRACE	DATA DESCRIPTION
1	9	Output values and messages
2	6	Acceptance criteria
3	13	Code revision specifications
4	9	Behavioral information
5	13	Code revision specifications
6	12	Additional testing requirements
7	1	Managerial input

PREPARED BY / DATE / REVIEWED BY / DATE / APPROVED BY / DATE



BCS DOCUMENT 10167 (SAMM)
CO 1888 1016 ORIG. 2/78

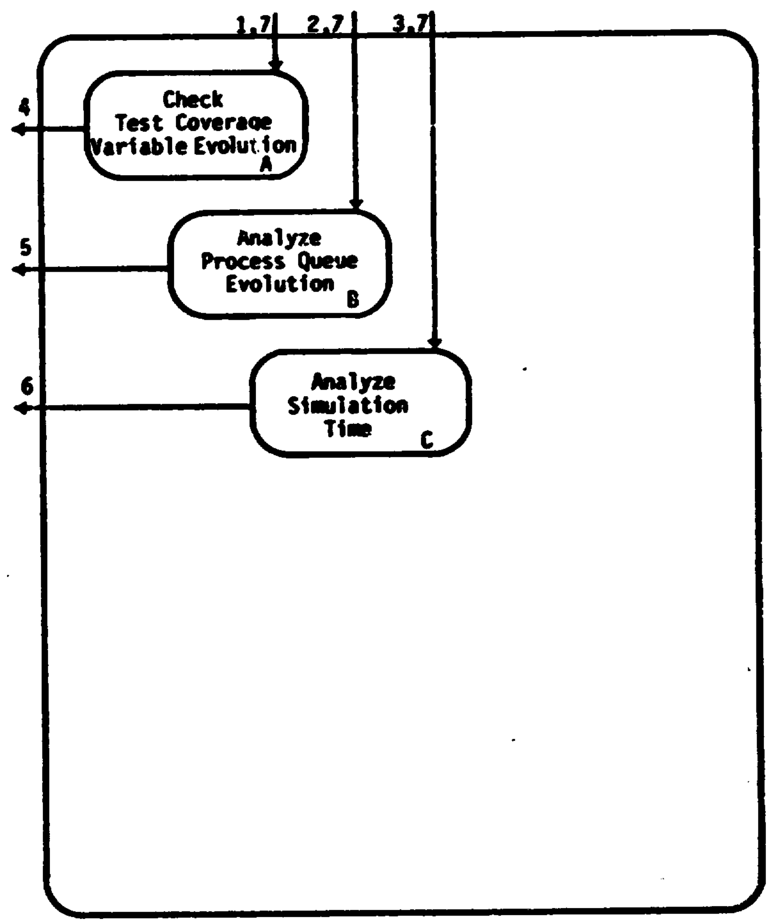
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCEB

TITLE: ANALYZE BEHAVIOR (PERFORMANCE)

DATA ID	TRACE	DATA DESCRIPTION
1	4	Histograms (frequency count, branch paths, statistics) (raw data)
2	4	Process queue snapshots (raw)
3	4	Simulated time output (raw)
4	6	Additional testing requirements
5	5	Code revision requirements - parallel processing schedules
6	5	Code revision (optimization) requirements
7	7	Managerial input



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10107 (SAMM)

CO 1000 1016 ORIG. 2/78

BGS — BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCEBA

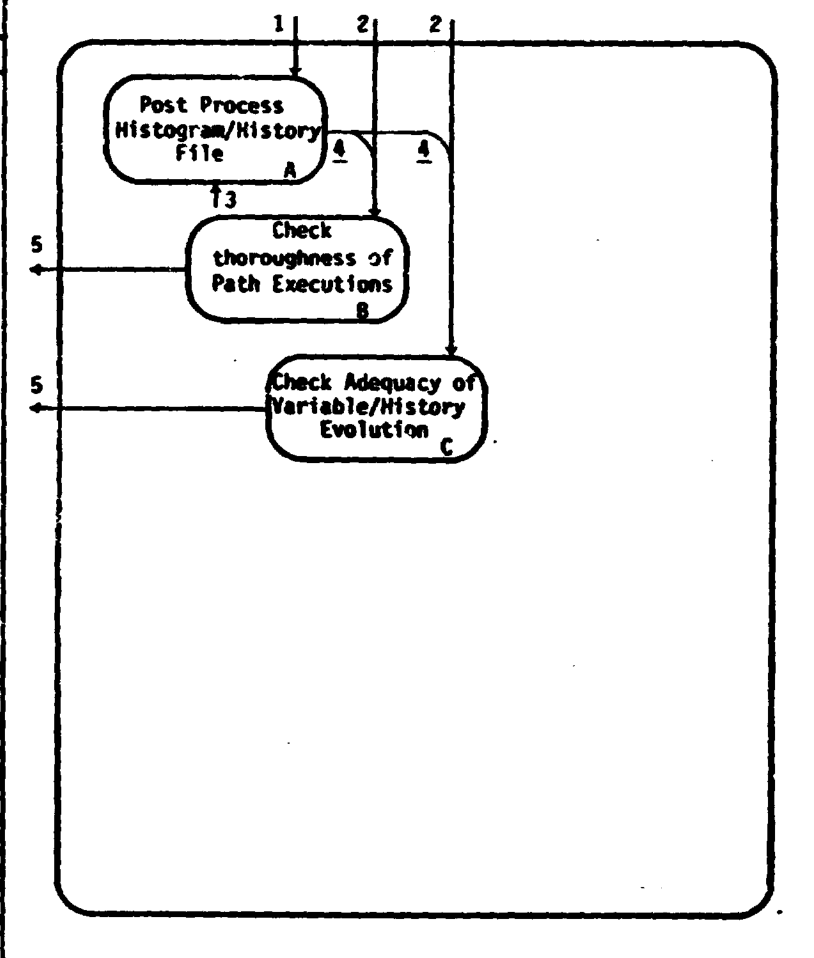
TITLE: CHECK TEST COVERAGE, VARIABLE EVOLUTION

DATA ID	TRACE	DATA DESCRIPTION
1	1	Raw histogram/history information
2	7	Managerial input
3	db	Program source listing
4		Annotated source listing
5	4	Requirements for additional tests

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10107 (SAMM)

CO 1000 1016 ORIG. 3/78



ACTIVITY DESCRIPTIONS

NODE CCEBA

TITLE CHECK TEST COVERAGE, VARIABLE EVOLUTION

ACTIVITY		RELATED DATA			
ID	DESCRIPTION	ID	SOURCE	DEST	NAME
A	See Section 3.7 of this document for a discussion of the output writer. It incarnation here is only one possible point of usage.				

