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)
SPECIFICATION AND EVALUATION SYSTEM DESIGN,
PART 2 Final Report (Science Applications,
Inc., Huntsville, Ala.) 211 p HC $7.75

Unclas

CSCL 09E G3/61 26773
NASA SOFTWARE SPECIFICATION
AND EVALUATION SYSTEM DESIGN
FINAL REPORT PART II

SOFTWARE VERIFICATION/VALIDATION TECHNIQUES

CONTRACT NAS8-31379

Prepared under the direction of:

Mr. Bobby C. Hodges
Marshall Space Flight Center
National Aeronautics and Space Administration

March 19, 1976

SCIENCE APPLICATIONS, INC.
2109 W. Clinton Avenue, Suite 800, Huntsville, AL 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>1-1</td>
</tr>
<tr>
<td>2. SOFTWARE SPECIFICATIONS</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1 THE LANGUAGE</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.1 Introduction</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.1.1 Need for SSL</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.1.2 Unique Features of SSL</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.1.3 Background</td>
<td>2-4</td>
</tr>
<tr>
<td>2.1.2 The Grammar</td>
<td>2-5</td>
</tr>
<tr>
<td>2.1.2.1 Metalanguage Description</td>
<td>2-5</td>
</tr>
<tr>
<td>2.1.2.2 Overview of SSL Grammar</td>
<td>2-6</td>
</tr>
<tr>
<td>2.1.2.3 Basic Vocabulary</td>
<td>2-8</td>
</tr>
<tr>
<td>2.1.2.4 Basic Language Elements</td>
<td>2-10</td>
</tr>
<tr>
<td>2.1.2.5 Requirement Declaration</td>
<td>2-14</td>
</tr>
<tr>
<td>2.1.2.6 Data Type and Variable Declaration</td>
<td>2-17</td>
</tr>
<tr>
<td>2.1.2.7 Constant Declaration</td>
<td>2-31</td>
</tr>
<tr>
<td>2.1.2.8 Data References</td>
<td>2-32</td>
</tr>
<tr>
<td>2.1.2.9 Expressions and Assertions</td>
<td>2-34</td>
</tr>
<tr>
<td>2.1.2.10 Module Descriptions</td>
<td>2-44</td>
</tr>
<tr>
<td>2.1.2.11 Subsystem Descriptions</td>
<td>2-55</td>
</tr>
<tr>
<td>2.1.3 Example</td>
<td>2-58</td>
</tr>
<tr>
<td>2.2 SEMANTIC DESCRIPTION OF SSL</td>
<td>2-64</td>
</tr>
<tr>
<td>2.3 OVERVIEW OF THE SSL TRANSLATOR</td>
<td>2-70</td>
</tr>
<tr>
<td>2.3.1 The Formal Software Requirement</td>
<td>2-70</td>
</tr>
<tr>
<td>2.3.2 Functional Design Overview</td>
<td>2-78</td>
</tr>
<tr>
<td>2.3.3 Detailed Design Notes</td>
<td>2-81</td>
</tr>
<tr>
<td>2.3.3.1 Assessing Data Availability</td>
<td>2-82</td>
</tr>
<tr>
<td>2.3.3.2 Assessing Consistency of Data Usage</td>
<td>2-83</td>
</tr>
<tr>
<td>2.3.3.3 Ordering Modules for Analysis</td>
<td>2-84</td>
</tr>
<tr>
<td>2.3.3.4 Construction and Closure of Dependency Matrices</td>
<td>2-85</td>
</tr>
<tr>
<td>2.3.3.5 Recursive Analysis Using Dependency Matrices</td>
<td>2-86</td>
</tr>
<tr>
<td>3. DATA BASE VERIFIER SUBSYSTEM DESIGN</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1 DML STATEMENT PROCESSING</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2 SUBSCHEMA INFORMATION PROCESSING</td>
<td>3-9</td>
</tr>
<tr>
<td>3.3 FUNCTIONAL REQUIREMENTS FOR THE DBVS</td>
<td>3-9</td>
</tr>
<tr>
<td>3.4 FUNCTIONAL SPECIFICATIONS FOR THE DBVS</td>
<td>3-23</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>4. STATIC CODE ANALYSIS</td>
<td>4-1</td>
</tr>
<tr>
<td>5. REFERENCES</td>
<td>5-1</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2-1</td>
<td>Syntactical Overview of an SSL Specification</td>
</tr>
<tr>
<td>2-2</td>
<td>Module Structure Chart for Example</td>
</tr>
<tr>
<td>2-3</td>
<td>SSL Description for Example</td>
</tr>
<tr>
<td>2-4</td>
<td>SRD for SSL Translator</td>
</tr>
<tr>
<td>2-5</td>
<td>Block Diagram for SSL Translator</td>
</tr>
<tr>
<td>2-6</td>
<td>A Module Ordering Example</td>
</tr>
<tr>
<td>2-7</td>
<td>An Example D Matrix</td>
</tr>
<tr>
<td>2-8</td>
<td>An Example D* Matrix</td>
</tr>
<tr>
<td>3-1</td>
<td>Subsystem Software Requirement Document for DBVS</td>
</tr>
<tr>
<td>3-2</td>
<td>DBVS Subsystem Preamble</td>
</tr>
<tr>
<td>3-3</td>
<td>DML_RECOGNIZER Module Description</td>
</tr>
<tr>
<td>3-4</td>
<td>INITIALIZE_SYSTEM Module Description</td>
</tr>
<tr>
<td>3-5</td>
<td>SUBSCHEMA_PROCESS Module Description</td>
</tr>
<tr>
<td>3-6</td>
<td>REALM_PROCESS Module Description</td>
</tr>
<tr>
<td>3-7</td>
<td>SET_PROCESS Module Description</td>
</tr>
<tr>
<td>3-8</td>
<td>RECORD_PROCESS Module Description</td>
</tr>
<tr>
<td>3-9</td>
<td>ERROR_PROCESS Module Description</td>
</tr>
<tr>
<td>3-10</td>
<td>OUTPUT_SUMMARY Module Description</td>
</tr>
<tr>
<td>3-11</td>
<td>Module Structure Chart for DBVS</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>SSL Special Symbols</td>
<td>2-9</td>
</tr>
<tr>
<td>2-2</td>
<td>SSL Reserved Words</td>
<td>2-11</td>
</tr>
<tr>
<td>2-3</td>
<td>Arithmetic Operations</td>
<td>2-37</td>
</tr>
<tr>
<td>2-4</td>
<td>Operation Hierarchy</td>
<td>2-40</td>
</tr>
<tr>
<td>2-5</td>
<td>Logical Operator Truth Table</td>
<td>2-41</td>
</tr>
<tr>
<td>2-6</td>
<td>Boolean and Relational Operations</td>
<td>2-42</td>
</tr>
<tr>
<td>2-7</td>
<td>Module Descriptions for Example</td>
<td>2-61</td>
</tr>
<tr>
<td>3-1</td>
<td>DBVS Description Outline</td>
<td>3-2</td>
</tr>
<tr>
<td>3-2</td>
<td>Character Table</td>
<td>3-4</td>
</tr>
<tr>
<td>3-3</td>
<td>FORTRAN DML Command Table</td>
<td>3-5</td>
</tr>
<tr>
<td>3-4</td>
<td>Allowable Keywords Within DML Statements</td>
<td>3-5</td>
</tr>
<tr>
<td>3-5</td>
<td>Identifier Sequence Elements</td>
<td>3-6</td>
</tr>
<tr>
<td>3-6</td>
<td>List Item Sequence Elements</td>
<td>3-8</td>
</tr>
<tr>
<td>3-7</td>
<td>Subschema Table Record Description</td>
<td>3-10</td>
</tr>
<tr>
<td>3-8</td>
<td>Realm Table Record Description</td>
<td>3-11</td>
</tr>
<tr>
<td>3-9</td>
<td>Set Table Record Description</td>
<td>3-12</td>
</tr>
<tr>
<td>3-10</td>
<td>Record Table Record Description</td>
<td>3-13</td>
</tr>
<tr>
<td>3-11</td>
<td>Error Status Table Record Description</td>
<td>3-15</td>
</tr>
<tr>
<td>3-12</td>
<td>Input Formats</td>
<td>3-18</td>
</tr>
<tr>
<td>3-13</td>
<td>Meta-Symbol Meanings</td>
<td>3-21</td>
</tr>
<tr>
<td>3-14</td>
<td>Outline of SSL Components</td>
<td>3-24</td>
</tr>
<tr>
<td>3-15</td>
<td>Module Descriptions for the DBVS</td>
<td>3-46</td>
</tr>
<tr>
<td>4-1</td>
<td>New Faces Capabilities</td>
<td>4-2</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Initial work performed under this contract (in response to SOW Tasks Phase A, B, and C (Item 1) consisted of conducting a survey and analysis of the existing methods, tools, and techniques that were being currently employed in the development of software. As a result of this effort, we made the following recommendations for the construction of reliable software which pertain to all phases of the software development cycle:

- The software requirements stage should result in a structured, formal document which leads naturally into the software specification stage. It should be produced by an experienced analyst working in conjunction with the user. Origination of key software testcases should be an integral part of this stage.

- Software functional design specifications should be carried out through a formal language which is capable of reflecting fidelity of design with software requirements.

- Program code should be implemented using a structured programming language in which control structures are operationally apparent and as few in number as tolerable. Hence a structured preprocessor should be employed for code implementation if a structured compiler language isn't available.

- A programming language which promotes standardization of methods for accessing and operating on stored data bases such as the CODASYL Data Manipulation Language should be adopted and employed for purposes of data base verification.
Software testing should be automated to establish user confidence while minimizing cost. Both static and dynamic testing are required. A static analyzer should enforce programming standards, while a dynamic analyzer should check the reliability of the code during execution. Structural and requirements testcase generators would greatly enhance the utility of the analyzers. A structural testcase generator produces data to test as many branches of the code as possible and should be employed for determination of software reliability. A requirements testcase generator produces data to determine the consistency of the code with the software requirements.

Maintenance documentation needs to be an integral part of software. Documentation guidelines need to be established.

As work progressed, we translated our analysis results into viable software designs for three of the SSES tools. This work was done in accordance with SOW Tasks Phase A, items 1, 3, 4a, 4b, 4e, and 4f and Phase C, items 1 and 2. In the remainder of this report, we present functional software designs for the Software Specifications Language (SSL) (including a language reference manual) and the Data Base Verifier Subsystem (DBVS). In addition, a detailed (build-to) software design is described for the new capabilities to be incorporated into the static analyzer, FACES (FORTRAN Automated Code Evaluation System).
2. SOFTWARE SPECIFICATION

A primary goal of the SSES system is to provide early software feasibility and testing. Commensurate with this goal we have defined the Software Specification Language (SSL) for which the design is contained within this section. SSL is capable of representing all the information inherent in a functional block diagram of a software system. In addition, it is capable of (a) explicitly denoting the internal structure of data, (b) more completely denoting module interdependencies, and (c) expressing the input and output variables of each module. However, the most distinctive aspect of the language is the ability to attach requirement attributes to modules and variables which may be used in performing consistency checks.

The experience gained thus far in using SSL indicates that it requires slightly more effort than is normally applied to functional design. However, the effort is rewarded during detailed design as the module interconnections are much more evident and module functions tend to be more uniform and tractable.
2.1 THE LANGUAGE

2.1.1 Introduction

SSIJ (Software Specification Language) is a new formalism for the definition of specifications for software systems. The language provides a linear format for the representation of the information normally displayed in a two-dimensional module inter-dependency diagram. In comparing SSL to FORTRAN or ALGOL, one finds the comparison to be largely complementary to the algorithmic (procedural) languages. SSL is capable of representing explicitly module interconnections and global data flow, information which is deeply imbedded in the algorithmic languages. On the other hand, SSL is not designed to depict the control flow within modules. We refer to the SSL level of software design which explicitly depicts inter-module data flow as a functional specification.

We wish to express our appreciation to Mr. Bobby Hodges of Data System Laboratory, George C. Marshall Space Flight Center for his guidance and support in the performance of this task.

2.1.1.1 Need for SSL

The current state of the art in software development permits insufficient formal evaluation prior to implementation. Such questions as:

- Are all requirements fulfilled?
- Have all software elements been defined?
- Are the element interconnections consistent?

cannot be answered in a manner that is independent of the designer's opinion. The intent of SSL is to formalize, through a language, the statement of the functional specification for a software system. Given this formal statement expressed in SSL and a translator for the SSL language, an independent evaluation of the software may begin much earlier in the development cycle.
In addition to evaluation, other aspects of SSL can aid both the designer and implementer. Several things that are characteristically omitted or inadequately performed during early design but required in SSL are:

- A complete and consistent statement of the software requirement
- Unambiguous communication of software organization to the detailed designer
- Enumeration of intraprogram consistency checks (assertions) useful during checkout.

A translator also provides tables and summaries for the final software documentation and a software element cross reference file. The latter could be used to statically verify the fidelity of the final code to original specifications.

