General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)
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

SCIENCE APPLICATIONS, INC.
2109 W. Clinton Ave., Suite 800, Huntsville, Ala. 35805
(205) 533-5900
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. CORRELATION OF SCOPE OF WORK TASKS TO SECTIONS OF THE FINAL REPORT</td>
<td>2</td>
</tr>
<tr>
<td>3. SSES METHODOLOGY</td>
<td>5</td>
</tr>
<tr>
<td>3.1 Software Specification and Evaluation System (SSES) Design Overview</td>
<td>5</td>
</tr>
<tr>
<td>3.2 Software Requirements Methodology</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Software Specification Language</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Elements of the SSL Computation Model</td>
<td>13</td>
</tr>
<tr>
<td>3.3.2 The Language</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Structured Fortran Preprocessor</td>
<td>17</td>
</tr>
<tr>
<td>3.5 Static Analyzer</td>
<td>19</td>
</tr>
<tr>
<td>3.6 Data Base Verifier</td>
<td>20</td>
</tr>
<tr>
<td>3.7 Dynamic Analyzer</td>
<td>21</td>
</tr>
<tr>
<td>3.8 Structural Test Case Generator</td>
<td>22</td>
</tr>
<tr>
<td>4. SSES BENEFITS AND UTILIZATION EXPERIENCE</td>
<td>23</td>
</tr>
</tbody>
</table>

**Appendix**

| A | SSES Software Development Example | A-1 |
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Technical Documentation for SSES Components</td>
<td>3</td>
</tr>
<tr>
<td>2-2</td>
<td>Correlation of SOW Tasks to Final Report Sections</td>
<td>4</td>
</tr>
<tr>
<td>3-1</td>
<td>Augmented Development Cycle</td>
<td>7</td>
</tr>
<tr>
<td>3-2</td>
<td>Software Development Process (for SPACELAB)</td>
<td>9</td>
</tr>
<tr>
<td>3-3</td>
<td>Software Requirements Information</td>
<td>10</td>
</tr>
<tr>
<td>3-4</td>
<td>Software Error Occurrence and Cost</td>
<td>12</td>
</tr>
</tbody>
</table>

LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Comparative Software Productivity Rates</td>
<td>25</td>
</tr>
</tbody>
</table>
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.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured FORTRAN Preprocessor</td>
<td>NASA Structured FORTRAN Preprocessor User’s Manual Module Descriptions 1</td>
<td>Included in documentation with title</td>
<td>Separate documentation with title</td>
<td>Included in documentation with title</td>
<td>NASA Structured FORTRAN Preprocessor Flowcharts</td>
</tr>
<tr>
<td>Static Analyzer</td>
<td>FACES Unit Module Descriptions and Updates to Existing FACES Documentation</td>
<td>Updates to existing documentation</td>
<td>No changes to existing documentation</td>
<td>Separate documentation with title</td>
<td>FACES Flowcharts</td>
</tr>
<tr>
<td>Data Base Verifier</td>
<td>Data Base Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Analyzer</td>
<td>NASA Dynamic Analyzer Detailed Design Document Version II and Dynamic Analyzer FORTRAN Data Base</td>
<td></td>
<td></td>
<td>Included in documentation with title</td>
<td>Included in an Appendix to Design Document</td>
</tr>
<tr>
<td>Structural Test Case Generator</td>
<td>NASA Structural Analyzer Extension to Dynamic Analyzer Detailed Design Document</td>
<td></td>
<td></td>
<td>Separate documentation with title</td>
<td>Included in Design Document</td>
</tr>
</tbody>
</table>

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

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 SSAS 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
SYSTEM DESIGN

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

PRELIMINARY SOFTWARE DESIGN

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

MOST ERRORS IN LARGE SOFTWARE SYSTEMS ARE IN EARLY STAGES

IT PAYS TO CATCH SOFTWARE ERRORS EARLY

Figure 3-4. Software Error Occurrence and Cost
3.3 SOFTWARE SPECIFICATION LANGUAGE
(Task Bl: 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 }
{ENTRY} {module name} [(local variable list)];
{ASSUMES }
{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}
{EXECUTES ITERATIVELY } (module reference list})

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 under-
lying 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.
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 compiler. 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