BCS

ACTIVITY - DATA FLOW DIAGRAM

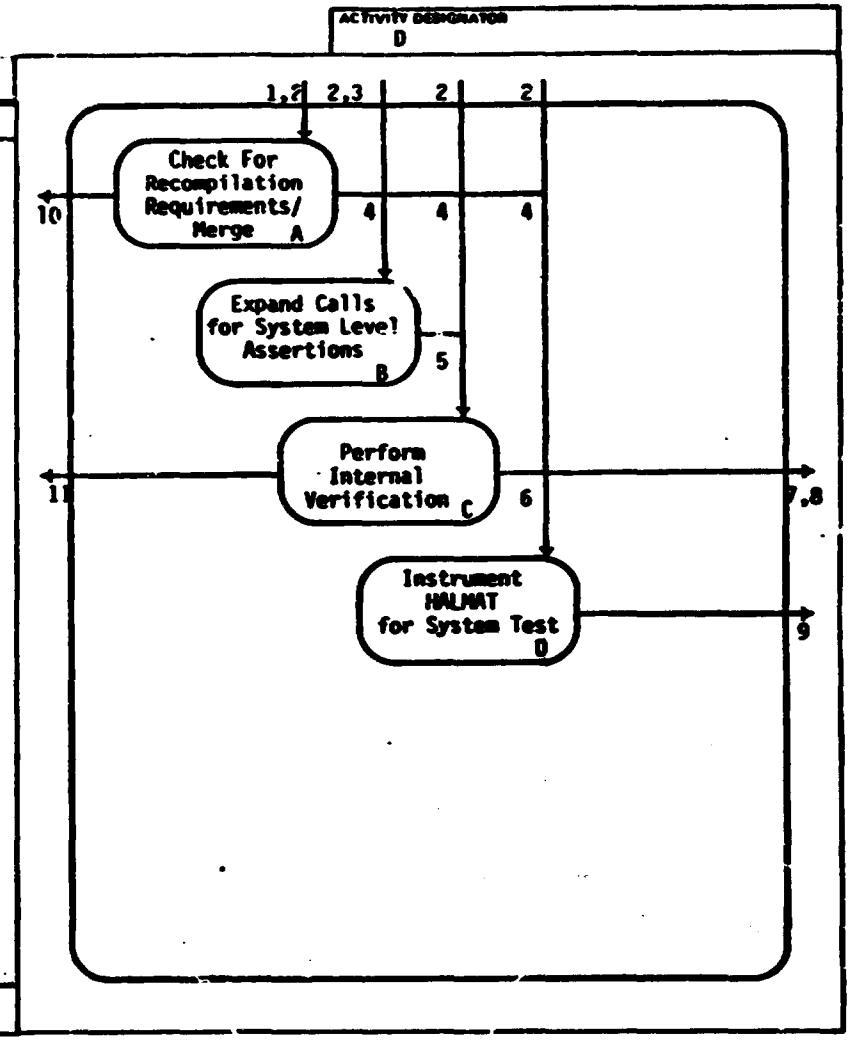
TITLE: INTEGRATE MODULES INTO SYSTEM

DATA ID	TRACE	DATA DESCRIPTION
1	6	HALMAT of each module
2	12	Managerial input/controlling parameters
3	6	HALMAT monitor files
4		Collection of modules forming system
5		Expanded HALMAT Monitor file
6		Further altered HALMAT Monitor file
7	15	Specification of system paths along which errors are suspected to exist
8	16	System flowgraph
9	8	HALMAT ready for execution as a complete system
10	7	Recompilation requirements due to factors such as "independent testing used outdated module templates"
11	7	Verification based revision requirements

PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT 80167 (SAMM)

CO 1000 1015 ORIG. 2/78



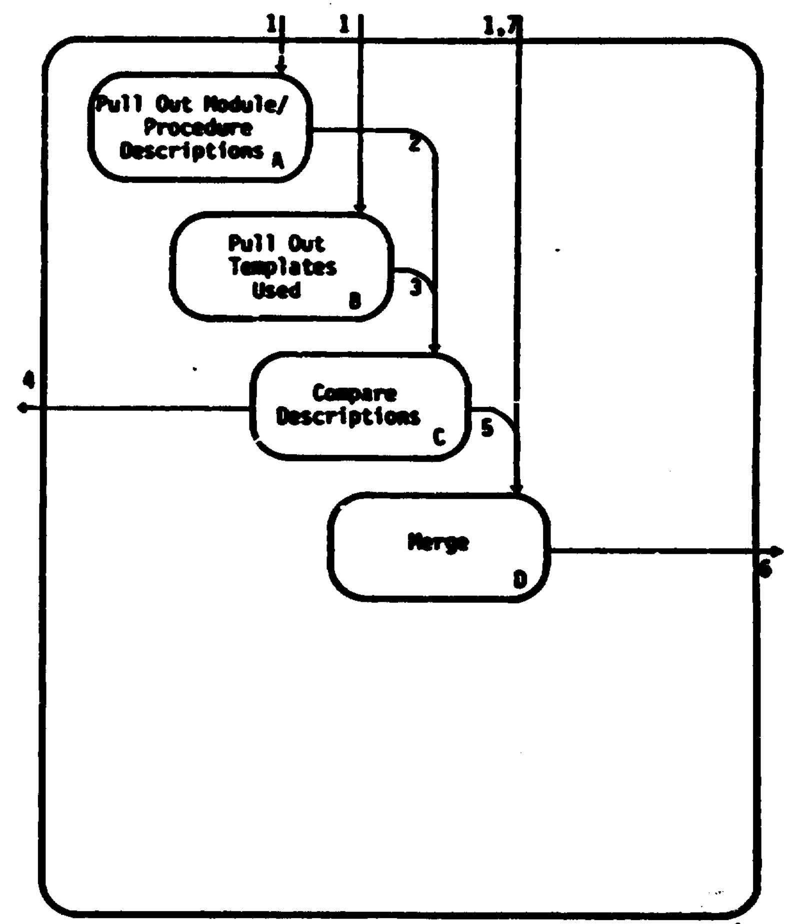
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATION: DA

TITLE: CHECK FOR RECOMPILATION REQUIREMENTS/MERGE

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT for each module
2		Procedure template "equivalents" (true description of procedures)
3		Procedure templates
4	10	List of mismatched procedure/templates => recompilation/revision requirements
5		List of modules which may be safely merged
6	4	System of legally merged modules
7	2	Directions as to merger (e.g., override) - may be unnecessary <u>or undesirable</u>



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

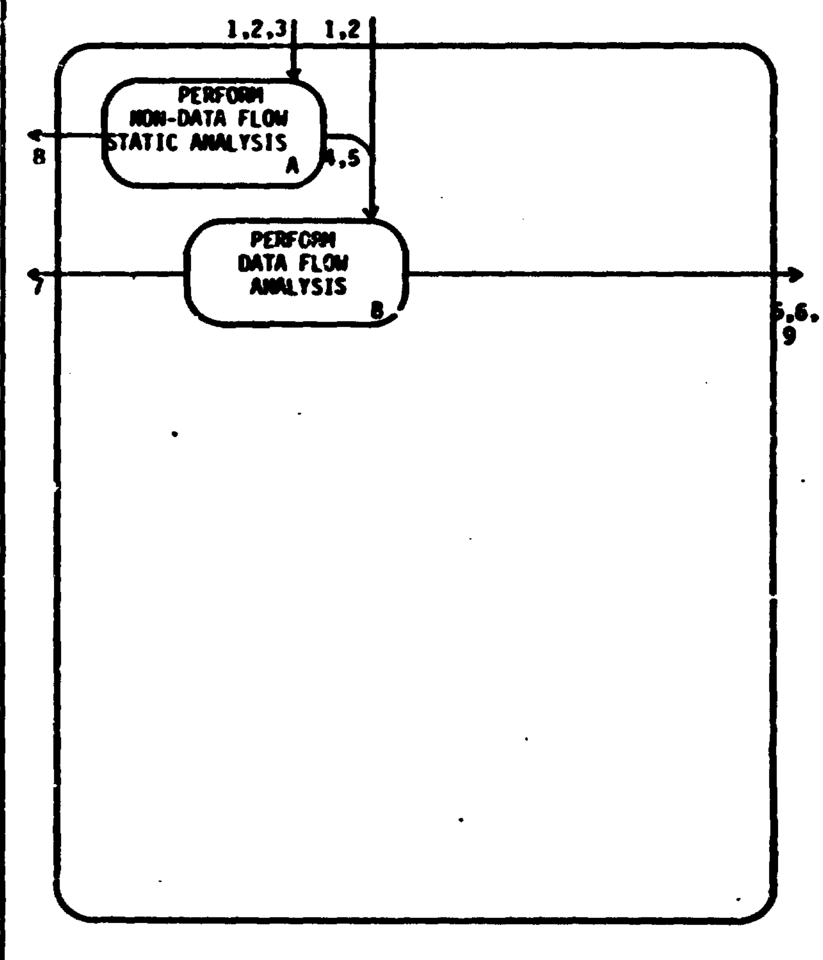
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: DC

