NASA Contractor Report 159008

NASA-CR-159008

INTEGRATED TESTING AND VERIFICATION SYSTEM FOR RESEARCH FLIGHT SOFTWARE - DESIGN DOCUMENT

Richard N. Taylor

BOEING COMPUTER SERVICES COMPANY
Space and Military Applications Division
Seattle, Washington 98124

NASA Contract NAS1-15253
February 1979

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

INTEGRATED TESTING AND VERIFICATION SYSTEM FOR RESEARCH FLIGHT SOFTWARE

Design Document

By Richard N. Taylor

Prepared Under Contract NAS1-15253

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

For

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

CONTENTS

CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

SECTION 1.0

Summary and Introduction

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 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. 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 the references.

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 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.

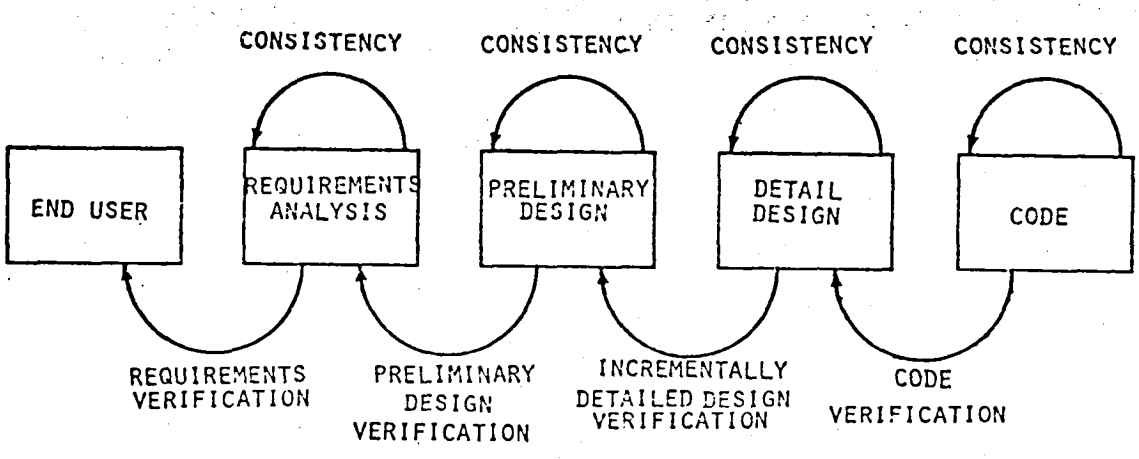


Figure 1.2-2 Lifecycle Verification

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."

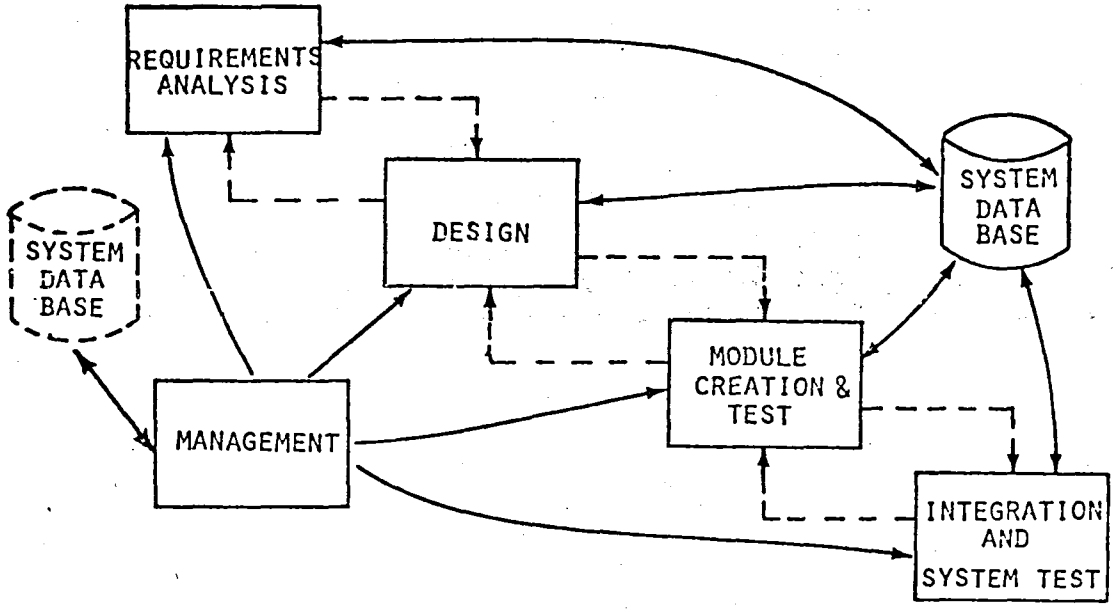


Figure 1.2-3 System Overview - Management of the Software Lifecycle and Data Flows

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.)

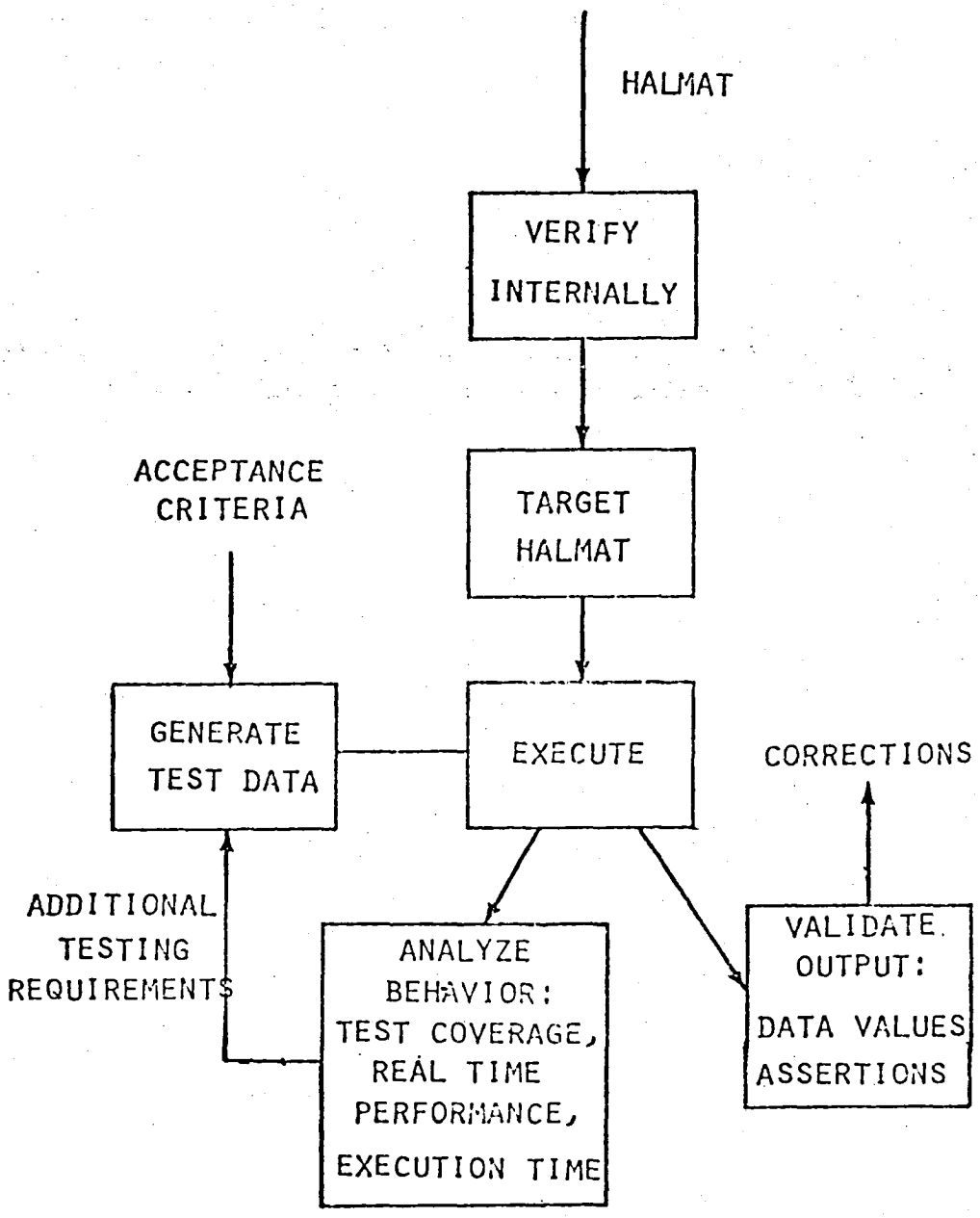


Figure 1.2-4 Source Code Verification and Testing

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 SAMM nodes CBC 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. 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.

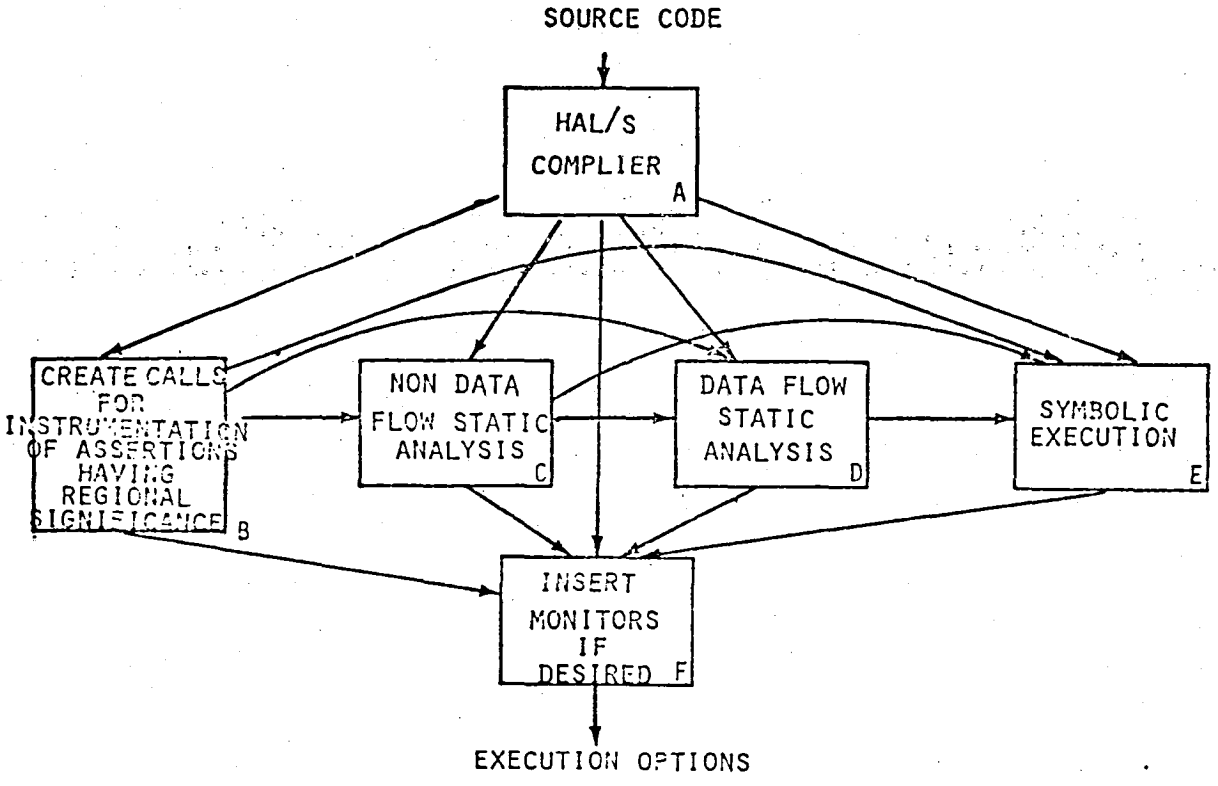


Figure 1.2-5 Module Verification Options

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.

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.

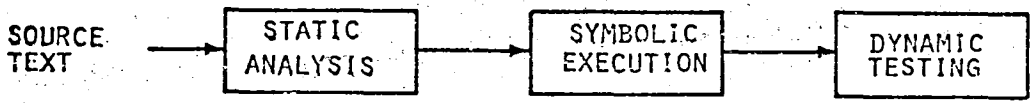


Figure 2.1 Basis for Integrated Verification Methodology

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 dfferent 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 Systems and Software Engineering Laboratory.

As verification of real time, concurrent process software is a poorly understood aspect of error detection called for in the requirements document, preliminary 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. To date, techniques for

detecting the following errors have been isolated: uninitialized shared (COMPOOL) variables, some forms of uselsss synchronization, simple deadlock situations, and unsafe alteration of shared data. Substantial progress in this area is anticipated as investigation continues.

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, and the FSIM compiler capability.

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, cheaper flight software.

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 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 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 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 listing produced by Phase I of the compiler is excellent in format and content. Since it is a "standard" form, as possible all messages should be related directly to this listing. Some information can be directly added to the listing by post-processing it and adding new fields.

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, second, a pointer (offset) to the proper HALMAT statement within the paragraph.

The various "paragraphs" 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 paragraphs will have expressions phased 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 necessarliy have to be removed - 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 files.

In summary, the following tag fields are tentatively identified as being associated with each monitor paragraph:

1. Pointer to SMRK instruction
2. Offset to relevant HALMAT instruction
3. Active/Inactive (and what determined that)
4. Level (module, system, et. al)
5. Monitor type (assert, keep, error monitor)
6. Development status
7. Monitor origin (which facility caused its creation)

Additional fields may be identified during later design phases. Not all fields may be required on every monitor.

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 and scales
3) Specification of relationships among units and scales
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.

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;
/*UNIT_RELATIONS:                    Inches = 12* feet;
                 watt = volt*amp; */;

DECLARE  /* UNIT: feet */ f1, f2;
DECLARE  /* UNIT: inches */ i1, i2;
DECLARE  /* UNIT: volts*/ v;
DECLARE  /* UNIT: amps*/ a;
DECLARE  /* UNIT: watts*/ w;
```

```
(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 node. 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.

2. Specification of Elementary Units and Scales

A. 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;
```

B. Scale

Elementary scale factors are declared differently from elementary units, since there is not a clear need for a facility like objective 2) above.

The declaration of elementary scales will appear as follows:

/* ELEMENTARY_SCALE: list of integer scales,
where each integer is a power of 2 */

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. Specification of Relationships Between Units and Scales

After all units/scales 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 */;

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. If desired, however, efforts could be made to extend the technique cited in reference 3 to allow this. The resulting algorithm may not be as efficient or flexible, though. The impacts of such a change should be carefully considered.)

The utility of constant values in relationships is subject to the considerations of Section 1. 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.

4. Declaration of Variables Having Units/Scale Attributes

- Variables may have both units and scale attributes.
- All scales and 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: previously declared scale */;

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.

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 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
```

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 the reference?

3) Ease of implementation of declaration processing. Some modifications to the HAL/S compiler will clearly be required. Further investigation into the compilers structure will be required to determine if the syntax described above is suitable.

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 ANSI 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, Cooke, 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 must be expanded in the appropriate areas. 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 present 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 a paper describing the results in reference 5 [Taylor and Osterweil, 1978]. The problem has many facets, but the prospect for significant results is good. The design incorporates the initial results, and is extensible allowing the inclusion of future results. See the later part of this section for a full discussion of this research.

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.

6. In order to generate the most helpful error messages and to provide analysis paths for 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. Thus this process should not present much overhead.

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

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 informations format and its retrieval. The ramifications of multi-level static analysis (i.e. static analyzer on the module level, then on the program level, then on the multi-process level) needs to be explored, as regards the internal database.

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.

Static Data Flow Analysis Of Concurrent Process Software

To illustrate the new difficulties inherent in analysis of concurrent programs, consider the HAL/S program fragment in Figure (3.4.2-1). We actually wish to consider two versions of this program: one including the statement marked with the asterisk, and one omitting it.

The figure consists of a main program and two tasks, T1 and T2. When scheduled, a task may execute in parallel with other tasks and any executing programs, subject to synchronization constraints. Note that in this example both tasks reference global variable i. Task T1 executes in parallel with the main program, initializing variable i in the process. Task T2 also executes in parallel with the main program when it, in turn, is scheduled. Its correct execution depends upon i having been properly initialized. Consider the version of the program omitting the marked statement. Depending on the implementation, once the schedule T1 statement is executed, the system scheduler may elect to run task T1 to completion before any further statements in the main program are executed. If so, all is well. If, however, multiple processors are used or time slicing is employed, the main program may progress to the schedule T2 statement before variable i has been initialized. The potential then exists for T1, T2, and the main program to execute in parallel, with variable i being referenced before it is defined.

The presence of the wait statement resolves the difficulty by requiring termination of T1 before T2 is scheduled, thereby assuring that i will be defined before it is referenced.

In order to clearly distinguish the essential difference between the two example programs, and to form a basis on which we may formulate an analysis algorithm, we introduce the concept of a process-augmented flowgraph.

```
main: program;
declare integer i;
     t1:task;
        i = 0;
        .
        .
        .
     close t1;
     t2:task;
        i = i+1;
     close t2;
schedule t1 priority (50);
        .
        .
        .
*wait for t1;
 schedule t2 priority (50);
        .
        .
        .
 close main;
```

Figure 3.4.2-1 A HAL/S Program Fragment

A process-augmented flowgraph (paf) is a graph representing a set of communicating concurrent processes, formed from the individual process flowgraphs joined by special edges (arrows) indicating all synchronization/communication constraints. In HAL/S, statements causing such constraints include schedule, wait, task, and close. The special arrows join these statements to appropriate points in the cooperating processes' graphs.

Specifically, to form the paf an arrow must be created for each ordered pair of nodes of each of the types: (schedule, task) and (close, wait).

Returning to our example, Figure (3.4.2-2) is a paf for the program omitting the wait statement, showing the three processes, T1, T2, and MAIN, all acting in parallel. The paf of Figure (3.4.2-3) shows the synchronizing effect of including the wait statement: T1 must terminate before initiation of T2. This distinction, as noted, is the basis for the error.

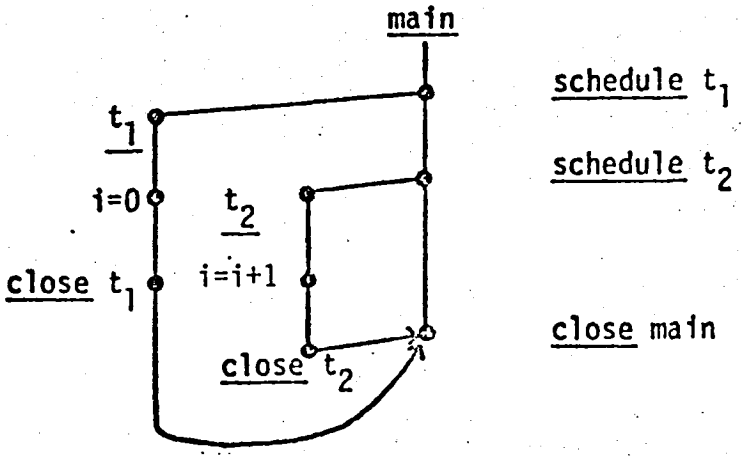


Figure 3.4.2-2 The paf For The Fragment In Figure 3.4.2-1 Without The Starred Statement

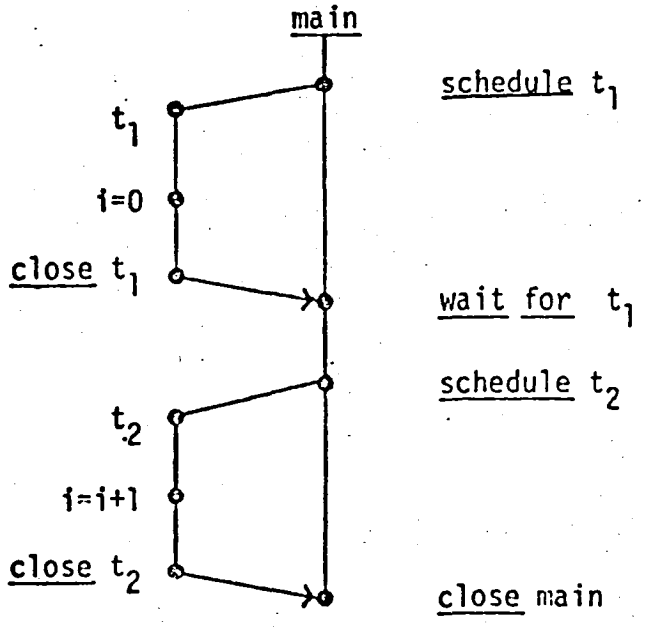


Figure 3.4.2-3 The paf For The Fragment In Figure 3.4.2-1

We now show that existing static analysis algorithms are able to detect the error in our example by analyzing the paf and making special allowance for the special edges.