2.1.1.2 Unique Features of SSL

The major contribution of SSL is the formal approach it brings to a phase of software development previously relegated to heuristic techniques as discussed above. Within this framework, there are several unique technical features possessed by SSL. First, the projection of a specialized form of software requirements onto the objects being defined establishes a rationale for the software structure not present in other methodologies. These requirements are an important aspect of consistency checking when evaluating a specific functional design. Second, the incorporation of levels of abstractions directly in a design methodology is a step forward in software engineering. Lastly, an automated SSL translator is being designed that is one of several interlocking software design and evaluation tools collectively called Software Specification and Evaluation System (SSES). SSES includes a static code analyzer, a dynamic code analyzer, and a test case analyzer. The specific capability that SSL brings SSES is the ability to test and evaluate software design early in the development cycle.
SSL also incorporates a flexible data abstraction capability and places emphasis on assertions as a means of describing the dynamic behavior of the software being designed. Although neither of these is unique, they are relatively new concepts in the field of computer science.

2.1.1.3 Background

In evaluating a new software system, particularly a programming language, it is important to trace the historical developments to which it relates and upon which it is based. The MIL (Module Interconnection Language) system [1] was a principal contributor to the concepts of data creation and data availability restrictions among modules within SSL. Guidelines imposed for the partition of programs into subsystems are derived from the principles embodied in the concept of levels of abstraction [2]. Module descriptions in SSL are a linearized form of the information available in the two-dimensional diagrams referred to as structure charts [3]. The data description capability is largely the same as that of PASCAL [4]. The syntax for expressions is derived from, but not identical to, that of ALGOL 60 [5]. Assertions in SSL have the form and appearance of those in the language NUCLEUS [6].
2.1.2 The Grammar

The material in this section is arranged in the form of a reference guide to the language, and not tutorially in the manner of a user's manual. To aid the reader, a cross reference index is provided in the last section.

2.1.2.1 Metalanguage Description

For the purposes of automatic translation and unambiguous communication, it is desirable to express SSL via a formal grammar. The vehicle selected for this purpose is the Backus-Naur-Form (BNF) metalanguage [5]. BNF has the advantages of being well known and compact in representation. In addition, most formal methodologies for analyzing grammars are based upon BNF representation.

Any nontrivial language contains an infinite number of legal sentences. Each sentence, in turn, is composed of the concatenation of strings; strings are composed of characters. A grammar uses strings as operands and combines them under the operation of concatenation to finitely depict, all legal sentences. The way in which this is done in BNF can best be interpreted via an example. Consider the following production:

\[
\text{<ab>} ::= \text{a}\mid \text{b} \mid \text{<ab> a}
\]

Sequences of characters enclosed within the brackets < > represent metalinguistic variables called nonterminal symbols. The marks " ::= " and " | " are metalinguistic connectives meaning "is composed of" and "or" respectively. Any string not a nonterminal or connective denotes itself and is called a terminal symbol. Juxtaposition of symbols between connectives in a formula, such as the example, signifies that the symbols must be in the exact order denoted. The above production indicates that \text{<ab>} may have the values:
In BNF, the null string is designated by <empty> ::=

SSL is represented as a context-free grammar which means:

- There exist a finite number of productions of the type of the above example.
- The left part of each production (i.e., left of ::=) consists of a single nonterminal symbol.
- There exists a unique nonterminal symbol (called the distinguished symbol) which is in the right part of no production except its own.

2.1.2.2 Overview of SSL Grammar

Prior to examining the detailed structure of SSL components, it will be useful to identify the overall structure of a software specification expressed in SSL. Figure 2-1 depicts the sequencing of the syntactical items used to describe an SSL specification.

A specification consists of one or more subsystems, each but the first having a name. The first subsystem is referred to as the "main" subsystem and each subsystem is composed of a preamble and one or more module descriptions. The preamble defines the local environment for the subsystem.
Figure 2-1  Syntactical Overview of an SSL Specification
(constants, requirements, data formats, etc.) and the module
descriptions indicate operational aspects of program units
(program units are subprograms, procedures, etc.).

In the following subsections, the detailed syntactical
descriptions will be presented.

2.1.2.3 Basic Vocabulary

The basic vocabulary of SSL consists of special
symbols, letters, digits, and reserved words. Each special
symbol (Table 2-1) is primarily a single character except
where limited computer character fonts require the concatena-
tion of two characters. Where a special symbol consists of
more than one character, it must be written without an inter-
vening blank. Subsequently, special symbols other than ".", "@", and "_" will be referred to as delimiters. Each char-
acter in Table 2-1 is available within the ANSI standard codes
[7] for ASCII-8, EBCDIC-8, and HOLLERITH-256. Substitutions
may be necessary if an SSL translator is implemented in an
environment not conforming to the standard character codes.

Letters and digits do not have individual meanings
but are used to construct identifiers, numbers, and reserved
words. The following basic productions enumerate these ele-
ments of the vocabulary:

\[
\begin{align*}
\text{<letter> } &::= a | b | \ldots | z \\
\text{<digit> } &::= 0 | 1 | \ldots | 9
\end{align*}
\]
TABLE 2-1  SSL SPECIAL SYMBOLS

| +    |   | [   |
| -    |   | ]   |
| *    |   |     |
| /    |   |     |
| =    |   | <=  |
| ,    |   | >=  |
| ;    |   |     |
| ...  |   |     |
| <    |   |     |
| >    |   |     |
| (    |   |     |
| )    |   | @   |
|     |   |     |
Reserved words (Table 2-2) are composed entirely of sequences of letters. In this document, they are normally underlined. A reserved word may not contain imbedded blanks and must always be followed by a blank or a delimiter.

The construct

/* any sequence of symbols not containing "*/ */

may be inserted between any two identifiers, numbers, delimiters, or reserved words. It is called a comment any may be removed from the program text without altering its meaning.

2.1.2.4 Basic Language Elements

2.1.2.4.1 Identifiers

Syntax

<identifier> ::= <letter>|<identifier> <letter>

|<identifier> <digit>|<identifier> _

Examples

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5ad</td>
</tr>
<tr>
<td>b27</td>
<td>sr$p</td>
</tr>
<tr>
<td>cr_14dr</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>Fulfil</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Accesses</td>
<td>Fulfills</td>
</tr>
<tr>
<td>Analog</td>
<td>Global</td>
</tr>
<tr>
<td>And</td>
<td>Implies</td>
</tr>
<tr>
<td>Array</td>
<td>In</td>
</tr>
<tr>
<td>Assume</td>
<td>Input</td>
</tr>
<tr>
<td>Assumes</td>
<td>Inputs</td>
</tr>
<tr>
<td>Boolean</td>
<td>Integer</td>
</tr>
<tr>
<td>Case</td>
<td>Iteratively</td>
</tr>
<tr>
<td>Char</td>
<td>Modify</td>
</tr>
<tr>
<td>Conditionally</td>
<td>Modifies</td>
</tr>
<tr>
<td>Constant</td>
<td>Module</td>
</tr>
<tr>
<td>Constants</td>
<td>Of</td>
</tr>
<tr>
<td>Constraint</td>
<td>Or</td>
</tr>
<tr>
<td>Constraints</td>
<td>Output</td>
</tr>
<tr>
<td>Create</td>
<td>Outputs</td>
</tr>
<tr>
<td>Creates</td>
<td>Real</td>
</tr>
<tr>
<td>Digital</td>
<td>Receive</td>
</tr>
<tr>
<td>Doubleprecision</td>
<td>Receives</td>
</tr>
<tr>
<td>End</td>
<td>Record</td>
</tr>
<tr>
<td>Entry</td>
<td>Requirement</td>
</tr>
<tr>
<td>Equ</td>
<td>Requirements</td>
</tr>
<tr>
<td>Execute</td>
<td>Satisfies</td>
</tr>
<tr>
<td>Executes</td>
<td>Satisfy</td>
</tr>
<tr>
<td>False</td>
<td>Set</td>
</tr>
<tr>
<td>File</td>
<td>Subjectto</td>
</tr>
<tr>
<td>For</td>
<td>Subsystem</td>
</tr>
<tr>
<td>Forall</td>
<td>Text</td>
</tr>
<tr>
<td>From</td>
<td>To</td>
</tr>
</tbody>
</table>
REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

Semantics

Identifiers must begin with an alphabetic character and contain only letters, digits, and the "_" symbol. The latter is known as the break character. Identifiers have no inherent meaning, but serve as identification for variables, modules, subsystems, and other elements of a software specification.

Identifiers may be of arbitrary length but must be unique within the first twelve characters. No identifier may be equivalent to the first twelve characters of a reserved word. The same identifier may not be used to denote two different quantities within a subsystem with the exception of field names in different records.

2.1.2.4.2 Numbers

Syntax

< unsigned integer> ::= <digit> | <unsigned integer> <digit>

< sign > ::= + | -

< exponent part > ::= e <unsigned integer>
| e <sign> <unsigned integer>
| d <unsigned integer>
| d <sign> <unsigned integer>

< decimal number > ::= <unsigned integer>
| <unsigned integer> .
| <unsigned integer>

< unsigned number > ::= <decimal number>
| <decimal number> <exponent part>
Examples

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>3,746</td>
</tr>
<tr>
<td>14d10</td>
<td>XII</td>
</tr>
<tr>
<td>3.7 e-5</td>
<td>e+7</td>
</tr>
<tr>
<td>0.2</td>
<td>.14</td>
</tr>
</tbody>
</table>

Semantics

Decimal numbers have their conventional arithmetic meaning. The exponent is a scale factor expressed as an integral power of 10. A number expressed with neither a scale factor nor a decimal fraction is assumed to be of type integer. A number which uses the "d" form for the exponent part is assumed to be double precision. Otherwise, the number is assumed to be type real. Note that if a number contains a decimal point, at least one digit must precede and succeed the point.

2.1.2.4.3 Logical Values

Syntax

<logical value> ::= true | false

Semantics

- Logical values have their conventional meaning and may be defined by describing their combination under the operations "union" and "intersection". The union of the logical value true with any other logical value always yields the result true. The intersection of the logical value false with any other logical value always yields the result false.
2.1.2.5 Requirement Declaration

The several parts of the requirement declaration are used to identify the data flow between the software package being described and other parts of the total system. In addition, they identify processing steps (called transductions) and restrictions (called constraints) which are attached to both modules and variables.

Syntax

\[
\begin{align*}
\text{<requirement declaration> ::= <requirement or requirements>} \\
\text{<requirement statement group> end}
\end{align*}
\]

\[
\begin{align*}
\text{<requirement or requirements> ::= requirement} \\
\quad \text{| requirements}
\end{align*}
\]

\[
\begin{align*}
\text{<requirement statement group> ::= <requirement statement part>} \\
\quad | <\text{requirement statement group} ; <\text{requirement statement part}>
\end{align*}
\]

\[
\begin{align*}
\text{<requirement statement part> ::= <input part>} \\
\quad | <\text{output part}| <\text{transduction part}>
\quad | <\text{constraint part}>
\end{align*}
\]

2.1.2.5.1 Input and Output Parts

An input is a system level input (or stimulus) which a software package receives from an external source. An output is a system level response which has a purpose beyond the immediate concern of the software package being described.
Syntax

\[ <\text{input part}> :: = <\text{input or inputs}> \text{ <entire variable list}> \]

\[ <\text{output part}> :: = <\text{output or outputs}> \text{ <entire variable list}> \]

\[ <\text{input or inputs}> :: = \text{input|inputs} \]

\[ <\text{output or outputs}> :: = \text{output|outputs} \]

Examples

- \text{input} state\_vector;
- \text{inputs} mass, velocity, distance;
- \text{output} concordance\_list;

Semantics

A variable may be in both an input and an output list. A variable in an output list not used within the subsystem other than in the module in which it is initialized is not required to have a requirement transduction attribute. The structure of all variables in input and output lists must be described within the variable statements of the subsystem preamble. Each subsystem preamble must have a requirement declaration with an output part.

2.1.2.5.2 Transduction Parts

Transductions are identifiers representing processing steps. They are derived by first writing a high level pseudo-program to "transduce" the input variables into the output variables and then extracting and listing the major verbs of the program. Just as the processing steps of the pseudo-program may be nested, the transductions may likewise be nested. Ideally, for each subsystem there should be from three to seven transductions that are not nested within any others.
Syntax

<transduction part> ::= <transduction or transductions>

<transduction clause> | <transduction part> ; <transduction clause>

<transduction or transductions> ::= transduction

| transductions

<transduction clause> ::= <transduction list>

|<transduction list> in <transduction list>

<transduction list> ::= <transduction identifier>

|<transduction list>, <transduction identifier>

<transduction identifier> ::= <identifier>

Examples

- transduction sum_expense, sub_deduct in tax_compute;
  write_paycheck;

- transductions save_options; read_card in parse;

Semantics

Within a transduction clause, each processing step represented by a transduction identifier to the left of in must be a substep of the processing steps listed on the right of in. Each transduction identifier represents a unique processing step, but may be reused to show different substep relationships. Substep relationships must be consistent, i.e., the complete set of substep relationships partially order the transduction identifiers.
2.1.2.5.3 Constraint Parts

Syntax

\[
\begin{align*}
<\text{constraint part}> & ::= <\text{constraint or constraints}> \\
& <\text{constraint list}> \\
<\text{constraint or constraints}> & ::= \text{constraint} \\
& | \text{constraints} \\
<\text{constraint list}> & ::= <\text{constraint identifier}> \\
& | <\text{constraint list}>, <\text{constraint identifier}> \\
<\text{constraint identifier}> & ::= <\text{identifier}> \\
\end{align*}
\]

Examples

- \text{constraint} carpool_size ;
- \text{constraints} max_targets, minimum_distance ;

Semantics

Each constraint identifier defined must be attached as an attribute to some module in the subsystem.