TITLE: PERFORM INTERNAL VERIFICATION

DATA ID	TRACE	DATA DESCRIPTION
1	4	HALMAT for all modules
2	2	Managerial input
3	5	HALMAT Monitor File
4		System call graph
5	6	Refined HALMAT Monitor File
6	7	Specifications of system paths on which errors are suspected to exist
7	11	Error messages and documentation
8	11	Comprising revision requirements
9	8	System flowgraph



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT (SAMM)

CD 1000 1016 ORIG. 2/76

ORIGINAL PAGE IS OF POOR QUALITY

ACTIVITY DESCRIPTIONS

NODE DC

TITLE PERFORM INTERNAL VERIFICATION

ACTIVITY

ID	DESCRIPTION

The structure and purpose of this activity closely resembles that of node CBCC. The further breakdown of this node and its related data items will closely follow that of CBCC.

RELATED DATA

ID	SOURCE	DEST	NAME

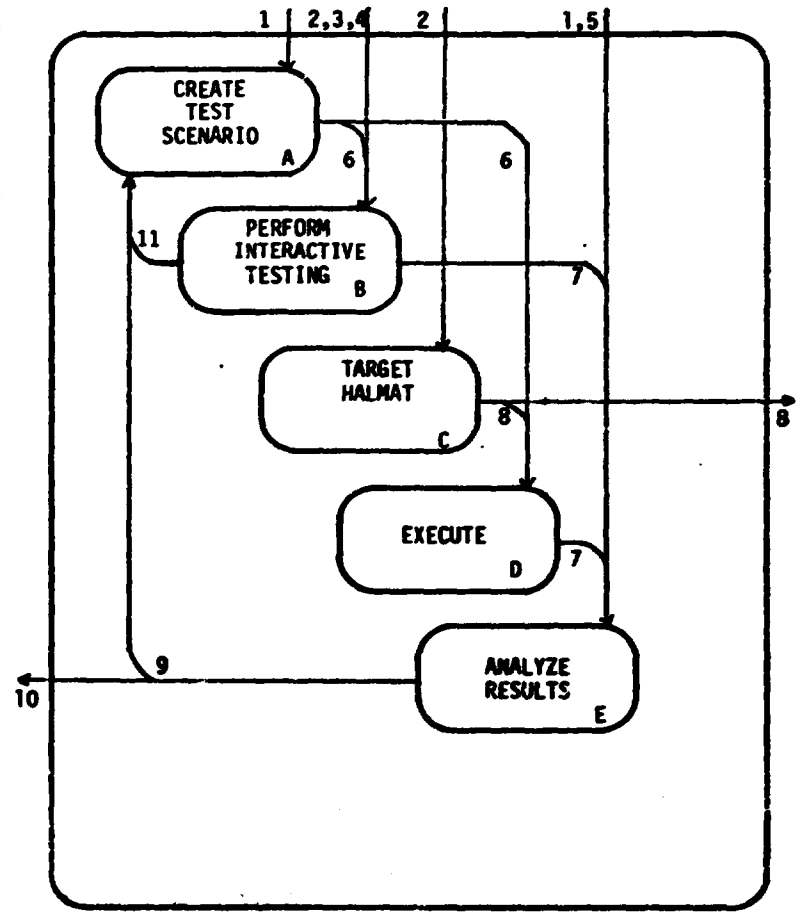
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: E

TITLE: TARGET AND TEST SYSTEM

DATA ID	TRACE	DATA DESCRIPTION
1	13	System acceptance criteria
2	8	HALMAT Comprising System
3	15	Suspect system path specifications
4	16	System flowgraph
5	12	Managerial input
6		Test data
7		Output: program output & monitoring information
8	14	Executable code
9		Requirements for additional test data
10	9	Module revision requirements
11		Generated test values



PREPARED BY | DATE | REVIEWED BY | DATE | APPROVED BY | DATE

REF. DOCUMENT 10167 (SAMM)

CO 1009 1016 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE E

TITLE TARGET AND TEST SYSTEM

ACTIVITY

ID	DESCRIPTION

Further decompostion of the activities and data items associated with this node will closely parallel that with node CC, Module Test. See that node for details.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS BOEING COMPUTER SERVICES COMPANY
A Division of The Boeing Company

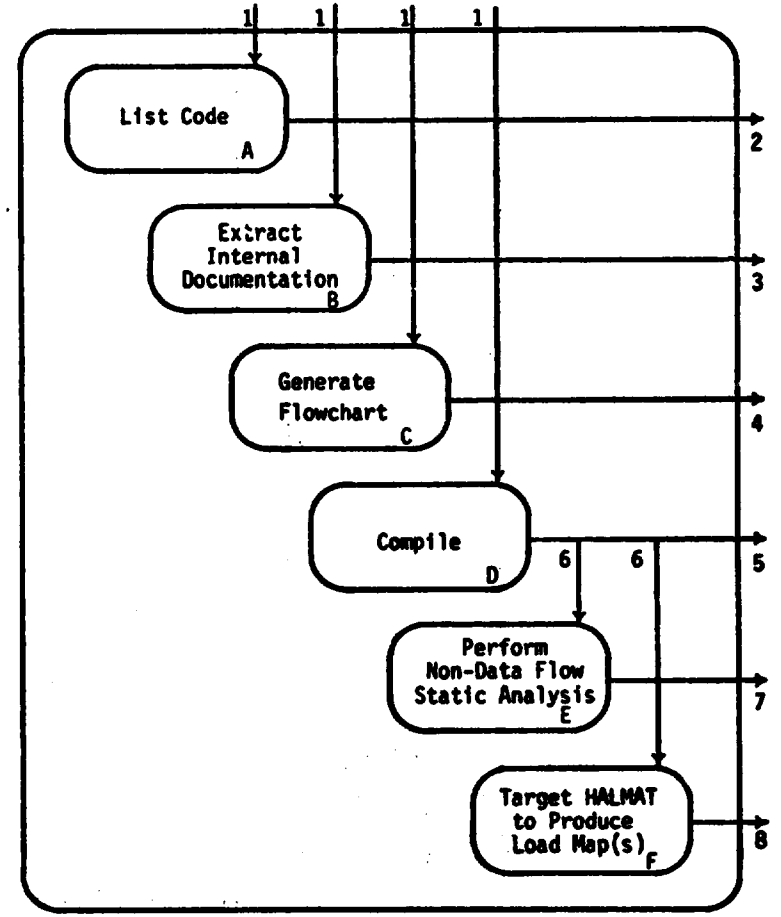
ACTIVITY - DATA FLOW DIAGRAM

TITLE: DOCUMENT EXISTING SYSTEM

DATA ID	TRACE	DATA DESCRIPTION
1		Source code
2		Listing
3		Internal documentation
4		Flowchart
5		Compiler documentation
6		Augmented HALMAT
7		Static analysis information
8		Load map(s)

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

ACTIVITY DESIGNATOR
ROOT



REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 2/76

ACTIVITY DESCRIPTIONS

NODE Root

TITLE Document Existing System

ACTIVITY

ID	DESCRIPTION
B	This activity will extract specially marked comments (such as /** or cc) which comprise various levels of internal program documentation. Several levels of documentation may exist- system, task, procedure, and local. If conventions are adopted regarding the formatting of the various levels, under user control this activity will extract and print the desired comments. A more sophisticated capability could be designed which would, for example, print the comments and transfer conditions associated with a particular path through a program. The path specification would be the same format as used for the timing estimate and the symbolic executor. See Section 3.7 for a discussion of the output writer.