As background, we briefly describe the operation of these algorithms. The algorithms to be employed are described in (Fosdick & Osterweil, 1976). The purpose of these algorithms is to infer global program variable usage information from local program variable usage information, and then to infer documentation, verification, and error detection results from the variable usage results. The local variable usage is represented by attaching two sets of variables, gen and kill, to each program flowgraph node. The global data usage is represented by attaching two sets, live and avail, to each node. The algorithms presented in (Fosdick & Osterweil, 1976) 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' 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.

As an example, let us see how this information can be used to determine the possibility of an uninitialized reference to a variable v in a given program. We begin by annotating the graph so that v is placed in the gen set of a node if and only if v is defined at the node, and all variables v are placed in the kill set of the program start node. For the purposes of this example let us also hypothesize the existence of a ref set at each node. Let v be placed in the ref set of a node n if and only if v is referenced at n. Now suppose the avail sets are computed. Next compute $ref(n) \cap \overline{(ref(n) \cap avail(n))}$. If this intersection set contains v at a node, n, then whenever node n is executed there will be a possibility that there will be an uninitialized reference to v. If the intersection is void, no uninitialized reference can occur at n.

In (Fosdick & Osterweil, 1976) it is shown that many similar analytic results can be obtained by appropriate selections of gen and kill criteria and corresponding interpretations of live and avail sets.

Bearing this discussion in mind, let us now consider the paf of Figure (3.4.2-2). Suppose the gen set for each node contained all variables defined at that node, the ref set of each node contained all variables referenced at that

node, and the kill set of the start node contained all variables. Now suppose the avail sets are computed in such a way as to insure the original definition - namely that for all execution sequences leading to n, v was most recently in a gen, rather than a kill, set. Then, if

$$ref(n) \cap \overline{ref(n) \cap avail(n)}$$

is computed, we find that i is in this intersection set for the node n representing the statement referencing i. This indicates the possibility of an undefined reference to i at n. This is the desired analytic capability.

Unfortunately the avail sets for nodes of a paf must be computed somewhat differently than for nodes of an ordinary flowgraph in order to assure the correct function of the above analysis. In an ordinary single-process flowgraph, the avail set at a node n must be given by:

$$(*)\ avail(n) \leftarrow \bigcap_{\substack{\text{all immediate} \\ \text{predecessors} \\ \text{of } n,\ n'_i}} \overline{(kill(n'_i)\ n\ avail(n'_i))}\ u\ gen(n'_i)$$

Algorithms for computing avail assure that this condition holds for all nodes upon termination.

Now suppose that node n represents a wait statement in a paf. Then one edge entering n must be a special synchronization edge. If a variable, v is initialized as a result of executing the process preceding that edge, then it is certain that v must be initialized at n regardless of whether it has been initialized prior to execution of any other edges entering n. Hence we see that if w is a wait node of a paf, and w' is the node at the other end of the special edge entering w, then for the purposes of this uninitialized variable reference analysis avail(w) must be computed by:

$$\text{avail}(w)\ \ \text{gen}(w')\ \ (\text{avail}(w')) \qquad \bigcap_{\substack{\text{all predecessors of } w, x, \\ \text{except } w'}} \Big[\ \ [\text{gen}(x)\ \ (\text{avail}(x))]\ \Big]$$

Similar adjustments must be made in order to assure that the other analytic results on single process flowgraphs which are described in reference 4 (Fosdick & Osterweil, 1976) can be obtained for concurrent program paf's.

Similarly, there are more intricate adjustments which must be made in order to correctly compute the live sets at schedule nodes of a paf. Once made, however many analytic results in reference 4 (Fosdick & Osterweil, 1976) which employ live sets can also be obtained for concurrent program paf analysis.

Our preliminary investigation indicates that the algorithms described in reference 4 (Fosdick & Osterweil, 1976) can be altered to compute live and avail for paf's as described above without affecting their highly efficient execution speeds.

All of the above results are predicated upon the existence of the paf, yet construction of the paf is not a trivial or insignificant activity. For our initial studies we have chosen to consider only programs whose synchronization is carried out entirely by the schedule and wait statements. Under this simplifying assumption the construction of the paf is rather straightforward, involving basically a depth-first search. It is important to note, however, that HAL/S contains certain synchronization constructs for which paf construction is more difficult, and in some cases impossible. Further research will be needed to determine the largest subset of the language which is comfortably amenable to static data flow analysis.

Our work has shown that the paf has additional significance as the basis for detecting various process coordination errors. For example, consider the program whose paf is presented in Figure (3.4.2-4). The HAL/S program contains a significant anomaly which should be noted. If during execution the else branch of

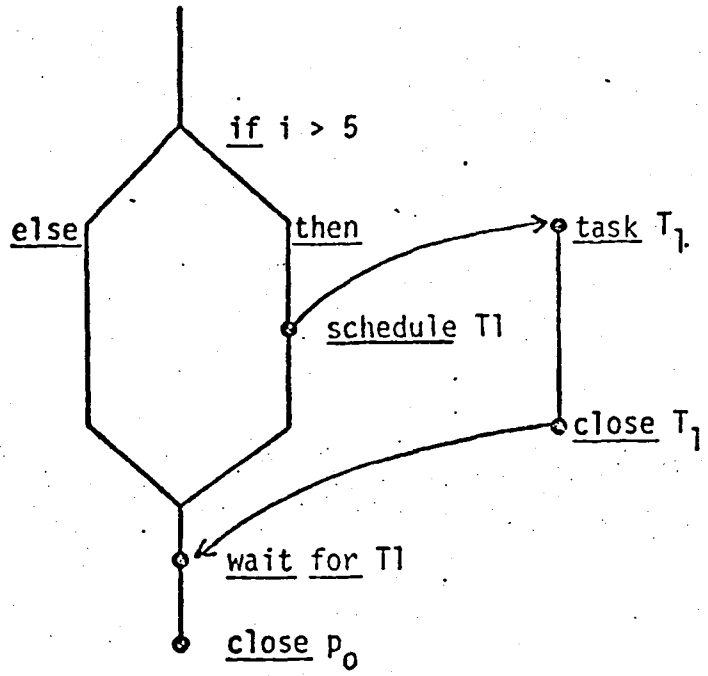


Figure 3.4.2-4

the if is taken, task T1 will not be scheduled, yet later the program will wait for it. Strictly speaking this is legal in HAL/S and the wait will have no effect, but it may be very indicative of a logic error. A reader of the program would likely assume that T1 was guaranteed to have been scheduled, else why wait for it? To document such statement usage, and for detection of other similar errors, an anomaly analysis is desirable.

Let us assume that the HAL/S program is represented by a paf. But now suppose that the gen set of every node consists of the processes scheduled at that node and the kill set of every node consists of the processes waited for at that node. Now consider the special synchronization edges to be no different than other edges and determine the avail sets for the paf nodes. Clearly if a process is not in the intersection of the avail and kill sets for some node n then it is possible to wait at n for a process that has not been scheduled immediately prior.

Similar kinds of analysis can detect useless synchronization (waiting for an event which is guaranteed to have transpired), erroneous synchronization (such as

waiting for a process guaranteed to not have even started), and scheduling a process in parallel with itself.

Unscheduled, yet declared, processes are detected by using live analysis to show the absence of any path from the start of a main program to the open node of such process flowgraphs. Simple deadlock situations may be detected by examining the paf for cycles involving (at least) two (close, wait) edges.

3.5 Symbolic Execution. - The design and recommended usage of the symbolic executor presented in the SAMM diagrams has been strongly motivated by consideration of two existing symbolic executors. Clarke's ATTEST system (reference 7) [Clarke, 1976] and Howden's DISSECT system (references 8,9) [Howden, 1977; Howden, 1978a] were both examined and experiences with them evaluated. The result is basically an amalgam of the two. The guiding principles here have been to define a tool which has a feasible implementation and, most importantly, addresses an important user need.

Experience with ATTEST has shown that efforts to prove the executablity of a given path by solving the associated system of constraints can meet with success, though substantial effort is involved. Additionally, as a by-product of showing executability, test data can be automatically generated which will force execution down the selected path. (Test values is a more precise term, as formatting of the values into what is required by the input handler must be performed manually.)

ATTEST has also served as a testing ground for data and program structure. A "standard" algebraic manipulation package was employed in the program . This experience will prove valuable when implementation of the MUST system is undertaken.

Howden's DISSECT system has been beneficial in indicating the user features which are most valuable in controlling the executor, and in providing overall perspective on the utility of such systems. With regard to user features, the design presented contains many different areas in which the user may interact and control actions. Specification of paths, data values to use, and output desired are all under user control. The DISSECT system is regarded as the model in this area. Refinements and revisions to DISSECT's abilities will be required, of course, as the symbolic executors design is deepened.

Concerning perspective, the DISSECT system was carefully evaluated after its construction to see how well it detected programming errors in comparison with other techniques. In a later paper (reference 10) [Howden, 1978b], Howden examined error detection schemes in general for their efficacy, through a series

of experiments. The result of both these considerations was that, in the presence of a suite of error detection tools (such as static and dynamic analyzers), symbolic executors did not detect significantly more errors than the combination of the other tools. One cannot conclude from this, however, that symbolic executors have no place in a programming environment. It does indicate that the tasks should be carefully chosen to which symbolic executors are put.

We believe that while the symbolic executor should be built with capabilities to perform automatic error detection, the user should have the potential to be heavily involved in the process. An example of such involvement is to have the executor indicate all output values which are affected by a given input value, or, for a given output, display all the inputs which are involved in its computation. In the latter example the user would be involved in aiding the identification of all paths (or classes of paths) which lead to the output. The system presented allows these types of activities in addition to the more classical functions of symbolic executors.

The possibilities of using a symbolic executor to analyze and refine "error" messages coming from the static anaylzer do not seem as promising as was once hoped. Experience with DISSECT, for example, indicated that pathwise checking for division by zero errors was not profitable. Significant user interaction is required to examine most of these questionable errors. The design we present allows for the symbolic executor to examine these situations, but further research and experience will be required to evaluate this more fully. We believe that using a symbolic executor to check the functional effect of paths in the manner described above, and in the classical sense (printing formulas and performing algebraic manipulations, in contrast to only printing simple variables which are "affected"), will prove most valuable.

One feature is present in the design which has no direct analogue in either ATTEST or DISSECT. A machine state resulting from partial execution of actual code may be passed to the symbolic executor to (at least partially) specify the values to be used during symbolic execution. Further analysis of this features utlity and the difficulty of implementing it is recommended. Such analysis will be

possible when the details of the executor's structure are worked out. Note that this type of facility begins to blur the distinction between a symbolic executor and an interactive debugger. Further analysis of the similarities between the two tools is recommended.

3.6 Dynamic Analysis.

3.6.1 Assertion Facility.

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.

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, 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.

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. The compiler procedures to parse expressions will thus be available. Examination of the compiler structure will be required to determine if this is truly feasible. (The semantic actions may be too closely tied to the parsing.)

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

Lastly, the problem of adding the new HALMAT (which represents the instrument) to the existing HALMAT (which represents the program) must be closely considered. No significant difficulties are anticipated, rather, care must be exercised to ensure that all pointer references, counters, and so forth in HALMAT's triple structure are maintained in a consistent manner.

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 particularly 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 compositon of the functions.

A list of supporting capabilities follows:

1. The assertion processor(s) will accept a control file as secondary input. This will allow information such as selective instrumentation commands to override commands embedded in the source text. Information regarding standard "assertion functions" to be used may also be supplied.

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 must 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 machines word size).

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

5. The facilities which actually insert the instrumentation should provide 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.

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

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 ::= to be determined
range ::= (constant (: constant))
path expression ::= PATH special label WAS path
    PATH path
path ::= to be determined
end ASSERT stmt ::= /* END ASSERT special label */ ;
invariant mark ::= /* END VARIANT special label */ ;
statistics value ::= special label ((name))

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.

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. If no label is given to an ASSERT statement, its instrumentation may only be controlled by a binary switch. This "external mechanism" is a user supplied function to the assertion processor (SAMM node CBCD).

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).

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 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 follows 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 assertions with the power of the first order predicate calculus. Such assertion capability is very useful in efforts to formally demonstrate program correctness. As far as instrumented assertions are concerned, however, substantial effort must be expended to determine what restrictions should be placed upon quantifier completion in order to guarantee feasible instrumentation.

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. 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.

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 originating in the current scope, will occur in reference contexts in the same (static) scope following the OUTPUT assertion. The scope of the OUTPUT specification is from the beginning of the program unit (procedure, task, or program) to 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 ) )
      normal function
svalue list ::= svalue ( , svalue )*
svalue ::= expression
      COUNT
      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). If they are unlabeled their instrumentation may be controlled by only a single (binary) switch.

2. The KEEP GLOBAL statement specifies a list of functions which are to be applied to 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 applied 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.

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. At least three special values may be referred to in the function definition: COUNT, VALUE, and STMTTYPE. COUNT refers to the execution count for the current statement. VALUE refers to the value, if any, for the current statement. VALUE is defined as follows.

Statement-type	Value
Assignment	the value assigned to the left side of the assignment
If	the value of the comparison

STMTTYPE refers to the statement types enumerated under stmt-type above.

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. 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.

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.

Rationale

The primary motivation for the provision of the KEEP statements 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 a debugger (SAMM node CCCD), 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.

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 valve
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 (1:6)(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 1.1(X) = 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.

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 analysis (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.

Rather than restate here all the particular documentation items generated, the reader is referred to the SAMM diagrams and the description of the assertion facility. Improvements to the HAL/S compiler are noted under non-data flow static analysis. (In general the compiler was judged as producing excellent cross reference maps and other information associated with compilation. This evaluation was based on reviewing the HAL/S 360 User's Guide and test programs run on the 360/370 compiler. The differences in output between the NASA-LRC compiler and the 360 compiler were not considered.)

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.

		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	Purser & 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			✓		✓											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	Purser & 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						
J-2	Core storage limit exceeded				✓							1			1					2	
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 1 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	Purser & Syntax Checker	Cross-reference Maps	Units/Scale Checking	Standards Checker	Termination Conditions	Coersion 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)

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. Since only the functional requirements are considered, and since requirements relating only to the detailed design are not relevant here, 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 possible exception of the interactive tools. They may require their own user interface. Adequate documentation and HELP messages will be provided, but without being burdensome.

4.1.2 Only nodes CCC, Execute and Debug, and CBCCC, Execute Symbolically, are 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 the interactive debugger and operating system interfaces (such as files needed in collecting run time statistics) verification activities are generally independent of a particular code generator.)

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 CCB, 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 CCCD, Interactive Debugging.

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

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 statements 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. (Extravagent 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 subscripts) 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. - Only a very few of the verification facilities presented in the design require any knowledge of the final (machine) representation of the code. The interactive debugger is clearly implementation dependent, as are the file system interfaces required by run-time monitors (both for error checking and assertion/statistics processing). Any activities associated with the run-time monitor, such as process queue snapshot generation, are also implementation dependent. With the exception of the interactive debugger, these dependencies are straightforward, and their resolution should be clear.* No undue

requirements are thus placed on any Interpretive Computer Simulator (ICS) utilized. Since interactive debuggers and ICS's serve similar purposes it is not anticipated that cooperation will be required. If cooperation is required, a suitable interaction should be achievable with some effort.

* All verification tool-generated monitors are contained directly in the code.

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 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 production 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. Further investigation into design of some of the listed tools will increase their value, of course. The symbolic executor is the most notable member of this category. Also involved in an implementation would be provision of general support capabilities, such as a manager for the database described in Appendix B.

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
CADC		Simulate design
CBB		Verify module code to design
CBAC	3	Answer questions about specified code segments (a language intelligent text editor)

SAMM Designator	Priority	Description
CBCB	1	Create monitor calls from assertions having regional significance
CBCD/DD	1	Instrument HALMAT for module/system tests
CBCAAA/D		Perform basic HAL/S to HALMAT translation
CBCAAB	1	Process (translate to monitors) local assertions and keep statements
CBCCB/DCB	1	Perform data flow analysis
CBCCC/DCC	2	Perform symbolic execution

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)

SAMM Designator	Priority	Description
CCB/EB		Target HALMAT to executable/simulation code
CCCA		Load and produce load maps
CCCB		Monitor HAL/S execution (System monitor)
CCCD	2	Debug/Test Interactively
CCDBAA	1	Post-process Histogram/History File
DA	2	Check for recompilation requirements and merge modules into a single system
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, CCC, CCDBA, CCDA (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.)

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" 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

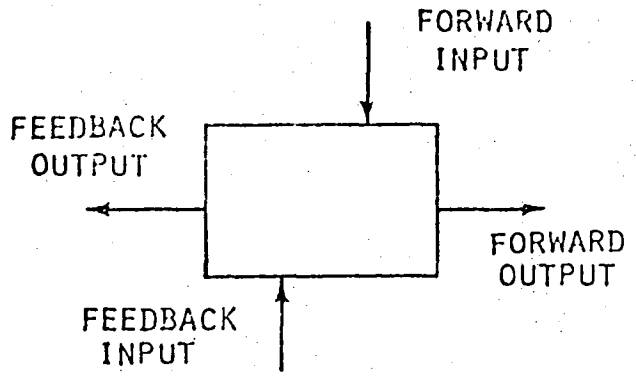


Figure A-1 SAMM Activity Cell With All Possible Inputs and Outputs

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

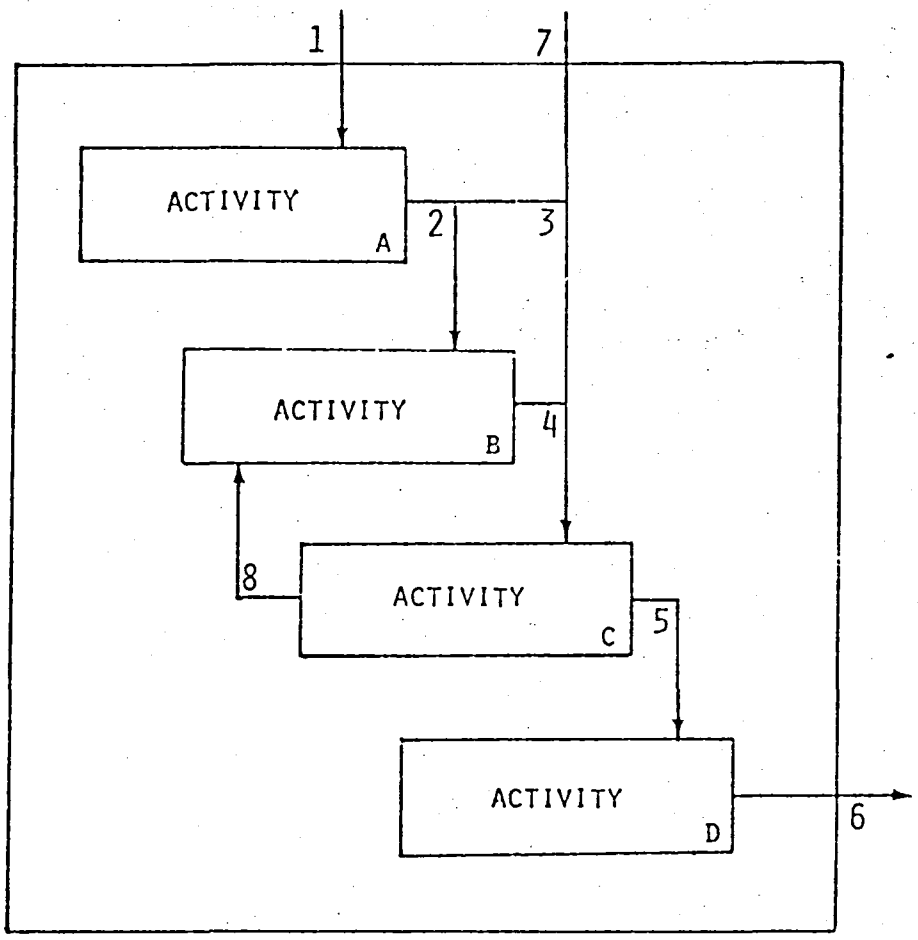


Figure A-2 Sample SAMM Diagram

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.