<table>
<thead>
<tr>
<th>Very Few Programmer Interactions</th>
<th>Aron (No System Testing)</th>
<th>Corbato (System Tested)</th>
<th>SSES* (System Tested)</th>
</tr>
</thead>
<tbody>
<tr>
<td>39 HOL Lines/Man Day</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Some</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

*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 predetermined 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."
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:
   see Following Sheets
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
DETAIL BOX DIAGRAM FOR COLLECT 1 AND COLLECT 2

**Input**
- Blank and non-blank characters
- Words of length ≤ 12
- End of telegram

**Collect 1**
- Read characters
- Group non-blank characters into words
- Truncate words at length 12

**Output**
- Word array of length 12

**Input**
- Words of length ≤ 12
- End of telegram

**Collect 2**
- Flag words of ≤ 12
- Flag end of telegram

**Output**
- Over-length message
- EOF flag
USING SOFTWARE SPECIFICATION LANGUAGE

1. MODULES AND INTERCONNECTIONS
   DESCRIPTION OF SYSTEM AT HIGH LEVEL OF DETAIL

2. AUTOMATIC TRANSLATION

3. CORRECTION OF INCONSISTENCIES

4. MODULES AND INTERCONNECTIONS
   MORE DETAILED DESCRIPTION

TO DETAILED DESIGN
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.
### Sample Table of Contents from an SSL Report

<table>
<thead>
<tr>
<th>SUBSYSTEM MAIN</th>
<th>VARIABLES</th>
<th>REQUIREMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_TELEGRAM</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GET_WIND</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHAR</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>CHARGE_COUNT</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TELEGRAM</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EOF_FLAG</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WORD</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WORD_COUNT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COLLECT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PRINT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SUBSYSTEM 1.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GET_CHAR</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FILL_BUFFER</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A_CHAR</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CHARACTER_FI</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>BUFFER</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>EOF_FLAG</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CHAR_INDEX</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>READ</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>SEPARATE</td>
<td>2</td>
<td>21</td>
</tr>
</tbody>
</table>
SOFTWARE SPECIFICATION LANGUAGE

Sample SSL Source Program

The subroutine GET-WORD fulfills the requirements transductions COLLECT 1 and COLLECT 2.
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

SUMMARY OF VARIABLE/MODULE CONNECTIONS

1. A_VAR
2. CHRG_STS
3. ELTIMM
4. FTR NL
5. MPL
6. MFC_STS

SSL Requirement to Module Connectivity Matrix
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

SUMMARY OF REQUIREMENT/IMPLEMENTATION CONNECTIONS

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. COLLECT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. PRINT</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SSL Module to Module Connectivity Matrix
SOFTWARE SPECIFICATION AND EVALUATION SYSTEM

SUMMARY OF MODULE/MODULE CONNECTIONS

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CBT TELECOM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. CBT MCFB</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. CBT CHAR</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SSL Variable to Module Connectivity Matrix
Structured to Unstructured Fortran Preprocessor

SOURCE PROGRAM (STRUCTURED FORTRAN AND ANSI FORTRAN STATEMENTS) → CONVERT STRUCTURED STATEMENTS TO ANSI FORTRAN CONSTRUCTS → NEW ANSI FORTRAN SOURCE PROGRAM
Structured Preprocessor Listing

SUBROUTINE GETCH (ICAR, EOF)

INTEGER FLAG, N
LOGICAL EOF, LEN
DATA EOF, LEN / .FALSE., .TRUE. /

BLANK, ICHAR
FOR (I = 1, 12)
   IF (EOF .AND. CHAR) = BLANK
   END FOR
   LEN = .FALSE.

