SCIENCE
Applications
INCORPORATED

(NASA-CR-150299) NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM: SOFTWARE VERIFICATION/VALIDATION TECHNIQUES Final Report (Science Applications, Inc., Huntsville, Ala.) 60 p HC A04/MF A01

N77-26828

Unclas
G3/61 31776

SAI-78-556-HU

NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

FINAL REPORT

SOFTWARE VERIFICATION/VALIDATION TECHNIQUES

CONTRACT NAS8-31554

Prepared under the direction of
Mr. John Capps
Marshall Space Flight Center
National Aeronautics and Space Administration

April 22, 1977

S C I E N C E   A P P L I C A T I O N S, I N C.
2109 W. Clinton Ave., Suite 800, Huntsville, Ala. 35805
(205) 533-5900

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The purpose of this report is to present an overview of a software development system built by Science Applications Inc., of Huntsville, Alabama, under the direction of the Data Systems Laboratory of NASA, Marshall Space Flight Center. The system, called the Software Specification and Evaluation System (SSES), was designed for the effective and efficient specification, implementation, and testing of computer software programs. The system as implemented will produce structured FORTRAN or ANSI FORTRAN programs, but the principles upon which SSES is designed allow it to be easily adapted to other high order languages.

2. CORRELATION OF SCOPE OF WORK TASKS TO SECTIONS OF THE FINAL REPORT

This final report describes the results of the work performed in fulfilling the scope of work tasks for contract NAS8-31554. These tasks were (A) to complete the detailed design of the Software Specification and Evaluation System (SSES), and (B) to implement the critical SSES components. In fulfillment of Task A, an overview of SSES is presented (Section 3.1 of this report), along with an example which depicts the use of SSES in the development of reliable software (Appendix A of the Final Report).

The remainder of Section 3 of the Final Report reflects the work performed in accordance with the specifications of Task B. Most of the SSES components developed under Task B resulted in new software tools for which many forms of technical documentation such as design documents, user's manuals, operation guides, listings, and flowcharts were produced. The chart appearing in Figure 2-1 contains a summary of the documentation delivered for each new software tool as well as for the Software Requirements Methodology and the Data Base Verifier. Since this documentation is very detailed, the sections of this final report pertaining to the new or modified software components present only overviews of the work performed in each area. A summary of the Final Report sections and their relationship to the scope of work tasks is presented in Figure 2-2.

TECHNICAL DOCUMENTATION FOR SSES COMPONENTS

SSES Component	Design Document	User's Manual	Operation Guide	Listing	Flowcharts
Software Requirements Methodology	Software Requirements Methodology Design Specifications				
Software Specification Language	NASA Software Specification Language Translator Unit Module Descriptions	Introduction to Formal Specification Technique and SSL	NASA Software Specification Language Operation Guide	Separate documentation with no title	NASA Software Specification Language Translator Flowcharts
Structured FORTRAN Preprocessor	NASA Structured FORTRAN Preprocessor Unit Module Descriptions [1]	NASA Structured FORTRAN Preprocessor User's Manual	Included in User's Manual	Separate documentation with no title	NASA Structured FORTRAN Preprocessor Flowcharts
Static Analyzer	FACES Unit Module Descriptions and Updates to Existing FACES Documentation	Updates to Existing FACES Documentation	No changes to existing documentation	Separate documentation with no title	FACES Flowcharts
Data Base Verifier	Data Base Verifier Design				
Dynamic Analyzer	NASA Dynamic Analyzer Detailed Design Document Version II Revision 0 and Dynamic Analyzer FORTRAN Data Base	NASA Dynamic Analyzer and Structural Analyzer User's Manual	Included in User's Manual	Separate documentation with no title	Included in an Appendix to Design Document
Structural Test Case Generator	NASA Structural Analyzer Extension to Dynamic Analyzer Detailed Design Document	NASA Dynamic Analyzer and Structural Analyzer User's Manual	Included in User's Manual	Separate documentation with no title	Included in Design Document

1. Other design documentation includes:

   Calling Hierarchy for Modules Constituting the NASA Structured FORTRAN Preprocessor
   COMMON Names and COMMON Variables Referenced in the NASA Structured FORTRAN Preprocessor
   Cross Reference of Modules and COMMON Names in the NASA Structured FORTRAN Preprocessor

Figure 2-1, Technical Documentation for SSES Components

Final Report

Task	Section	Title
A	3.1	Software Specification and Evaluation System (SSES) Design Overview
	Appendix A	SSES Software Development Example
B1	3.2	Software Requirements Methodology
	3.3	Software Specification Language
B2	3.4	Structured FORTRAN Preprocessor
B3	3.5	Static Analyzer
	3.6	Data Base Verifier
	3.7	Dynamic Analyzer
	3.8	Structural Test Case Generator

Figure 2-2. Correlation of SOW Tasks to Final Report Sections

3. SSES METHODOLOGY

3.1 SOFTWARE SPECIFICATION AND EVALUATION SYSTEM (SSES) DESIGN OVERVIEW (Task A: SSES Design Completion)

Early in 1975, SAI and NASA jointly began a software R&D effort to develop a methodology which could reduce the effort expended in a typical software test and verification activity without sacrificing confidence in performance, thus improving the cost effectiveness of the overall software development. The Software Specification and Evaluation System (SSES) has been developed to achieve these goals. The system includes special-purpose languages and automatic requirements/code verification and validation tools designed to improve the quality assurance, traceability, testability and maintainability of the final software product.

The SSES comprises a set of integrated components based on the following software development phases:

- For the Requirements/Specification phase, a requirements methodology was developed to insure the integrity and feasibility of the software requirements. This methodology includes a prescription for the necessary content of the software requirements specification. Also, there is a formal Software Specification Language (SSL). This language is used to formally describe the overall software system (or functional) structure and, thereby provide a firm foundation for the software design process. SSL automatically provides for the traceability of requirements and checks element interconnection consistency.

- For the Coding phase, language disciplines for the promotion of reliable software have been identified and incorporated into a high-level, structured FORTRAN language. This language is translated to ANSI 3.9 FORTRAN through a preprocessor. Further work in this area includes the formulation of a complete programming methodology to alleviate questionable coding practices and, thus, increase reliability and flexibility.

- For the Verification and Validation phase, there is a Static Code Analyzer, a Data Base Verifier a Dynamic Analyzer, and an Automatic Structural Test Case Generator. The Static Code Analyzer is used to enforce technical coding standards and to document pertinent program information to be used during other V&V activities. The Data Base Verifier is used to analyze the program's accessing specifications and construct tables which describe the stored data base. (This tool exists in design only and will not be implemented until FORTRAN CODASYL standards have been set.) The dynamic analyzer is used to dynamically analyze the software system's execution characteristics, providing execution path trace and variable trace information. In order to provide adequate test case coverage, an automatic test case generator is used to test the final software product.

The application of the SSES components, the methodologies, reliability disciplines, and software tools, to the software development cycle are pictorially presented in Figure 3-1.