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.

```
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} ;

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 }
```

REFERENCES

Appendix C

1. [Osterweil, 1977a] "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] "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] "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, L. 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, Redordo 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 Spare 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]"Requirements Expression and Verification Aid," Proceedings 3rd International Conference on Software Engineering, Atlanta, Georgia, pp. 101-108 (May, 1978).

20. [Johnson, M. S.,] 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.

SAMM DIAGRAMS

Appendix D

BGS

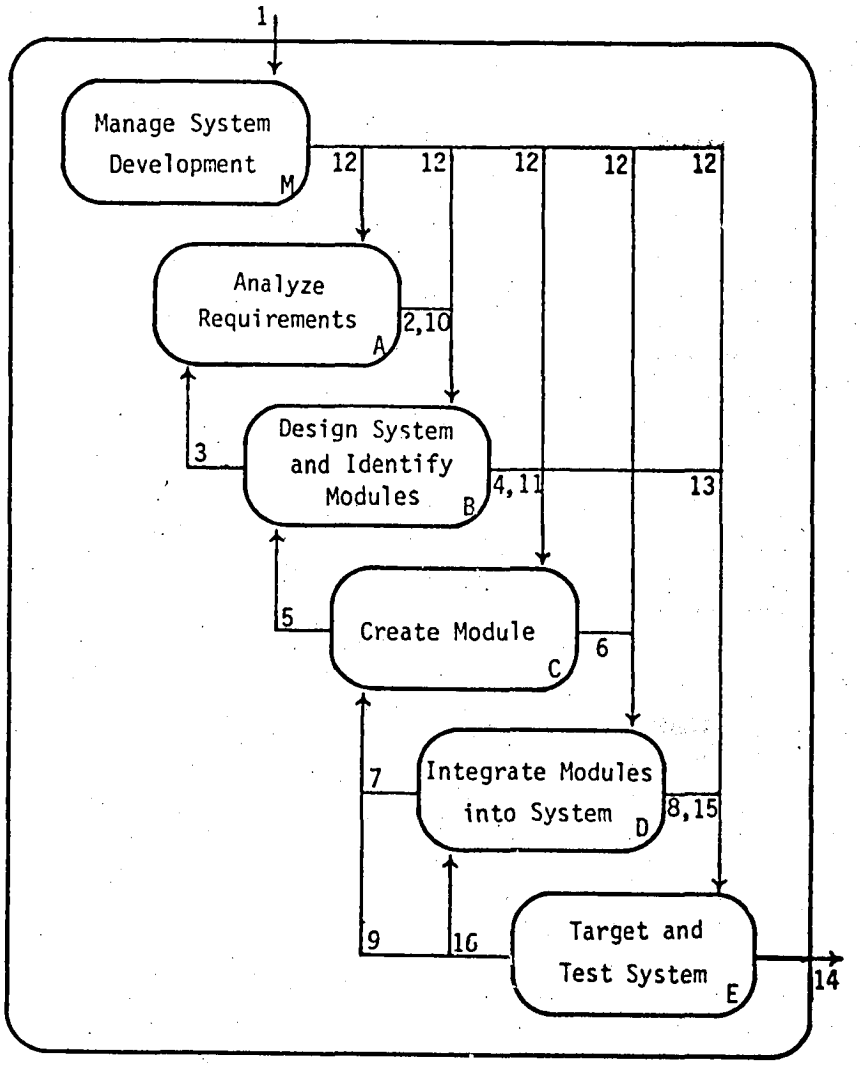
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: ROOT

TITLE: SYSTEM CREATION

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		Generated test data
16		Machine state resulting from partial execution (for use by the symbolic executor)

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

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.)		

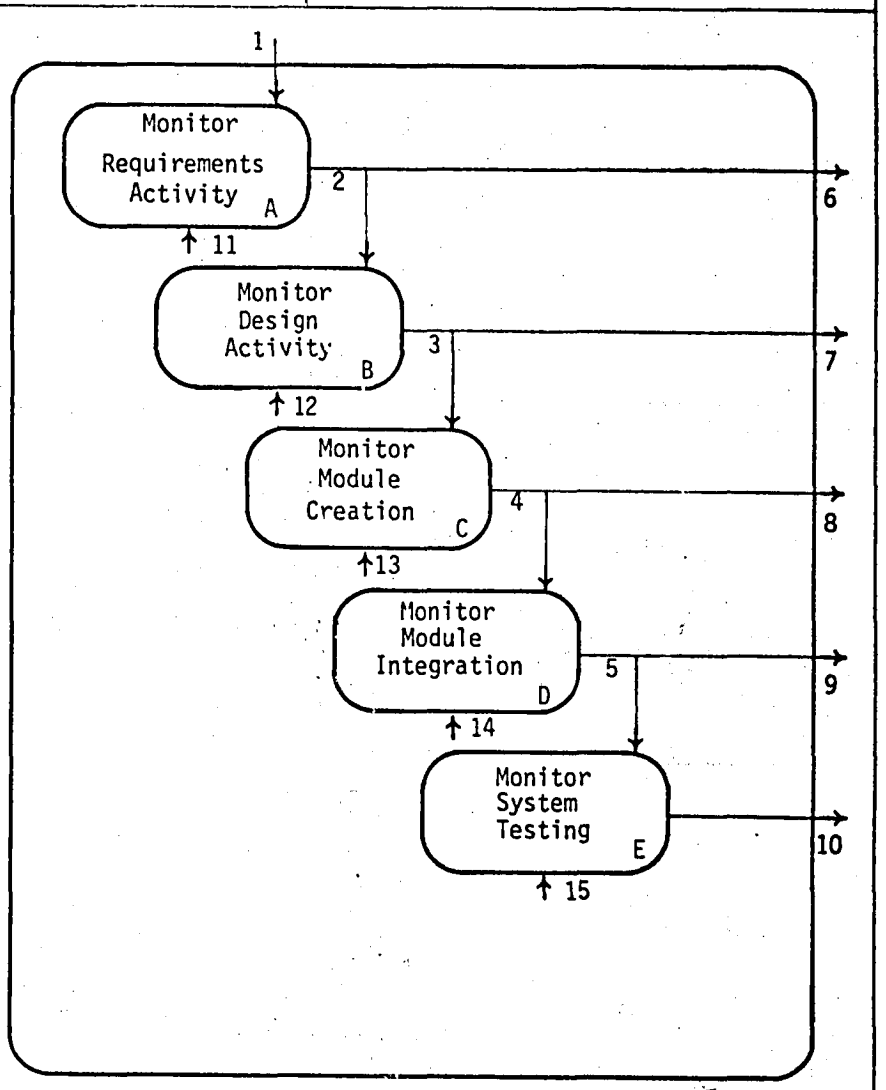
BCS BOEING COMPUTER SERVICES COMPANY

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 1015 ORIG. 2/78

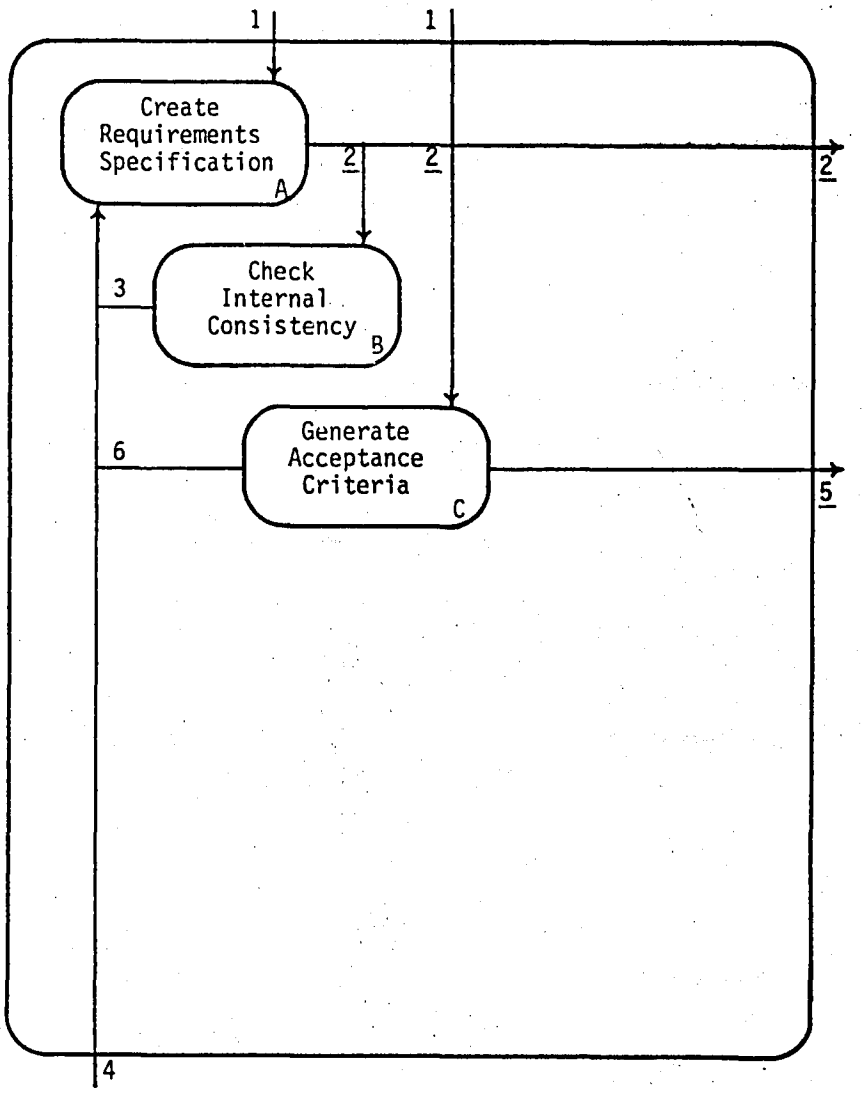
BCS BOEING COMPUTER SERVICES COMPANY

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 10167 (SAMM)

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

BCS BOEING COMPUTER SERVICES COMPANY

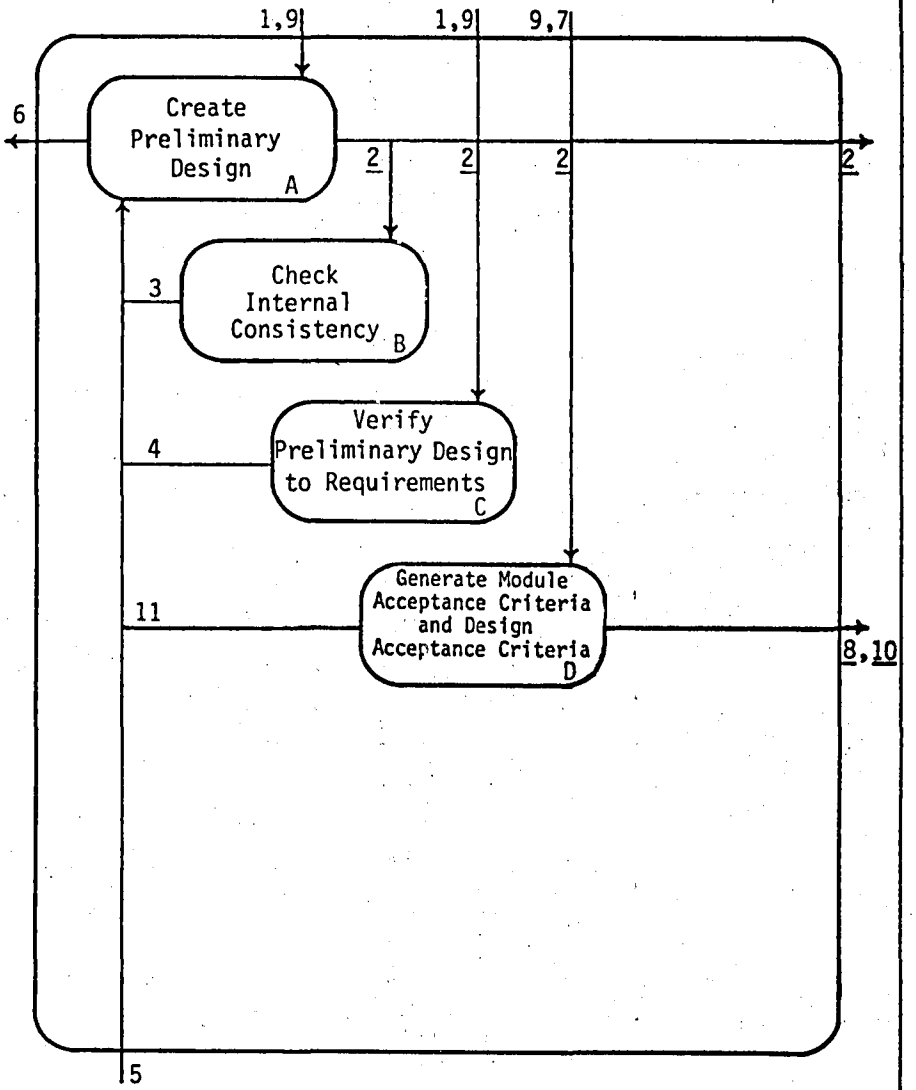
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: B

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 101E 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 INC

BCS BOEING COMPUTER SERVICES COMPANY

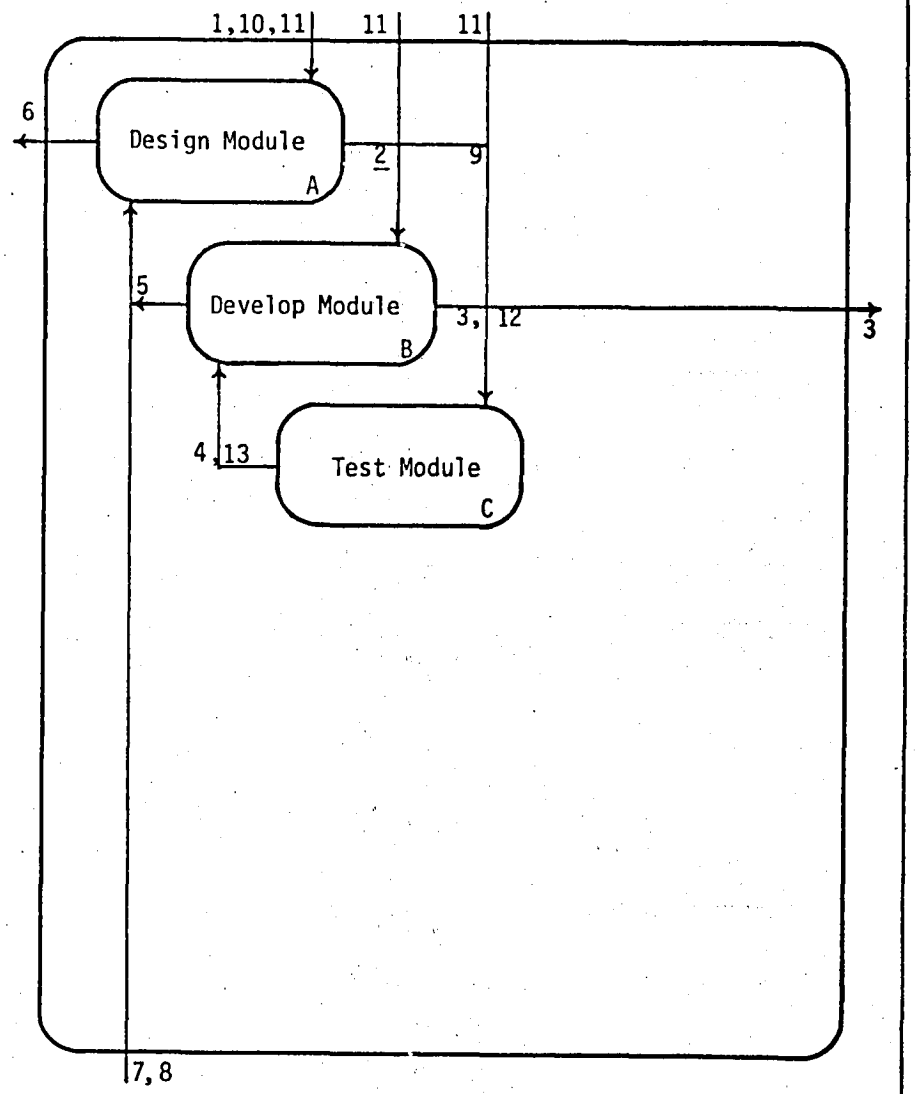
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: C

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		Generated test data
13		Machine state (at intermediate points of execution - for use by symbolic executor)