RELATED DATA

ID	SOURCE	DEST	NAME

Appendix E

INTEGRATED TESTING AND VERIFICATION SYSTEM FOR RESEARCH FLIGHT SOFTWARE

REQUIREMENTS DOCUMENT

Contract Number NASI - 15253

May 1978

Prepared by:

Boeing Computer Services Company
Space & Military Applications Division
P.O. Box 24346
Seattle, Washington 98124

PRECEDING PAGE BLANK NOT FILMED

Appendix E: Requirements Document

CONTENTS

CONTENTS (Continued)

Appendix E: Requirements Document

1.0 PROBLEM STATEMENT

The production of reliable software is in general, a difficult, slow, and expensive process. Tools and methodologies addressing this issue are recent, often fragmentary, and restricted in scope and applicability. Production of reliable flight software is more difficult yet, as real time and multi-task requirements compound the problem. Advanced tools are required to aid in the timely production of reliable, real time, flight systems.

2.0 GOAL

The study's goal is to benefit the NASA researcher by designing a unified set of automated tools within the MUST programming environment to aid in the documentation, verification, and testing of flight software.

3.0 AUDIENCE

3.1 Programmers. The capabilities provided by the verification system will be of greatest utility to programmers writing the flight software. All capabilities will be of interest.

3.2 Program Managers. Program managers will primarily be interested in aspects of the documentation produced, though the generic verification capabilities will be of interest as well, as they may in principle be applied to requirements and design analysis. This latter ability is not considered fundamental to the problem at hand, but the algorithms employed by this work will be directly applicable in the verification of a suitable specification language.

Documentation features of interest include statistics charting a program's execution history and an indication of coding practices employed in terms of some predefined parameters.

4.0 ENVIRONMENT

Several environmental considerations will affect the design of the verification and testing system. First are the characteristics of the user community. Second are the general characteristics of research flight hardware and software. Third are the characteristics of the MUST program and its constituents.

4.1 User Community.

4.1.1 Problem Orientation. The users of the MUST system are researchers devoted to addressing particular NASA problems. As engineers and programmers they are familiar with computing concepts and may effectively use sophisticated tools without extensive "handholding."

4.1.2 Interactive and Batch Operation. Most users will heavily utilize the interactive features of MUST; thus the verification and testing capability should be oriented this way. Batch usage is still preferred by some, however, so the capabilities must be effectively usable in both modes.

4.2 Research Flight Hardware and Software.

4.2.1 Hardware.

4.2.1.1 Flight Computers. Flight computers tend to be small, one-of-a-kind machines, though more advanced machines are appearing. They often have little supporting software and place tight space and time constraints on applications programs. Assembly language coding is most often the rule and absolute patches are by no means unknown. Floating point features are often absent, or if present, unacceptably slow. Thus the use of hardware real arithmetic is often circumvented by software fixed point computations which invite scaling and precision errors.

4.2.1.2 Ground Based System. Large general purpose computer systems are available to NASA researchers for ground based support. MUST is hosted on such a system (a large CDC machine supporting the programming language Pascal).

4.2.2 Software.

4.2.2.1 Research Orientation. Since the subject software is research oriented, rapid evolution is common, with the attendant requirement of constantly updated documentation. Further, rapid evolution requires the rapid production of correct code. Often a multidisciplinary team of researchers will address a single problem. Utilizing the verification and testing capabilities should aid in the smooth integration of independently produced pieces of software.

4.2.2.2 Real Time Constraints. Supporting flight operations requires the software to operate within strict real time bounds. For example, on board equipment may produce a signal which must be processed ten time a second. Many such constraints may reside with a large system, requiring complex scheduling of functions.

4.2.2.3 Parallel Operations and Data Pools. In response to real time constraints, or for logical clarity, a system may be constructed with several independent, possibly parallel, modules accessing a common data base. A typical model might involve navigation, guidance, and display modules, while the data base would contain global parameters, such as position, attitude, and speed. Programming concerns would include data base consistency and proper ordering of module executions, as each module needs the guarantee that the data base is fully updated when accessed, and that necessary information is present and correct. The actual implementation of such a system may involve a single processor being time shared among the modules, or each module executing on a separate processor.

4.3 MUST. The above considerations have, of course, been the motivational and guiding forces in the design of the MUST programming environment, in which this verification and testing system will be imbedded. Its important components are described below.

4.3.1 Interactive Software Invocation System - ISIS. The use of ISIS as the primary user interface, invoking tools and managing data, makes it an important point of integration. Since the user sees MUST, and therefore the verification and testing capability, through ISIS, the design and use of the system must be consistent with the ISIS philosophy, presenting no implementation or invocation peculiarities. The output produced by various aspects of the verification and testing facility will be entered, for example, as books in an ISIS library.

The relational data base capabilities of ISIS may prove to be especially useful in holding representations of a user program. As futher descriptions of ISIS become available, this will require investigation.

4.3.2 HAL/S.

4.3.2.1 General Charactersitics. The HAL/S language is by far the largest environmental concern. As the prime programming language of the MUST environment, the verification and testing system will be closely focussed on it. Particular attention will be paid to the real time features of HAL/S, as real time issues and shared data pools are critical in flight software, as noted above. These general characteristics will most profitably be addressed within the specific semantics of the HAL/S language and run time environment, yet the algorithms developed and used will be general in character. This is a natural approach, but is especially important in view of the fact that the new Department of Defense programming language may be adapted for NASA use within a few years. The NASA standard version of HAL/S is used by MUST. Any language additions or alterations will require coordination with the language standard control group.

4.3.2.2 Language Richness. The HAL/S language is quite rich in programming constructs - perhaps too rich. Several constructs have somewhat awkward semantics, and special cases are frequent. Examination of the language features will be necessary, therefore, to see if any pose particularly difficult problems for the verification and testing capability, such as adversely affecting the detection of certain classes of errors. On this basis a decision will be made as to whether the verification capability should take cognizance of those identified features. An example in this category is the NAME facility. Either spurious error messages will be generated or some error phenomena may be missed if names are used without restraint. The problem is one of aliasing, and no satisfactory solution yet exists.

4.3.2.3 Implementation Dependent Features. Several language limitations and operations are implementation defined, such as the exact operation of the real time executive. Implications of this when concerned with the validity of identical HAL/S programs running on different machines will be examined.