FIND FIRST NON-BLANK CHARACTER
   ICHAR = BLANK
   DO WHILE (ICAR .EQ. BLANK .AND. EOF)
       CALL GETCH (ICAR, EOF)
   END DO

COLLECT WORD
   I = 1
   DO WHILE (EOF .AND. ICHAR .EQ. BLANK)
       IF (I .EQ. 1)
          THEN
              LEN = ICHAR
              I = 1 + 1
          ELSE IF (LEN .GT. 6)
              THEN
                  WRITE (6, 1)
                  FORMAT (1H1, 10HWORD OVERFLOW)
                  LEN = .TRUE.
              END IF
         END IF
   CALL GETCH (ICAR, EOF)
   END DO

RETURN

1 STATEMENTS REMAINING.
5 ERRORS FOUND IN THIS ROUTINE.
42 CARDS READ.
45 CARDS OUTPUT.
0 PREPROCESSOR CHARACTERS FOUND.
7 TOTAL CHARACTERS.
31 TOTAL CHAR.
2 TOTAL WHILES,
3 TOTAL IF'S,
7 TOTAL FORS,
0 TOTAL TEST CASES.
0 PROGRAM UNITS PROCESSED.
GENERATED ANSI FORTRAN

L 21.0 ( JUN 14, 1987)

C PROGRAM NAME = URAN - MAIN, OPT = 12, LINECNT=50, SIZE=1741K;
SIMUL CL=YES, UOT=MI, ILEGA, LAM=1, MP, N=11, 10, 3, MANAGE

IN 0002 SUBROUTINE GETIN ( ICHAK, EOT )
IN 0003 INTEGER ICHAK(LZ)
IN 0004 LOGICAL EOT, LEN
IN 0005 DATA IJLANK / 4K /
IN 0006 IF ( I .LT. 12 )
IN 0007 I = 1
IN 0008 IF ( I ) = IJLANK
IN 0010 CONTINUE
IN 0011 CONTINUE
IN 0012 LEN = ICHAK
IN 0013 ICHAK = IJLANK
IN 0014 IF ( I .LT. 1 )
IN 0015 IF ( ICHAK .LE. IJLANK .AND. .NOT. EOT )
IN 0016 CALL GETCN ( ICHAK, EOT )
IN 0017 10 TO 99996
IN 0018 CONTINUE
IN 0019 I = 1
IN 0020 IF ( I .LT. 1 )
IN 0021 I = 1
IN 0022 IF ( ICHAK .LE. IJLANK .AND. .NOT. EOT )
IN 0023 10 TO 99996
IN 0024 I = I + 1
IN 0025 I = 1 + 1
IN 0026 10 TO 99992
IN 0027 CONTINUE
IN 0028 IF ( I .LT. LEN )
IN 0029 10 TO 99991
IN 0030 WRITE ( 6, 1 )
IN 0031 LEFMT ( 13, 134WCN )
IN 0032 LEN = LEN + 1
IN 0033 CONTINUE
IN 0034 99992 CONTINUE
IN 0035 CALL GETCN ( ICHAK, EOT )
IN 0036 10 TO 99995
IN 0037 CONTINUE
IN 0038 RETURN
IN 0039 END
STATIC CODE ANALYZER

REPORT SUMMARIES THAT PINPOINT POORLY CONSTRUCTED SOFTWARE SEGMENTS

CHECKS TOTAL PROGRAM CONSISTENCY BETWEEN SUBROUTINES

- TYPES OF VARIABLES
- DIMENSIONS OF VARIABLES
- SUBROUTINE CALLS
FARIS PRIMARY LISTING REPORT

PROGRAM MAIN
INTEGER I, K, C(12)
LOGICAL ETF
COMMON IACF, JACF, KACF

210 C Она
COMMON IS UNLABELLED.

IACF = 1
JACF = 1
LEN = .FALSE.
ETF = .FALSE.
IF (1
   .GT. 160
   GOTO 67554
   G0 59558
   I = 1, 160
   CALL (BLOCK ) INGRO, ECF
65553 CONTINUE
65599 CONTINUE
STOP
END

*** EXPLANATIONS ***

Static Code Analyzer Output
STATIC ANALYZER Report on GET-WORD

19X THERE EXISTS A PATH SUCH THAT A LOCAL VARIABLE IS UNINITIALIZED.
DYNAMIC CODE ANALYZER (DCA)

**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

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
**NASA DYNAMIC ANALYZER INPUT REQUESTS **

- DIALECT: STRUCTURED
- ANALYZE: ALL
- REPORT: MAIN
  - REPORT: GETHC
  - REPORT: GETCAR
- $EOF

END OF JJ8 STREAM

Dynamic Analysis Report requested for GET-KORD.