2.1.2.6 Data Type and Variable Declarations

Explicit description of data and the ability to define and use new data types is one of the greatest assets of SSL. A new data type may be described directly as part of a variable declaration, or described independently for subsequent use.

Syntax

\[
\begin{align*}
<\text{type declaration}> & ::= <\text{type or types}> \\
& <\text{type definition}> \\
& | <\text{type declaration}> ; <\text{type definition}> \\
<\text{type or types}> & ::= \text{type} | \text{types} | \text{global type} \\
& | \text{global types} \\
\end{align*}
\]
<type definition>::= <identifier> = <type>
<type>::= <simple type> |<structured type>
                  |<pointer type>

<variable declaration>::= <variable or variables>
                          <variable definition>
                          |<variable declaration> ; <variable definition>

<variable or variables>::= variable | variables
<variable definition>::= <identifier list> :< type>
                          |<identifier list> :<type> ; <for clause>
                          |<identifier list> :<type> ; <subjectto clause>
                          |<identifier list> :<type> ; <for clause>;

<subjectto clause>
<for clause>::= for <transduction list>
<subjectto clause>::= subjectto <assertion list>
<assertion list>::= <assertion>
                          |<assertion list> ; <assertion>

<identifier list>::= <identifier> |<identifier list> ,
                          <identifier>

Semantics

A type declaration list is used to define new data types. Each type is named and may be referenced by the identifier to the left of "=" in the <type definition> production. The normal scope of a type identifier is the subsystem in which it is defined. However, the scope of a global type is the entire SSL program. Global types may be defined only in the main subsystem.
A data type need not be named if it is defined intrinsic to the variable declaration. Both type and variable declarations may use data types defined and named elsewhere. Examples of both are given in the following subsections.

The <for clause> of the variable declaration is used to attach requirement attributes. Requirement attributes limit the availability of variables within the modules of the subsystem. All variable declarations must contain a for clause with the exception of output variables identified in the requirement statement.

The <subjectto clause> identifies the global assertions associated with the variables being declared. A global assertion is one that must be true upon exit from the module creating the variable, and true on both entry and exit of modules using the variable.

2.1.2.6.1 Simple Types

Simple types are data types for which the designer, using SSL, need not define the internal structure or the internal structure has previously been defined and named.

Syntax

\[
\text{<simple type>} ::= \text{<basic type}> | \text{<scalar type}> \\
| \text{<subrange type>} | \text{<type identifier>}
\]

\[
\text{<type identifier>} ::= \text{<identifier>}
\]

Semantics

A type identifier must previously have been used to the left of an "=" in a type definition.
2.1.2.6.1.1 Basic Types

The basic data types are those which are implicitly defined by the SSL language.

<basic type> ::= integer | real | boolean
               doubleprecision | char | analog | text

Examples

- variables I, J, K: integer; for count_people;

- variables height: real;
  for record_status;
  subjectto height >0.0 ;
  height <= 10.0 ;
  employed: boolean ;
  for record_status ;

Semantics

The types integer, real, boolean, and doubleprecision have the conventional meaning. The type char indicates a single unit of hollerith information. Type text indicates hollerith data with unspecified length. Since the length of a text item varies, it may not be combined with other variables in forming structured data types. The type analog designates a data item which contains analog signal information. Like the text type, it may not be combined with other variables to form structured types.
2.1.2.6.1.2 Scalar Types

Scalar types are used to designate a finite number of disjoint states which a variable may represent. In conventional programming languages, it is customary to declare the variable of type \texttt{integer} and assign it only the cardinal numbers 1, 2, ..., \texttt{n} where each value represents one state of the several possible.

Syntax

\begin{verbatim}
<scalar type> ::= ( <identifier list> )
\end{verbatim}

Examples

- \texttt{type marital_status = (single, married, divorced)};
- \texttt{variable ms: marital_status; for emp_record;}
- \texttt{variable color: (Red, blue, yellow, green)};

Semantics

Conceptually, the elements of scalar types are ordered regardless of whether or not the underlying set of states is ordered. The order is always the same as that of the identifiers in the identifier list. This enables a designer to use relational tests (<, >, etc.) in assertions involving scalar type variables.

2.1.2.6.1.3 Subrange Types

Subrange types are used to designate a subset of integers or scalars which a data item may assume.

Syntax

\begin{verbatim}
<subrange type> ::= <constant> .. <constant>
<constant> ::= <unsigned integer>
| <sign> <unsigned integer>
| <constant identifier>
\end{verbatim}
|<sign> <constant identifier>  
|<logical value>  

< constant identifier> ::= <identifier>  

**Examples**  

- **variables**  
  weight: 10..350; **for** ins_compute;  
  dependents: 0..15; **for** tax_compute;  

- **type**  
  color = (purple, blue, red, yellow, green, black);  
  variable primary_color: blue..green;  

**Semantics**  

A subrange simply indicates the least and largest constant values an item may assume. The lower bound (left-most constant in the production) must be less than the upper bound. A subrange with bounds expressed in types other than integer or scalar is not permitted.  

Constant identifiers may arise from two contexts. The first is the appearance of an identifier to the left of "=" in a constant declaration. The second (illustrated by the above example) is the appearance of an identifier as a scalar element. Constant identifiers arising from the second context may not be preceded by a unary sign.  

2.1.2.6.2 **Structured Types**  

A structured type is a data type composed of more elementary data types.
Syntax

< structured type> ::= <array type> | <record type> 
| <digital type> | <set type> 
| <sequence type>

Semantics

An SSL structured data type is used to indicate the general form and content of a data structure, not precise implementation word and storage formats. In SSL, the following definitions are used:

Array - A fixed number of data items, all of the same type and length and accessed by computed index.

Record - A fixed number of data items, each of fixed length, and each equally accessible.

Digital - A record having additional restrictions which are discussed in a subsequent subsection.

Set - An element of the powerset of a finite number of basic elements.

Sequence - A variable number of data items, all of the same type and length; however, each element is not equally accessible at all times.

Stronger connotations (such as elements of an array are sequentially stored) are not implied by the semantics of SSL.
2.1.2.6.2.1 Arrays

An array is a fixed number of data elements, each of the same type and length and each equally accessible. Elements of an array are ordered and each element is accessed by a cardinal number called its index.

Syntax

\[
<\text{array type}> ::= \text{array}\ [\text{<index list>}]\ \text{of}
\]
\[
\text{<component type>}
\]

\[
<\text{index list}> ::= \text{<index type>} | \text{<index list>}
\]
\[
\text{<index type>}
\]

\[
<\text{index type}> ::= \text{<simple type>}
\]

\[
<\text{component type}> ::= \text{<type>}
\]

Examples

• \underline{variable} matrix: \text{array [1..10, 0..20] of real; for ta;}

• \underline{type} people = (adams, buckles, jones, smith);
  \text{variable employee: array [people] of 1..50; for ta;}

Semantics

Index types must have a finite range and be ordered. This requirement eliminates index type of integer, real, and double-precision. However, subranges of integers are permitted. For the purpose of ordering, false <true for boolean type indices.

2.1.2.6.2.2 Records

A record is a structure containing a number of components called fields. Fields are not constrained to be of identical type but must be of fixed length. A single record type is permitted to have variants.

Syntax

<record type> ::= record <field list> end

<field list> ::= <fixed part> | <fixed part> ;

<variant part> |<variant part>

<fixed part> ::= <record section> |<fixed part> ;

<record section>

<record section> ::= <field identifier list> : <type>

<field identifier list> ::= <field identifier>

|< field identifier list> , <field identifier>

<variant part> ::= case <tag field> <type identifier>

of <variant list>

<variant list> ::= <variant> | <variant list> ;

<variant>

<tag field> ::= <field identifier> :

<field identifier> ::= <identifier>

<variant> ::= <case label list> : ( <field list> )

|<case label list> : ( )

<case label list> ::= <case label> |<case label list>,

< case label>

<case label> ::= <constant>
Examples

- **Type** employee = Record

  Number: Integer;
  Salary: Real;
  Name: Array [1..24] of char

End;

- **Variable** machine_part: Record

  Part_No, Order_Quantity: Integer;
  Weight: Real

End;

for customer_billing;

- **Type** complex = Record real_part, imag_part: real End;

- **Type** farm = (peaches, cotton, soybeans);

Type land_use = Record

  Owner_Name: Array [1..12] of char;
  Plot_No: Integer;
  Case_Crop: Farm of

  peaches: (tree_count: Integer);
  cotton, soybeans: (plant_date: Integer;
                      herbicide, insecticide: boolean)

End;

- **Variable** sizes: Array [1..10] of Record

  Height: Integer;
  Weight: Real

End;
for health_file_update;
subjectto
height >0; height <120;
weight >0.0; weight <500.0;

Semantics

Fields may not be of basic types text or analog. A record may be a component of another record, but a digital type may not. The scope of a field identifier is the smallest record in which it is defined. Field identifiers with disjoint scopes may be reused. Access of a component is always by the field identifier and never by a computed value.

The type associated with the tag field of a variant must contain only a finite number of elements. This limits it to boolean, subrange, and scalar. All elements of the type must appear in some case label list of the variant. If the field list for case label L is empty, the form is:

L : ( )

A record may contain only one variant part and it must succeed the fixed part. However, a variant may contain variants. That is, it is possible to have nested variants. All field names of the same record must be unique even if they are in different variants.

2.1.2.6.2.3 Digital Types

Digital types are a restricted form of records to represent real time digital signals.

Syntax

<digital type> ::= digital <fixed part> end
Example

- **Variable**
  
  ```
  Signal_In: digital
  Valve_1: boolean;
  LOX_Switch: 1..3;
  Command: (Idle, stopped, running)
  End;
  for check_status;
  ```

**Semantics**

Due to their physical interpretation, the type of components within digital types may only be boolean, scalar, or subrange. Digital types may not have variant parts and they may not be used as components of any other type.

### 2.1.2.6.2.4 Set Types

Set types represent elements of powersets over a finite set of elements called the base type. Conceptually, a set type variable may be viewed as a bitstring of length equal to the number of elements in the base type. Each bit is associated with a unique element and is "on" or "off" if the element is a member or not a member of the powerset.

**Syntax**

```
<set type> ::= set of <base type>
<base type>::= <simple type>
```

**Examples**

- **Type**
  
  ```
  members = (father, mother, big_sister,
             little_sister, big_sister, little_brother);
  ```
  
  **Variable**
  
  ```
  family: set of members; for arrange;
  ```

- **Variable**
  
  ```
  Even_numbers: set of -10..10;
  ```
  
  **for** compute_something;
Semantics

The base type must be either scalar or subrange.

2.1.2.6.2.5 Sequence (File) Types

A sequence differs from an array in that it may vary dynamically in length and is referenced through a "window" called its buffer (not by computed index). Examples of physical representations of sequences include linked lists and mass storage files.

Syntax

<sequence type> ::= <file or sequence> of <type>
<file or sequence> ::= file | sequence

Examples

- Variable Assembly: sequence of record
  part_name: array [1..6] of char;
  order_no: integer;
  drilled, punched, stamped, purchased:
    boolean
  End; for update_orders ;

- Type roster_entry = record
  name: array [1..20] of char;
  rank: 1..16; base_code: 1000..5000
  End;
  Variable roster: file of roster_entry;
  for assign_new_base ;
Semantics

All components of sequences must be of identical type and length. A sequence may not have sequence type or text type components. Furthermore, digital and analog types may not be combined as sequences.

2.1.2.6.3 Pointer Types

Variables of type pointer are "bound" to a particular type. That is, the contents of a pointer is used to indicate a second variable, and the second variable is required to be of a predetermined, specific type.

Syntax

<pointer type> ::= @ <type identifier>

Examples

- Type combination = record n, p: integer 
End ;
Variable comb_ptr: @ combination; for select_band;

- Type weather_station = record hi,lo: integer;
  rain: real End;

  Variable ws_ptr: @ weather_station;
  for record_temperature;

Semantics

The contents of a pointer may be altered, but the data element the pointer indicates is always of the same type.
2.1.2.7 Constant Declarations

In SSL, constant declarations may appear in the preamble of any subsystem and are used to communicate actual values or parameters to the detailed designer. Normally, a constant declaration would be used only for critical values for which the effects are to be isolated in the final code.

Syntax

<constant declaration> ::= <constant or constants>
<constant definition list>
<constant or constants> ::= constant | constants
<constant definition list> ::= <constant definition>
|<constant definition list>;<constant definition>
<constant definition> ::= <identifier> = <constant>
|<identifier> = <simple type>

Examples

- Constant a = 10.0 ; max_count = Integer;
- Constants Low = true;
  Tax_cut = 1..5 ;

Semantics

An identifier declared equal to a simple type indicates that the exact value is not known at the time of specification, but will be provided before implementation. An identifier used in a constant declaration may subsequently be used any place that a constant (of the same type) may be used.
2.1.2.8 Data References

Data elements may be referenced by variable name, by selected component, or pointer. A variable has components only if it is a record, digital signal, file, or array.