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

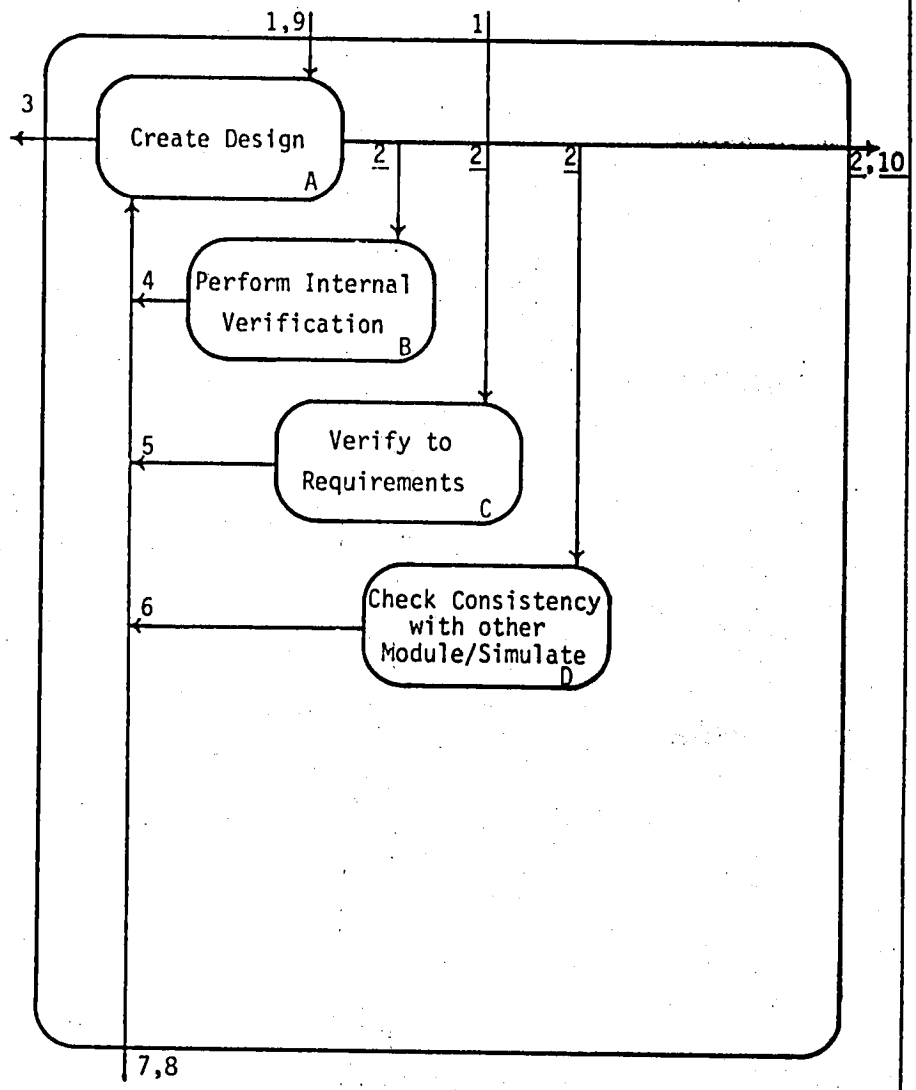
BCS BOEING COMPUTER SERVICES COMPANY

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



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1030 1015 ORIG. 2/78

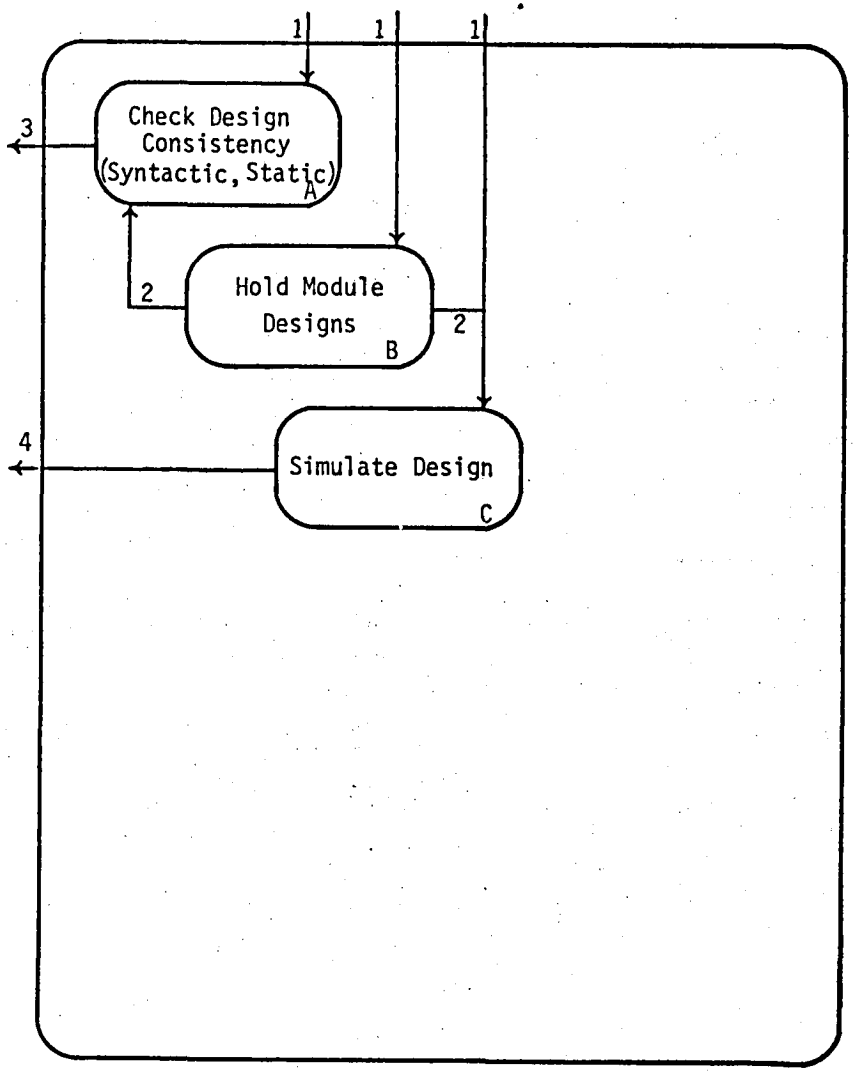
BGS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CAD

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



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 100C 1015 ORIG. 2/78

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

BCS BOEING COMPUTER SERVICES INC

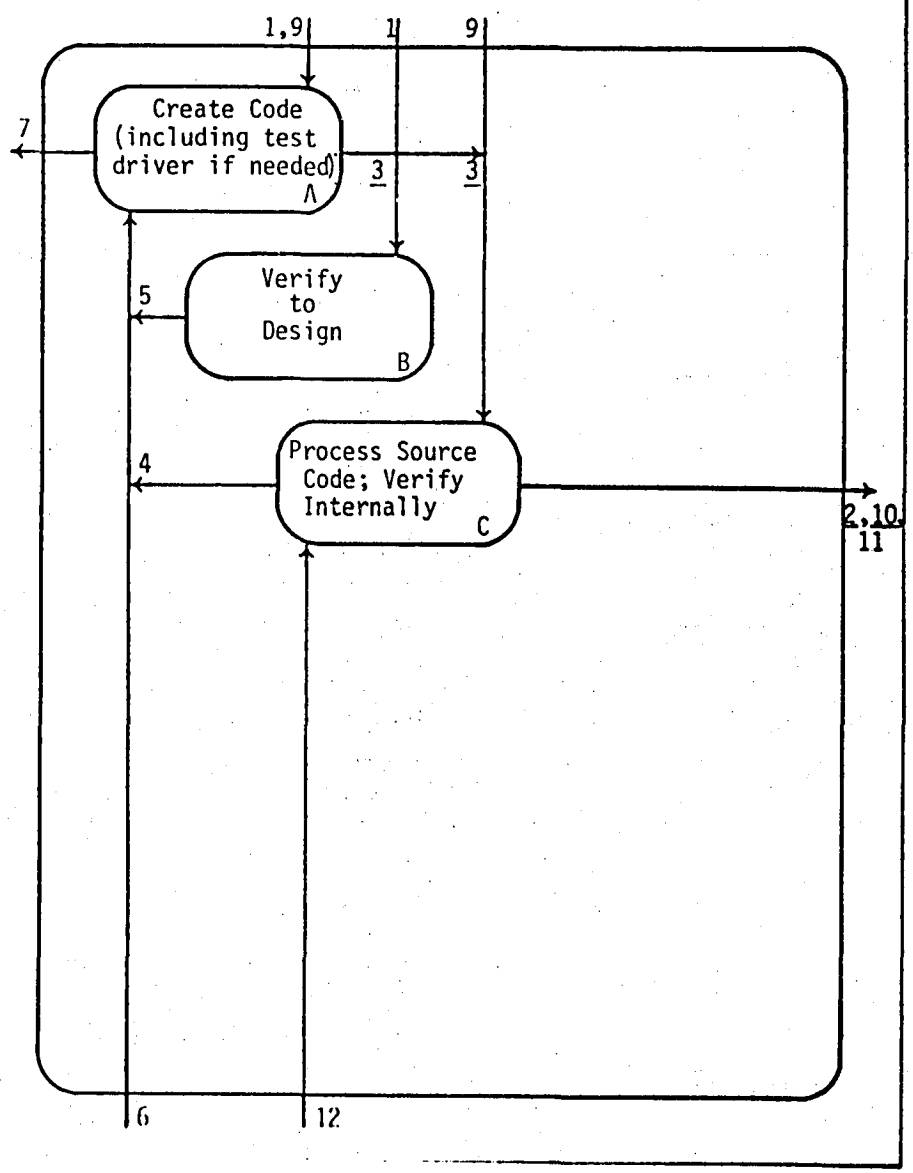
BOEING COMPUTER SERVICES COMPANY

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	3	HALMAT Monitor File for system testing
11	12	Generated test data
12	13	Machine state from incomplete execution



PREPARED BY DATE | REVIEWED BY DATE | APPROVED BY DATE

REF. DOCUMENT 10167 (SAMM)

CG 800 PUB. ORIG. 2/78

BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

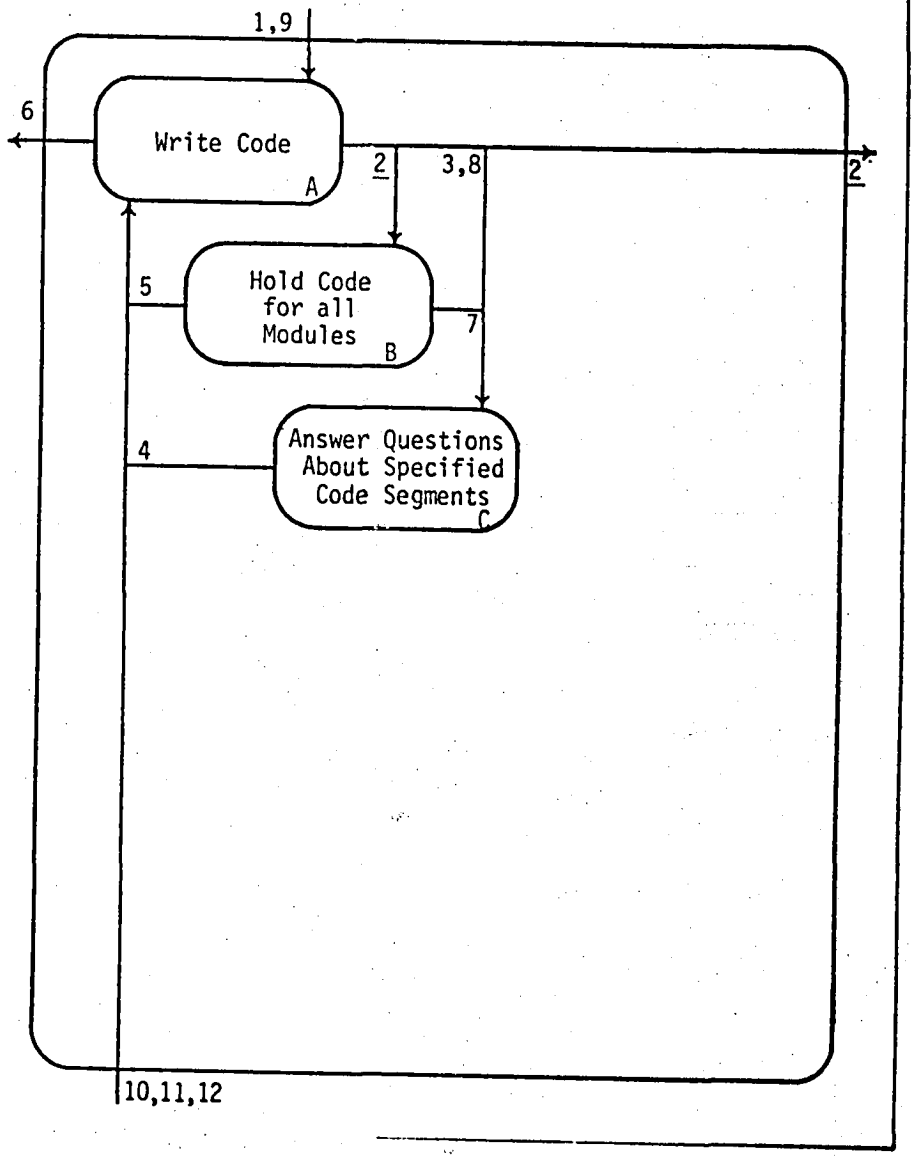
ACTIVITY DESIGNATOR: CBA

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 oversight
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 10167 (SAMM)



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 BOEING COMPUTER SERVICES COMPANY

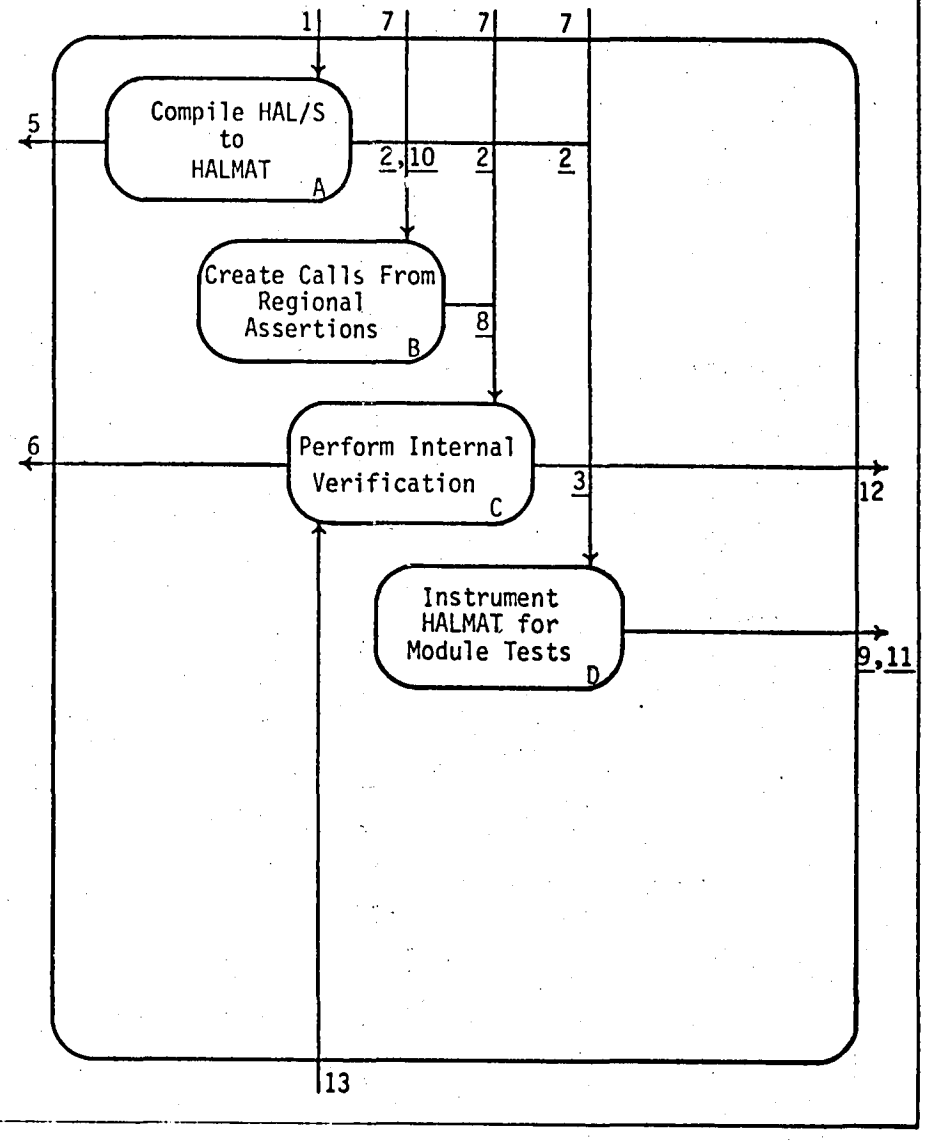
ACTIVITY - DATA FLOW DIAGRAM

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	10	HALMAT Monitor File containing monitor information to be expanded at system level
12	11	Generated test data
13	12	Machine state from partial execution

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



ACTIVITY DESCRIPTIONS

NODE CBC

TITLE

ACTIVITY

ID	DESCRIPTION
D	This activity "merges" the HALMAT and the HALMAT Monitor File - HALMAT level instructions which implement the active (or selected) monitors are inserted into the program HALMAT.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS BOEING COMPUTER SERVICES INC.

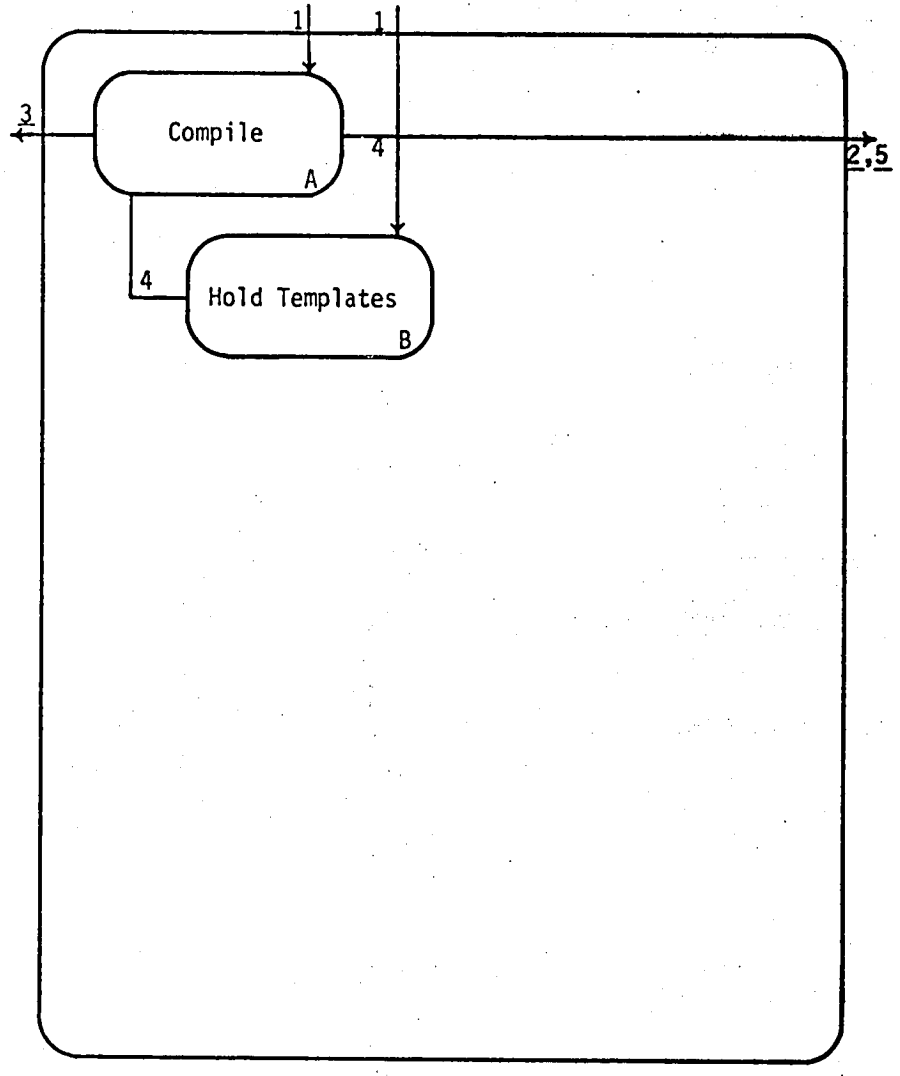
BCS BOEING COMPUTER SERVICES COMPANY

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 and source listing
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)