4.3.3 HAL/S Compiler System.

4.3.3.1 Checking and Documentation. Some checking and documentation features exist as normal parts of the compiler. Unless there is strong reason to act otherwise, these capabilities should be retained and not duplicated. As an example, the Symbol and Cross Reference Table lists all points where variables are referenced.

4.3.3.2 HALMAT. The HAL/S compiler systems produce a fairly high level intermediate language, HALMAT. This language may well be suitable as a primary input to the capability, allowing most of the verification and testing functions to be separate from the verification and testing compiler internals, but still utilizing the compiler's syntax analysis capabilities. HALMAT currently has several unused operation codes which may be utilized by the verification capability to communicate new high level "statements," such as assert, to the analysis modules. Doing so should require only minor changes to the compiler.

4.3.3.3 Pascal Implementation. The portion of the compiler which generates HALMAT has been translated by NASA-Langley from the original XPL/360 version to CDC Pascal. Comprehensive documentation is available for this implementation.

4.3.3.4 Functional Simulation - FSIM. Though not a part of the NASA Pascal based HAL/S compiler, FSIM is available on some Intermetrics compilers. Some of its features, such as provision of an execution time estimate, seem quite useful. FSIM's full capabilities will be examined to see if it should be interfaced with the verification and testing facility, or if perhaps the most valuable features should be made a part of the verification features directly.

4.3.4 Documentation Capabilities.

4.3.4.1 RNF. RNF is the Pascal based text processing system used by MUST. RNF provides extensive features for formatting text into justified paragraphs, pages, lists, and so forth. A simple macro facility is also included.

4.3.4.2 Graphical Code Representation. A Pascal based facility provides another component of the documentation system. Given a description of (almost) any programming language and a program written in that language, the system will produce a structured flowchart of that program. Some interface/modification of this system may be required, if, for example, assertions or unit specifications are to appear in the diagrams.

4.3.5 Meta - Assembler. MUST's meta-assembler is a facility which might allow HALMAT to be targeted to several different computers. Verification and testing functions which are closely tied to specific implementations may require interface with the meta-assembler, or possibly knowledge of what the meta-assembler actually produces.

4.3.6 Interpretive Computer Simulator. This system allows a bit-by-bit simulation of an actual target program to be run on the large computer hosting MUST. Some of the run time tests may be suitable for inclusion here, and statistics could be gathered from a simulation run. Further examination of the system's capabilities and potential will be required.

4.3.7 HALSTAT. Since in-line code and absolute patches may still be used in the MUST/HAL environment, cognizance should be taken of tools available to analyze the consistency of actual load modules. Such a tool, HALSTAT, has been produced by Intermetrics. In its current form it may not be suitable for direct inclusion in the system, but its capabilities bear close examination.

5.0 FUNCTIONAL CAPABILITIES

Functional capabilities can be broadly divided into the three categories of documentation, verification, and testing. This division is based upon the type of information produced, and not necessarily on the verification and testing methods used. Indeed, detection of certain types of errors may involve the interaction of several different verification and testing capabilities, or the use of existing tools, such as the compiler.

5.1 Documentation. Note that some capabilities here may already be provided by the compiler system; inclusion here is for completeness sake, and does not imply duplication.

5.1.1 A Cross Reference Map. This is a table which for every variable and label, shows the location and nature of every reference and definition. As such it should be a useful aid to debugging and desk checking, as well as a tool for standards checking.

HAL/S has a number of functional classes of variables. Special prominence shall be given to each of the classes below. Each of these specialized cross references is intended to focus attention on a different aspect of the program's structure and functioning. As such they should facilitate specialized debugging, testing and analysis of the program.

5.1.1.1 LOCK Group Variables. All variables of each LOCK group will be listed. For each variable there will be a list of the UPDATE blocks accessing the variable.

5.1.1.2 COMPOOL Variables. All variables of each COMPOOL will be listed. Points of reference and definition for each variable will be enumerated.

5.1.1.3 EVENT Variables. All accesses to each EVENT variable will be listed.

5.1.1.4 Unprotected Shared Data. Notation will be produced for all variables which are shared among processes, yet which do not belong to a LOCK group or a COMPOOL.

5.1.2 Implicit Type Conversions. Documentation will be produced to describe cases where operand types are not properly matched and are automatically coerced into matching. Often such coercions are not intended by the programmer and produce erroneous results. Hence this documentation is intended to call to the programmer's attention possible unexpected consequences of existing code.

5.1.3 Extraction of Internal Documentation. A facility for extracting imbedded commentary and reformatting it into external documentation will be supplied. Internal commentary may take the form of comment statements or assertions. The assertion capability is outlined in a later section of this document.

5.1.4 Process Dependency Documentation. A representation will be given indicating the dependencies of program and task processes. A dependent process may continue to exist only as long as its parent; if the parent terminates, so does the dependent, whether or not it is finished. As discussed later, this may cause errors. A clear statement of such dependencies will enable the programmer to be aware of all the process interrelationships.

5.1.5 Event Scheduling Statement Cross Reference. A table will be provided showing where all event scheduling statements appear in a body of program text. The event scheduling statements are: SCHEDULE, TERMINATE, WAIT, and CANCEL. SET, RESET, and SIGNAL may also be considered in this category. If a programmer or analyst is shown where all of these statements are located, it becomes easier to grasp and analyze the real time structures of the program. Thus this documentation should aid debugging, desk checking and test design.

5.1.6 Call Graph. A representation of the calling structure of the program will be given. This representation will show where each procedure is called, and what procedures are used within a given procedure.

5.1.7 Query Facility. A feature will be provided enabling the programmer to assess the impact of proposed coding changes, in the sense of knowing what modules/procedures will be affected by changing a given piece of code. This feature may also be used to determine what sections of code were executed in establishing the values of a given set of variables at a given point in the program. This query facility is thus a more sophisticated version of the call graph mentioned above, enabling the user to obtain more detailed information in response to more detailed requests. The exact capabilities to be provided will be determined later. The University of Texas FAST system will be examined as a source of model features.

5.1.8 Reentrancy Notation. All procedures used in a multiprocessing situation will be examined for reentrancy. Any characteristics which inhibit reentrancy will be noted. This checking will involve examination of sub-procedures used and any update blocks present.

5.2 Verification. Verification is the process of proving the absence or showing the presence of program errors. No technique exists (or can exist) to fully verify a program, but the following classes of errors will be detected.

5.2.1 Detection of Illegal Data Usage. This includes errors such as referencing an undefined variable, and definition/redefinition anomalies.

5.2.1.1 Detection of Undefined Variables.
Example:

```
PROC: PROCEDURE;
      DECLARE INTEGER, I, J INITIAL (1);
      J = I;
      .
      .
      .
CLOSE PROC;
```

Variable I is referenced before it is defined; possibly the programmer meant the declaration to be: DECLARE INTEGER, J, I INITIAL (1);. The reference to the undefined variable I would be caught by simple static analysis.

5.2.1.2 Definition/Redefinition Anomalies. An example definition/redefinition anomaly follows:

```
PROC1: PROCEDURE;
       DECLARE INTEGER, K, L, M, N;
       DECLARE ...
          .
          .
          .
       K = M + 1;
       L = N + M;
       K = (M+N) L;
          .
          .
          .
CLOSE PROC1;
```

The assignment statement K = M + 1; is useless in this context, as K is redefined two statements later, without being referenced in between. The presence of such a statement does not make the program erroneous, but it does suggest the computation performed is not the one intended. Since this anomaly would be flagged as a result of a static analysis scan, the programmer would be wise to review the code in question.