Syntax

<variable> ::= <entire variable>

|<component variable>
|<referenced variable>

2.1.2.8.1 Entire Variables

<entire variable> ::= <identifier>

Semantics

A reference to an entire variable includes all fields of a record or digital signal, all elements of an array, or all records of a file. If the data element is a simple, unstructured variable (integer, boolean, etc.) it may only be referenced as an entire variable.

2.1.2.8.2 Component Variables

Syntax

<component variable> ::= <indexed variable>

|<field designator>
|<file buffer>

<indexed variable> ::= <array variable>

[<expression list>]

2-32
<array variable> ::= <variable>

<expression list> ::= <expression>|<expression list>, <expression>

<field designator> ::= <record variable> . <field identifier>

<record variable> ::= <variable>

<file buffer> ::= <file variable> @

<file variable> ::= <variable>

Examples

Char_Array[15]
Inverse_Matrix[5, I, 16]
Employee.Name
Owner[15].Accessed_Value
Name_Record.Character[6]
Transaction_File@
Transaction_File@.Date
Transaction_File@.Date.Month

Semantics

Indexed variables have the conventional meaning. Field designators denote which field component of a record or digital signal type is to be selected. A file buffer variable designates the current active element of the sequence of elements that comprise the file.
Since arrays, files, and records can be combined in various ways (a record of records, file of arrays, array of records, etc.) a component variable can be arbitrarily complex. It is recommended that data structures be as limited in complexity as the problem permits.

2.1.2.8.3 Referenced Variables

Syntax

\[
\text{<referenced variable>} ::= \text{<pointer variable>} @ \\
\text{<pointer variable>} ::= \text{<variable>}
\]

Examples

\[
\text{Symbol\_Pointer} @ \\
\text{Student\_Name[6]} @ \\
\text{Assembly}.\text{Manufacturer}@ \\
\]

Semantics

The data structure denoted by the contents of the pointer variable is substituted for the referenced variable in expression evaluation.

2.1.2.9 Expressions and Assertions

Expressions arise in two contexts: subscripts of arrays and as terms within assertions. Assertions may appear in either variable declarations or module descriptions.
2.1.2.9.1 Arithmetic Expressions

Arithmetic expressions in SSL are similar to those in other high level languages. Results of expressions are single valued with type determined by the operation and the constituent operands.

Syntax

<arithmetic expression> ::= <term> | <sign> <term>
    | <arithmetic expression> <sign> <term>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<factor> ::= <primary> | <factor> ** <primary> | <set>

<primary> ::= <constant identifier> | <unsigned number> | <variable> | <function designator>
    | (<arithmetic expression> )

<set> ::= [ <element list> ]

<element list> ::= <empty> | <element> | <element list>, <element>

<element> ::= <expression> | <expression> .. <expression>

<multiplying operator> ::= * | /

<function designator> ::= <function identifier> (<expression list> )

<function identifier> ::= <identifier>
Examples

a + b
3.0 * sin ( r + 1.0)
2 * (ifix(c) + blank_common.icount)
name.field1
name.set + [joe, fred]

Semantics

Mixed mode expressions are prohibited with the exception of the exponentiation operator as indicated in Table 2-3. In Table 2-3, any operand of type integer may be replaced by an operand of type integer subrange. The symbol "dp" indicates double precision. The unary "+" may be used with any operand permitting a unary "-", but is semantically superfluous (i.e. + is the identity operation). If a type is not included in the operand type columns of Table 2-3 then its use with the designated operator is not permitted. Note, however, that integer and integer subrange are interchangable.

SSL does not contain intrinsically defined functions. All function identifiers are accepted, but it is suggested that those embodied in the proposed implementation language be adopted for each specification. Function types are not explicitly declared, but must be consistently used throughout the specification. In addition to the basic types (integer, real, etc.), the permissible function types include scalar and subranges of integers and scalars.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>V1 Type</th>
<th>V2 Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Arithmetic Negation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Integer</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>Real</td>
<td>Real</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>dp</td>
<td>dp</td>
<td></td>
<td>dp</td>
</tr>
<tr>
<td>+,-</td>
<td>Addition, Subtraction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Integer</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>Real</td>
<td>Real</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>dp</td>
<td>dp</td>
<td></td>
<td>dp</td>
</tr>
<tr>
<td>+,-</td>
<td>Set Union, Set Difference</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set</td>
<td>Set</td>
<td></td>
<td>Set</td>
</tr>
<tr>
<td>*</td>
<td>Set Intersection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set</td>
<td>Set</td>
<td></td>
<td>Set</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication, Division</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Integer</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>Real</td>
<td>Real</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>dp</td>
<td>dp</td>
<td></td>
<td>dp</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>Integer</td>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>Real</td>
<td>Real</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>Real</td>
<td>Real</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>dp</td>
<td>dp</td>
<td></td>
<td>dp</td>
</tr>
<tr>
<td></td>
<td>dp</td>
<td>dp</td>
<td></td>
<td>dp</td>
</tr>
</tbody>
</table>
2.1.2.9.2 Boolean Expressions

Combining arithmetic expressions with the boolean operations produces the expressions used in SSL assertions and array subscript lists.

Syntax

<expression> ::= <implication> | <expression> equ <implication>

<implication> ::= <boolean term> | <implication> implies <boolean term>

<boolean term> ::= <boolean factor> | <boolean term> or <boolean factor>

<boolean factor> ::= <boolean secondary> | <boolean factor> and <boolean secondary>

<boolean secondary> ::= <boolean primary> | !<boolean primary>

<boolean primary> ::= <logical value> | <arithmetic expression> | <relation> | (<assertion>)

<relation> ::= <arithmetic expression><relational operator><arithmetic expression>

<relational operator> ::= <|=|>|>=|<|=| in
Examples

Rate = 7.0
Value and Qual
a>b IMPLIES c>0.0
S =< t Equ p<t
Color in [red, green, yellow]
abs (buffer @.velocity) <16.0 and weight >= 14.0

Semantics

The arithmetic and boolean operators are grouped into hierarchial levels as exhibited in Table 2-4. Operations are performed in the order of highest hierarchial level first followed by equal hierarchial levels from left to right. This sequence may be overridden by parentheses, in which case the innermost operations are performed first. The meaning of the logical operators ¬ (not), and, or, implies, and equ (equivalent) is given in Table 2-5.

Table 2-6 depicts the required operand types for the boolean and relational operators. For set types, the symbols "[ ]" stand for the empty set. When comparing set types to scalars, the base type of the set must be the same as that of the scalars. The operators <, <=, =, >=, >,¬ = stand for less than, less than or equal, equal, greater than or equal, greater than, and not equal respectively. Relational operators (other than in ) may be used to compare arrays of equal length composed of characters, in which case they denote alphabetical ordering.
<table>
<thead>
<tr>
<th>Level</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Equ</td>
</tr>
<tr>
<td>2</td>
<td>Implies</td>
</tr>
<tr>
<td>3</td>
<td>Or</td>
</tr>
<tr>
<td>4</td>
<td>And</td>
</tr>
<tr>
<td>5</td>
<td>¬</td>
</tr>
<tr>
<td>6</td>
<td>&lt;, &lt;=, =, &gt;=, &gt;, ¬ =, In</td>
</tr>
<tr>
<td>7</td>
<td>+, −</td>
</tr>
<tr>
<td>8</td>
<td>*, /</td>
</tr>
<tr>
<td>9</td>
<td>**</td>
</tr>
<tr>
<td>b1</td>
<td>false</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>b2</td>
<td>false</td>
</tr>
<tr>
<td>¬ b1</td>
<td>true</td>
</tr>
<tr>
<td>b1 And b2</td>
<td>false</td>
</tr>
<tr>
<td>b1 Or b2</td>
<td>false</td>
</tr>
<tr>
<td>b1 Implies b2</td>
<td>true</td>
</tr>
<tr>
<td>b1 Equ b2</td>
<td>true</td>
</tr>
<tr>
<td>Operator</td>
<td>Operation</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>&lt;, &lt;, =,</td>
<td>Compare</td>
</tr>
<tr>
<td>&gt;=, &gt;, ≠</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>In</td>
<td>Set Inclusion</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Logical Inversion</td>
</tr>
<tr>
<td>And</td>
<td>Logical &quot;And&quot;</td>
</tr>
<tr>
<td>Or</td>
<td>Logical &quot;Or&quot;</td>
</tr>
<tr>
<td>Implies</td>
<td>Logical Implication</td>
</tr>
<tr>
<td>Equ</td>
<td>Logical Equivalence</td>
</tr>
</tbody>
</table>
2.1.2.9.3 Assertions

Assertions are conditions which may assume only true/false values. They are attached to variables at their point of declaration and to modules. Module assertions depict entry and exit data conditions.

Syntax

<assertion> ::= <expression><forall clause>
<forall clause> ::= <empty>| forall identifier = <set>

Examples

• $a[i] = 0.0$ forall $i = [1..n-1]$
  $b.c[j] = t[k]$ forall $j = [1,3,4..16]$ forall $k = [16..30]$

• big>small

• code = 1 implies (eof equ true)

Semantics

The scope of the identifier in the <forall clause> is the assertion in which it is used and must not overlap that of a local or global variable of the same name. Its type is assumed to be the base type of the set within the <forall clause>. The set must represent a finite number of elements and may not be empty.

The expression within the assertion may assume only the values true and false. If the <forall clause> is present, the expression is evaluated once for each unique value which the <forall identifier> can assume from the set.
2.1.2.10 Module Descriptions

Modules are basic system objects in an SSL system description. In using SSL, one identifies for each module:

- The module name
- Input and output data
- Conditions placed on data upon entry to and exit from the module
- Dependence of the module on environmental objects and other modules

The rule of correspondence between input and output data is not stated in SSL. Its statement is a function of detailed design.

Syntax

\[
\text{<module description>} ::= \text{<module statement>}; \\
\qquad \text{<module definition part> end} \\
\text{<module definition part>} ::= \text{<module definition statement>} | \text{<module definition part> ;} \\
\text{<module definition statement>} ::= \text{<assumes statement>} \\
| \text{<satisfies statement>} | \text{<fulfills statement>} \\
| \text{<accesses statement>} | \text{<modifies statement>} \\
| \text{<creates statement>} | \text{<uses statement>} \\
| \text{<receives statement>} | \text{<transmits statement>} \\
| \text{<executes statement>}
\]
2.1.2.10.1 Module Statement

The module statement is always the first statement of a module description. It identifies the module by name and declares the local variables (if any).

Syntax

<module statement> ::= <module or entry> <module identifier> <release variable group>

<module or entry> ::= module | entry

<release variable group> ::= <empty> | (<release variable list>)
<release variable list> ::= <release variable> | <release variable list> ; <release variable>

<release variable> ::= <variable> | <local variables>
<local variables> ::= <identifier list> : <simple type>
,module identifier> ::= <identifier>

Examples

- module matrix_multiply;
- entry push_stack (stack_item: stack_entry);
- module permutation (m, n: integer; elements: p_array);
Semantics

A module statement introduced by module can only be referenced from within the subsystem in which it is declared.

A module statement introduced by entry can be referenced only from subsystems other than the one in which it is declared.

Release variables occur both in module statements and virtual references within execute statements. Local variables within a release group serve strictly for communication between the module and those calling it. In this respect, they differ from global variables declared in the subsystem preamble which serve to communicate among modules having common requirement attributes. Local variable identifiers must be unique throughout a subsystem. Only the module statements introducing entry modules are permitted release variables which are not local variables. The variables of a release group for a module statement of an entry module must agree in type, number, and sequence to each virtual reference to it from other subsystems.

2.1.2.10.2 Assumes and Satisfies Statements

The assumes and satisfies statements specify truth conditions for data.

Syntax

<assumes statement> ::= <assume or assumes>
<assertion list>
<satisfies statement> ::= <satisfy or satisfies>
<assertion list>
<assume or assumes> ::= assume | assumes
<satisfy or satisfies> ::= satisfy | satisfies
Examples

- **Assume** \( a > 0.0 \);

- **Satisfies** `big_sister in family; count \( \neg \) = 0`;

**Semantics**

The assumes statement specifies data conditions that must be true upon module entry. The satisfies statement specifies data conditions that must be true upon module exit. Variables used in assertions must be either local variables in the release set or in the availability set pertinent to the module. (The availability set consists of those variables having requirement attributes which subsume all requirement attributes of the module.)

2.1.2.10.3 Fulfills Statement

The fulfills statement attaches requirement attributes to a module.

**Syntax**

```
<fulfills statement> ::= <fulfil or fulfills> <requirement attribute list>

<requirement attribute list> ::= <attribute identifier> | <requirement attribute list>, <attribute identifier>
<attribute identifier> ::= <transduction identifier> | <constraint identifier>
<fulfil or fulfills> ::= fulfil | fulfills
```
Examples

- `fulfills size_constraint, cluster;`
- `fulfil name_list ;`

Semantics

All modules must have at least one transduction identifier attached as a requirement attribute. All attribute identifiers must be declared in the preamble to the subsystem in which the module is declared.