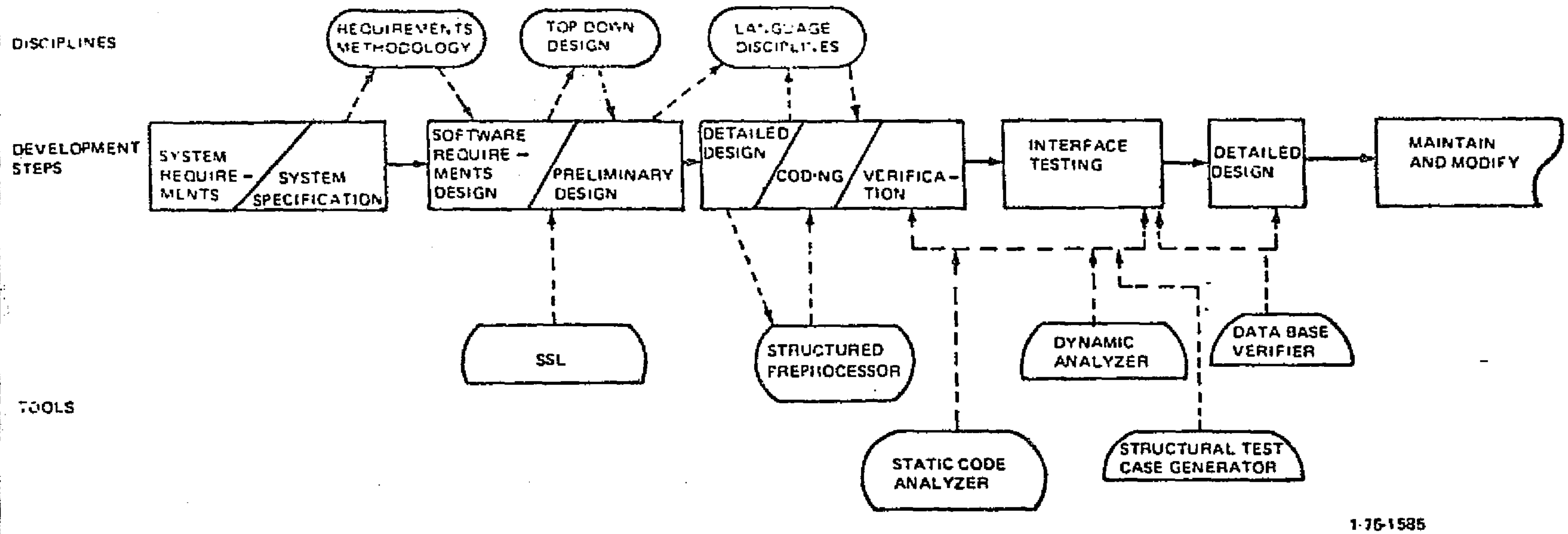


Figure 3-1. Augmented Development Cycle

3.2 SOFTWARE REQUIREMENTS METHODOLOGY (Task B1: Software Requirements Methodology Design)

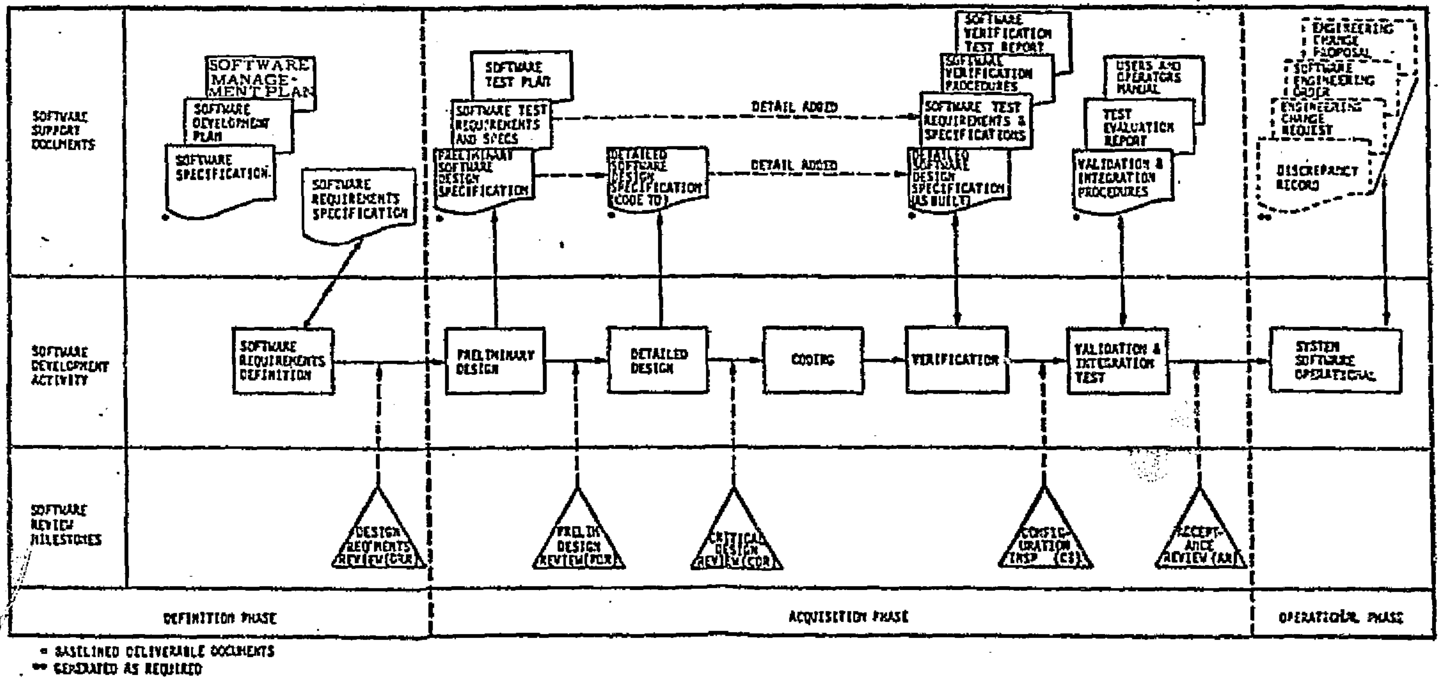
In the area of software requirements, a method of stating requirements which enhances clarity, consistency, completeness, traceability, and testability had to be defined. These requirement expressions represent all the relationships between the input and output and between the to-be-produced product and its environment without unnecessarily limiting the possible configurations of that product. Using SPACELAB software, an initial consideration of the approach that was developed for NASA to use in developing software requirements specification documents is presented in the following paragraphs.

As depicted in Figure 3-2, the software development process consists of activities, documents, and reviews. In order for the reviews to be maximally effective, the software and supporting documentation needs to be clearly expressed and sequentially traceable. In particular, with regard to the design requirements review (DRR), the software requirements specifications should be a function of (and must bridge the gap between) the prior activity--system design (not depicted) and the succeeding activity--preliminary software design. Consequently, the software requirements specification, whatever its particular format, should contain the information listed in Figure 3-3.

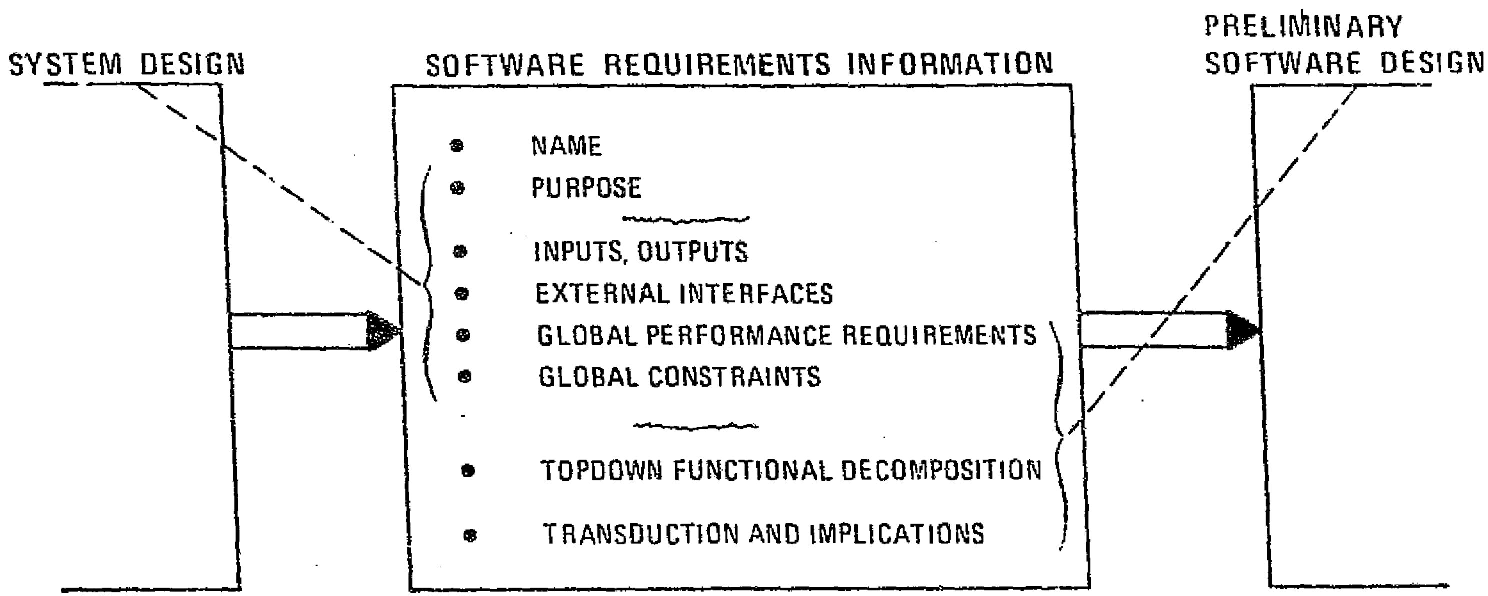
The method in which such information is expressed should probably be project or personnel dependent. Some factors affecting the choice of method are:

- training and background of requirements developers
- desired breadth of requirements visibility
- generic type of software
- allocated finances and other resources

One specific format (for SPACELAB software) will be suggested in the Software Requirements Design Specifications to be delivered as part of the task work. The design document will depict the key aspects of the Software Requirements Methodology.



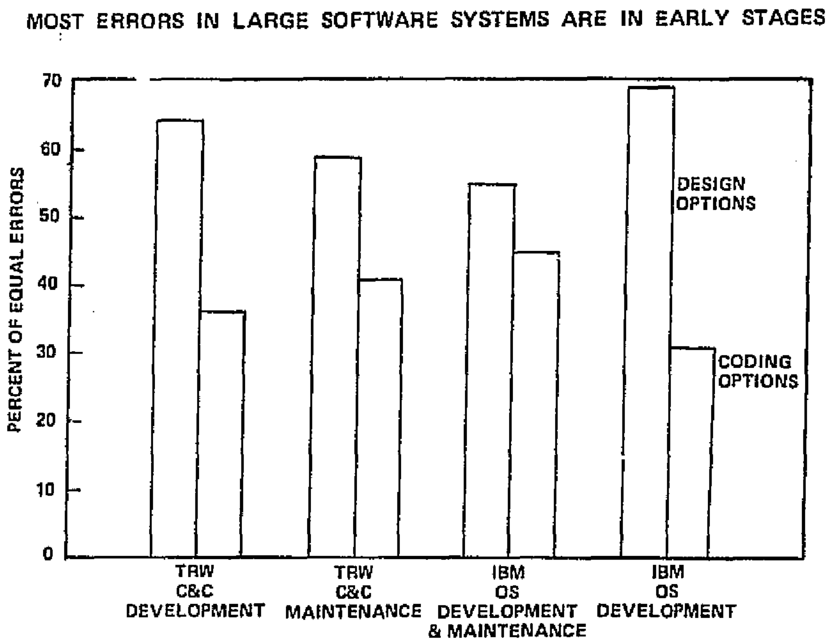
SOFTWARE DEVELOPMENT PROCESS (FOR SPACELAB)
Figure 3-2



SOFTWARE REQUIREMENTS INFORMATION
FIGURE 3-3

Of all the software development phases, requirements definition is undoubtedly the most important. The kind of information depicted in Figure 3-4 illustrates the quality and cost advantage that can be gained through a careful execution of this initial stage of software development.