Definition/Undefinition anomalies can take several forms and involve variables in virtually all classes. All such errors will be detected.

5.2.1.3 Illegal Data Usage Across Procedure Boundaries. The above data flow anomalies, using an undefined variable and defining/redefining a variable, can be detected by the static analyzer across procedure boundaries as well. Full recognition is made of a program's branching logic. The above examples are illustrative only, and do not reflect the complexity of errors which are detectable. The following program illustrates how an error may occur across procedure boundaries.

```
FOO: PROGRAM;
     DECLARE INTEGER, I, J, N;
     .
     .
     .
     BAR: PROCEDURE ASSIGN (X);
          DECLARE INTEGER, X;
          X = X + 1;
          WRITE (5) 'THIS IS THE', X, '-TH TIME';
     CLOSE BAR;
     I = 0;
     READ (4) N;
A:   IF N  0 THEN
          CALL BAR ASSIGN (I);
     ELSE
          CALL BAR ASSIGN (J);
          .
          .
          .
     J = 0;
B:   CALL BAR ASSIGN (J);
          .
          .
          .
     GO TO A;
CLOSE FOO;
```

Suppose -1 is the first value read for variable N. Then in the statement labeled A, BAR will be called with J as its argument. J is uninitialized at this point, and BAR has not been called before. Thus the assignment statement in BAR references an undefined variable. Static analysis will detect this and flag it as a possible error. The call to BAR at B is correct however, as J is defined at this point, regardless of the value read for N. No error flag will be raised at that point.

5.2.2 Detection of Unexecutable Code. A programmer may unknowingly create a section of code to which there is no path, either when originally writing a program or performing maintenance on an existing program. Static analysis coupled with symbolic execution can detect a large number of these situations. Consider the following code fragment:

```
DO FOR I = 1 TO 10;
    .
    .
    .
    If I = 10 THEN GO TO OUT;
END;
    X = X + 10;
OUT: Y = Y + 10;
```

Clearly the statement X = X + 10; is unexecutable. This condition will be detected by the verification and testing capability. It should be noted, however, that not all unexecutable paths will be detected, as this is precluded by theoretical results (namely, that the halting problem is unsolvable).

5.2.3 Deadlock Detection. A HAL/S multitask program may be written so that a cyclic wait (deadlock) situation occurs. Consider the following example.

```
    DECLARE EVENT LATCHED, EV1, EV2;

T1: TASK;
    /* some computation */
    RESET EV2;
    WAIT FOR EV1;
    SET EV2;
CLOSE T1;
T2: TASK;
    /* somewhat less computation */
    RESET EV1;
    WAIT FOR EV2;
    SET EV1;
CLOSE T2;

    SET EV1;
    SET EV2;
SCHEDULE T1 PRIORITY (50);
SCHEDULE T2 PRIORITY (50);
```

Depending upon the actions of the real time executive, events EV2 and EV1 may be reset by tasks T1 and T2 (respectively) "simultaneously." In the absence of external influences, both tasks will wait indefinitely, essentially for each other. This simple example of potential deadlock can be detected statically, as can some more complex examples. For some situations, however, symblic execution may be required to attempt to generate conditions under which deadlock can occur. Other examples may require instrumentation for monitoring these conditions at run time. This distribution of error detecting capabilities among several verification and testing tools is expected to be common in the facility designed.

5.2.4 Illegal COMPOOL Data Usage in a Multitask Environment. A group of processes may be structured such that compool data is properly defined and used only if the processes execute in a certain order. The possible existence of conditions under which this ordering could be violated will be noted.

Example:

```
COMMON: COMPOOL;
    DECLARE INTEGER, I, J;
```

```
        .
        .
        .
    CLOSE COMMON;

    BAZ: PROGRAM;
        DECLARE INTEGER, M, N;
    /* compool template also included */
        INIT: TASK;
            I = 0;
        CLOSE INIT;
        USE: TASK;
            I = I + 1;
        CLOSE USE;
          .
          .
          .
        READ (4) M, N;
        SCHEDULE INIT PRIORITY (M);
        SCHEDULE USE PRIORITY (N);
    CLOSE BAZ;
```

In this example the scheduling of INIT and USE depend upon variables M and N. If N M, USE will execute first, causing an uninitialized variable to be used. As with deadlock, the detection of this type of error will be distributed among several functions. Compool data membership and usage is documented, as are the statements controlling the execution of processes. Static detection of ordering requirements will generate a message, and run time instrumentation may be inserted to check for actual violation.

5.2.5 Data Inconsistencies Resulting From the Termination of Dependent Processes. The program will be examined to see what types of errors may occur when the parent of a dependent process is terminated, causing its sons to be terminated as well. Warnings of inconsistencies in shared data which may arise will be provided. The following example indicates such an inconsistency.

```
ONE_OF_TWO: PROGRAM;
        .
        .
        .
UPDATE_POSITION: TASK;
        /* reference compool */
CLOSE UPDATE_POSITION;
        .
        .
        .
  TERMINATE;
CLOSE ONE_OF_TWO;
DATA_BASE: COMPOOL;
        .
        .
        .
CLOSE DATA_BASE;
TWO_OF_TWO: PROGRAM;
        .
        .
        .
  NAVIGATION: TASK;
        .
        .
        .
        /* reference compool */
```

```
CLOSE NAVIGATION;
    .
    .
    .
CLOSE TWO_OF_TWO;
```

Suppose that task UPDATE_POSITION is executing when its parent, ONE_OF_TWO, reaches the TERMINATE statement. If the task is only partially done, the data base will be left in an indeterminate state. If TWO_OF_TWO's NAVIGATION task then accesses the data base, erroneous results will ensue. Warning of such a situation will be provided by the static analysis, and run time checks may by inserted for monitoring.

5.2.6 Units Specification. A facility will be added to the HAL/S language (possibly as a specially processed comment) to allow the programmer to specify in what units the value of a variable is assumed to be stored. This declaration will be specified at the point of normal declarations. Checking for consistency will be performed at procedure bondaries; checking may be attempted during expression evaluation.

Example:

```
Declare speed integer /* units: feet/second */;
Declare velocity /* units: furlongs/fortnight */;
Declare height /* units: cubits */
```

5.2.7 Scaling and Precision Specification. On machines with inadequate or non-existent floating point units, scaler computation may be performed using fixed point quantities where the programmer keeps track of the implied decimal (or binary) point. The declaration of this convention will be done in a manner analogous to the units specification. Checking of proper scaling and precision will be performed throughout expression evaluation as well as across procedure invocation boundaries.

A sample declaration might appear as follows:

```
Declare float3 integer /* scale: 3 */;
```

implying that float3 has three digits to the right of an implied binary point. Only variables with compatible scales could be added and subtracted. In assignment context the resulting scale from expression evaluation would be checked for compatibility with the declared scale of the receiving variable.

The precise form of the declaration will be determined later.

5.2.8 Violation of Language Restrictions. Language violations which will be checked here include division by zero and exceeding the maximum subscript of a matrix. Of course it may not be possible to completely verify that these will not occur before actual program execution; for some instances run time monitors will be required.

5.2.9 Alteration of Termination Conditions. A common programming error is the writing of infinite loops, when such action is not intended. This often occurs because the variables involved in the termination condition are not altered during execution of the body of the loop. A check will be made to verify that such a change is possible; if not, a warning message will be printed. It should be stressed that such checking will not be infallible in the detection of infinite loops, it will only be an aid.