2.1.2.10.4 Accesses Statement

The accesses statement is used to indicate which environmental objects (chiefly peripherals) are utilized by a module.

Syntax

```plaintext
<accesses statement> ::= <access or accesses>
  <environmental object list>

<access or accesses> ::= access | accesses

<environmental object list> ::= <environmental object identifier> | <environmental object list>,
                             <environmental object identifier>
<environmental object identifier> ::=<identifier>
```
Examples

- **Access** line_printer;
- **Accesses** real_time_clock, system_disk;

Semantics

For each environmental object there must be a unique identifier for which the scope is the entire specification.

2.1.2.10.5 Receives and Transmits Statements

The receives and transmits statements are used to indicate real time data activity such as is associated with telecommunications, analog, and digital signals.

Syntax

<receives statement> ::= <receive or receives>
  <from clause> | <receives statement> ; <from clause>

<from clause> ::= <entire variable list> from
  <environmental object identifier>

<transmits statement> ::= <transmit or transmits>
  <to clause> | <transmits statement> ;
  <to clause>

<to clause> ::= <entire variable list> to
  <environmental object identifier>
\[
\begin{align*}
\text{\langle receive or receives\rangle} & : = \text{receive} \mid \text{receives} \\
\text{\langle transmit or transmits\rangle} & : = \text{transmit} \mid \text{transmits} \\
\text{\langle entire variable list\rangle} & : = \text{\langle entire variable\rangle} \\
& \mid \text{\langle entire variable list\rangle, \langle entire variable\rangle}
\end{align*}
\]

Examples

- Receive weight from \text{strain\_gage_1};
- Transmit course\_correction to \text{ground\_control};

Semantics

The scope of the environmental object name is the entire specification. Note that components of structured variables may not be transmitted or received.

2.1.2.10.6 Creates, Modifies, and Uses Statements

The creates, modifies, and uses statements distinguish between input and output data variables. They may also indicate how the two are related in a manner short of a rule of correspondence. A complete rule of correspondence (algorithm) is a task of detailed design and not of SSL.

Syntax

- \text{\langle creates statement\rangle} : = \text{\langle create or creates\rangle}
  - \text{\langle create list\rangle}
- \text{\langle modifies statement\rangle} : = \text{\langle modify or modifies\rangle}
  - \text{\langle modify list\rangle}
<modify list> ::= <variable list><using clause>
| <modify list> ; <variable list><using clause>
<create list> ::= <entire variable list><using clause>
| <create list> ; <entire variable list><using clause>

<uses statement> ::= <use or uses> <variable list>

<create or creates> ::= create|creates

<modify or modifies> ::= modify|modifies

<use or uses> ::= use|uses

<using clause> ::= <empty>|using <variable list>

<variable list> ::= <variable>| <variable list>, <variable>

Examples

• create employee_array using name_file;

• modifies count, fica_rate using tax_table, salary_scales;

• modify pressure-weight [4], names [10]. initials;

• uses cluster@, transaction_file;
Semantics

The order of the variable references in any variable list has no significance.

The variables within a using clause or a uses statement are input variables. A variable may be both input and output. An input variable in a using clause indicates that its contents are instrumental in determining the final contents of the output variables within the same statement extending to the first semicolon on the left.

The presence of a variable in the output list of a creates statement indicates the first use (in a dynamic sense) of that variable. This does not mean, however, that the variable may not appear previously in the sequential listing of the SSL program. The implication of the creates statement is that all variables in the output list are first computed or initialized in the module being described. All variables declared in the subsystem preamble must appear as an output variable in exactly one creates statement within the subsystem unless it is a release variable of an entry module.

All variables appearing in a creates, modifies or uses statement (other than the output list of the creates statement) must be in the availability set for the module. A variable is in the availability set of a module if the transduction requirement attributes of the variable subsume all the transduction requirement attributes of the module.
2.1.2.10.7 Execute Statement

The execute statement designates modules which are called by the module being described. It may indicate that specific modules are called iteratively, conditionally, or both.

Syntax

<executes statement> ::= <execute or executes>
   <call list>|<executes statement>; <call list>

<call list> ::= <module reference list>|<module reference list> <call list tail>
   |<call list tail>

<call list tail> ::= <iteratively clause>
   |<conditionally clause>

<iteratively clause> ::= iteratively <module reference list> | iteratively <call list tail>

<conditionally clause> ::= conditionally <module reference list>

<execute or executes> ::= execute|executes

,module reference list> ::= <module reference>
   |<module reference list>, <module reference>

<module reference> ::= <concrete reference>
   |<virtual reference>
<concrete reference> ::= <module identifier>

<virtual reference> ::= <subsystem identifier>.<module identifier><release variable group>

Examples

- Execute matrix_multiply, cluster.group (pointer@);

- Execute iteratively suba, subb;
  conditionally subc, subd, sube;

- Executes sqrt; iteratively cos conditionally sin;

Semantics

The order of module identifiers in the module reference lists is not significant. The domain of either an iteratively or conditionally clause extends to the next semicolon. An iteratively clause may overlap another clause.

Presence of a module identifier in a iteratively clause connotates that it is called from within a loop. Presence in a conditionally clause connotates the module is not always called. If present in neither, the module is called unconditionally but not from within a loop.

A concrete reference is a call to a module within the same subsystem. A concrete reference may never be to an entry module. A virtual reference is a call to a module of a different subsystem and must always be to an entry module.
Within the release variable group, the local variable format must be used for variables never before defined. A variable may have been defined in the preamble to the subsystem or in the last module statement. The entry module to which the virtual reference refers must have the same release list with respect to number, order, and type of variables. All variable types used in a virtual reference release list must be either intrinsically defined (boolean, real, text, etc.) or global types.

2.1.2.11 Subsystem Descriptions

Subsystems are independent software units, each with its own requirement declaration. Subsystems may not share global variables but communicate via the release group variables of virtual references and entry modules. The only identifiers with scope greater than a single subsystem are global type identifiers, environment object identifiers, subsystem identifiers, and function identifiers.

Syntax

<subsystem description> ::= <subsystem preamble> ;
   <module description list> end

<module description list> ::= <module description>
   | <module description list>; <module description>

<subsystem preamble> ::= <preamble declaration list>
   | subsystem <subsystem identifier> ; <preamble declaration list>
<subsystem identifier> ::= <identifier>

<preamble declaration list> ::= <preamble declaration>
   |<preamble declaration list> ; <preamble declaration>

<preamble declaration> ::= <requirement declaration>
   |<type declaration>|<variable declaration>
   |<constant declaration>

<subsystem description list> ::= <subsystem description>
   |<subsystem description list> ; <subsystem description>

<specification> ::= <subsystem description list> end

Example

* Requirement transduction sort_descend; input n,
  sort_array; output sort_array end;

Variable sort_array: array \([1..1000]\) of real;
  for sort_descend;
  subject to sort_array[i] > 0.0 for all i =
  \([1..n-1]\);
  n:1..1000; for sort_descend;

Module sort;

  fulfills sort_descend;
  accesses card_reader, line_printer;
  creates n, sort_array;
  modifies sort_array using n, sort_array;
satisfies \( \text{sort}_i \geq \text{sort}_{i+1} \) 
\text{forall } i = [1..n-1] 

End
End
End;

Semantics

Each subsystem must have a requirement declaration that contains at least one transduction identifier and one output variable. There must also be at least one module description. The first subsystem declared (called the "main" subsystem) does not have a subsystem identifier; all others must have a unique identifier. The scope of the subsystem identifier is the entire specification.

The nonterminal symbol \(<\text{specification}>\) is the distinguished symbol of the SSL grammar.
2.1.3 Example

The example of this section was selected to demonstrate both the descriptive level of SSL and as many language elements as possible. The requirement of the problem may be stated as follows [8]:

"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."

To complete the problem statement, several assumptions are necessary. The following alternatives were selected for the purpose of this exposition:
The character stream from which the telegrams are constructed resides on a drum having fixed length records; the record length itself is left as an implementation option.

• The chargeable word count is the value to be printed and overlength words count as one word.

• If a physical end of file is encountered before the logical end of the data stream, an error message and the partial telegram is printed.

The software is organized into four modules as indicated by Figure 2-2. The purpose of each module is given in Table 2-7. Figure 2-3 contains the SSL description of the telegram processor. The right margin of the statement listing contains reference notes to subsections containing detailed descriptions of the language elements used.

A careful examination of Figure 2-3 will indicate an interesting application of the subsystem capability. The subroutines GET_CHAR and FILL_BUFFER occupy a separate subsystem with the sole purpose of handling file I/O. The characteristics of the device on which the telegrams are stored are encapsulated within these two modules.
Figure 2-2 Module Structure Chart for Example

NOTES:

“A” CALLS “B” CYCLICALLY

“A” CALLS “B” CONDITIONALLY

“A” USES SYSTEM SERVICE “B”
<table>
<thead>
<tr>
<th>MODULE</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_TELEGRAM</td>
<td>Collects words belonging to each telegram and prints them in a neat manner along with the chargeable word count.</td>
</tr>
<tr>
<td>GET_WORD</td>
<td>Collects characters into words and prints error messages denoting over-length word or physical record end of file.</td>
</tr>
<tr>
<td>GET_CHAR</td>
<td>Returns the next character in the telegram file.</td>
</tr>
<tr>
<td>FILL_BUFFER</td>
<td>Enters the next physical record from the drum into the character buffer.</td>
</tr>
</tbody>
</table>
/* beginning of main subsystem preamble */

require

transduction

\begin{itemize}
\item collect_in
\item print;
\item output
\item telegram, charge_count
\end{itemize}

end;

variable telegram: text;

charge_count: integer;

\begin{itemize}
\item for print;
\item subject to charge_count > 0
\end{itemize}

word_count: integer;

\begin{itemize}
\item for print;
\item subject to word_count > charge_count
\end{itemize}

word: array [1..12] of char;

\begin{itemize}
\item for print;
\item subject to word_count = charge_count
\end{itemize}

eof_flag: boolean;

\begin{itemize}
\item for print
\end{itemize}

end: /* end of main subsystem preamble */

/* main routine to collect words and */
/* print telegram with chargeable word count*/

module get_grammar:

\begin{itemize}
\item fulfills print;
\item creates telegram, charge_count using word;
\item creates word_count;
\item modifies word_count;
\item uses eof_flag;
\item accesses line_printer
\end{itemize}

executes cyclically get_word:

\begin{itemize}
\item satisfies
\item eof_flag or word_count = 0
\end{itemize}

end:

/* subroutine to collect characters into */
/* words */

module get_word:

\begin{itemize}
\item fulfills collect;
\item executes cyclically l.o.get_char(char, char, eof_flag);
\item creates word, eof_flag;
\item accesses line_printer /*prints error messages */
\end{itemize}

end: /* end of main subsystem */

Figure 2-3 SSL Description for Example
/* beginning of i_o subsystem preamble */

subsystem i_o:

requirement

input character_file;

transductions

read in separate;

output a_char, eof_flag

end;

/* parameterize record length */

constant record_length = integer;

type character_record = array [1..record_length] of char;

variable character_file: sequence of character_record:

for read;

buffer: character_record;

for separate;

a_char: char;

for separate;

char_index: integer;

for separate;

eof_flag: boolean;

for separate;

end; /* end of subsystem preamble */

/* subroutine to fetch next */

/* character from file */

gt_char (a_char; eof_flag) :

fulfills separate;

executes conditionally fill_buffer;

modifies char_index;

creates a_char using buffer [char_index], eof_flag;

creates character_file, char_index;

satisfies eof_flag implies a_char = buffer [char_index];

end;

/* subroutine to fetch next physical */

/* record from character file */

module fill_buffer:

fulfills read;

assumes char_index = record_length;

accurate as disk;

creates buffer, eof_flag using character_file;

satisfies:

eof_flag implies buffer = character_file;

end

end; /* end of subsystem */
end; /* end of specification */

Figure 2-3 SSL Description for Example (continued)
2.2 SEMANTIC DESCRIPTION OF SSL

Our semantic description of SSL is in terms of sets and set functions grouped into n-tuples. The initial construct is the Vertex Correlation Tuple which forms the basis for definition of a single subsystem of the software structure. This tuple consists of a set of nodes, some of which have module names attached. Virtual nodes do not represent program modules, but evocations of entry nodes of separate subsystems. Modules are represented by concrete nodes. Special types of concrete nodes, called entry nodes, are a distinctive characteristic of subsystems other than the main subsystem and have special properties. First, they must have attached module names, so they can be referenced by other subsystems. Second, (as we shall see in the Requirements Graph) their predecessor may only be the root node of the subsystem which is not permitted an attached module name.