SOFTWARE ERROR OCCURRENCE AND COST



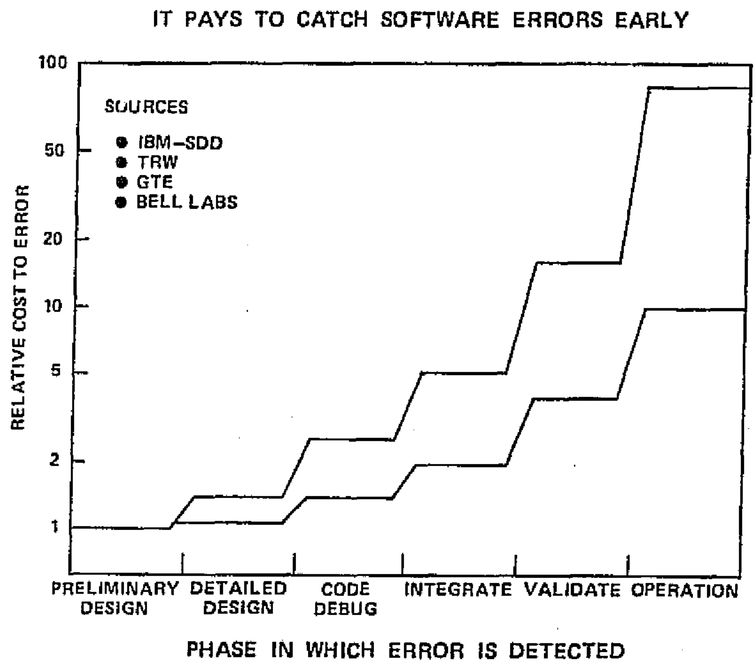


Figure 3-4. Software Error Occurrence and Cost

3.3 SOFTWARE SPECIFICATION LANGUAGE

(Task B1: Software Specification Language Implementation)

The purpose of SSL (Software Specification Language) is to aid in the process of defining systems and modules in order to alleviate software interface errors and improve requirements/design traceability. A formal description of the syntax and semantics exists which has enabled the construction of an automatic translator. The translator makes a series of nontrivial consistency checks based primarily on a system flow model that is assumed to exist apart from the SSL description and which originated in the software requirements specification. The essence of the flow model is checked implicitly by several features within the language.

3.3.1 Elements of the SSL Computation Model

SSL is a machinable design analysis tool with a formal syntax and semantic description. It does not impose artificial restrictions on data flow or software architecture but does insist that both conform to a separately developed system flow model. This affords the opportunity to perform extensive nontrivial consistency verification and develop a document that aids communication of design intentions and testing criteria to subsequent phases. The basic elements underlying an SSL description are data structures, modules, levels of abstraction, and requirements.

Anyone with an understanding of data declarations in such procedural languages as ALGOL and PL/I can easily grasp the concepts of variable and data structure in SSL. The language provides a small number of basic data structures. It also provides a small number of basic data types for which there is a direct implementation or trivial extension of a direct implementation on most hardware. These types may be used to affix attributes to simple variables or combined to describe composite variables.

Generally, a module is understood to be a program unit that can be understood independently of the rest of the system. Examples are COBOL paragraphs, ALGOL procedures, and FORTRAN subroutines. Modules are further combined into higher entities called levels of abstraction under the interconnection operation.

Levels of abstraction are sets of modules, embedded within a larger system, having several distinct properties:

P1. A level of abstraction is a set of modules which may share global data (and perhaps hardware features) among themselves, but not with modules outside the set. In any case, a subjective commonality of function or purpose binds all modules within a set.

P2. A subset of the modules with property P1 (called entry or external modules) can be referenced only from modules in other levels.

P3. There is a unidirectional dependence among the sets (i.e. a higher level may reference an entry module of a lower level but not vice versa).

In SSL, there are four components to a requirement: input, output, transduction, and constraint. Input and output are named variables corresponding to system level stimuli and responses. Constraints are simply named-entities attached as attributes to various objects within a described system. Their higher or conceptual meaning is not directly representable in SSL. Transductions are also named-entities attached as attributes to objects, but their purpose is to capture, via a partial ordering, the flow model underlying the module decomposition.

3.3.2 The Language

Systems described in SSL are partitioned into one or more subsystems where each subsystem corresponds to a level of abstraction. Within each subsystem one or more modules are described nonprocedurally. Module description statements permit module connections and data flow to be depicted in a variety of ways, subject to the restraints imposed by the underlying flow model. The flow model (i.e., requirements) and data structures are defined in a subsystem preamble. A partial ordering of transductions is specified in the preamble.

Modules are the focal point of an SSL description of a decomposition. Information represented about modules includes input variables, output variables, called modules, and transduction attributes which guard all interconnections. The general form of a module description in SSL is:

```
{MODULE} {module name} [(local variable list)];
{ENTRY }
        {ASSUMES  } {assertion list};
        {SATISFIES}

        FULFILLS   {transduction and constraint list};

        ACCESSES   {environment list};

        USES       {variable or component list};

        CREATES    {variable list} USING   {variable or
                                            component list};
        MODIFIES   {variable or component list}
                 USING { variable or component list};

        EXECUTES   [ITERATIVELY  ] ({module reference list})
                   [CONDITIONALLY]

    END MODULE;
```

Transduction attributes play a role in limiting the access scope of global data accessed in the MODIFY and USE statements or USING clause. For example, each transduction attribute of the module must be either the same as some transduction attribute of the variable or a successor (in the partial ordering sense) of some attribute of the variable. The effect of this rule is to limit the use of a variable to specific subnetworks. Similarly, in order for one module to reference a second module within the same subsystem, the first module must have transduction attributes that imply at least one attribute of the second module. This insures that the module ordering will generally correspond to the transduction ordering which, in turn, corresponds to some underlying flow model. Yet, the rule is not excessively constraining. The produced module network is seldom a simple restatement of the system level flowchart. The preliminary designer has considerable freedom within which to decompose the flow processes.

3.4 STRUCTURED FORTRAN PREPROCESSOR (Task B2: High Level Language Disciplines Determination and Structured Preprocessor Selection)

The goal of consistently producing reliable software dictates certain criteria for the structure of the program language employed. A list of criteria which an ideal programming language should satisfy was derived from studying programming languages that promote reliable code implementation. These criteria are as follows:

- The language should follow naturally from a top down approach and should be able to reflect the problem at hand.
- The language promotes a sequential implementation.
- Control structures should be explicitly clear and should be kept to a minimum.
- The language should exhibit the same syntax structure for semantically similar constructs.
- The language should allow indentation and a type of modularization that clearly defines the boundary of each module and allows each module to be clearly and completely locally understood.
- The language should have meaningful reserved words.
- The language should allow the programmer to write often used constructs with a minimum of detail.
- The language should offer a nonrestrictive placement of comments which facilitates trouble-free usage.
- Side effect changes of data should be made explicit and restricted to a minimum.
- Data types and other information crucial to correct execution should be explicitly specified preferably in several different ways.

- The language should have a context-free syntax.
- The language should be amenable to automatic code analysis.
- Machine overhead of often used constructs should be kept to a minimum.

Attempting to find a language that satisfied the above criteria while simultaneously acknowledging NASA's wide use of FORTRAN influenced us to consider a structured FORTRAN preprocessor as a language vehicle. Existing structured preprocessors were evaluated to determine which ones incorporated a large number of the criteria listed above. A preprocessor developed by the U.S. Army Missile Command at Redstone Arsenal was selected as the basis of our work. It featured three primary control structures for structured programming: the concatenation capability, the IF-THEN-OR IF-ELSE construct, and the DO WHILE construct. The FOR and TEST CASE constructs were added for user convenience. The preprocessor accepts structured FORTRAN source statements as input, and generates corresponding ANSI 3.9 FORTRAN statements. These generated source statements can then be used as input to an ANSI FORTRAN complier. Moreover, the structured FORTRAN preprocessor provides for automatic identification of nesting levels.

In addition, the original preprocessor as well as all subsequent modifications were designed with transportability as a priority. To date, the structured FORTRAN preprocessor has been readily implemented on the IBM 360 and 370, CDC 6600, UNIVAC 1108, PDP 10 and 11, and the SEL computing systems.

3.5 STATIC ANALYZER
(Task B3: Static Code Analyzer Implementation)

After the desired software modules have been coded and compiled, the next step in producing reliable software is to verify and validate the code using software tools. From the SSES repertoire, the logical component to use first is the static code analyzer. The static code analyzer accepts ANSI FORTRAN source code as input, evaluates the code according to intramodule and intermodule considerations, and produces appropriate output which identifies parts of the code which are likely candidates for inconsistencies and errors. Proper technical coding standards, good programming style, and appropriate program structure are all checked during the evaluation of the source code. To satisfy the task requirements in the area of static analysis, the following capabilities were added to the NASA static analyzer, FACES:

- EQUIVALENCE and EXTERNAL statements are flagged.
- Unlabeled COMMONs are flagged.
- DIMENSION statement and variable which contain an adjustable (variable) dimension are flagged.
- Arithmetic IFs are flagged.
- Targets of branches should not be other branches, especially single GO TOs.
- Occurrences of error-prone FORTRAN statements such as ASSIGN statement, assigned GO TO, and PAUSE are flagged.

These new features represent a significant increase in the overall effectiveness of the NASA static analyzer.

3.6 DATA BASE VERIFIER
(Task B3: Data Base Analysis Tool Design)

The approach to data base verification was based on CODASYL's (Conference of Data Systems Language) view of a data base management system. The CODASYL organization has been engaged in the development of language standards for describing extensions to existing high level languages (e.g. COBOL and FORTRAN) which will allow access and operation on the data base components as well as describe the part of a data base which resides on permanent storage. According to CODASYL's definition, a data base management system is a system which manages and maintains data in a non-redundant structure for the purpose of being processed by one or more applications. In a data base management system, an applications programmer writes a program in a higher order programming language such as FORTRAN or COBOL which has been augmented to incorporate Data Manipulation Language (DML) commands. The DML statements provide interfaces between application programs and data bases during execution.

Our data base verification subsystem concentrates on the FORTRAN applications program written in ANSI FORTRAN which has been extended to include Data Manipulation Language (DML) commands. It accepts CODASYL FORTRAN Data Manipulation Language source code as input, and statically analyzes the program. Data base description tables are then constructed which describe the stored data base that the program accesses and manipulates. Finally, it prints a report containing a summary of all the information collected about the components and the structure of the stored data base. The user must then establish the consistency and validity of the stored data base within the framework of the program descriptions by cross referencing these tables.