5.2.10 Consistency of the Load Module. Since a HAL/S load module may be a collection of several separately compiled programs and data pools, checks will be made to guarantee that uniform descriptions of compools and common procedures are used by all programs. This is especially important in view of the fact that non-HAL code may be present, including some absolute patches. In addition, a reference map will be produced showing the locations of all variables. The HALSTAT tool will be carefully examined for guidance when considering the provision of these features.

5.3 Testing. Testing includes all activities taken at or near run time.

5.3.1 Histogram Coverage. A histogram will be produced showing the execution frequency for all statements of a program. Untested statements are thus apparent, and an indication of branch paths taken will be provided. (This information serves as an important guide to optimization as well.)

5.3.2 General Monitoring. Many capabilities are possible in this classification, with the following being among the most important.

5.3.2.1 Event Variable Activity. A report would be produced indicating at what times, with respect to the real time clock, the values of event variables changed, and to what values they were changed. Since program and task names have process events associated with them, this report would also indicate the times of their entering and departing the process queue.

5.3.2.2 Process Queue Snapshots. At specified intervals or times a snapshot would be produced showing what processes were currently in the queue, and in what state: active, wait, ready, or stall. If stalled, an indication would be given as to the condition causing the stall.

5.3.2.3 Selective Variable Monitoring. At each point of change a message would be produced indicating the new value of the variable and the statement number causing the change.

5.3.2.4 Selective Procedure Invocation Monitoring. A report similar to variable monitoring would be produced, but indicating what procedures had been called, from where, and the values of the parameters.

5.3.3 Assertions. Assertions are statements which allow the user to describe the expected behavior of a program. As "statements," they could be inserted in HAL/S programs as specially processed comments or even as a new HAL/S statement type. The actual syntax will be decided upon during the design phase. The basic assert statement, possibly phrased as assert < boolean expression >, when instrumented, is semantically equivalent to the executable statement:

IF NOT < boolean expression > THEN SEND $\underline{\text{ERROR}}_{i;j}$;

where $\underline{\text{ERROR}}_{i;j}$ corresponds to assertion_violation. A simple use of this assert statement might appear as follows:

```
      .
      .
CALL SUB1 ASSIGN (X);
Y = 14.0 N + 3 J;
ASSERT ( X + Y < J );
Z = J / ( X + Y );
      .
      .
      .
```

Presumably when the code was written the programmer was aware that his calculations "guaranteed" X + Y < J. Indicating that by an assert statement documents his understanding while inserting a check for errors which may have arisen due to later modifications (such as to SUB1), misunderstandings, implementation errors, and so forth.

More advanced assertion statements will allow checking of a range of variables and values, and over a program region. An assertion in this category might appear as assert global values (x,y) (1:10); indicating that if the values of variables x and y ever deviate from the range 1 to 10 in any region of the program, the assertion has been violated. The instrumentation for such as assertion would involve checking the values of x and y at each point they are changed, to assure they lie in the proper interval.

The actual design of the specific assertion statements to be implemented is a requirement of the study. Of a particular interest to the real time programmer will be assertions involving event variables, to assert (and thus check for) proper event sequencing. Note that the error handling capabilities provided by HAL/S may enable much of the assertion checking instrumentation to be implemented within the HAL/S language.

Overall, the assertion facility should contain the following features.

1) The notion of a region over which the assertion is valid. This may be a single statement, or an entire procedure. The translator must determine all the relevant points at which to check the assertion.

2) Levels of assertions. The ability to suppress checking (instrumentation) of assertions below a certain level should be provided as a compile-time option.

3) Some quantifiers which may apply to the boolean expression. The full power of first-order predicate calculus would be desirable, but at least a "∀" should be supplied.

4) An "invariant" clause, to allow statements such as assert x<y invariant; for a specified region.

5) A threshold concept. The user would be enabled to specify a limit on the number of times a particular assertion may be violated, before some drastic action (such as terminating the program) is taken.

5.3.4 Timing Assessment. A capability will be provided for estimating the execution time of a given program on a given machine. Input would be required, of course, describing the target machine.

5.4 Debugging Tool. The verification tools provided are envisioned as interfacing with a program debugging tool. Such a tool would permit the generation of program snapshots, setting of checkpoints, and dynamic alteration of variable values. The tool will be highly interactive, but shall be usable from batch as well. Additional capabilities may be added as the design of the tool and its relationship to the verification facility is elaborated. (Some of the functional capabilities listed above, such as variable evolution tracing, may be included as part of the debugging tool.)

6.0 DESIGN/IMPLEMENTATION PLAN

It is not envisioned that all the above capabilities will be implemented at once. A phased implementation is anticipated, with increasingly powerful (and thus increasingly expensive) verification capabilities being added at each step. With this in mind, the capabilities have been divided into six categories, based upon utility of the features to the user and the scope of analysis required. The categorization is not rigid, in that the distinction between some categories for certain features is somewhat arbitrary. A first implementation would almost certainly go beyond implementing only category one; most likely the first three categories would be produced.

The design of the verification and testing capability will accommodate such an implementation. The design produced should be easily amenable to expansion or contraction of capabilities. Thus, for example, if only category one and two features were desired, the implementation should succeed well without the presence of any category three capabilities. The categories are hierarchical, however, in that an implementation of category four could assume the presence of the first three.

6.1 Simple Documentation.

Cross reference maps
- Variable
- Lock Group
- COMPOOL
- Shared Data
- Event Scheduling Statements

Process dependency

Implicit type conversions

Extraction of internal documentation

Call graph

6.2 Local Information.

Histograms

Symbolic post-mortem dump

Local assertions: boolean expressions
levels
threshold
quantifiers

Intraprocedural detection of:
uninitialized variables
definition/redefinition anomalies

Run time monitors for zero division, overflow, etc.

Variable and procedure monitoring

Scaling specification and intra-procedural checking

Simple detection of unexecutable code

6.3 Multi-Procedural Information.

Units specifications and interprocedural checking

Scale specifications and interprocedural checking

Interprocedural checking of:
uninitialized variables
definition/redefinition anomalies

Regional assertions

Load module analysis

6.4 Separate Compilation/Multi-Processing Information.

FAST-like query facility

Reentrancy checking

Illegal COMPOOL usage

Termination of dependant processes

Event chronology

Queue snapshots

Simple Deadlock detection

6.5 Debugging/Performance Estimates.

General debugging system:

- breakpoints
- traces
- variable alteration

Check of termination conditions

Timing estimate

6.6 Difficult Issues.

Refinement of above analysis:

- Unexecutable code
- Definition/redifinition anomalies
- Uninitialized variables
- Deadlock detection
- COMPOOL usage
- Violation of language rules

REFERENCES

1. [Osterweil, 1977a] Osterweil, Leon J., "A Methodology for Testing Computer Programs," Proceedings AIAA Conference on Computers in Aerospace, Los Angeles, California, pp. 52-62 (October 1977).

2. [Osterweil, Brown and Stucki, 1978] Osterweil, Leon J.; Brown, John R.; Stucki, Leon G., "ASSET: A Lifecycle Verification and Visibility System," in Proceedings COMPSAC 78, Chicago, Illinois pp. 30-35 (November 1978).

3. [Karr and Loveman, 1978] Karr, Michael and Loveman, David B., "Incorporation of Units into Programming Languages," Communications of the ACM, Vol. 21, No. 5, pp. 385-391 (May 1978).