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 INC

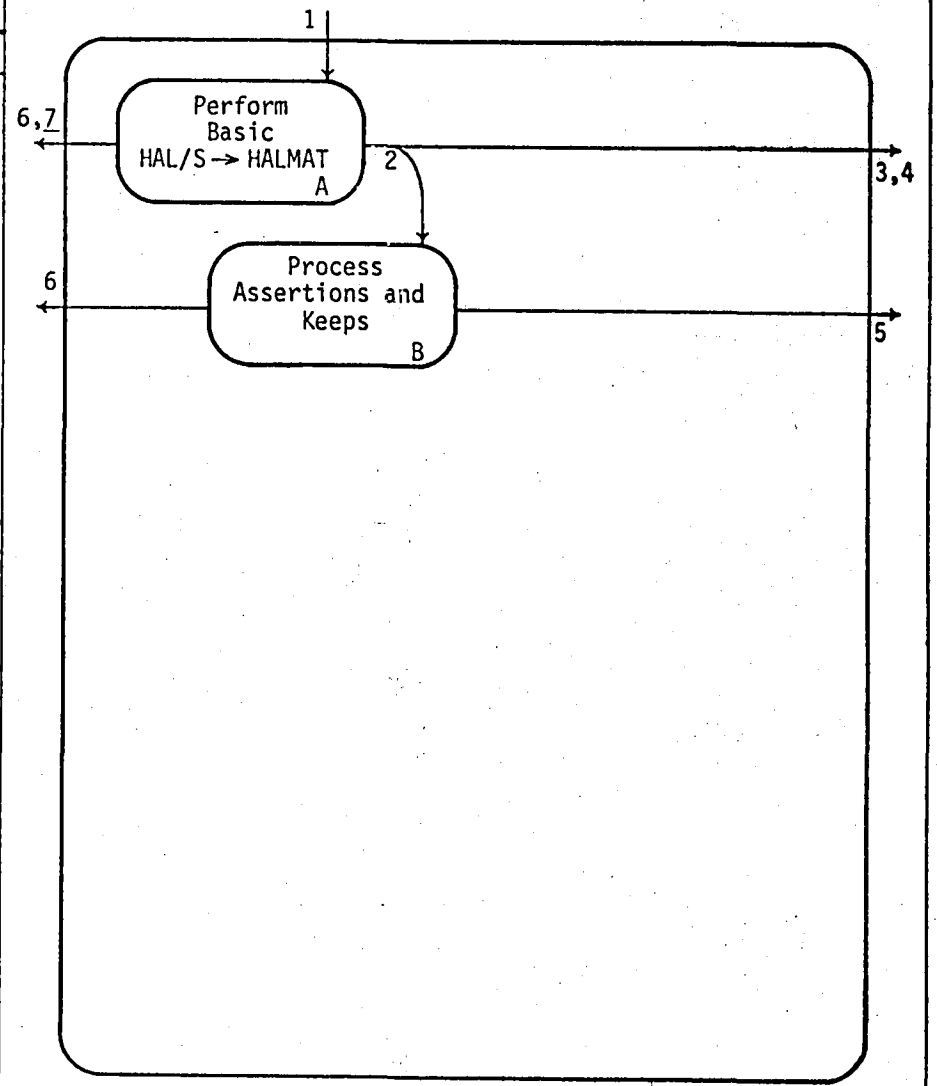
BCS BOEING COMPUTER SERVICES COMPANY

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) ASSERTS and KEEPS (and ENDS)
3	2	Stock HALMAT
4	4	Templates
5	5	Intermediate representation of asserts, keeps, etc. Contains pointers into HALMAT
6	3	Error messages (Syntax errors)
7	3	Program source listing



PREPARED BY DATE | REVIEWED BY DATE | APPROVED BY DATE

REF. DOCUMENT 10167 (SAMM)

CO 1003 1016 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CBCAA

TITLE

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

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.

RELATED ACTIVITIES

SOURCE	DESTINATION

BCS BOEING COMPUTER SERVICES COMPANY

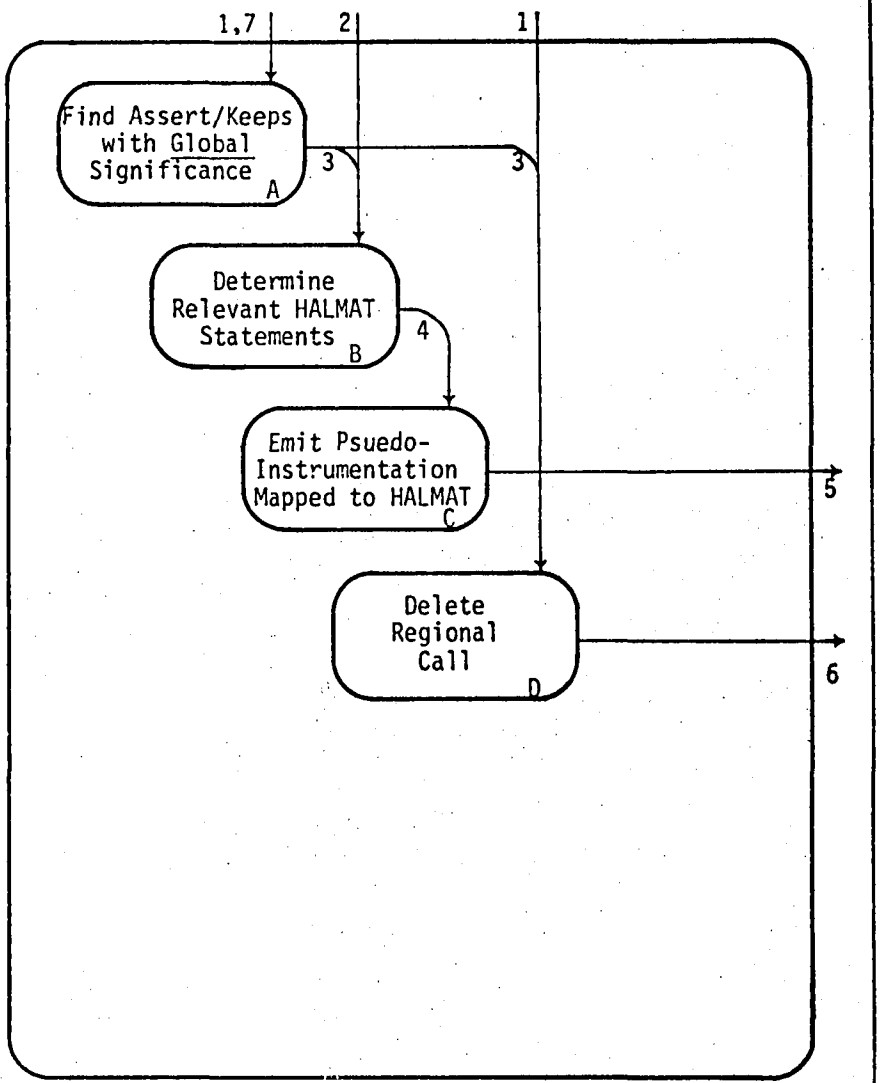
ACTIVITY - DATA FLOW DIAGRAM

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. DOCUMENT 10167 (SAMM)

BCS BOEING COMPUTER SERVICES COMPANY

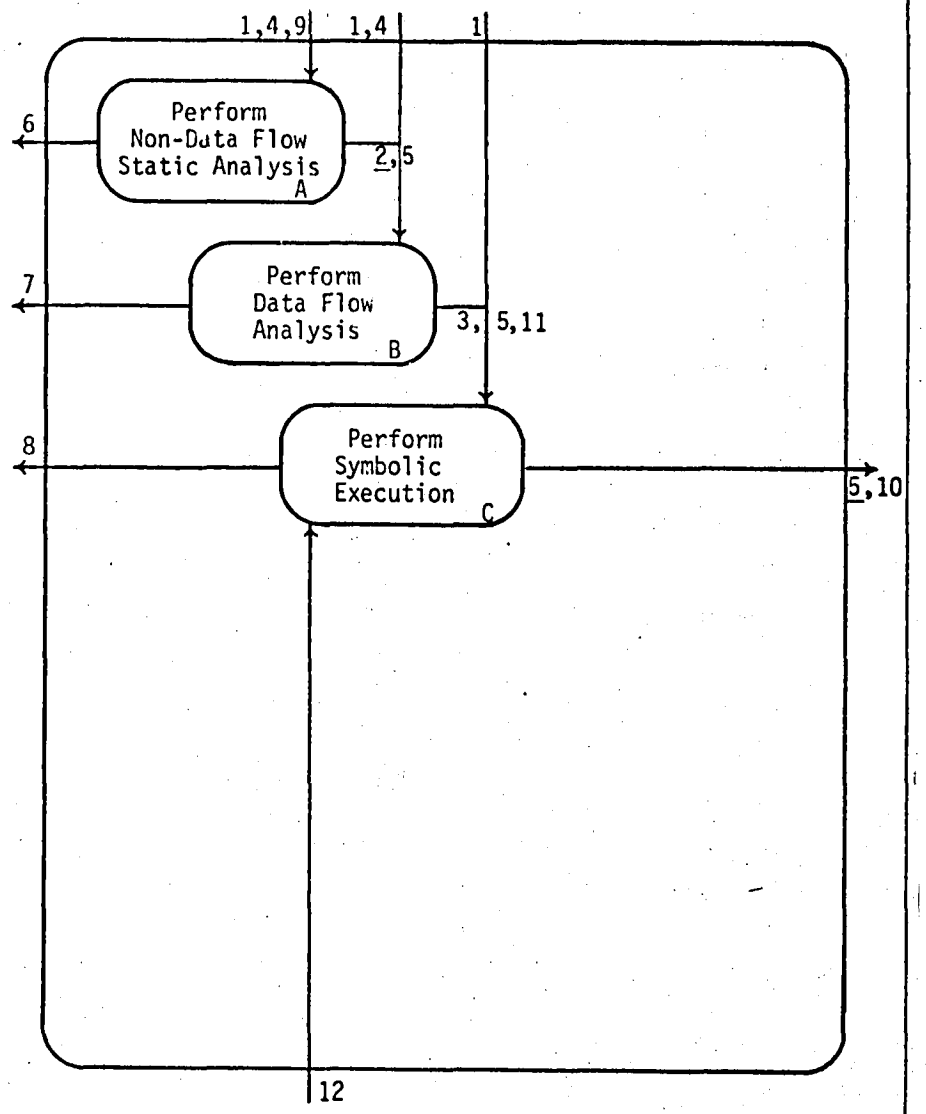
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCC

TITLE: PERFORM INTERNAL VERIFICATION

DATA ID	TRACE	DATA DESCRIPTION
1	2	HALMAT
2		Call graph
3		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 comprising revision requirements or the code
7	6	
8	6	
9	8	HALMAT Monitor File
10	12	Generated test data (for a specified program path)
11		Annotated program flowgraph
12	13	Machine state from partial execution

NOTE: Use of symbolic executor to examine this path is possible, and therefore indicated. However, significant human interaction with the executor is required and its utility is questionable.



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015, 0800, 2778

BCS

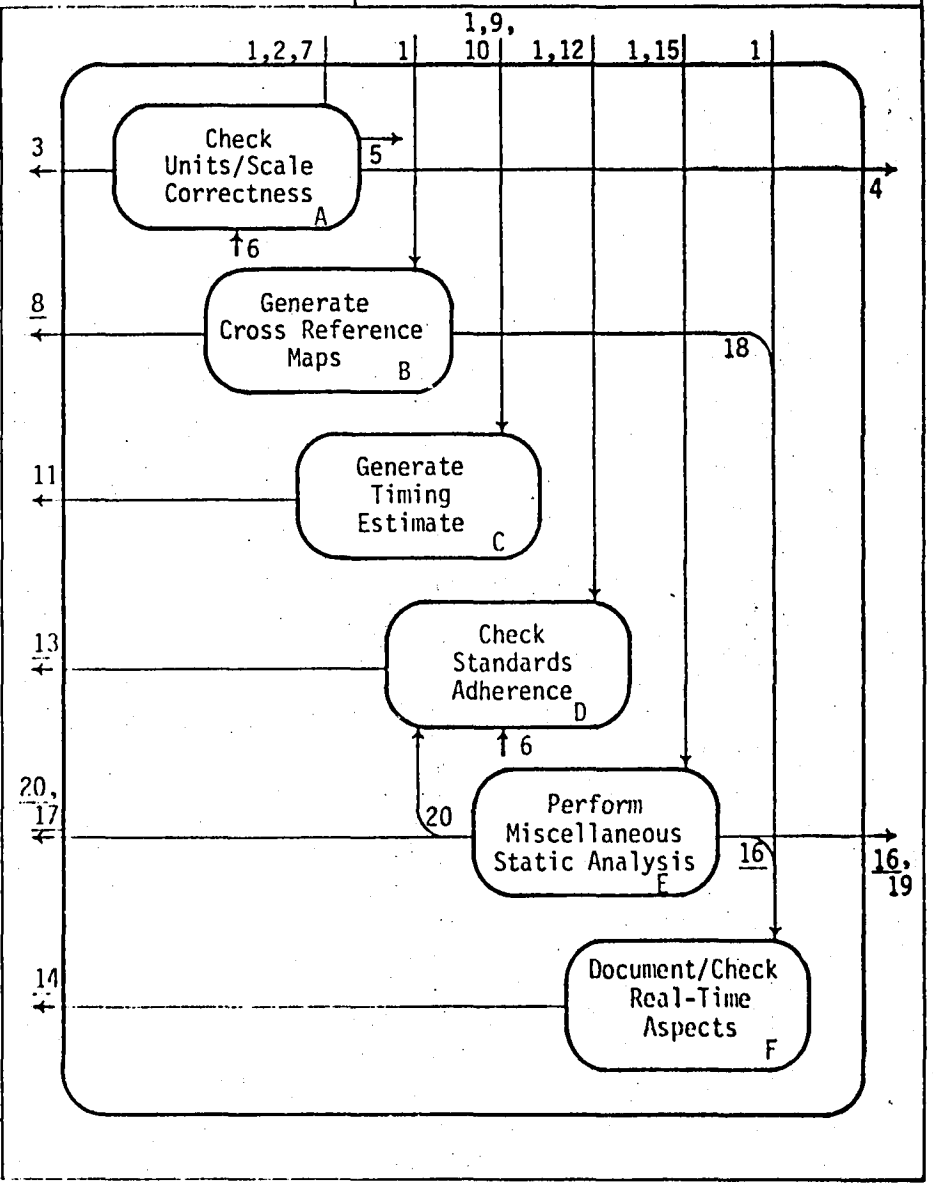
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCA

TITLE: PERFORM NON-DATA FLOW STATIC ANALYSIS

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT
2	9	Processed units/scale declarations vectors associated with variables matrices of relationships
3	6	Error messages
4	5	Specification for application of conversion factors to HALMAT
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 report, statistical summary
14	6	Documentation
15	4	Managerial input
16	2	Call graph
17	6	Error messages, documentation
18		Listing of shared variables
19	5	Monitors to add to Halmat
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.

BCS BOEING COMPUTER SERVICES COMPANY

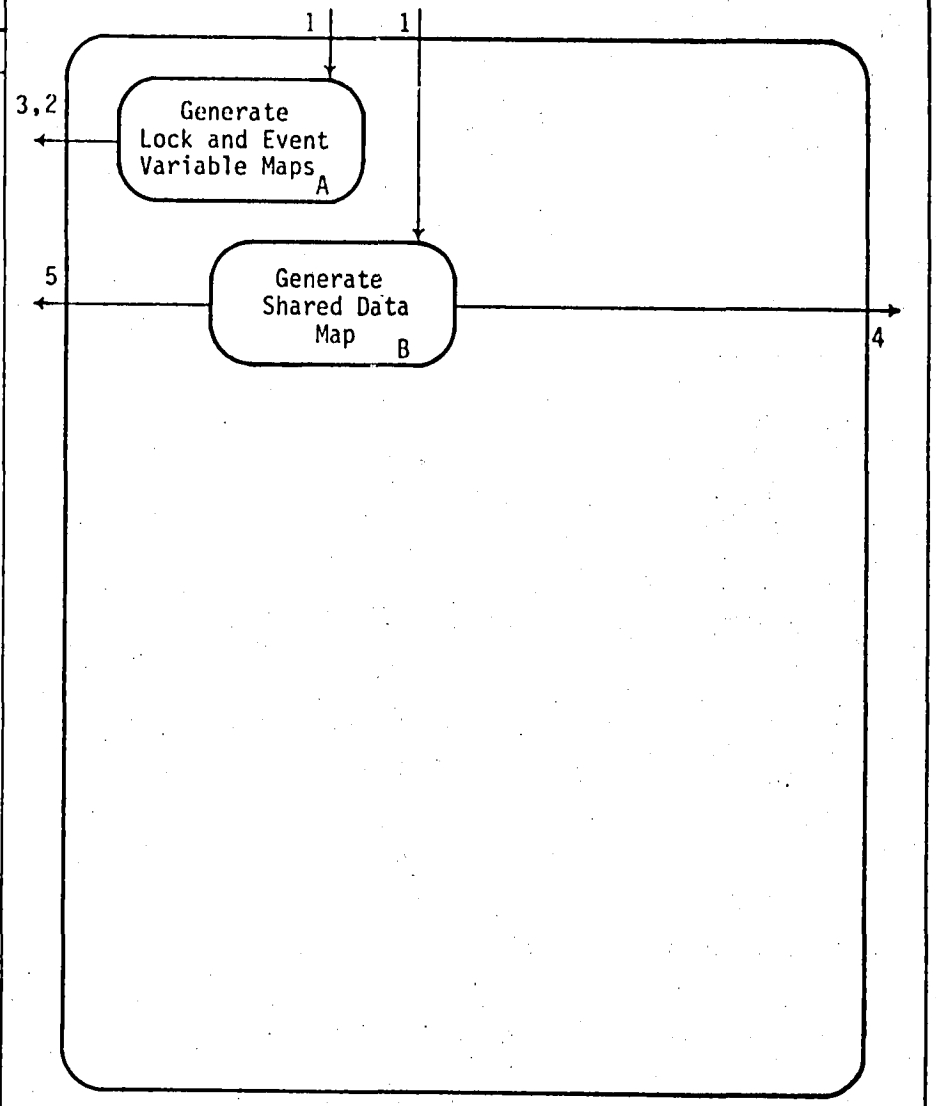
A Division of The Boeing Company

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 10167 (SAMM)

CO 1000 1016 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 information produced by activities A and B could easily be added onto the source listing produced by the compiler.
-	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

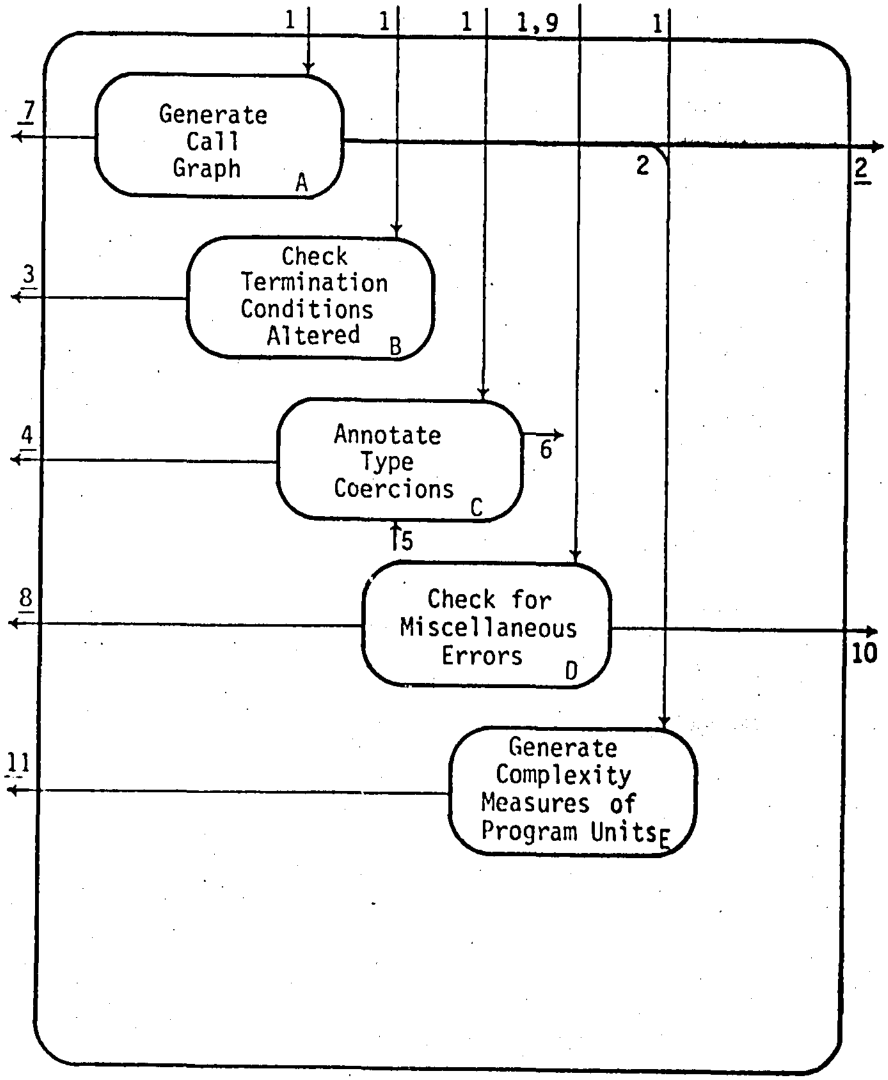
ID	SOURCE	DEST	NAME

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	19	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

ACTIVITY DESCRIPTIONS

NODE CBCCAE

TITLE

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).