3.7 DYNAMIC ANALYZER
(Task B3: Dynamic Code Analyzer Implementation)

Continuing the code verification and validation process using the SSES methodology, the next logical software tool to execute would be the dynamic analyzer. The dynamic analyzer accepts either structured or ANSI FORTRAN source modules (or a combined stream of both types of modules) as input. The static analysis section of the dynamic analyzer recognizes all the necessary statement types, and sets up a program graph of the source code which emphasizes branch nodes. The program graph is constructed from the target program by assigning to each program statement (line) a node on the graph and using the edges between these nodes to represent control flow of the program. A decision-to-decision (DD) path is a path which begins and ends on a decision or branch node. The DD paths are important because they are used as indicators for inserting probes into the code. One probe is placed for each DD path in the program-graph. The instrumented source code is then written to a file which may be attached in the same computer run or a later one. After this file has been attached, compiled, and loaded (or link edited) with the Dynamic Analyzer run time package, the module is executed and run time statistics are collected. When the execution is completed, the third component of the Dynamic Analyzer, the trace analysis package, reads and interprets the data collected in the previous step. A detailed module test report, including a node/statement list, a DD path analysis, and a monitored variable list along with a summary report of the effectiveness of module testing is produced. (A sample test report is presented in Appendix A.) These reports provide the author of the software a comprehensive dynamic analysis of the software modules. The author can then determine by inspection which areas of code are most critical. Since the testing coverage is documented, the author has a reference for any further testing of the software modules regardless of whether modifications are necessary.

3.8 STRUCTURAL TEST CASE GENERATOR
(Task B3: Structural Test Case Generator Implementation)

The structural test case generator assists in the generation of test data sets that will exercise desired segments of code. It accepts structured FORTRAN code as input and performs several different functions for the user. First of all, it determines the total number of execution paths from entrance to exit in the module, based on some assumptions concerning the looping structure. Other functions of the test case generator are determinations of minimum and maximum coverage tests and a measure of probable testing effectiveness for these two testing alternatives. For the first calculation, the minimum number of distinct test cases which must be produced to meet the testing goal of covering each DD path in at least one of the tests is computed. This set of test cases represents the "best case" situation for testing purposes. In the next calculation, the structural test case generator determines the number of distinct test cases required to satisfy the execution of all DD paths which represents a "worst case" situation. Dividing these minimum and maximum number of tests by the number of execution paths yields a minimum and maximum (probable) testing confidence measure, respectively. In effect, this measure reflects how thoroughly, in terms of total possible execution paths, the program would be tested by using the minimum or maximum number to achieve DDP coverage. Resultant low values indicate that a high level of confidence can be placed in program behavior based on the DD path coverage tests.

The remaining test case generator function is a potential path selection which takes into account the previously calculated measurements. The cover selector portion of the output report prescribes an ordered selection of DD paths in a sequence of steps, which will number between the minimal to maximal values, to be executed in order to achieve complete DD path coverage. With the output generated from this automatic code analysis tool, a user can make a quick, more productive selection of paths for test data generation.

4.0 SSES BENEFITS AND UTILIZATION EXPERIENCE

All of the SSES components previously described except the data base verifier have been implemented. During implementation, productivity figures were kept on the Dynamic Analyzer and the Software Specification Language Preprocessor which were developed using as much of SSES as was available. Table 1-1 contains these figures and their comparisons with industry standard productivity estimates. The fact that personnel training and familiarity with the SSES components is not reflected in the figures must be taken into account when viewing Table 1-1. The productivity rates for the SSES component development for which many programmer interactions occurred show a 2 to 1 benefit ratio in comparison with the Aron figures. This increase in productivity represented a corresponding cost reduction in the development of reliable software which was one of the original objectives of SSES.

Experience has shown the Software Specification Language to be a useful tool in evaluating the early design efforts prior to further expenditure of resources. The primary contribution of the language is an existence proof that higher order verification is possible. This is accomplished by two basic semantic rules that relate the decomposition to a system flow model without demanding the system architecture be a simple restatement of the model. Simultaneously, the system encourages use of modularity, high level data types, and levels of abstraction.

The Structured FORTRAN Preprocessor was used in the development of SSL, the Dynamic Analyzer, and the Structural Test Case Generator. It promoted "built-in" software reliability by allowing the implementors to use structured programming, and its ease of use simplified the coding of these software tools.

The static analyzer FACES has been used for a portion of the shuttle structural testing data acquisition system. FACES was applied after the software had been debugged. Error conditions were detected in 6.5% of the source code analyzed, and one half of

1% of these errors were "fatal". If FACES had been applied at the beginning of the debugging phase, the benefits would have been greater.

Table 1-1. COMPARATIVE SOFTWARE PRODUCTIVITY RATES

	Aron (No System Testing)	Corbato (System Tested)	SSES* (System Tested)
Very Few Programmer Interactions	39 HOL Lines/ Man Day		50
Some	19	5	
Many	6		12

*Using SSL, Structured Preprocessor, FACES

APPENDIX A

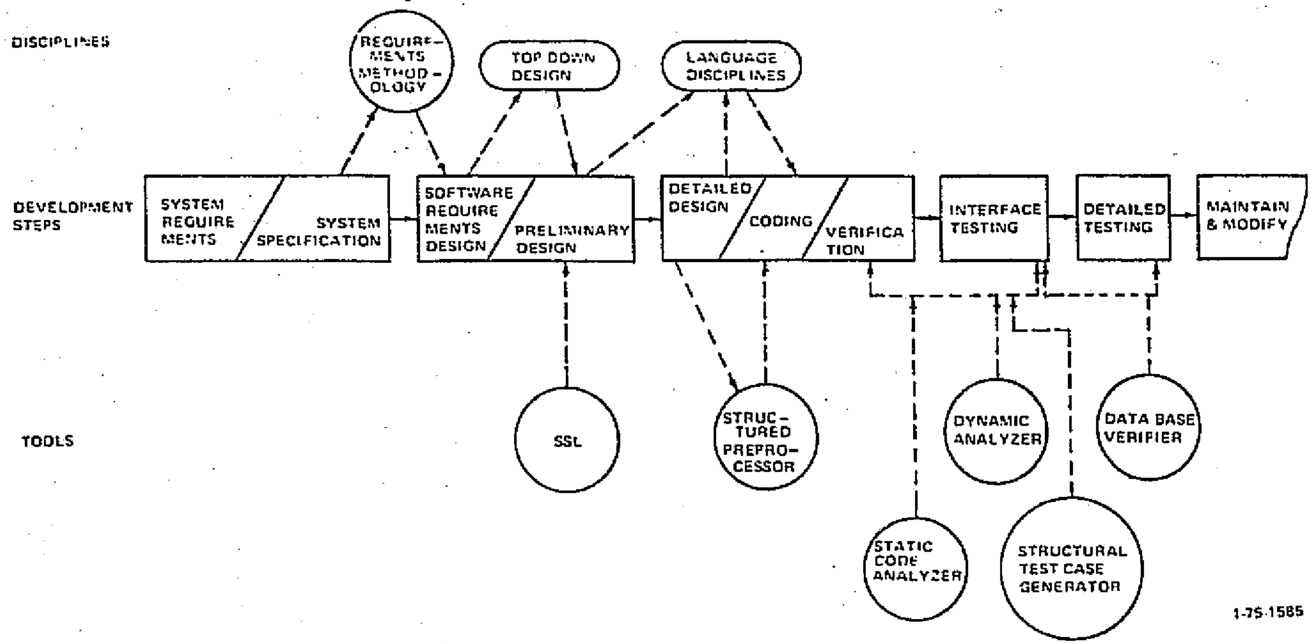
SSES SOFTWARE DEVELOPMENT EXAMPLE

APPENDIX A

SSES SOFTWARE DEVELOPMENT EXAMPLE

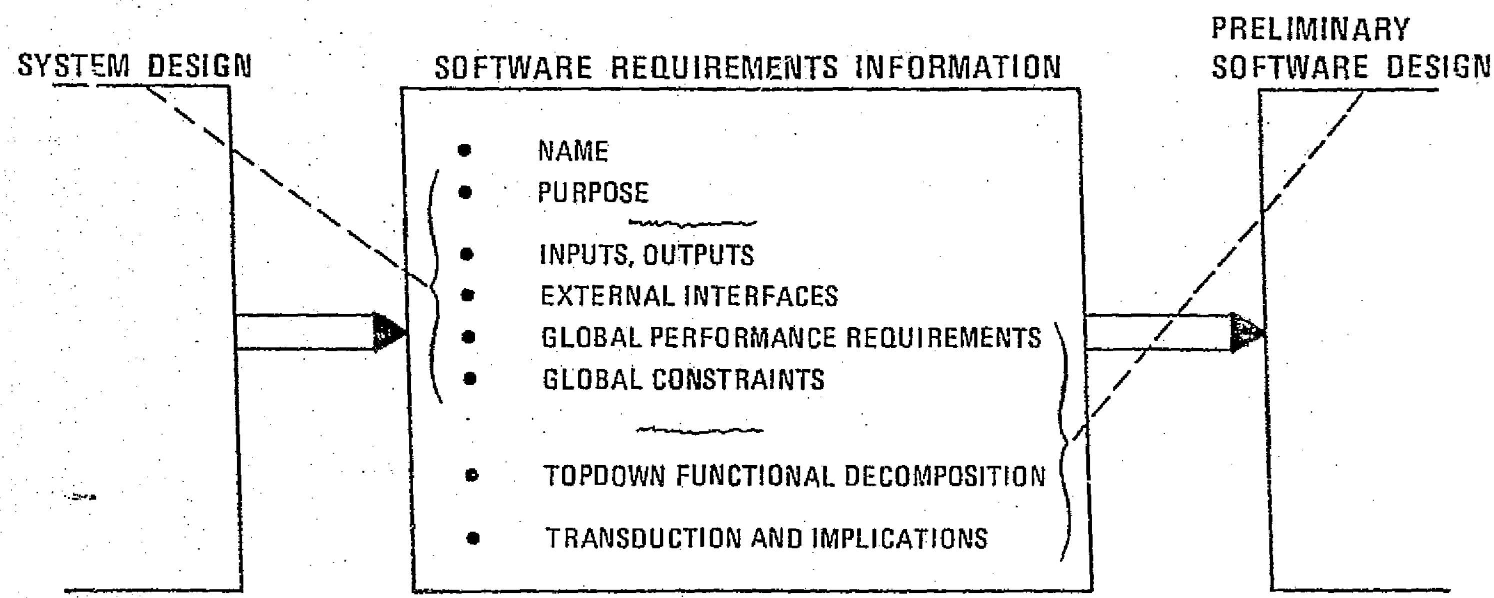
The following pages contain an example of a computer program developed by the NASA SSES Software Development System. The program is intended to solve the problem appearing on the next page. For this program we have written the following SSES documents and listings: The Software Requirements Specification, the Software Specification Language, the Structured Preprocessor Listing, the ANSI FORTRAN Listing, the Static Analyzer Listings, the Dynamic Analyzer Listings and the Structural Test Case Generator Listing.