Vertex Correlation Tuple

\[ \chi = (N,C,V,G,Z,M,\text{mod}) \]

where

(1) \( N \) = a finite set of labeled nodes
(2) \( C \) = a subset of \( N \) called concrete nodes
(3) \( V \) = a subset of \( N \) called virtual nodes; \( V \cap \text{C} = \emptyset \)
(4) \( G \) = a subset of \( C \) called entry nodes
(5) \( Z \) = a distinguished node called the root
(6) \( M \) = a finite set of module names
(7) \( \text{mod} : N \rightarrow M \): a module name mapping function

(a) If \( n \in C \), then cardinality (\( \text{mod} (n) \)) = 0 or 1.
(b) If \( n \in G \), then \( \text{mod} (n) \neq \emptyset \); entry nodes are required to have an attached module name.
(c) If \( n \in V \), then \( \text{mod} (n) \neq \emptyset \); virtual nodes are required to have an attached module name.
(d) If \( G \neq \emptyset \), then \( \text{mod} (Z) = \emptyset \); the root nodes of subsystems other than the main subsystem are not permitted to have attached module names.
The module mapping function (mod) implies:

1. Not all concrete nodes have an associated module name.
2. Module names associated with virtual nodes are not necessarily unique; each evocation of an entry node is represented by a distinct virtual node.

The nodes in \( \chi \) will be collected into a weakly-connected digraph based on a mapping of nodes to subsets of requirements. (For reasons of clarity and stand-alone interpretation, we are using "requirements" in the sense we used "transductions" in the previous part.) To accomplish this, the requirements are partially ordered by implication. If \( R \) is a set of requirements, \( r_1, r_2 \in R \), we write \( r_1 \leq r_2 \) (read "\( r_1 \) is implied by \( r_2 \)"). For example, consider a line editor. The requirements are:

\[
\text{edit, search_file, delete_line, add_line}
\]

The implications between these requirements are as follows:

1. delete_line \( \leq \) edit
2. add_line \( \leq \) edit
3. search_file \( \leq \) edit
4. Search_file \( \leq \) delete_line
5. search_file \( \leq \) add_line
6. delete_line \( \neq \) add_line, add_line \( \neq \) delete_line:
   i.e., \{delete_line, add_line\} is unordered.

With nodes mapped onto subsets of a set of partially ordered requirements, it is possible to define the predecessor relationship between nodes by a function. The function (called simply the predecessor function) requires that \( r_2 \leq r_1 \) in order for \( n_1 \) to be a predecessor of \( n_2 \) where \( r_1 \) is a certain requirement attribute of \( n_1 \) depending on \( r_2 \).
The Requirements Graph

\[ \Pi = (\chi, R, B, \text{req}, \text{pr}) \]

where

1. \( \chi \) = A Vertex Correlation Tuple
2. \( R \) = a finite set of labeled requirements, partially ordered by implication
3. \( B \) = a base set of \( R \); if \( r_1 \in B \), then \( \not\in R \)
   \[ r_2 \in R, \ r_1 \neq r_2 \] such that \( r_1 < r_2 \)
4. \( \text{req}: \mathbb{N} \rightarrow 2^R \)
   = the requirements mapping set function; a subset of \( R \) is associated with each node
5. \( \text{pr}: \mathbb{N} \rightarrow 2^N \)
   = the predecessor set function
   a. \( \text{pr}(Z) = \emptyset \); if \( n \neq Z \) then \( \text{pr}(n) \neq \emptyset \)
   b. If \( n_1 \in \text{pr}(n_2) \) then we define \( (n_1, n_2) \) to be adjacent in a digraph sense
   c. If \( r_2 \in \text{req}(n_2) \), \( n_1 \in \text{pr}(n_2) \) then there exists \( r_1 \in \text{req}(n_1) \) such that \( r_2 \leq r_1 \)
   d. If \( n_1, n_2 \in \text{pr}(n_3) \), \( r_1 \in \text{req}(n_1) \), \( r_2 \in \text{req}(n_2) \) then \( r_1 \neq r_2 \), \( r_2 \neq r_1 \), i.e., the requirement attributes if taken one each from the predecessors of a node are unordered
   e. If \( n_1 \in \text{pr}(n_2) \) then \( n_1 \in C \); i.e., virtual nodes may not have successors
   f. If \( n_3 \in V \), \( n_1 \in \text{pr}(n_3) \), \( n_2 \in \text{pr}(n_3) \) then \( n_1 = n_2 \neq \emptyset \); i.e., virtual nodes have exactly one predecessor
   g. If \( n_1 \in \text{pr}(n_2) \), \( n_2 \in V \) then \( \text{req}(n_1) = \text{req}(n_2) \); i.e.,
virtual nodes have the same requirement attributes as their predecessor

(h) If n \in G then pr(n) = \{Z\}

In addition to modules at graphical nodes, the other resources available to a software system are environment objects and data objects. Data objects are created at one and only one point within the system for use elsewhere; environment objects are not "created" but may be used or "accessed" at various points within the structure. A further distinction is that data objects, but not environment objects, have requirement attributes. The requirement attributes of data objects are not necessarily the same as those of the creating node. These attributes delimit where within a structure a data object may be used.

Object Distribution Graph

\[ \Lambda = (\pi, Q, E, cr, drq, acc) \]

where

(1) \( \pi \) = A Requirements Graph

(2) \( Q = DUL \) a finite set of labeled data objects;
    \( DUL = \phi \); if \( d \in D \) then \( d \) is called a global data object; if \( d \in L \) then \( d \) is called a local data object

(3) \( E \) = a finite set of labeled environment objects

(4) \( cr : C \rightarrow 2^D \) = object creation set function
    (a) If \( d \in D \), \( d \in cr(n_1), d \in cr(n_2) \) then \( n_1 = n_2 \); i.e., an object is created at one and only one node
    (b) If \( d \in D \) then \( d \in cr(n) \) for some \( n \); i.e., all data objects are created within the system

(5) \( drq : D \rightarrow 2^R \) = data requirements set function

(6) \( acc : C \rightarrow 2^E \) = environment access set function; \( e \in acc(n) \) if \( e \in E \) and \( e \) is accessed at concrete node \( n \)
Thus far, the constituents of a software structure that have been developed are:

1. Nodes with attached modules
2. Node interconnections based on requirements ordering
3. Data objects created and environment objects accessed.

In order to complete the interconnection graph, it is necessary to depict the utilization of data objects. A global data object is assigned requirement attributes via the drq function and those attributes are compared to those of potential user nodes. If n is a concrete node then d, a data object, is available for use only if for each \( r_1 \in \text{req}(n) \) there exists \( r_2 \in \text{drq}(d) \) such that \( r_1 \leq r_2 \) or d is a member of the release set of n or an immediate successor of n. In effect, data object requirement attributes place an upper bound on the generality of modules permitted use as well as limit the object's scope to specific segments of the graph.

**Subsystem Structure Graph**

\[ \Sigma = (\Delta, \text{rel}, \text{av}, \text{use}) \]

where

1. \( \Delta = \) an Object Distribution Graph
2. \( \text{rel}:(N,N) \rightarrow 2^Q = \) release set function;
   if \( \text{rel}(n_1,n_2) \neq \emptyset \) then \( n_1 \in \text{pr}(n_2) \); i.e., \( n_1 \) and \( n_2 \) must be adjacent in a digraph sense if \( n_1 \) may release objects to \( n_2 \)
3. \( \text{av}:N \rightarrow 2^Q = \) availability set function:
   a) If \( d \in D, n \in C, \text{av}(n), r_1 \in \text{req}(n) \) then \( r_1 \leq r_2 \) for some \( r_2 \in \text{drq}(d) \); i.e., each requirement attribute of a concrete node must be subsumed by at least one of the requirement attributes of the global data object.
(b) If \( d \in L, n_1 \in N, d \in \text{av}(n_1) \) then either \( d \in \text{rel}(n_1, n_2) \) or \( d \in \text{rel}(n_2, n_1) \); i.e., local data objects may be shared only by pairs of nodes that are adjacent in a digraph sense.

(c) If \( d \in \text{av}(n), n \in V \) then \( d \in \text{rel}(n_1, n) \) where \( \text{pr}(n) = \{n_1\} \); i.e., a data object available at a virtual node must have been released by its parent.

(d) If \( d \in \text{rel}(c, v) \) then \( d \in \text{av}(c) \) where \( c \in C, d \in \mathcal{Q} \), i.e., concrete nodes may release to virtual nodes only those data objects which are elements of the availability set of the concrete node.

(4) \( \text{use:} N \rightarrow 2^D \) = usage set function; \( d \in \text{use}(n) \) if \( d \in \text{av}(n) \) and d "is used" at n as determined by the designer.

The definition of the System Structure Graph is the final step in describing the semantics of SSL. The ultimate goal is an automated version of SSL which will:

- Permit partial system definition with consistency checking at the appropriate level of detail, and
- Participate in the design by supplying derived information and indicating the next logical steps in an incremental development.

Accomplishing this will depend on discerning higher order relationships among the functions describing SSL. This involves identifying theorems and algorithms that derive the set defined by one function through the sets defined by other functions. A subsequent section represents our efforts in this direction.
2.3 OVERVIEW OF THE SSL TRANSLATOR

In this section we give the formal software requirement for the translator, the functional design, and notes on detailed design. Significantly more detailed information exists on the functional design, including a description in the SSL language, than that presented here. The format and notation used for the functional design description in this report is more conventional.

The notes on detailed design concentrate on the more critical semantic and structure analysis phase. Included are:

- A method for determining the legitimacy of data/module interconnections
- A method for ordering modules in a manner that facilitates module interconnection and recursive analysis
- A method for constructing a matrix over which module interconnection and recursive analysis can be performed
- Rules for determining which modules are potentially recursive.

2.3.1 The Formal Software Requirement

In our terminology, the software requirement is a detailed declaration of the actions to be performed by a single software package. It includes what is to be done as well as initial estimates of "when" and "how well" various actions are to be performed. The constituent parts of the software requirement decomposition are:

Direction - A general statement of the boundaries of the problem

Input - Data or documents available to the software system or subsystem from external sources
Transductions - A list of processing steps to be performed, each of which translate a stimulus into a response

Output - Data or documents produced by the software for external purposes

Constraints - A list of capacities, design objectives, or resources to be observed

Preconceptions - A list of specific design alternatives to be observed

Implications - A binary relation existing between certain pairs of transductions, indicating which are substeps of others.

Of the seven divisions, the constraints and preconceptions are optional. If all transductions are independent, the implications may also be omitted.

The level of detail required for the SRD is generally greater than could be expected of a casual user of data processing services. For this reason, it is expected that the SRD will be completed by an experienced computer specialist after reviewing the user requirements. Completing the SRD is the beginning step of functional design.

Figure 2-4 depicts the SRD for the SSL translator. There are three major segments within the transductions: program analysis, structure analysis, and report summarization. The implication relations between transductions is represented in Figure 2-4 by indentation within the transduction division rather than within a separate implications division.

The only system (or requirement) level input is the SSL source language. System output consists of eight items. In addition to the source listing, syntax and semantic errors, these consist of:
Module concordance - An alphabetical list of all modules; for each module the following information is given: 1) modules it calls, 2) modules which call it, 3) variables referenced, 4) environmental objects referenced, and 5) requirement attributes

Variable concordance - An alphabetical list of all variables; for each variable the following information is given: 1) requirement attributes and 2) modules which reference it

Requirement concordance - An alphabetical list of all requirement transductions and constraints; for each, the following information is given: 1) modules to which attached as an attribute, 2) variables to which attached as an attribute, 3) (for transductions) the transductions immediately above and below it in a partially ordered sense

Summary - A summarization of the number of modules, variables, errors, etc. by subsystem

Index - A cross reference guide to facilitate access to parts of the SSL generated report
DIRECTION

Implement an automated translator for Software Specification Language (SSL).

INPUT

SOURCE: A set of logically connected SSL statements.