RELATED DATA

ID	SOURCE	DEST	NAME

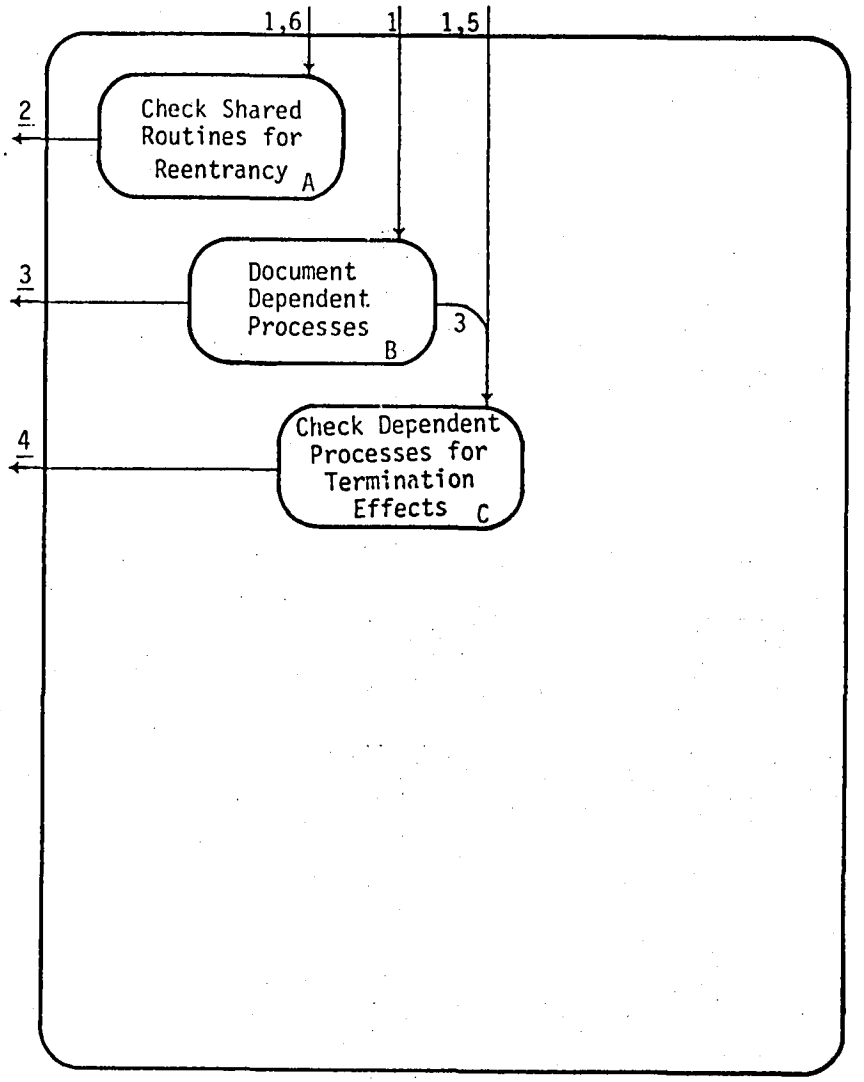
BGS BOEING COMPUTER SERVICES COMPANY

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)

CD 3000 0656 0001 2718

ACTIVITY DESCRIPTIONS

NODE CBCCAF

TITLE

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 inline functions should declare no data

RELATED DATA

ID	SOURCE	DEST	NAME

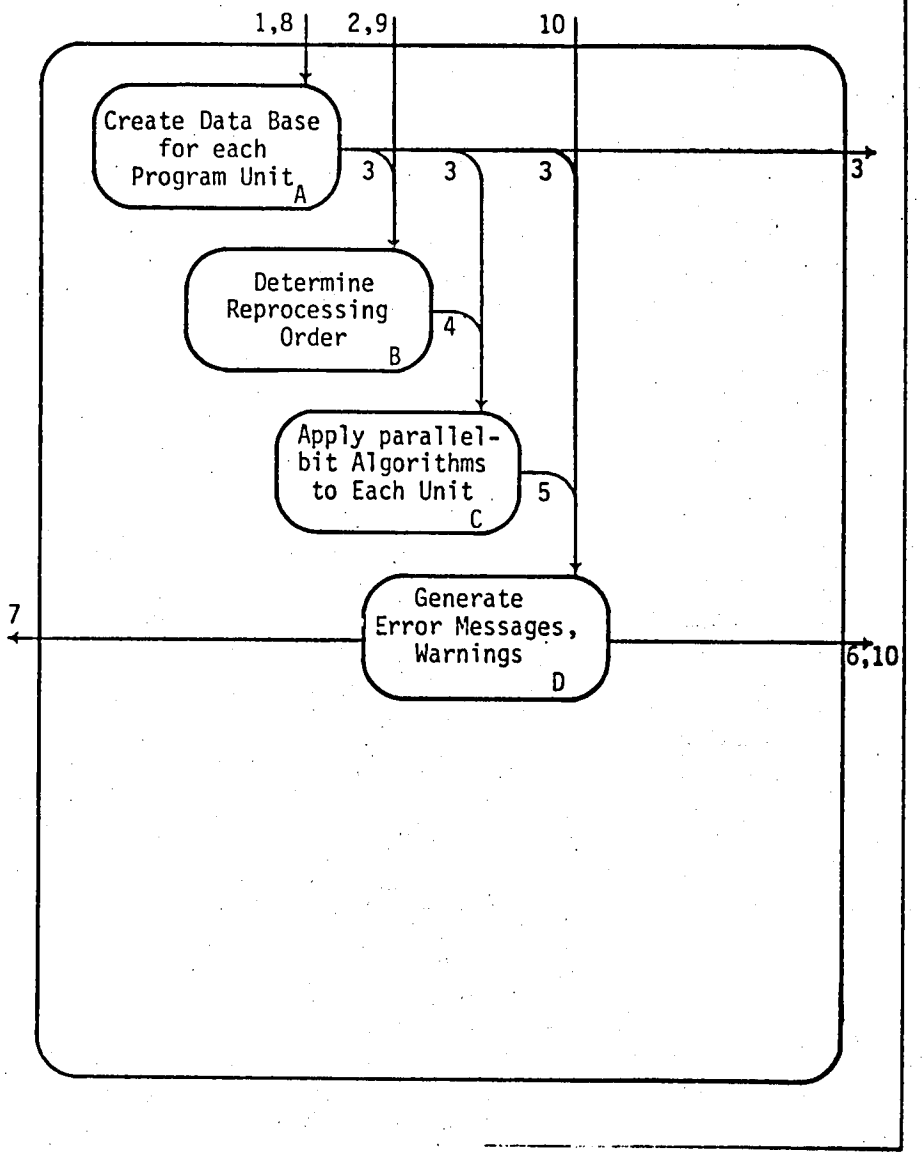
BCS

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	11	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	Helpful user oriented error messages
8	4	Provision for non-existent program units
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 10167 (SAMM)

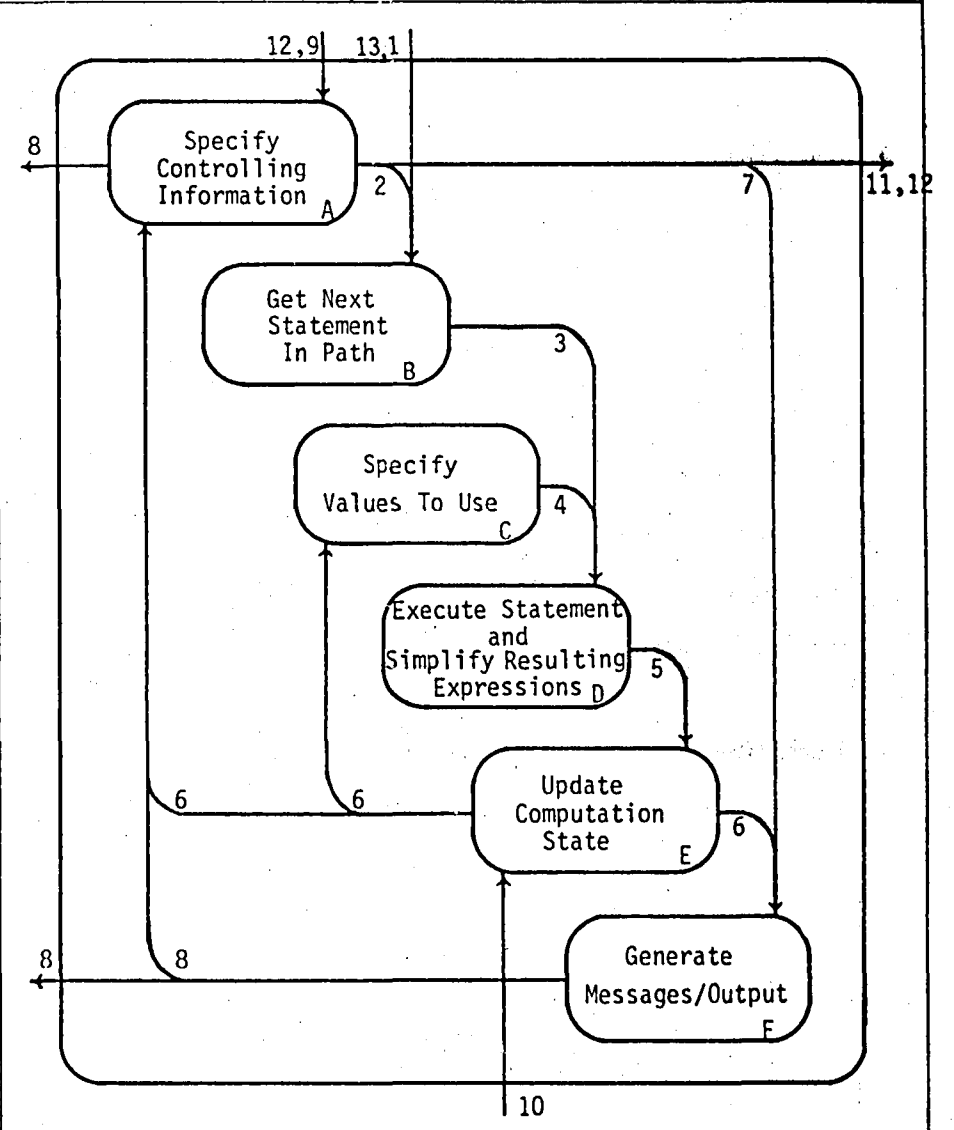
BGS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CBCCC

TITLE
PERFORM SYMBOLIC EXECUTION

DATA ID	TRACE	DATA DESCRIPTION
1	1	HALMAT (program specification)
2		Determination of what to do on branching, loops, etc.
3		The statement to execute
4		Values to be used in statement execution
5		Updates to computation state: statement pointer and data values
6		Computation state
7		Specification of output desired
8	8	Output/Error messages
9	3	(Possibly incomplete) path specification from static analyzers
10	12	Specification of machine state (from partial execution)
11	10	Generated test data (to force execution of a specific path)
12	5	HALMAT Monitor File
13	11	Program flowgraph



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1016 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CBCCC

TITLE PERFORM SYMBOLIC EXECUTION

ACTIVITY

ID	DESCRIPTION
	This breakdown specifies an incremental symbolic executor: path selection, value specification, output generation, and constraint solving can all occur after "execution" of each statement. As such it allows highly interactive use, yet may also be used in batch mode given adequate controls and path specifications.

RELATED DATA

ID	SOURCE	DEST	NAME

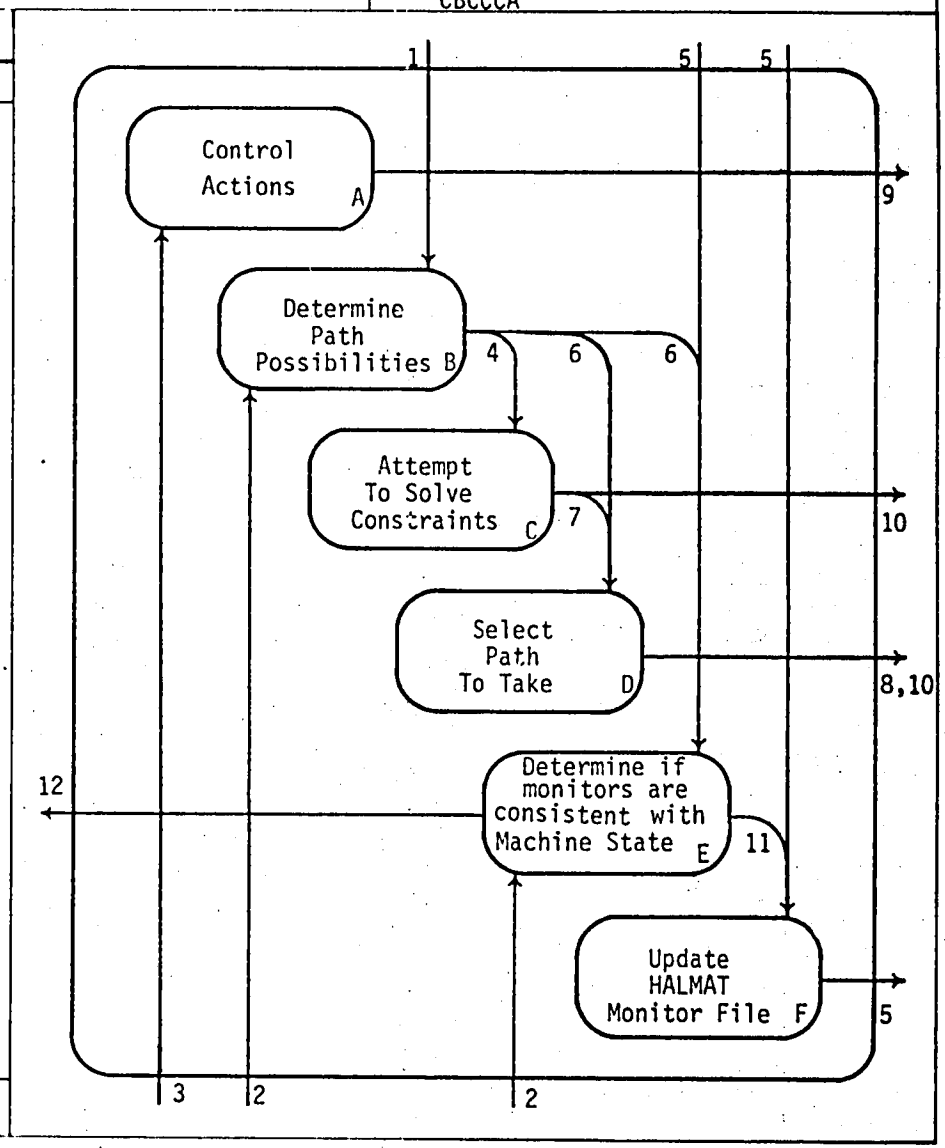
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CBCCCA

TITLE: SPECIFY CONTROLLING INFORMATION

DATA ID	TRACE	DATA DESCRIPTION
1	9	(Possibly incomplete) path specifications
2	6	Machine state (instruction pointer and data values)
3	8	Output/Error messages
4		Constraints on following a particular path
5	12	HALMAT Monitor File
6		List of potential paths
7		Feasibility of a particular path
8	2	Path specification
9	7	Specification of desired output
10	11	Test data required to force execution of a particular path
11		Monitor file update requirements
12	8	Indication of verified assertions



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

BCS BOEING COMPUTER SERVICES COMPANY

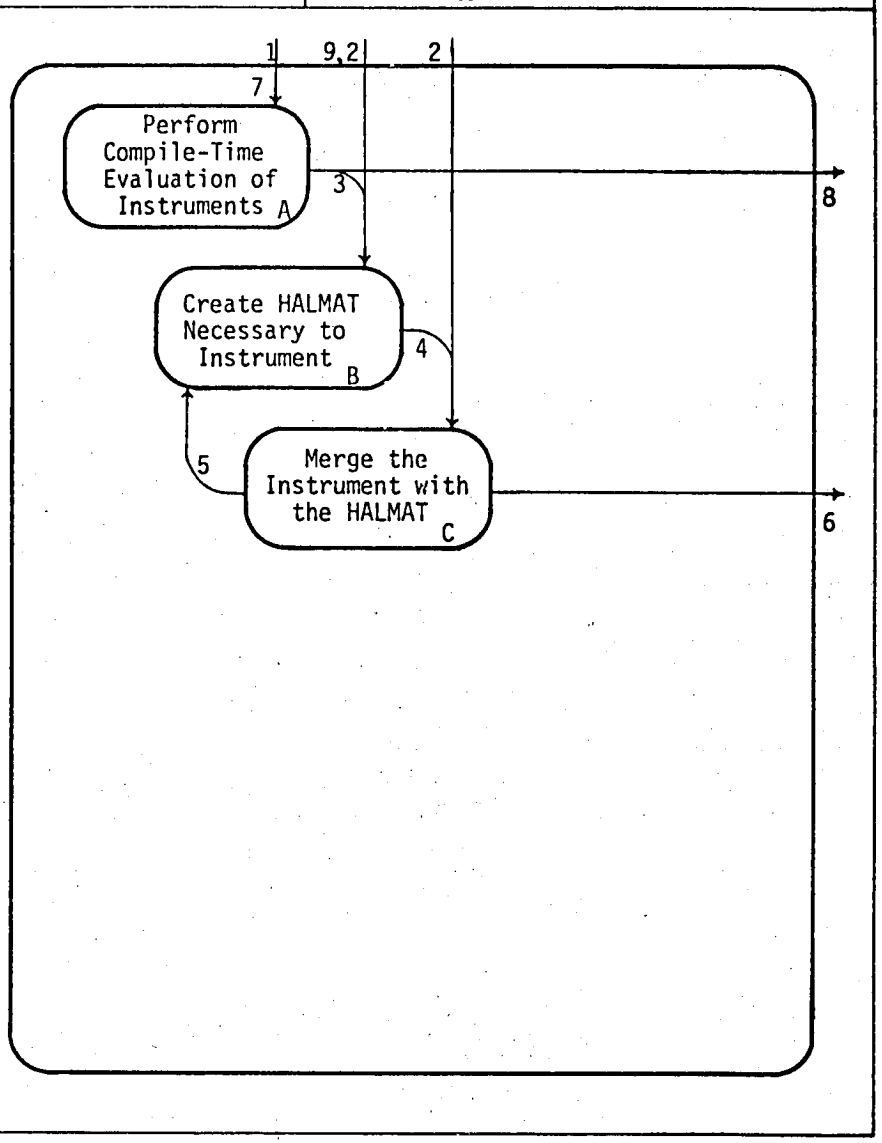
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
CBCD