SOFTWARE SPECIFICATION & EVALUATION SYSTEM (SSES)



PROBLEM

"A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of pre-determined size into a buffer where it is to be processed. The words in the telegram are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word."


SYSTEM DESIGN
SOFTWARE REQUIREMENTS INFORMATION
PRELIMINARY
SOFTWARE DESIGN
NAME
PURPOSE
INPUTS, OUTPUTS
EXTERNAL INTERFACES
GLOBAL PERFORMANCE REQUIREMENTS
GLOBAL CONSTRAINTS
TOPDOWN FUNCTIONAL DECOMPOSITION
TRANSDUCTION AND IMPLICATIONS

SOFTWARE REQUIREMENT SPECIFICATION

1. Name: Telegram Processing Program

2. Purpose: See Previous Page

3a. Inputs : character stream on a drum of fixed length records
3b. Outputs: printed telegram with detailed changes

4. External Interfaces: Drum, Printer

5. Global Performance Requirements: Must run in 32K

6. Global constraints: Must run on a PDP-8

7. Functional Decomposition : } see Following Sheets

8. Transductions and Implications: }

REQUIREMENTS ACTIVITIES AND TRANSDUCTIONS

Print 1	:	Collect words into telegrams
Print 2	:	Print whole telegrams
Print 3	:	Print all telegram charges
Collect 1	:	Collect characters into words
Collect 2	:	Print overlength word messages and physical record end of file messages
Separate	:	Return next character in telegram file
Read	:	Enter next physical record from drum into character buffer

Collect 1, Collect 2 $\subseteq$ Print 1

Read $\subseteq$ Separate

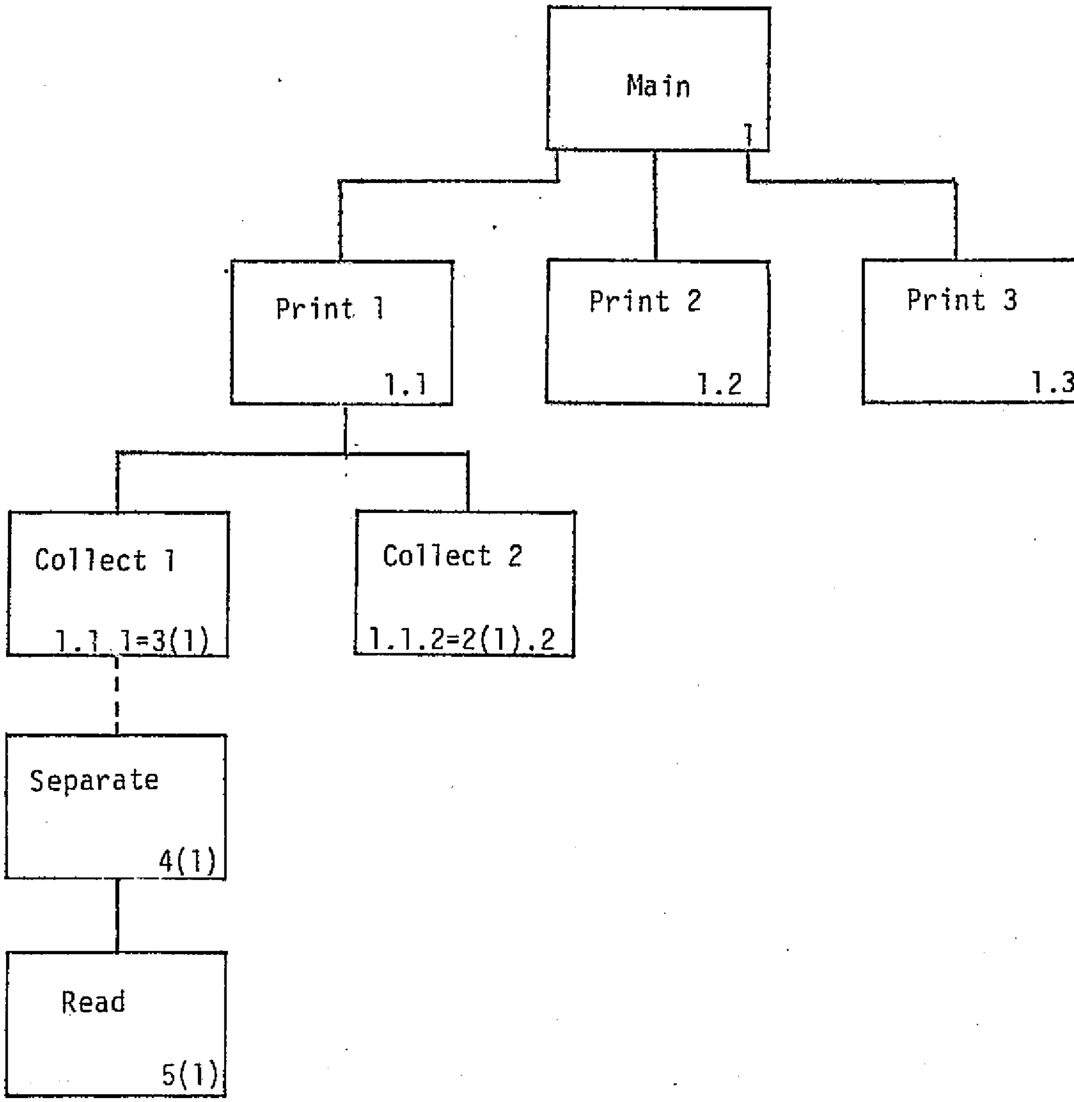
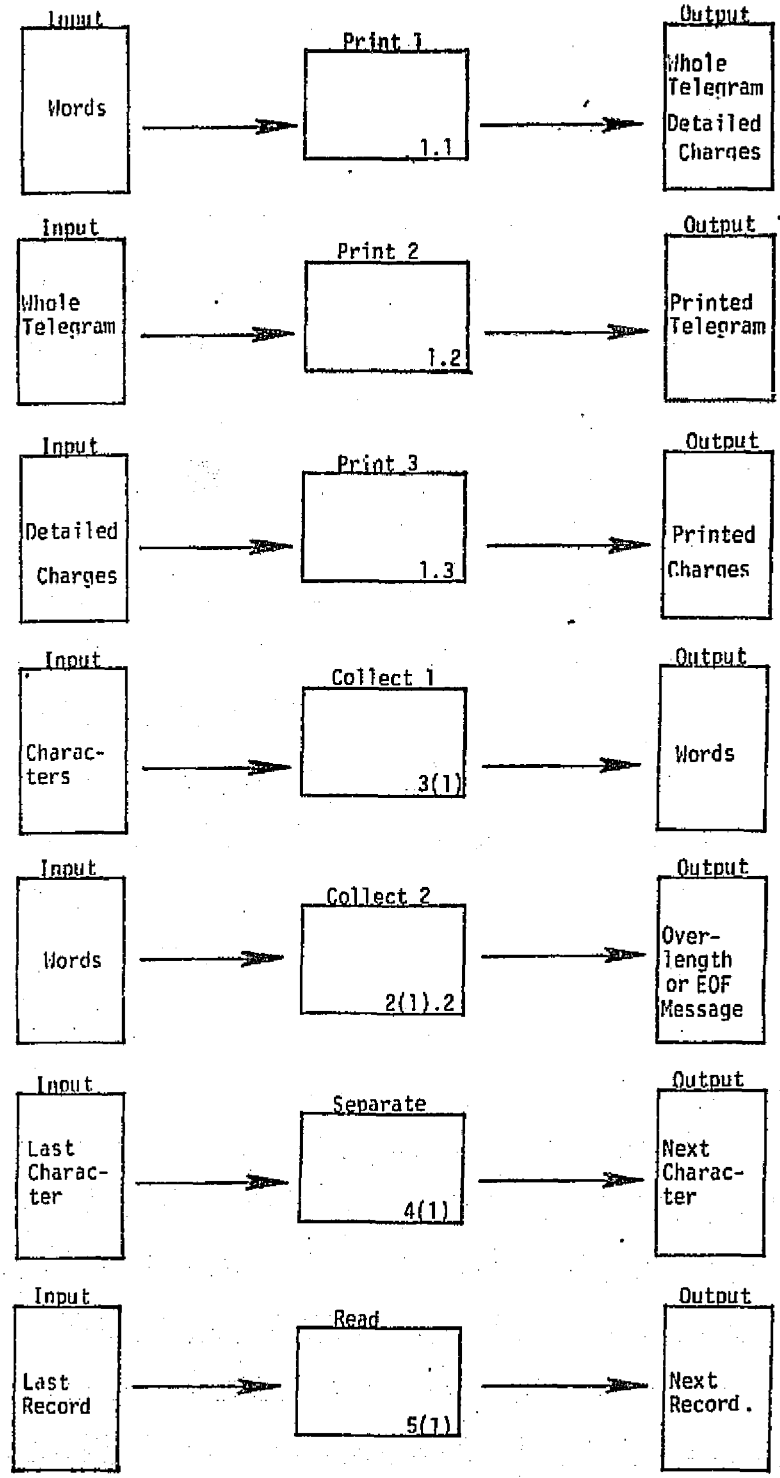
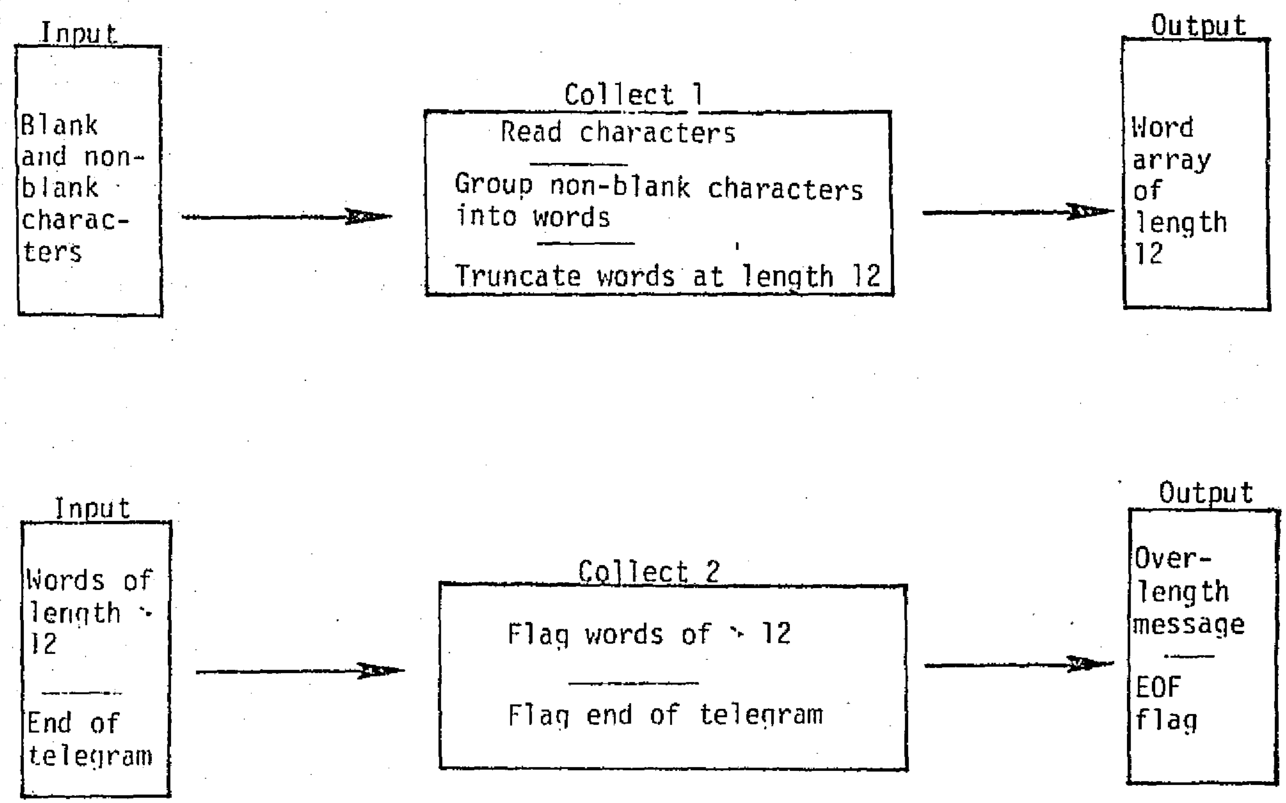