4. [Fosdick and Osterweil, 1976] Fosdick, Lloyd D. and Osterweil, Leon J., "Data Flow Analysis in Software Reliability," Computing Surveys, Vol. 8, No. 3, pp. 305-330 (September 1976).

5. [Taylor and Osterweil, 1978] Taylor, Richard N. and Osterweil, Leon J. "A Facility for Verification, Testing, and Documentation of Concurrent Process Software," in Proceedings COMPSAC 78, Chicago, Illinois, pp. 36-41 (November, 1978).

6. [Osterweil, 1977b] Osterweil, Leon J., "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis," Department of Computer Science Technical Report No. CU-CS-110-77, University of Colorado; Boulder, Colorado (May 1977).

7. [Clarke, 1976] Clarke, Lori A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 215-222 (September 1976).

8. [Howden, 1977] Howden, William E., "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, pp. 266-278 (July 1977).

9. [Howden, 1978a] Howden, William E., "DISSECT - A Symbolic Evaluation and Program Testing System," IEEE Transactions on Software Engineering, Vol. SE-4, No. 1, pp. 70-73 (January 1978).

10. [Howden, 1978b] Howden, William E., "Functional Program Testing," Proceedings COMPSAC 78, Chicago, Illinois, pp. 321-325 (November 1978).

11. [Stucki, 1976] Stucki, Leon G., "The Use of Dynamic Assertions to Improve Software Quality, MDC G6588, McDonnell Douglas Astronautics Company-West (November 1976).

12. [Chow, 1976] Chow, T. S., "A Generalized Assertion Language," Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, pp. 392-399 (October 1976).

13. [Lloyd & Lipow, 1977] Lloyd, D. K. and Lipow, M., Reliability: Management Methods, and Mathematics. Second edition, published by the authors, Redondo Beach, CA, 1977.

14. [Sukert, 1977] Sukert, A. N., "A Multi-Project Comparison of Software Reliability Models," Proceedings, 1977 Computers in Aerospace Conference, L.A., CA, pp. 413-421 (October 1977).

15. [Thayer, et. al., 1976] Thayer, T. A., Lipow, M., Nelson, E. C., "Software Reliability Study," TRW-SS-76-03, TRW Defense and Space Systems Group, Redondo Beach, California (March 1976).

16. [Huang, 1978] Huang, J. C., "Detection of Data Flow Anomaly Through the Use of Program Instrumentation," Technical Report UH-CS-78-4, Department of Computer Science, University of Houston (July 1978).

17. [Johnson, 1977] Johnson, David B., "Program Analysis with the Aid of a Data Management System," Masters' thesis, Department of Computer Science, The University of Texas at Austin (August 1977).

18. [Browne, & Johnson, 1978] Browne, J. C. and Johnson, David B., "FAST - A Second Generation Program Analysis System," Proceedings of the 3rd International Conference on Software Engineering, Atlanta, Georgia, pp. 142-148 (May 1978).

19. [Stephens and Tripp, 1978] Stephens, Sharon A. and Tripp, Leonard, "Requirements Expression and Verification Aid," Proceedings 3rd International Conference on Software Engineering, Atlanta, Georgia, pp. 101-108 (May 1978).

20. [Johnson, 1978] Johnson, Mark Scott, "The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment," Doctoral thesis, Department of Computer Science, University of British Columbia (1978).

21. [Osterweil and Fosdick, 1976] Osterweil, L. J. and Fosdick, L. D. "DAVE-A validation, error detection, and documentation system for FORTRAN programs," Software-Practice and Experience, Vol. 6, 473-486 (1976).

22. [Allen, 1969] Allen, F. E. "Program Optimization," in Annual Review in Automatic Programming, Pergamon Press, New York, pp. 239-307 (1969).

23. [Allen and Cocke, 1976] Allen, F. E. and Cocke, J. "A program data flow analysis procedure," Communications of the ACM, Vol. 19, no. 3, pp. 137-147 (March 1976).

24. [Hecht and Ullman, 1975] Hecht, M. S. and Ullman, J. D., "A simple algorithm for global data flow analysis problems," SIAM Journal of Computing, Vol. 4, pp. 519-532 (December 1975).

25. [Riddle, et.al., 1977] Riddle, W., Bristow, G., Drey, C. and Edwards, B., "Anomaly Detection in Concurrent Programs," Department of Computer Science Technical Report, #CU-CS-147-79, University of Colorado, Boulder, Colorado (January 1977).

26. [Peterson, 1977] Peterson, J. L., "Petri Nets," Computing Surveys, Vol. 9, Number 3, pp. 223-252 (September 1977).

27. [Reif, 1978] Reif, J., "Data Flow Analysis of Communicating Processes," University of Rochester (1978).

28. [Barth, 1978] Barth, J. M., "A Practical Interprocedural Data-Flow Analysis Algorithm,"Communications of the ACM, Vol. 21, No. 9, pp. 724-736 (September 1978).

29. [Bollacker, 1978] Bollacker, L. A. "An Algorithm for Detecting Unexecutable Paths Through Program Flow Graphs," Department of Computer Science Technical #CU-CS-112-78, University of Colorado, Boulder, Colorado (January 1978).

30. [Gallucci, 1978] Gallucci, M., "Report on Path-Generating Algorithm," Department of Computer Science Internal SVG memo #93, University of Colorado, Boulder, Colorado (May 1978).

31. [Osterweil, 1975] Osterweil, L. J., "Depth-First Search Techniques and Efficient Methods for Creating Test Paths" Dept. of Computer Science Technical Report #CU-CS-077-75, University of Colorado, Boulder, Colorado (August 1975).

32. [Hopcroft and Tarjan, 1973] Hopcroft, J. and Tarjan, R., "Algorithm 447: Efficient Algorithms for Graph Manipulation" Communications of the ACM, Volume 16, No. 6, pp. 372-378, (June 1973).

33. [Hecht, 1977] Hecht, M. S., Flow Analysis of Computer Programs, Elsevier North-Holland, New York, ISBN 0-444-00216-2 (1977).

34. [London, 1977] London, T., "The Semantics of Information Flow," PhD Thesis, Department of Computer Science, Cornell University (1977).

35. [Clarke and Ogden, 1978] Clarke, Lori and Ogden, Neal, "Top-down Testing with Symbolic Execution" Digest for the Workshop on Software Testing and Test Documentation", Ft. Lauderdale, Florida (December 1978).

36. [Howden, 1977b] Howden, William E., "Symbolic Testing - Design Techniques, Costs and Effectiveness" National Bureau of Standards, NBS-GCR-77-89 (May 1977).

37. [Winters, Ogden, Clarke, 1978] Winters, Daryl; Ogden, Neal; Clarke, Lori, "A Definition of AID, The ATTEST Interface Description Language" COINS Technical Report 78-15, University of Massachusetts, Amherst, MA (December 1978).

38. [Howden, 1978c] Howden, William E., "An Evaluation of the Effectiveness of Symbolic Testing", Software-Practice and Experience, Vol. 8, pp. 381-397 (1978).

39. [Hall, 1971] Hall, Andrew D. Jr., "The Altran System for Rational Function Manipulation - A Survey," Communications of the ACM, Vol. 14, No. 8 (August 1971)

40. [Knuth, 1969] Knuth, Donald E., The Art of Computer Programming Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, Mass. (1969).