TITLE
INSTRUMENTATION, et al

DATA ID	TRACE	DATA DESCRIPTION
1	3	Instrument file (parallel to HALMAT)
2	2	HALMAT
3		"Pared-down" instrument file (only selected instruments will be inserted)
7	7	Controlling/overriding input
4		HALMAT which represents the instrument
5		Updated HALMAT (pointers changed)
6	9	Fully updated HALMAT
8	11	Instruments (calls) which could not/ need not be expanded at this level - will be expanded at system level
9	7	Precompiled assertion procedures

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

BCS BOEING COMPUTER SERVICES COMPANY

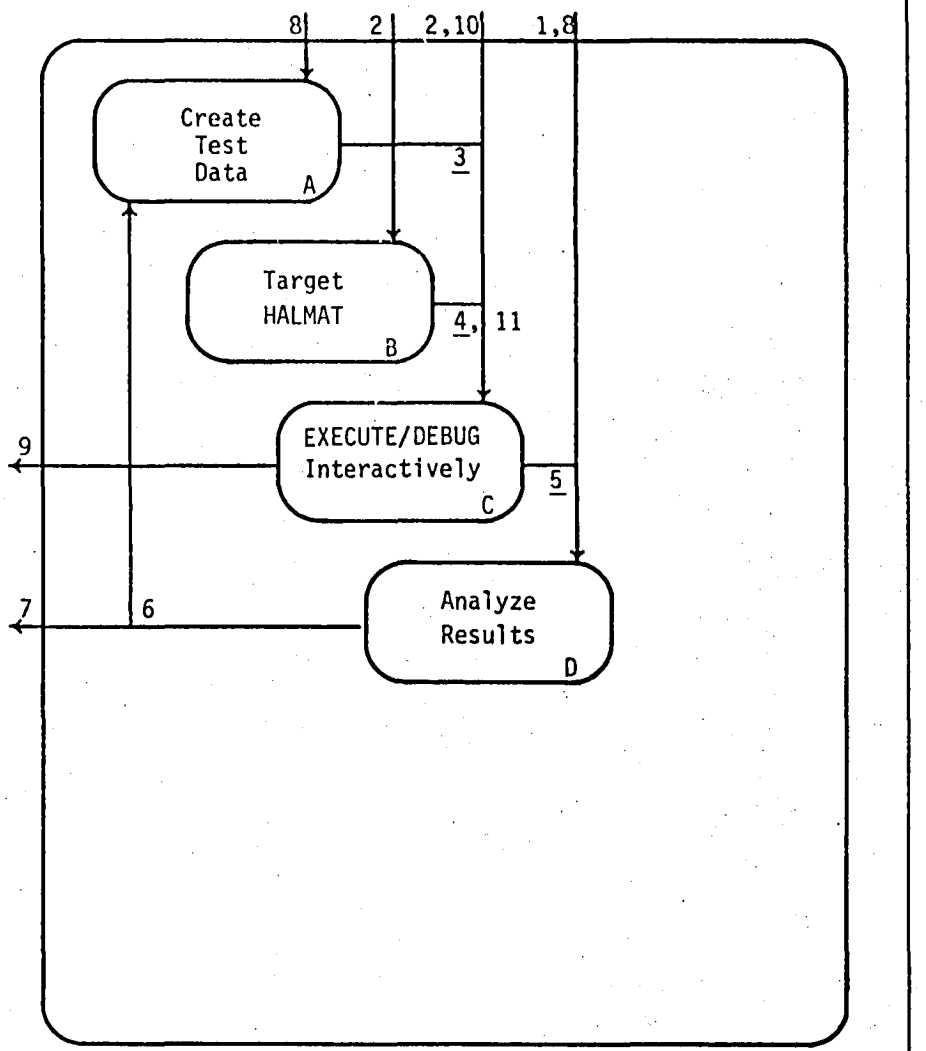
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CC

TITLE: TEST MODULE

DATA ID	TRACE	DATA DESCRIPTION
1	11	Managerial input
2	3	HALMAT (includes instrumentation)
3		Test data
4		Executable code
5		Results (output)
6		Requirements for additional test data
7	4	Code revision specifications
8	9	Module acceptance criteria
9	13	Partial machine state for use by symbolic executor
10	12	Generated test data (from symbolic executor)
11		Mapping of variables to target machine "symbols". (Required, along with the symbol table contained in the HALMAT, for generation of post-mortem dumps)

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

ACTIVITY DESCRIPTIONS

NODE CC

TITLE Test Module

ACTIVITY

ID	DESCRIPTION
	Three aspects of this activity, nodes A, C, and D 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 correctness. Such an apparatus is especially useful during retesting which is required as the result of program modifications.
B	Given HALMAT and a specificationof 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.
C	The interactive debugging of any program will be dependent on the target machine supporting such activities. Instrumentation to support such actions may be inserted at any of the previous instrumentation steps. In addition to providing the "standard" functional capabilities expected of interactive debuggers, the interactive debugger should be of sufficient sophistication

RELATED DATA

ID	SOURCE	DEST	NAME

ACTIVITY DESCRIPTIONS

NODE CC (continued) TITLE Test Module

ACTIVITY

ID	DESCRIPTION
	to allow the user to specify: 1) an (arbitrary) point to begin execution. 2) a point at which to halt execution. 3) initial values for all variables. 4) Which variable values to display when execution halts. Such a capability will greatly facilitate effective functional testing.

RELATED DATA

ID	SOURCE	DEST	NAME

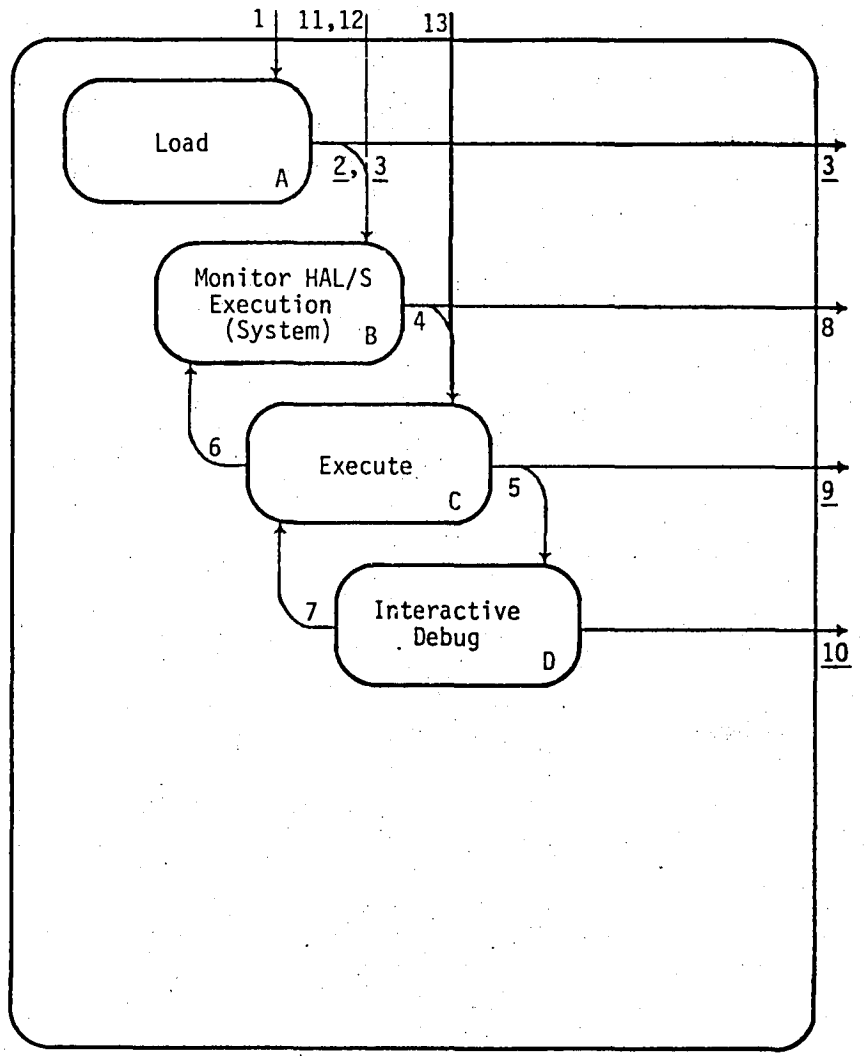
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCC

TITLE: EXECUTE AND DEBUG INTERACTIVELY

DATA ID	TRACE	DATA DESCRIPTION
1	4	Executable code ("relocatable binary")
2		Load image
3	5	Load output-documentation
4		System control over executing program
5		Machine/program state
6		Calls to system monitor
7		Revised machine/program state
8	5	Raw performance/histogram/history data/post-mortem dump
9	5	Program output
10	5	Performance/execution characteristic output (trace/session transcript)
11	2	Symbol table (from HALMAT)
12	11	Mapping of symbol table to target machine symbols
13	3,10	Test data



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CCC

TITLE Execute/Debug Interactively

ACTIVITY

ID	DESCRIPTION
D	Interactive debuggers have been around for some time and they are well understood. No "novel" features are planned for this debugger, except the ability to aid in transfering the machine state resulting from partial execution to the symbolic executor. For an excellent consideration of interactive debugging systems, including an extensive annotated bibliography, see (Johnson, M.S., 1978). When the design for this debugger is embarked upon the basic principles guiding the design of the total environment must be held paramount: the facilities of the tool should not overlap those of another, capabilities should not appear which would better occur elsewhere, and the user modes it will appear in must be remembered. It is our contention that the presence of a suite of verification tools, notably static and dynamic analysis, will remove much of the need for interactive debuggers. The most profitable use of the interactive debugger will be in performing functional testing. Some of the acceptance criteria received by node CCA, Create Test Data, may only require that a relatively small portion of code be executed. If the debugger allows the user to start execution at (almost) any location

RELATED DATA

ID	SOURCE	DEST	NAME

ACTIVITY DESCRIPTIONS

NODE CCC

TITLE

ACTIVITY

ID	DESCRIPTION
	(using a sufficient set of user supplied initial data values), stop execution at any point, and display values at arbitrary points, then such testing can be performed easily and cheaply.

RELATED DATA

ID	SOURCE	DEST	NAME

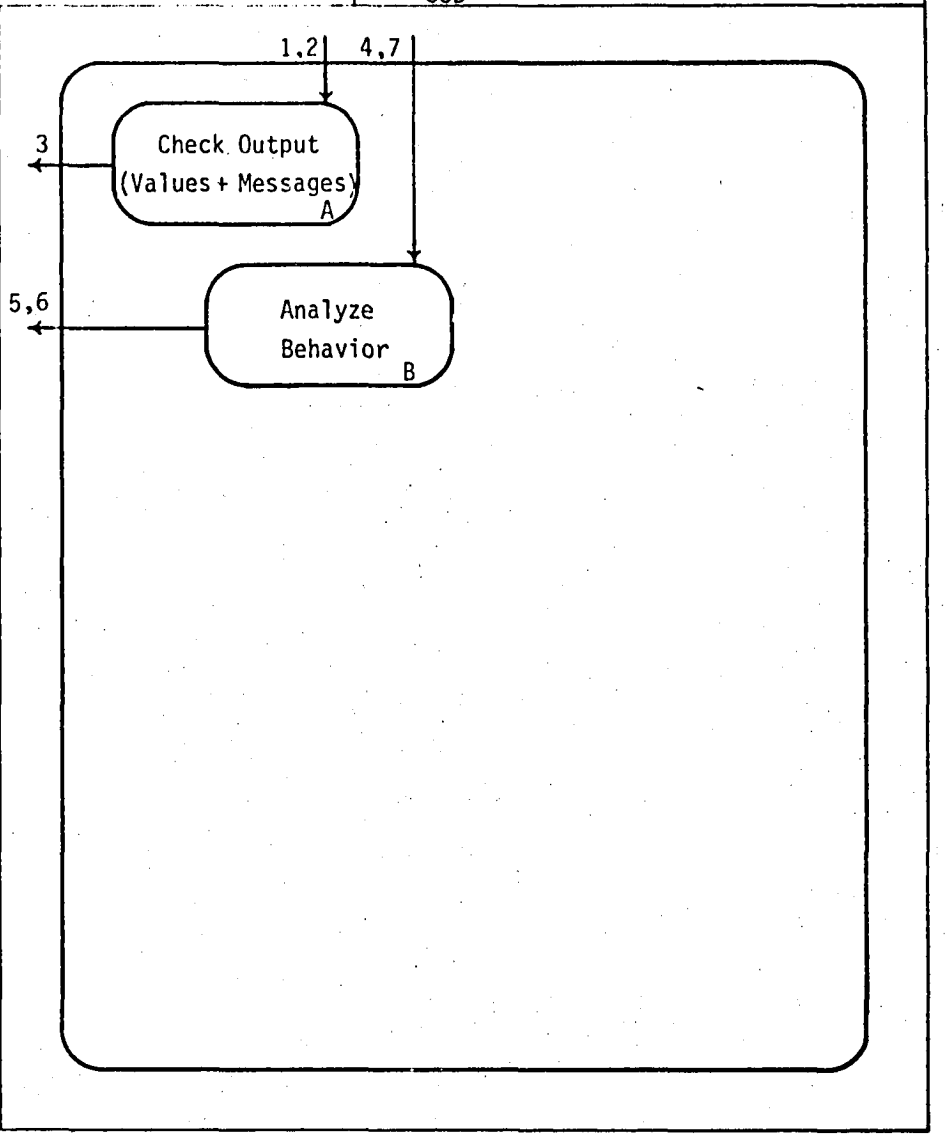
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCD

TITLE: ANALYZE RESULTS

DATA ID	TRACE	DATA DESCRIPTION
1	5	Output values and messages
2	8	Acceptance criteria
3	7	Code revision specifications
4	5	Behavioral information
5	7	Code revision specifications
6	6	Additional testing requirements
7	1	Managerial input



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

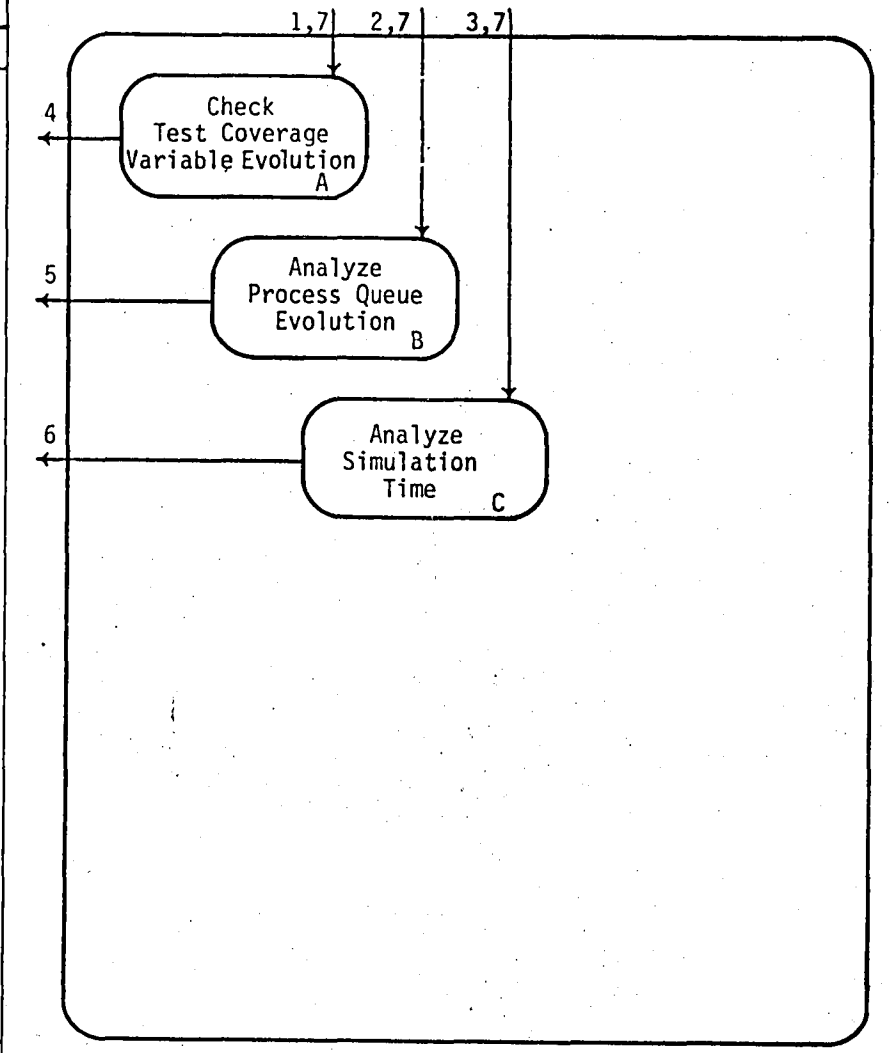
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCDB

TITLE: ANALYZE BEHAVIOR (PERFORMANCE)

DATA ID	TRACE	DATA DESCRIPTION
1	4	Histograms (frequency count, branch paths, statistics) + variable maximum and minimums
2	4	Process queue snapshots
3	4	Simulated time output
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 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

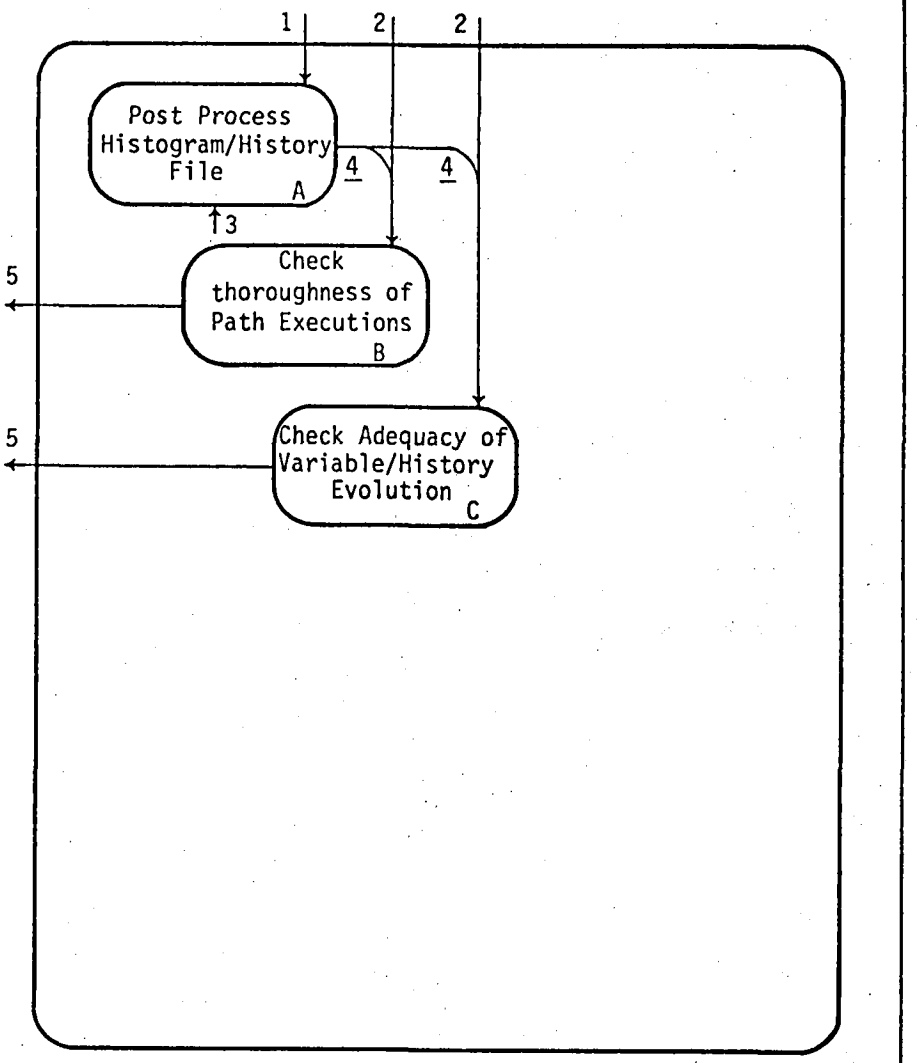
BCS BOEING COMPUTER SERVICES COMPANY

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: CCDBA

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 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