TABLE OF CONTENTS

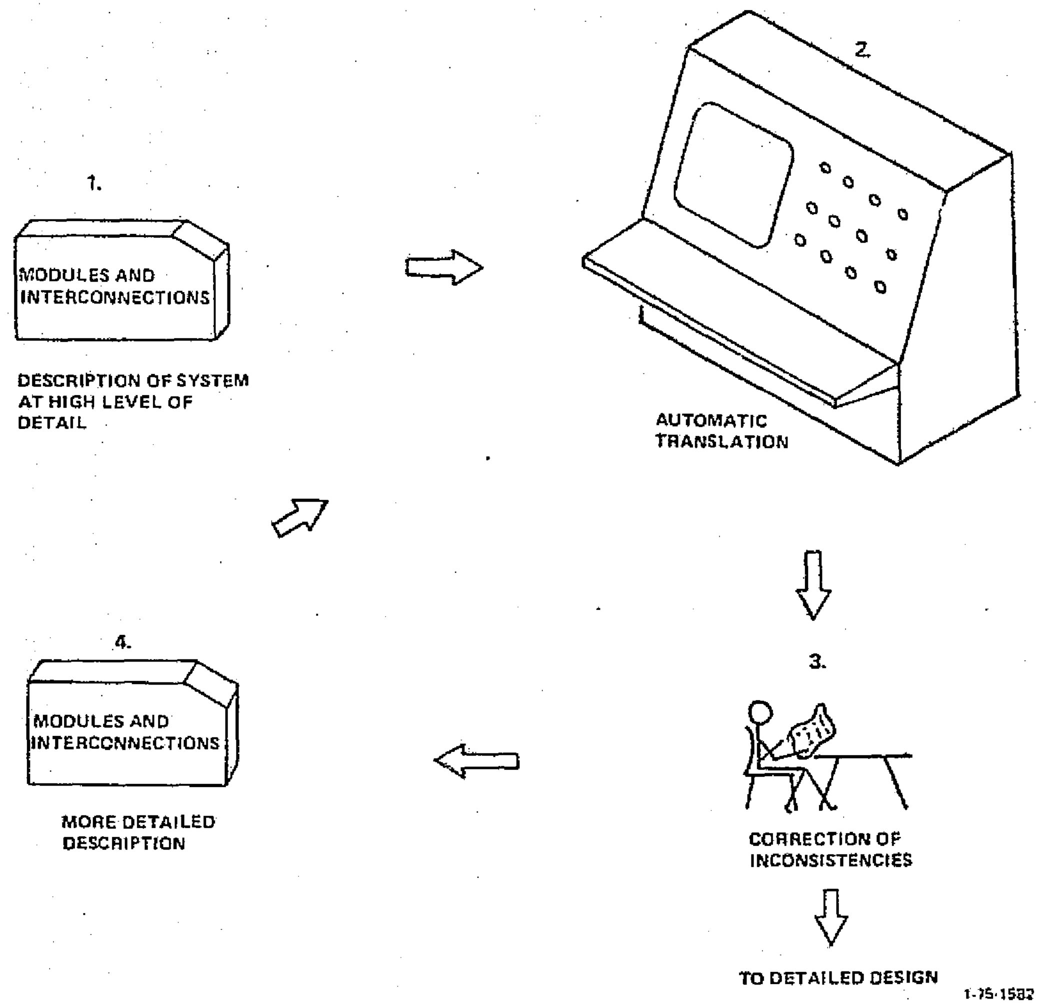


OVERVIEW DIAGRAM



DETAILED DIAGRAM FOR COLLECT 1 AND COLLECT 2

USING SOFTWARE SPECIFICATION LANGUAGE



SOFTWARE SPECIFICATION LANGUAGE

- A semi-automated tool that assists in conversion from written requirements to computer code structure.

- Checks the consistency of the logical flow of data and computations sequence to generate the desired output for a given input.

- Provides requirements traceability.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

SPECIFICATION AND EVALUATION SYSTEM

REPORT TABLE OF CONTENTS

Sample Table of Contents from an SSL Report

SOFTWARE SPECIFICATION LANGUAGE

```
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

    S[illegible]FILE ONEALL[illegible]
    /* BEGINNING OF MAIN SUBSYSTEM [illegible] */                                          ***
       REQUIREMENT                                                                        ***
         TRANSDUCTION  COLLECT IN PRINT;                                                  ***
         OUTPUT            TELEGRAM, CHARGE_COUNT;                                        ***
       END REQUIREMENT;                                                                   ***
       VARIABLE TELEGRAM: TEXT;                                                           ***
                 CHARGE_COUNT: INTEGER;                                                   ***
                    FOR PRINT;                                                            ***
                    SUBJECTTO  CHARGE_COUNT >= 0;                                         ***
                 WORD_COUNT: INTEGER;                                                     ***
                    FOR PRINT;                                                            ***
                    SUBJECTTO  WORD_COUNT >= CHARGE_COUNT;                                ***
                 WORD: ARRAY(12) OF CHAR;                                                 ***
                    FOR PRINT;                                                            ***
                 EOF_FLAG: BOOLEAN;                                                       ***
                    FOR PRINT;                                                            ***
    /* END OF PREAMBLE */                                                                 ***
    /* MAIN ROUTINE TO COLLECT WORDS AND PRINT TELEGRAM WITH WORD COUNT */                ***
      MODULE GET_TELEGRAM;                                                                ***
        FULFILLS  PRINT;                                                                  ***
        CREATES   TELEGRAM, CHARGE_COUNT  USING  WORD;                                    ***
        CREATES   WORD_COUNT;                                                             ***
        MODIFIES  WORD_COUNT;                                                             ***
        USES      EOF_FLAG;                                                               ***
        ACCESSES  LINE_PRINTER;                                                           ***
        EXECUTES                                                                          ***
          ITERATIVELY  (GET_WORD);                                                        ***
        SATISFIES                                                                         ***
          EOF_FLAG  OR  WORD_COUNT = 0                                                    ***
      END MODULE;                                                                         ***
    /* PROCEDURE TO COLLECT CHARACTERS INTO WORDS */                                      ***
      MODULE  GET_WORD;                                                                   ***
        FULFILLS  COLLECT;                                                                ***
        EXECUTES                                                                          ***
          ITERATIVELY  (I_O.GET_CHAR (A_CHAR:CHAR; EOF_FLAG) );                           ***
        CREATES   WORD, EOF_FLAG;                                                         ***
        ACCESSES  LINE_PRINTER  /* PRINTS ERROR MESSAGES */                               ***
      END MODULE                                                                          ***
      END SUBSYSTEM; /* END OF MAIN SUBSYSTEM */                                          ***
```

The subroutine GET-WORD fulfills the requirements transductions COLLECT 1 and COLLECT 2.