TRANSDUCTIONS

TA_PROG_ANALYSIS: Analyze the SSL program
TB_SYNTAX: Analyze the program syntax
TC_TOKEN: Reduce the next lexical token
   TD_CARD: Read and print the next card image
TC_SYN_ERR: Perform syntax error recovery procedures
TC_TABLES: Construct dynamic tables
   TD_REQ_ST: Process requirement statement
   TD_CONS_ST: Process constant statement
   TD_FUL_ST: Process fulfills statement
   TD_TYPE_ST: Process type statement
   TE_RECORD: Process record and digital forms

Figure 2-4. SRD for SSL Translator
TF ARRAYS: Process array forms
TF SUBRANGE: Process subrange forms
TF SCALARS: Process scalar forms
TF BASIC: Process basis type forms
TF POINTER: Process pointer forms
TE FILE: Process file and sequence forms

TF ARRAYS, TF_SUBRANGE, TF_SCALARS, TF_BASIC, TF_POINTER

TD_VAR_ST: Process variable statement
TE FOR: Process for clause
TE_POLISH: Analyze a polish string
TE_RECORD, TE_FILE

TD_MOD_ENT_ST: Process module and entry statements
TF_BASIC
TE_RELEASE

TD_SUBSYS_ST: Process subsystem statement
TD_EXEC_ST: Process executes statement
TE_RELEASE: Process release form
TF_BASIC

TD_USES_ST: Process uses statement
TE_DATA_LIST: Add items to data list
TD_MOD_CR_ST: Process modifies or creates statement
TE_DATA_LIST
TD_XMIT_RX_ST: Process transmit or receives statement
TE_DATA_LIST
TD_ASUM_SAT_ST: Process assumes and satisfies statements
TE_POLISH

Figure 2-4. SRD for SSL Translator (continued)
TA-STRUC ANALYSIS: Analyze the software decomposition structure

TB ELEMENTS: Ascertain all elements are consistent within themselves
- TC_SUB_DEF: Evaluate all subsystem definitions
- TC_MOD_DEF: Evaluate all module definitions
- TC_VAR_DEF: Evaluate all variable definitions
- TC_TYPE_DEF: Evaluate all type definitions
- TC_REQ_DEF: Evaluate all requirement definitions

TB SETS: Construct and evaluate all interelement relationships
- TC_REQ VS REQ: Ascertain requirement definitions are consistent
  - TD_REQ_CON: Construct requirement concordance lists
    - TE_MDR_MATRIX: Construct a module/data/requirement matrix
    - TE_NEXT_REQ: Find next alphabetically ordered requirement
    - TE_MOD_ATT: Construct list of modules to which requirement attached
    - TE_SUP_ATT: Construct list of superstep transductions

TC MOD VS REQ: Ascertain module hierarchy consistent with requirements
- TD_MOD_VECT: Construct a module requirement vector
- TD_CLOS_VECT: Perform closure over a requirement vector

TC MOD VS DAT: Ascertain modules use data consistent with requirements
- TD_AD_VECT: Construct a variable availability vector
- TD_MOD_VECT, TD_CLOS_VECT
- TD_DAT_CON: Construct data concordance list
  - TE_DAT_MATRIX: Construct data concordance list
  - TE_NEXT_DAT: Find next alphabetically ordered variable
  - TEREFER: Construct list of modules reverencing data object

Figure 2-4. SRD for SSL Translator (Continued)
TC_MOD_VS_REL: Ascertain module release sets are consistent
TC_MOD_VI_MOD: Ascertain module calling hierarchy consistent
TD_ORDER: Order modules by forward paths
TD_MATRIX: Construct D\(^+\) Matrix
TD_RECURSIVE: Perform recursive analysis
TE_POT_RECUR: Mark recursive modules
TE_LAT_HEAD: Mark latch and head modules
TD_INV_HIER: Construct inverse hierarchial list
TE_INV_MOD: Find next inverse call module
TC_REPORT_GEN: Generate report
TD_REQ_REPORT: Print requirement concordance
TD_DAT_REPORT: Print variable concordance
TD_MOD_REPORT: Print module concordance
TA_REPORT: Summaries and index report
TB_SUMMARY: Print summary of software report
TB_TOC: Print table of contents for report
TC_MOD_LIST: Print modules in alphabetical order
TD_NAME_PAGE: Print a single name and page number
TC_DAT_LIST: Print variables in alphabetical order
TD_NAME_PAGE
TC_REQ_LIST: Print requirements in alphabetical order
TD_NAME_PAGE

OUTPUT
SOURCE_LISTING: A line printer listing of the original SSL source
SYN_ERRORS: SSL syntax error diagnostics interleaved with the source listing

Figure 2-4. SRD for SSL Translator (continued)
SEM_ERRORS: A printed summary of all semantic incongruities
HIER_LIST: A list (alphabetical) of all modules which includes modules referenced, inverse hierarchy, data referenced, environment objects referenced, and requirement attributes.
DATA_LIST: A list (alphabetical) of all variables accompanied by the names of modules which use them and requirement attributes assigned.
REQ_LIST: A list (alphabetical) of all requirements cross referenced with modules, variables, and other requirements
SW_SUMMARY: A summarization of the software which includes counts for modules, variables, errors, etc.
INDEX_LIST: A cross reference guide for facilitating access to parts of the SSL generated report

IMPLICATIONS

(Implications are represented by indentation within the transductions substation.)

CONSTRAINTS

LANGUAGE: ANSI FORTRAN IV will be used to implement the translator
HOST_MACHINE: The translator will be written in a manner amenable to transportability; the host machines shall at least include the IBM S360/65 and Univac 1108.

Figure 2-4. SRD for SSL Translator (continued)
2.3.2 Functional Design Overview

The module decomposition follows closely the requirement decomposition of Figure 2-4. There are three phases. The first phase analyzes the SSL source input and constructs a hierarchical file containing all object attributes and interrelationships. The second phase analyzes the information within the hierarchical file for semantic errors, then generates a report on the software organization. The third phase prints a summary and generates an index to the report.

Figure 2-5 depicts a high level view of the software organization. The three phases depicted are executed once consecutively to produce the report from the SSL source program. The block labeled "SSL Translator Main Program" is the control program. The actions performed by each of the phases is discussed in the paragraphs below.

The first phase (controlled by the block labeled "Syntax Analyzer" in Figure 2-5) is the source program analysis phase. Its function is to read the source program and construct the file used in subsequent phases. The purposes of the principal blocks are as follows:

- Syntax Analyzer - The subroutines of this block parse input source statements, emit syntax diagnostics and pass the parse trees (polish notation) to the semantic analyzer
- Lexical Analyzer - The subroutines of this block read and print the source statements; from the source statements, the tokens (numbers, reserved words, delimeters, and identifiers) are collected and returned to the syntax analyzer
- Semantic Analyzer - The subroutines of this section collect the parse trees and, when one is sufficiently complete, passes it to lower level
Figure 2-5. Block Diagram for SSL Translator
subroutines; the lower level subroutines called by the semantic analyzer are differentiated by statement type and each constructs a specific part of the hierarchical file.

The second phase (controlled by the block labeled "Structure Analyzer" in Figure 2-5) is where the software interconnections are examined for consistency. Each element (subsystem, module, variable, etc.) is examined first for internal or self-consistency. Self-consistency includes each element being defined, referenced, and having all attributes assigned. After element analysis, set analysis takes place. Set analysis involves testing the consistency of all interelement references. This task is performed in two parts. The first part constructs the interelement relationships in the form of a set of boolean matrices. Semantic error analysis is carried out from the matrices and their representation is converted to a list structure. This list structure is used in the second part of set analysis to generate the software structure report.

The role of the second phase can be further clarified by examining the function of each of the blocks in Figure 2-5.

- **Structure Analyzer** - Control routine for phase 2
- **Element Analyzer** - On a subsystem by subsystem basis, the subroutines of this block examine the various elements that comprize the subsystem for intraelement consistency; the lower level subroutines that the element analyzer calls are differentiated on the basis of element type.
- **Set Analyzer** - Control routine for set analysis.
Set Construction - The subroutines of this block construct a data base containing interelement dependencies, and analyze the dependences for semantic errors; the lower level subroutines called by set construction are differentiated on the basis of element type pairs (modules vs. requirements, subsystems vs. modules, etc.).

Report Generation - The subroutines of this block generate the software structure report; the lower level subroutines called by report generation are differentiated on the basis of the various sections of the report (modules, variables, etc.).

The third phase (controlled by the block labeled "Post-Analysis Phase" in Figure 2-5) summarizes and prints an index for the report previously generated. The major blocks of this phase may be summarized as follows:

- Post-Analysis Phase - Control routine for the third phase.
- Summary Generator - Prints a summary of the software structure such as the number of modules, number of variables, and number of errors per subsystem.
- Index Generator - The subroutines of this block generate a index list for each module, variable, etc., including which page of the source listing and concordance listings it occurred.
- Record Formatter - Prints a single line of the index which includes a name with page numbers.

2.3.3 Detailed Design Notes

Phase 1 of the translator is a standard parser combined with data structure synthesis routines. Phase 3 is simply an output editor. The crucial subset is phase two in which the
interelement relations are analyzed semantically. The purpose of this section is to expound both algorithmically and theoretically on some of these relationships.

2.3.3.1 Assessing Data Availability

Recall that requirement attributes are attached to both modules and data objects as a means of providing requirements traceability. A secondary effect of requirement attributes is that they limit the availability of data objects; i.e., a data object may not be used at a concrete node unless all the requirements attributes of its module are equal to or implied by requirement attributes of the data object. Therefore, one might expect a close relationship between the requirement attribute functions (req, drq) and the availability function (av). This relationship is expounded below.

Let \( R = \{ r_1, r_2, \ldots, r_k \} \) be a set of requirements. Let \( P(d) = (p_1, p_2, \ldots, p_k) \) be a data object requirement vector where:

\[
P_j = \begin{cases} 
1 & \text{if } r_j \in \text{drq}(d) \text{ or } r_j < r_i \text{ where } r_i \in \text{drq}(d) \\
0 & \text{otherwise.}
\end{cases}
\]

Let \( Q(n) = (q_1, q_2, \ldots, q_k) \) be a node requirement vector where:

\[
q_j = \begin{cases} 
1 & \text{if } r_j \in \text{req}(n), n \in C \\
0 & \text{otherwise.}
\end{cases}
\]

Theorem

For any \( n \in C, d \in \text{av}(n) \) if and only if \( \sum_{i=1}^k q_i = P(d) \cdot Q(n) \).

\textbf{Proof} (necessary) assume \( n \in C \) and \( d \in \text{av}(n) \). Then, by definition of the av function, for any \( r_1 \in \text{req}(n) \) there exists an \( r_2 \) in \( \text{drq}(d) \) such that \( r_2 > r_1 \). So \( q_j = 1 \Rightarrow r_j \in \text{req}(n) \Rightarrow r_j < r_k \) for some \( r_k \in \text{drq}(d) \Rightarrow p_j = 1; \ldots, k \).

\( \sum_{i=1}^k q_i = P(d) \cdot Q(n). \)

The sufficiency part of the proof is carried out similarly.

\textbf{END OF PROOF}
Note that the theorem applies only to concrete nodes since virtual nodes, unlike concrete nodes, depend additionally on the release function (rel).

2.3.3.2 Assessing Consistency of Data Usage

Data object usage at a module is dependent upon its availability at that module. However, the two sets are derived from different perspectives and require cross checking. Anomalies should prompt the designer to re-think the requirement attributes assigned objects, a healthy exercist.

Let's begin by recalling that a data object, $d$, is not eligible to be used at a module, $n$, unless $d \in \text{av}(n)$. Let $D = \{d_1, d_2, \ldots, d_m\}$ be the set of data objects within the system. For some node $n$, let $U(n) = [u_1, u_2, \ldots, u_m]$, the usage vector, be a vector where

$$u_i = \begin{cases} 
1 & \text{if } d_i \in \text{use}(n), \ n \in \text{C where } \text{C is concrete node} \\
0 & \text{otherwise.} 
\end{cases}$$

Let $W(n) = [w_1, w_2, \ldots, w_m]$, the candidate vector, be a vector where

$$w_i = \begin{cases} 
1 & \text{if } d_i \in \text{av}(n), \ n \in \text{C}; \ i = 1, 2, \ldots, m \\
0 & \text{otherwise.} 
\end{cases}$$

Given $U(n)$ and $W(n)$ the usage set assigned to node $n$ is legitimate only if

$$u_i = u_i \cdot w_i ; \ i = 1, 2, \ldots, m$$

Furthermore, if the set is illegal, the culpable object is identified by the element of $U(n)$ for which the above test fails.
2.3.3.3 Ordering Modules for Analysis

The predecessor relation defined in SSL semantics only partially orders the nodes (modules). There exists more than one total order that adequately reflects the partially ordered properties of any nontrivial module set. It is necessary to select a (total) ordering algorithm in order to perform analysis in a deterministic manner.

The algorithm preserves the natural partial order of the modules. Definition of the following terms are necessary:

- $e$: A unique module called the root or entry module (of a subsystem)
- $\text{ord}(m)$: The order number of module $m$; initially zero for all modules
- $\text{pr}(m)$: The predecessor function for module $m$ as defined in the SSL semantics; $\text{pr}^{-1}(m)$ is the successor function
- $n$: The set of modules
- $\#S$: The cardinality of the set $S$

The algorithm is as follows:

1. Let $S_1 = e$; set ord $(e) = 1$
2. Let $p = 1$, $k = 1$
3. If $m_p \in S_k$ and there exists $m_q \in N \cap (\text{pr}^{-1}(m_p) - S_k)$, then
   a. For each $m_r$ such that $p < \text{ord}(m_r) \leq \#S_k$, increase ord $(m_r)$ by 1
   b. Define $\text{ord}(m_q) = p + 1$, $S_{k+1} = S_k \cup \{m_q\}$
   c. Increase $p$ and $k$ each by 1
   d. Return to step (3).
4. If $p > 1$, decrease $p$ by 1 and return to step (3); otherwise stop.
This algorithm assigns an order number to each module. Furthermore, the order numbers increase monotonically along forward (non-recursive) calling paths. Figure 2-6 illustrates the order number assignments for an arbitrary block diagram. Note that more than one order assignment combination fulfills the criterion of increasing order numbers along forward paths.

2.3.3.4 Construction and Closure of Dependency Matrices

A dependency matrix (or adjacency matrix) is an n x n boolean matrix where there are n modules. Rows and columns must be ordered equivalently to the order numbers acquired from the algorithm given above. The elements of the dependency matrix, D, are as follows:

\[
\begin{align*}
\text{d}_{ij} &= \begin{cases} 
\text{true} & \text{if module of order i references module of order j} \\
\text{false} & \text{otherwise}
\end{cases}
\end{align*}
\]

Once constructed, the rows of D yield "called" lists and the columns "called by" lists.