ACTIVITY DESCRIPTIONS

NODE CCDBA

TITLE

ACTIVITY

ID	DESCRIPTION
A	The source listing produced by the compiler is an excellent document to be annotated with the histogram information. Multiple statements per line are broken up to several lines, all statements are pretty printed, and the format is uniform.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS BOEING COMPUTER SERVICES COMPANY

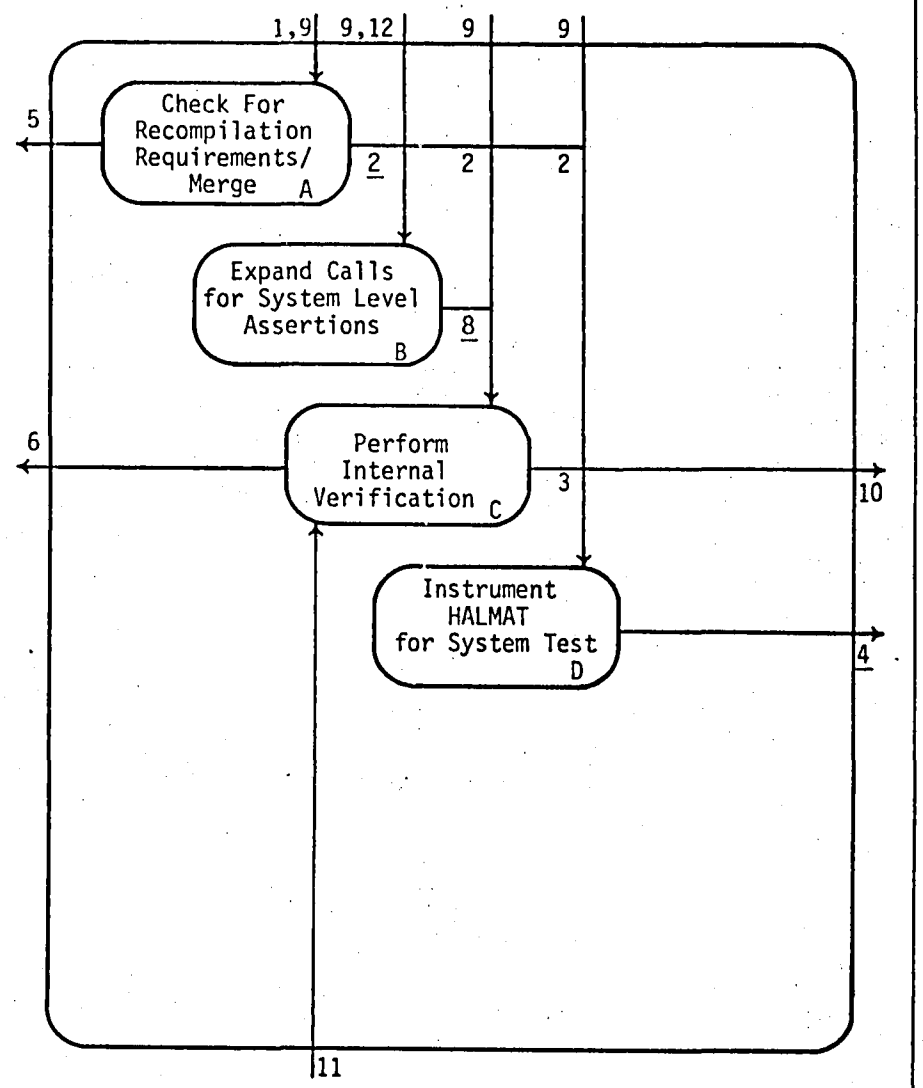
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
D

TITLE: INTEGRATE MODULES INTO SYSTEM

DATA ID	TRACE	DATA DESCRIPTION
1	6	HALMAT of each module
2		Collection of modules forming system
3		Further altered HALMAT Monitor File
4	8	HALMAT ready for execution as a complete system
5	7	Recompilation requirements due to factors such as "independent testing used outdated module templates"
6	7	Verification based revision requirements
8		Expanded HALMAT Monitor File
9	12	Managerial input/controlling parameters
10	15	Generated test data
11	16	Machine state from partial execution
12	6	HALMAT Monitor Files

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

BCS BOEING COMPUTER SERVICES COMPANY

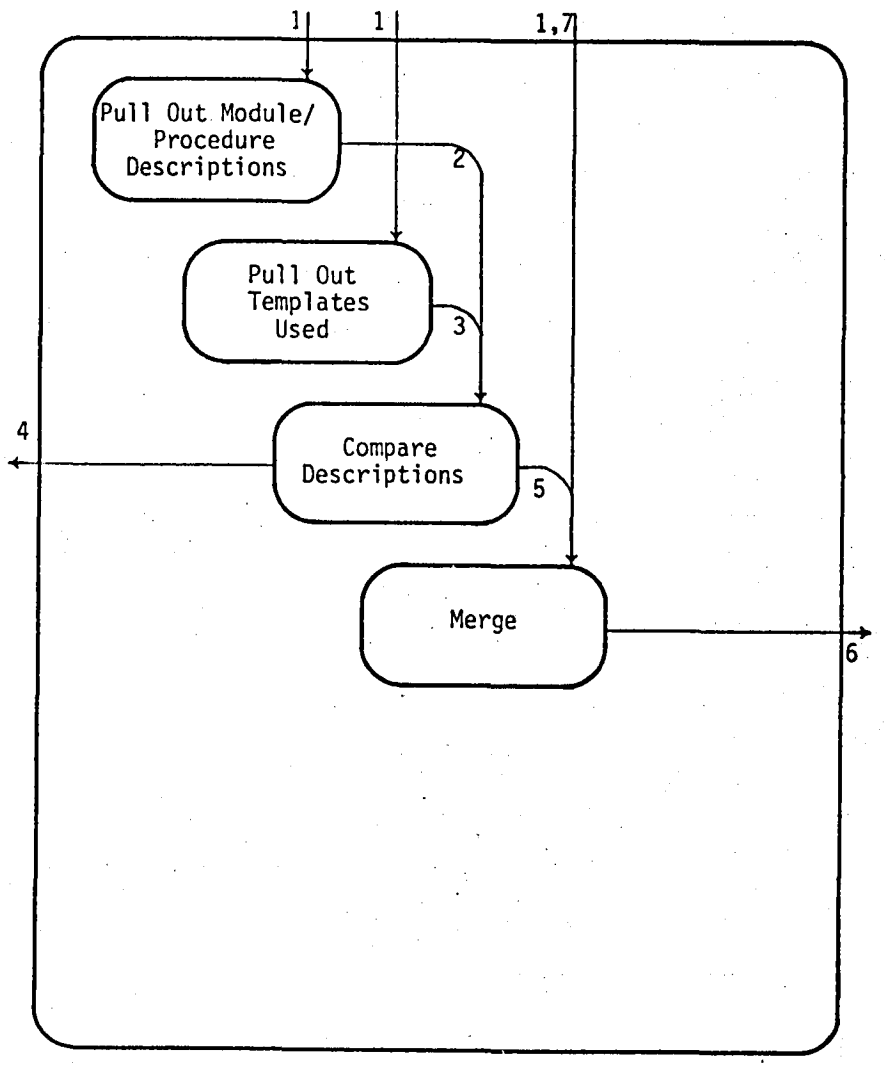
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
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	5	List of mismatched procedure/ templates ⇒ recompilation/revision requirements
5		List of modules which may be safely merged
6	2	System of legally merged modules
7	9	Directions as to merger (e.g., override) - may be unnecessary or undesirable

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

BCS

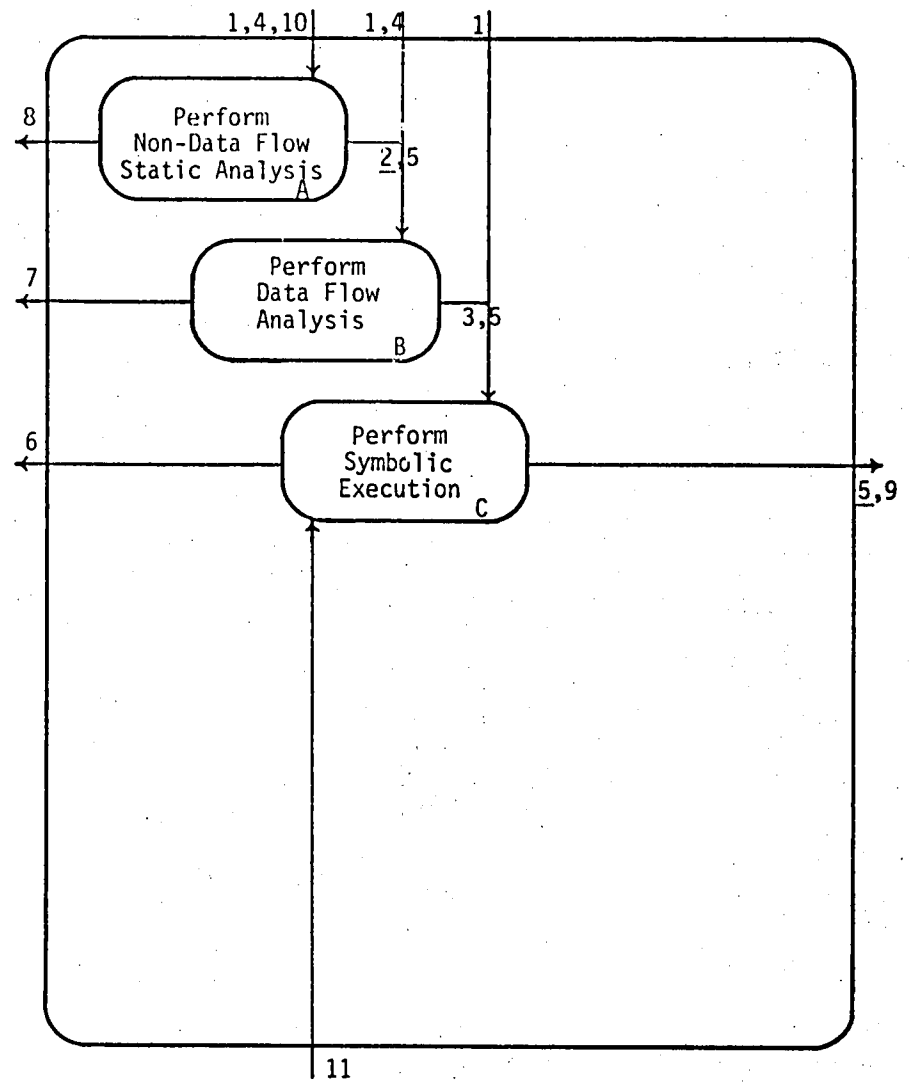
ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR: DC

TITLE: PERFORM INTERNAL VERIFICATION

DATA ID	TRACE	DATA DESCRIPTION
1	2	HALMAT for all modules
2		System call graph
3		Specifications of system paths on which errors are suspected to exist
4	9	Managerial input
5	3	Refined HALMAT Monitor File
6	6	Error messages and documentation comprising revision requirements
7	6	
8	6	
9	10	Generated test data
10	8	HALMAT Monitor File
11	11	Machine state from incomplete execution

PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE



REF. DOCUMENT 10167 (SAMM)

CO 1000 1015 ORIG. 2/78

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

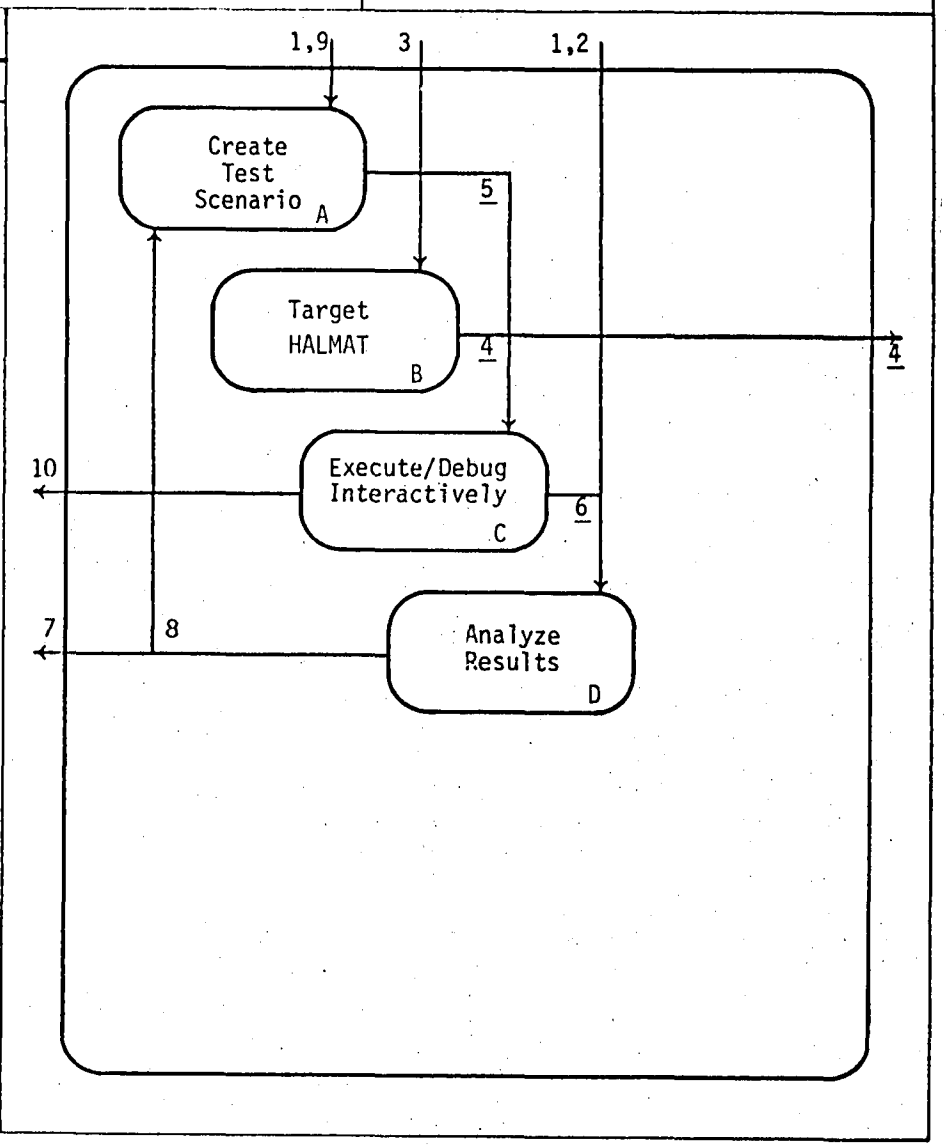
BCS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
E

TITLE
TARGET AND TEST SYSTEM

DATA ID	TRACE	DATA DESCRIPTION
1	13	System acceptance criteria
2	12	Managerial input
3	8	HALMAT comprising system
4	14	Executable code
5		Test data
6		Output
7	9	Module revision requirements
8		Requirements for additional test data
9	15	Generated test data
10	16	Machine state at a specified point of computation (for use by symbolic executor)



PREPARED BY	DATE	REVIEWED BY	DATE	APPROVED BY	DATE

REF. DOCUMENT 10167 (SAMM)

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 INC

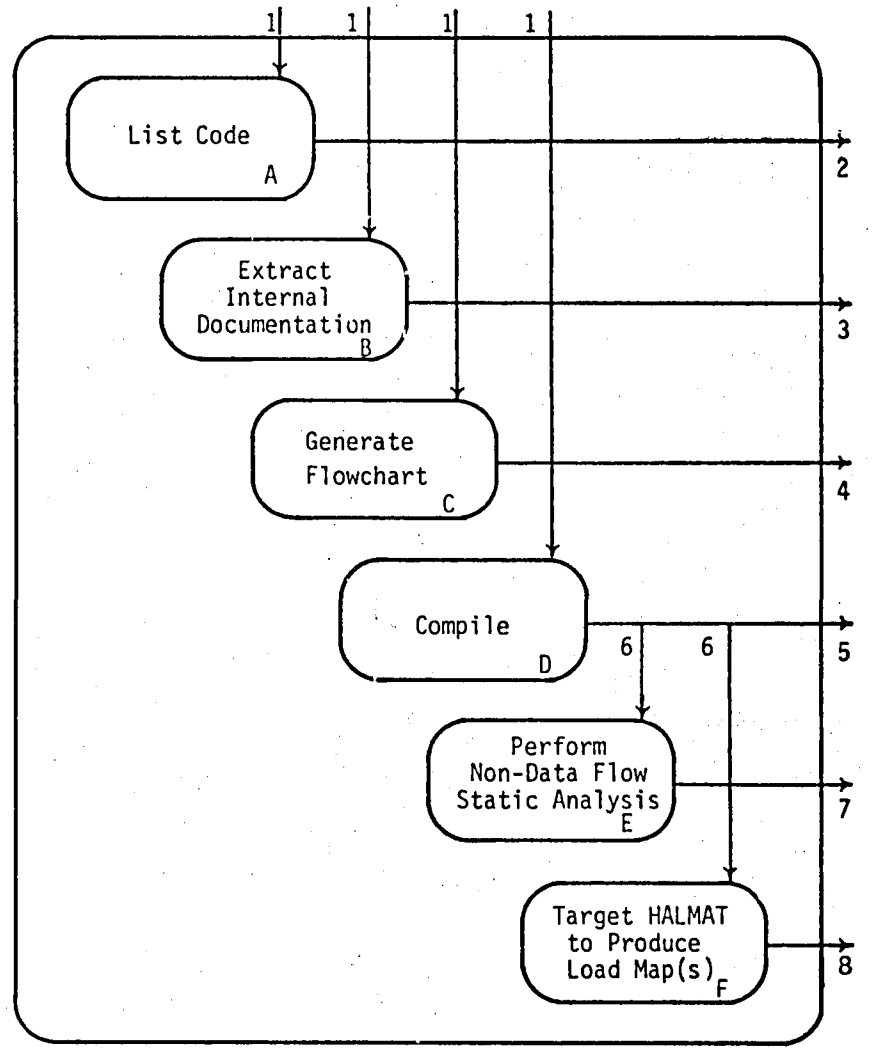
BGS

ACTIVITY - DATA FLOW DIAGRAM

ACTIVITY DESIGNATOR
ROOT

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

REF. DOCUMENT 10167 (SAMM)

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.

RELATED DATA

ID	SOURCE	DEST	NAME

BCS BOEING COMPUTER SERVICES INC

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

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

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 Systems. 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 times a second. Many such constraints may reside within 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 Representative. 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. 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);
         .
         .
         .
    GOTO 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 GOTO 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-existant 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_error$_{i:j}$;

where 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" $X + Y \leq J$. 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