Sample SSL Source Program

```
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

                                         SUMMARY OF VARIABLE/MODULE CONNECTIONS

                          1 2 3
                         --------
 1.  A CHAR              I . X .
 2.  CHARGE COUNT        I X . .
 3.  TELEGRAM            I X . .
 4.  EOT FLAG            I X X .
 5.  WORD                I X X .
 6.  WORD COUNT          I X . .
```

SSL Requirement to Module Connectivity Matrix

```
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

                              SUMMARY OF REQUIREMENT/MODULE CONNECTIONS

                     1 2 3
                    --------
 1.  COLLECT        I . X .
 2.  PRINT          I X . .
```

SSL Module to Module Connectivity Matrix

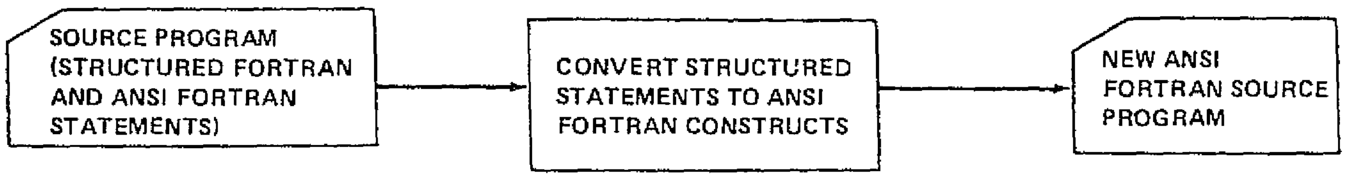
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

SUMMARY OF MODULE/MODULE CONNECTIONS

		1	2	3
1.	GET TELEGRAM	.	.	.
2.	GET WORD	X	.	.
3.	GET CHAR	.	X	.

SSL Variable to Module Connectivity Matrix

STRUCTURED TO UNSTRUCTURED FORTRAN PREPROCESSOR



1-75-1578

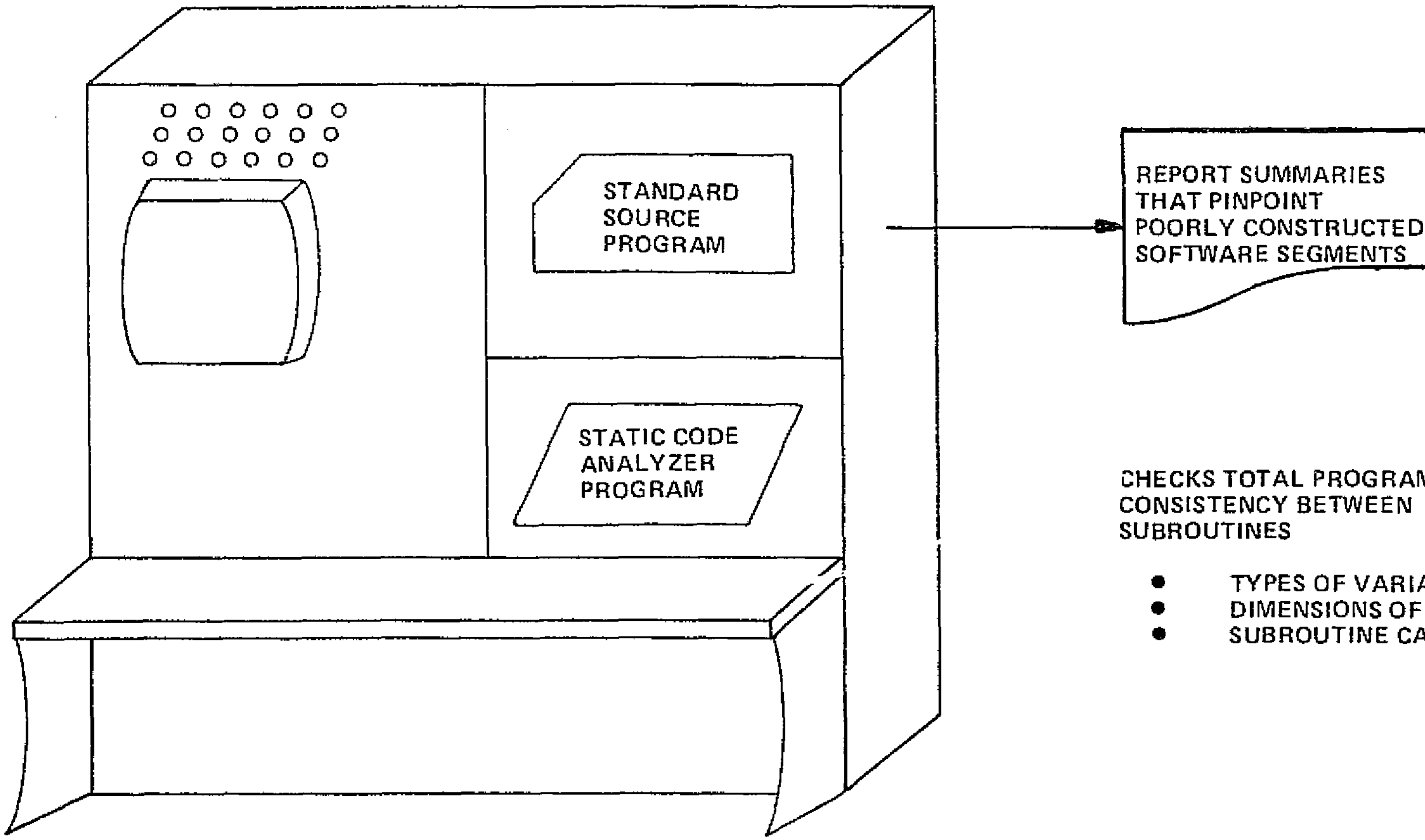
STRUCTURED PREPROCESSOR LISTING

```
      SUBROUTINE GETWD (IWORD, EOF)
C     FETCHES NEXT WORD FROM TELEGRAM FILE
C        IWORD = CHARACTERS IN WORD
C        EOF   = END OF FILE FLAG
C
      INTEGER IWORD(12)
      LOGICAL EOF, LEN
      DATA IBLANK / 4H     /
C
C     BLANK FILL WORD
      FOR (I = 1, 12)
         IWORD(I) = IBLANK
      END FOR
      LEN = .FALSE.
C
C     FIND FIRST NONBLANK CHARACTER
      ICHAR = IBLANK
      DO WHILE (ICHAR .EQ. BLANK .AND. .NOT. EOF)
         CALL GETCHR (ICHAR, EOF)
      END DO
C
C     COLLECT WORD
      I = 1
      DO WHILE (.NOT. ICHAR .EQ. IBLANK
     9             .AND. .NOT. EOF)
         IF (I .LE. 12)
            THEN
               IWORD(I) = ICHAR
               I = I + 1
            ELSE
               IF (.NOT. LEN)
                  THEN
                     WRITE (6, 1)
    1                FORMAT (1H , 13HWORD OVERFLOW)
                     LEN = .TRUE.
               END IF
         END IF
         CALL GETCHR (ICHAR, EOF)
      END DO
C
      RETURN
      END
  1 STATEMENT NUMBERS FOUND.

   0 ERRORS FOUND IN THIS ROUTINE.     42 CARDS READ.            45 CARDS OUTPU

   0 PREPROCESSOR ERRORS FOUND.        71 TOTAL CARDS READ.       31 TOTAL CARDS
             2 TOTAL DO WHILE'S,        3 TOTAL IF'S,              0 TOTAL OR IF
             2 TOTAL FOR'S,             0 TOTAL TEST CASE'S.
             3 PROGRAM UNITS PROCESSED.
```

GENERATED ANSI FORTRAN

```
L 21.8 ( JUN 74 )                                    OS/360  FORTRAN H

      COMPILER OPTIONS - NAME=  MAIN,OPT=02,LINECNT=56,SIZE=0000K,
                         SOURCE,EBCDIC,NOLIST,NODECK,LOAD,MAP,NOEDIT,ID,NOXREF
ISN 0002               SUBROUTINE GETWD (IWORD, EOF)
ISN 0003               INTEGER IWORD(12)
ISN 0004               LOGICAL EOF, LEN
ISN 0005               DATA IBLANK / 4H     /
ISN 0006               IF(  1
                     ..GT.  12
                     .)GO TO  99999
ISN 0008               DO  99998
                     .        I = 1, 12
ISN 0009               IWORD(I) = IBLANK
ISN 0010         99998 CONTINUE
ISN 0011         99999 CONTINUE
ISN 0012               LEN = .FALSE.
ISN 0013               ICHAR = IBLANK
ISN 0014         99997 IF(.NOT.
                     .              (ICHAR .EQ. BLANK .AND. .NOT. EOF)
                     .)GO TO 99996
ISN 0016               CALL GETCHR (ICHAR, EOF)
ISN 0017               GO TO 99997
ISN 0018         99996 CONTINUE
ISN 0019               I = 1
ISN 0020         99995 IF(.NOT.
                     .              (.NOT. ICHAR .EQ. IBLANK
                     .               .AND. .NOT. EOF)
                     .)GO TO 99994
ISN 0022               IF(.NOT.
                     .    (I .LE. 12)
                     .)GO TO  99993
ISN 0024               IWORD(I) = ICHAR
ISN 0025               I = I + 1
ISN 0026               GO TO 99992
ISN 0027         99993 CONTINUE
ISN 0028               IF(.NOT.
                     .    (.NOT. LEN)
                     .)GO TO  99991
ISN 0030               WRITE (6, 1)
ISN 0031             1 FORMAT (1H , 13HWORD OVERFLOW)
ISN 0032               LEN = .TRUE.
ISN 0033         99991 CONTINUE
ISN 0034         99992 CONTINUE
ISN 0035               CALL GETCHR (ICHAR, EOF)
ISN 0036               GO TO 99995
ISN 0037         99994 CONTINUE
ISN 0038               RETURN
ISN 0039               END
```