DYNAMIC ANALYZER COMMANDS
SUBROUTINE GETWO(IWORD, EOF)

FETCHES NEXT WORD FROM TELEGRAM

IWORD = CHARACTERS IN WORD
EOF = END OF FILE FLAG

INTEGER IWORD(12)
LOGICAL EOF, LEN
DATA IBLANK / 4H

BLANK FILL WORD
FOR ( I = 1, 12 )
IWORD(I) = IBLANK
END FOR
LEN = .FALSE.

FIND FIRST NONBLANK CHARACTER
ICHAR = IBLANK
DO WHILE ( ICHAR .EQ. IBLANK .AND. .).
CALL GETCHR(ICHAR, EOF)
END DO

COLLECT WORD
I = 1
DO WHILE ( .NOT. ICHAR .EQ. IBLANK )
IF ( I .LE. 12 )
THEN
IWORD(I) = ICHAR
I = I + 1
ELSE IF (.NOT. LEN )
THEN
WRITE(6,1)
FORMAT(1H, 13HWORD OVERFLOW)
LEN = .TRUE.
NT: ELSE INSERTED
ELSE
END IF
CALL GETCHR(ICHAR, EOF)
END DO
RETURN
END
**MODULE DETAIL REPORT FOR GETWD**

--- DDP ANALYSIS ---

<table>
<thead>
<tr>
<th>NODE</th>
<th>MEMBER NODES</th>
<th>ENTRY</th>
<th>DDP TYPE</th>
<th>CONDITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>FOR</td>
<td>( I .LE. 12 )</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>6</td>
<td>FOR</td>
<td>( I .GT. 12 )</td>
</tr>
<tr>
<td>3</td>
<td>4 7 8</td>
<td>10</td>
<td>DOW</td>
<td>( ICHAR.EQ.BLANK.AND..NOT.EOF )</td>
</tr>
<tr>
<td>4</td>
<td>6 9</td>
<td>11</td>
<td>DOW</td>
<td>( NOT.( ICHAR.EQ.BLANK.AND..NOT.EOF ) )</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>24</td>
<td>DOW</td>
<td>( NOT.( ICHAR.EQ.BLANK.AND..NOT.EOF ) )</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>10</td>
<td>IFS</td>
<td>( I .LE. 12 )</td>
</tr>
<tr>
<td>9</td>
<td>11 14</td>
<td>15</td>
<td>IFS</td>
<td>( ELSE-- )</td>
</tr>
<tr>
<td>9</td>
<td>13 15 16 17 18 20 21 22 23</td>
<td>10</td>
<td>IFS</td>
<td>( ELSE-- )</td>
</tr>
<tr>
<td>13</td>
<td>15 19 23 21 22 23</td>
<td>10</td>
<td>IFS</td>
<td>( ELSE-- )</td>
</tr>
</tbody>
</table>

--- MONITORED VARIABLE LIST ---

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>NAME</th>
<th>TYPE</th>
</tr>
</thead>
</table>

0 VARIABLES WERE MONITORED FOR THIS MODULE
### Module Testing Effectiveness Summary

<table>
<thead>
<tr>
<th>Module</th>
<th>Times Invoked</th>
<th># DD Paths</th>
<th># Executed*</th>
<th>% Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>100.0</td>
</tr>
<tr>
<td>GET4()</td>
<td>100</td>
<td>40</td>
<td>6</td>
<td>40.0</td>
</tr>
<tr>
<td>GETCAR</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>14</td>
<td>6</td>
<td></td>
<td>42.9</td>
</tr>
</tbody>
</table>

*(At Least Once)*

Dynamic Analyzer Run Time Report
### Dynamic Analyzer Run Time Report

#### Reproducibility of the Original Page is Poor

---

---

<table>
<thead>
<tr>
<th>DDP Number</th>
<th>Entry Count</th>
<th>Percent Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1200</td>
<td>80.0</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>6.7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>6.7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

---

| Totals     | 1500        | Percentage DD Paths Executed: 40.0 |

---

---
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.
**Cover Selector and Analytic Path Analysis Report for Module GETWO**

---

**Analytic Path Analysis**

5 -- Number of Decision Nodes
10 -- Number of Decision-to-Decision Paths (DDP's)
10 -- 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**