STATIC CODE ANALYZER



CHECKS TOTAL PROGRAM CONSISTENCY BETWEEN SUBROUTINES

- TYPES OF VARIABLES
- DIMENSIONS OF VARIABLES
- SUBROUTINE CALLS

```
FACES PRIMARY LISTING REPORT
****************************

               PROGRAM MAIN
               INTEGER IWORD(12)
               LOGICAL EOF
               COMMON IACE, JACE, KACE
** 210  COMPAL    COMMON IS UNLABELLED. <----------
               IACE = 1
               JACE = 1
               LEN = .FALSE.
               EOF = .FALSE.
               IF(  I
              ..GT.  100
              .)GO TO  99999
               DO  99998
              ,       I = 1, 100
               CALL GETWD (IWORD, EOF)
         99998 CONTINUE
         99999 CONTINUE
               STOP
               END



* * * E X P L A N A T I O N S * * *
```

Static Code Analyzer Output

FACES PRIMARY LISTING REPORT
****

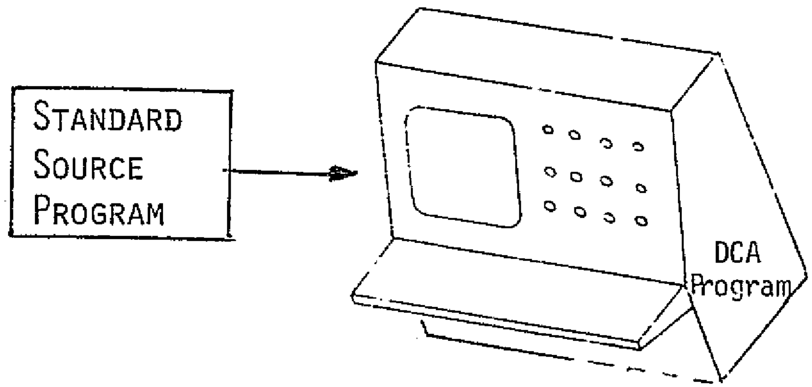
```
              SUBROUTINE GETWD (IWORD, EOF)
              INTEGER IWORD(12)
              LOGICAL EOF, LEN
              DATA IBLANK / 4H     /
              IF(   1
             ..GT.   12
             .)GO TO  99999
              DO  99998
             .      I = 1, 12
              IWORD(I) = IBLANK
       99998 CONTINUE
       99999 CONTINUE
              LEN = .FALSE.
              ICHAR = IBLANK
       99997 IF(.NOT.
             .           (ICHAR .EQ. BLANK .AND. .NOT. EOF)
             .)GO TO 99996
** 190  UNINT     LOCAL VARIABLE BLANK    IS UNINITIALIZED.
              CALL GETCHR (ICHAR, EOF)
              GO TO 99997
       99996 CONTINUE
              I = 1
       99995 IF(.NOT.
             .           (.NOT. ICHAR .EQ. IBLANK
             9              .AND. .NOT. EOF)
             .)GO TO 99994
              IF(.NOT.
             .    (I .LE. 12)
             .)GO TO  99993
              IWORD(I) = ICHAR
              I = I + 1
              GO TO 99992
       99993 CONTINUE
              IF(.NOT.
             .    (.NOT. LEN)
             .)GO TO  99991
              WRITE (6, 1)
            1 FORMAT (1H , 13HWORD OVERFLOW)
              LEN = .TRUE.
       99991 CONTINUE
       99992 CONTINUE
              CALL GETCHR (ICHAR, EOF)
              GO TO 99995
       99994 CONTINUE
              RETURN
              END
```


* * * E X P L A N A T I O N S * * *

19X   THERE EXISTS A PATH SUCH THAT A LOCAL VARIABLE IS UNINITIALIZED.

STATIC ANALYZER Report on GET-WORD

DYNAMIC CODE ANALYZER (DCA)



Outputs a Graphical Picture of Data Execution

- Dynamic Path Trace
- Number of Executions
- Percent % of Executions
- Variable Monitored
  - Initial Values
  - Min./Max. Values
  - First and Last Values

Step 1: Graph Analysis

- Path Traces
- ID of Nodes

Step 2: Generate Probes

- Manual Analysis of Nodes
- Create Control Cards
- Generate Test Data

Step 3: Dynamic Execution

- Input Control Cards
- Input Source Program
- Input Test Data
- Compile and Execute

```
** NASA DYNAMIC ANALYZER INPUT REQUESTS **

                    DIALECT     STRUCTURED
                    ANALYZE     ALL
                    REPORT      MAIN
                   (REPORT      GETWD )
                    REPORT      GETCHR
                    $EOF

END OF JOB STREAM
```

Dynamic Analysis Report requested for GET-WORD.

DYNAMIC ANALYZER COMMANDS

** MODULE DETAIL REPORT FOR GETWD **

--- NODE/STATEMENT LIST ---

STATEMENT NUMBER	NODE NUMBER	STATEMENT NUMBER LABEL	TEXT
1			SUBROUTINE GETWD (IWORD,EOF)
2		C	FETCHES NEXT WORD FROM TELEGRAM
3		C	IWORD = CHARACTERS IN WORD
4		C	EOF = END OF FILE FLAG
5		C	
6			INTEGER IWORD(12)
7			LOGICAL EOF,LEN
8			DATAIBLANK / 4H
9		C	
10		C	BLANK FILL WORD
11	1		FOR (I = 1,12)
12	2		IWORD(I) = IBLANK
13	3		END FOR
14	4		LEN = .FALSE.
15		C	
16		C	FIND FIRST NONBLANK CHARACTER
17	5		ICHAR = IBLANK
18	6		DO WHILE (ICHAR.EQ.BLANK.AND..
19	7		CALL GETCHR(ICHAR,EOF)
20	8		END DO
21		C	
22		C	COLLECT WORD
23	9		I = 1
24	10		DO WHILE (.NOT.ICHAR.EQ.IBLANK
25	11		IF (I.LE.12)
26			THEN
27	12		IWORD(I) = ICHAR
28	13		I = I+1
29	14		ELSE
30	15		IF (.NOT.LEN)
31			THEN
32	16		WRITE (6,1)
33	17	1	FORMAT (1H , 13HWORD OVERFLOW)
34	18		LEN = .TRUE.
35		COMME	NT: ELSE INSERTED
36	19		ELSE
37	20		END IF
38	21		END IF
39	22		CALL GETCHR(ICHAR,EOF)
40	23		END DO
41		C	
42	24		RETURN
43	25		END

** MODULE DETAIL REPORT FOR GETWD ** (Page 2)

--- DDP ANALYSIS ---

DDP NO.	BEGIN NODE	MEMBER NODES	END NODE	DDP TYPE	CONDITIONS
1	1	2 3	1		--ENTRY--
				FOR	(I .LE.12)
2	1	4 5	6	FOR	(I .GT.12)
3	6	7 8	6	DOW	(ICHAR.EQ.BLANK.AND..NOT.EOF)
4	6	9	10	DOW	.NOT.(ICHAR.EQ.BLANK.AND..NOT.EOF)
5	10		11	DOW	(.NOT.ICHAR.EQ.IBLANK.AND..NOT.EOF)
6	10		24	DOW	.NOT.(.NOT.ICHAR.EQ.IBLANK.AND..NOT.EOF)
7	11	12 13 21 22 23	10	IFS	8(I.LE.12)
8	11	14	15	IFS	8--ELSE--
9	15	16 17 18 20 21 22 23	10	IFS	8(.NOT.LEN)
10	15	19 20 21 22 23	10	IFS	8--ELSE--

--- MONITORED VARIABLE LIST ---

NAME	TYPE	NAME	TYPE

0 VARIABLES WERE MONITORED FOR THIS MODULE

*** MODULE TESTING EFFECTIVENESS SUMMARY ***

MODULE	TIMES INVOKED	# DD PATHS	# EXECUTED*	% COVERAGE
MAIN	1	2	2	100.0
GETWD	100	10	4	40.0
GETCHR	0	2	0	0.0
	TOTAL	14	6	42.9

*(AT LEAST ONCE)

Dynamic Analyzer Run Time Report

```
***  DETAILED TEST REPORT FOR MODULE GETWD   ***

MODULE GETWD    WAS INVOKED  100 TIME(S)

DDP NUMBER    ENTRY COUNT   PERCENT EXECUTION
     1           1200              80.0        |****************************************
     2            100               6.7        |***
     3              0               0.0        |
     4            100               6.7        |***
     5              0               0.0        |
     6            100               6.7        |***
     7              0               0.0        |
     8              0               0.0        |
     9              0               0.0        |
    10              0               0.0        |
                                               |----|----|----|----|----|----|----|----|----|----|
                                               0   10   20   30   40   50   60   70   80   90   100
                                                                 PERCENT EXECUTION
TOTALS

    10           1500                          PERCENTAGE DD PATHS EXECUTED  40.0
```

Dynamic Analyzer Run Time Report

STRUCTURAL TEST CASE GENERATOR

(Implemented)

- COVER SELECTOR - To gauge the number of execution paths in the program and to select an optimal cover for testing purposes.
- DDP CONDITION LINKER - To associate a series of decisions (in simplest form) with each execution path.
- NEXT TEST - To select the best next path for test case generation based on testing history data.

SAI

```
**  COVER SELECTOR AND ANALYTIC PATH ANALYSIS REPORT FOR MODULE GETWD    **

                    --- ANALYTIC PATH ANALYSIS ---

        5  -- NUMBER OF DECISION NODES
       10  -- NUMBER OF DECISION-TO-DECISION PATHS (DDP'S)

       16  -- NUMBER OF PRACTICAL EXECUTION PATHS

        1  -- MINIMUM NUMBER OF TEST CASES (I.E., BEST-CASE DDP COVERAGE )
    0.063  -- IMPACT ON EXHAUSTIVE TESTING

        6  -- MAXIMUM NUMBER OF TEST CASES (I.E., WORST-CASE DDP COVERAGE )
    0.375  -- IMPACT ON EXHAUSTIVE TESTING

                    --- COVER SELECTOR ---

    FIRST TEST DDP 1

          THEN ONE OF   DDP 2
          THEN ONE OF   DDP 3, DDP 4
          THEN ONE OF   DDP 5, DDP 6
          THEN ONE OF   DDP 7, DDP 8
          THEN ONE OF   DDP 9, DDP 10
```

STRUCTURAL TEST CASE GENERATOR REPORT