**Closure of D** - In the closure of the matrix D (denoted D⁺) an element d_{ij} will be true if there exists a sequence in D, d_{ip}, d_{pq}, d_{qr}, ..., d_{sj}, all of which are true.

One way of deriving D⁺ is by raising D to the nth power. The algorithm given here is much more efficient.

```plaintext
D: array [1..n, 1..n] of boolean;
i, j, k: integer;
for j = 1 to n do
begin
  for i = 1 to n do
    if D[i, j] and i ≠ j then
      for k = 1 to n do
        D[i, k] = D[i, k] or D[j, k]
  end
end;
```

2-85
Figure 2-6. A Module Ordering Example
2.3.3.5 Recursive Analysis Using Dependency Matrices

It has already been noted that "called" and "called by" lists are explicitly represented in D. What remains is the derivation of the recursive information required for the module concordance of the previous section. Specifically, the determination of head modules, latch modules, and potentially recursive modules. (A latch module, in the context of recursive analysis, is one that makes a recursive call. A head module is one to which a recursive call is made).

Theorem

If $d_{ij} = true$ and $j < i$ then

- $i$ is the order no. of a latch module and $j$ is the order no. of a head module of a recursive subsystem.

Proof

Assume $\text{ord}(m_1) = i$ and $\text{ord}(m_2) = j$. Since $d_{ij} = true$, $m_1$ calls $m_2$. If there exists a forward calling path from $m_2$ to $m_1$ then the call of $m_2$ by $m_1$ is clearly recursive with $m_1$ being the latch and $m_2$ being the head. So, assume there is no forward path from $m_2$ to $m_1$. If there does not exist a forward path from $m_2$ to $m_1$ then the path $(m_1, m_2)$ is a forward reference. This implies $\text{ord}(m_1) < j$, contradicting the original assumption.

End of Proof

Figure 2-7 is the D matrix for the dependency chart of Figure 2-6. It illustrates that 4 (Module I) is a latch module and 2 (Module E) is a head module.

Theorem

If $d_{ii}^+ = true$ and $i = \text{ord}(m)$ then $m$ is potentially recursive (i.e., there exists a path from $m$ back to itself).

Proof

By definition, $d_{ii}^+$ is true if there exists a sequence, $d_{ir}, d_{rs}, \ldots, d_{ti}$, all of which are true. This implies the existence of a reference path, $m, m_1, m_2, \ldots, m$, thus $m$ is potentially recursive.

End of Proof
## Figure 2-7. An Example D Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SAI-007
Figure 2-8 is the $D^+$ matrix for the dependency chart of Figure 2-6. It illustrates that 2 and 4 (Modules E and I) are potentially recursive.

Note that the existence of a recursive path does not necessarily prove that modules on the path are recursive. During execution, the recursive path may never be traversed. Note also that a recursive call made unconditionally by a latch module is a potential infinite loop.

The techniques above were discussed in the context of the module concordance. A subset of the same methods would apply also to the data concordance and to the analysis of the requirement transductions.
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>t</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
</tr>
</tbody>
</table>

Figure 2-8. An Example $D^+$ Matrix
3. DATA BASE VERIFIER SUBSYSTEM DESIGN

As a result of the study and analysis conducted under SOW task Phase A, item 3, we performed a high level design (i.e., software development through the requirements and functional specification stages) of a data base verifier subsystem (DBVS). The functions of this data base verifier subsystem are analysis of the Data Manipulation Language (DML) commands within a FORTRAN source deck, collection of pertinent descriptions of the stored data base as viewed by the program(s), and printing of the subschema information in a user oriented format. The accomplishment of these functions was the goal of each step in the DBVS software design.

At the requirements stage of the development of the DBVS, we produced a Subsystem Software Requirements Document (SSRD). This document was written in accordance with the requirements methodology that we recommended as a result of analysis performed during this contract period (cf. Part I, subsection 2.1 of this report). Subsection 3.3 contains the SSRD which formed the basis of the functional design of the DBVS. For the specification stage, we used the Software Specification Language (SSL) that was designed under this contract and which is explained in Part I, subsection 2.2 and Part II, section 2 of this final report and in the special report, "SSL-A Software Specification Language."

A general description of the two main phases of the data base verifier, DML Statement Processing and Subschema Information Processing, are presented in subsections 3.1 and 3.2. These subsections are outlined in Table 3-1.
TABLE 3-1. DBVS Description Outline

1. Data Base Verifier Subsystem
   1.1 DML Command Processing
       1.1.1 DML Statement Recognition
       1.1.2 DML Statement Parsing
       1.1.3 DML Statement Components Storage
   1.2 Subschema Information Processing
       1.2.1 Subschema, realm, set, record, privacy, or error information retrieval
       1.2.2 Subschema, realm, set, record, privacy, or error information tabulation
       1.2.3 Summary Reporting
           1.2.3.1 Subschema, realm, set, record, privacy, or error information printing
3.1 DML STATEMENT PROCESSING

As each source statement is read, the first label (after the statement number field) of a non-comment statement is isolated by the DBVS using the information in Table 3-2. If this label is succeeded by a left parenthesis, the DBVS compares this label with the keywords contained in the FORTRAN DML Command Table (cf. Table 3-2). When a match is found, the label is entered into the keyword sequence, and parsing of the statement continues. (During parsing of the DML statements, the identified components of each clause or statement, i.e., the keywords, identifiers, or list items, are placed in three sequences. The sequence types were chosen because this type may vary dynamically in length in response to the variance of the number of keywords, identifiers, and list items within a DML statement).

Within each DML clause, identifiers (which occur to the right hand side of the equal sign) must be isolated and entered into the identifier sequence. These identifiers could contain CODASYL keywords as shown in Table 3-4 or one of the items (some of which contain keywords) in Table 3-5.

Some DML statements such as FETCH, MODIFY, etc. allow specification of list items which must be isolated and entered into the list item sequence. According to CODASYL, the items fall into the two categories described in Table 3-6.

The processing of DML statements including the construction of the keyword, identifier, or list item sequences continues in the manner described above until the entire statement has been parsed. Then the appropriate module is evoked according to the DML statement and its associated subschema information.
<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
</tr>
<tr>
<td>B</td>
<td>O</td>
</tr>
<tr>
<td>C</td>
<td>P</td>
</tr>
<tr>
<td>D</td>
<td>Q</td>
</tr>
<tr>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td>F</td>
<td>S</td>
</tr>
<tr>
<td>G</td>
<td>T</td>
</tr>
<tr>
<td>H</td>
<td>U</td>
</tr>
<tr>
<td>I</td>
<td>V</td>
</tr>
<tr>
<td>J</td>
<td>W</td>
</tr>
<tr>
<td>K</td>
<td>X</td>
</tr>
<tr>
<td>L</td>
<td>Y</td>
</tr>
<tr>
<td>M</td>
<td>Z</td>
</tr>
</tbody>
</table>
### TABLE 3-3. FORTRAN DML COMMAND TABLE

<table>
<thead>
<tr>
<th>1. FETCH</th>
<th>9. ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. FIND</td>
<td>10. INVOKE</td>
</tr>
<tr>
<td>3. GET</td>
<td>11. READY</td>
</tr>
<tr>
<td>4. STORE</td>
<td>12. FINISH</td>
</tr>
<tr>
<td>5. MODIFY</td>
<td>13. ACCEPT</td>
</tr>
<tr>
<td>6. ERASE</td>
<td>14. USE</td>
</tr>
<tr>
<td>7. CONNECT</td>
<td>15. PRIVACY</td>
</tr>
<tr>
<td>8. DISCONNECT</td>
<td>16. QUIT</td>
</tr>
</tbody>
</table>

### TABLE 3-4. ALLOWABLE KEYWORDS WITHIN DML STATEMENTS

<table>
<thead>
<tr>
<th>1. SUBSCHEMA</th>
<th>14. DUPLICATE</th>
<th>27. INSERT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. SCHEMA</td>
<td>15. ANY</td>
<td>28. REMOVE</td>
</tr>
<tr>
<td>3. ALL</td>
<td>16. OFFSET</td>
<td>29. STORE</td>
</tr>
<tr>
<td>4. SET</td>
<td>17. FIRST</td>
<td>30. MODIFY</td>
</tr>
<tr>
<td>5. REALM</td>
<td>18. LAST</td>
<td>31. FIND</td>
</tr>
<tr>
<td>6. UPDATE</td>
<td>19. NEXT</td>
<td>32. GET</td>
</tr>
<tr>
<td>7. RETRIEVAL</td>
<td>20. PRIOR</td>
<td>33. ERASE</td>
</tr>
<tr>
<td>8. EXCLUSIVE</td>
<td>21. CURRENT</td>
<td>34. FETCH</td>
</tr>
<tr>
<td>9. PROTECTED</td>
<td>22. OWNER</td>
<td>35. ORDER</td>
</tr>
<tr>
<td>10. CONCURRENT</td>
<td>23. RSE</td>
<td>36. CONNECT</td>
</tr>
<tr>
<td>11. ERROR</td>
<td>24. USING</td>
<td>37. DISCONNECT</td>
</tr>
<tr>
<td>12. RECORD</td>
<td>25. DISPLAY</td>
<td>38. OTHER</td>
</tr>
<tr>
<td>13. KEY</td>
<td>26. PRIVACY</td>
<td>39. STATUS</td>
</tr>
</tbody>
</table>
TABLE 3-5. IDENTIFIER SEQUENCE ELEMENTS

NOTE: These CODASYL definitions are predicated on the working document, "FORTREV," of the ANS committee for the proposed revised FORTRAN, (X3.9). This document was printed in the March 1976 issue of SIGPLAN Notices under the title, "Draft Proposed ANS FORTRAN."

- Character constant - is an apostrophe followed by a non-empty string of characters followed by an apostrophe.
- Character expression - is used to express a character string consisting of a character primary alone or concatenated with other character primaries. A character primary may be a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, character function reference, or character expression enclosed in parentheses (cf. FORTREV 75-09-26, Section 4).
- Character variable - FORTRAN variable of type character.
- Data base key - Integer variable. (FORTREV's definition of variable excludes array elements.)
- Data base name - record name, set name, realm name, or character expression. (The first 3 are names in the subschema being used.)
- Data base names - data base name.
- Error phrase - contains keyword error and either statement number or subroutine (with arguments if applicable) name.
- Retain - contains keyword RETAINING and either 1) the keyword RECORD, REALM, SET or 2) keyword SET and appropriate data base names, or contains keywords RETAINING and MULTIPLE.
- Record selection expression - The various record selection expressions are given in Table 3-12 under the explanation of the FIND statement.
- Statement number - consists of one to five digits
- Subroutine name - consists of one to six letters or digits, the first of which must be a letter.
- Usage - one of each of the following groups of keywords: 1) PROTECTED, EXCLUSIVE, CONCURRENT 2) RETRIEVAL, UPDATE.
TABLE 3-6. LIST ITEM SEQUENCE ELEMENTS

- Input list item which must be one of the following: variable name, array element name, character substring name, array name, or array block item (cf. FORTREV 12.8.2.1).
- Implied-DO list item consisting of one of the following: a variable name, an array element name, a character substring name, an array name, or an array block item (cf. FORTREV 12.8.2.3).
3.2 SUBSCHEMA INFORMATION PROCESSING

After the identifier, keyword, and list item sequences have been constructed, the various components of the DML statement containing subschema, REALM, SET, record, privacy, and error information must be extracted from these sequences. This information is then appropriately entered into one of the following tables: subschema, realm, set, record, or error status. The record description for each of these tables is given in Tables 3-7 through 3-11, respectively. These record description tables contain specific information about the subschema which can be collected from the DML statements. Moreover, the printing of these tables in a highly readable format will provide the user with different descriptions of the data base components which were established by the DML statements. With these labels, the user can cross reference the information and thereby determine the consistency of the data base descriptions within the bounds of the applications program.

3.3 FUNCTIONAL REQUIREMENTS FOR THE DBVS

As stated previously in Part I of this report, the specifications for the CODASYL DML are not in final form. The pertinent section of the CODASYL FORTRAN Data Base Facility Journal of Development (November 25, 1975), written by the Data Base Manipulation Language Committee, was presented in the January 1976 monthly progress report in Appendix A. It should be used as a reference for understanding the subsystem software requirements document subsequently presented in Figure 3-1.
<table>
<thead>
<tr>
<th>Item Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subschema Name</td>
<td>(cf. FORTREV, Section 4.8.1)</td>
</tr>
<tr>
<td>Schema Name</td>
<td>(cf. FORTREV, Section 4.8.1)</td>
</tr>
<tr>
<td>Privacy Key (assoc. with INVOKE or PRIVACY statement)</td>
<td>(cf. FORTREV, Section 4.8.1)</td>
</tr>
<tr>
<td>DML Command Indicator</td>
<td>Integer constant</td>
</tr>
<tr>
<td>Indicates reference by one of the following DML commands:</td>
<td>Integer data</td>
</tr>
<tr>
<td>INVOKE</td>
<td></td>
</tr>
<tr>
<td>PRIVACY</td>
<td></td>
</tr>
<tr>
<td>Line Number</td>
<td>Integer constant</td>
</tr>
<tr>
<td>Indicates line number of DML command in listing</td>
<td>Integer data</td>
</tr>
<tr>
<td>Item Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Realm Name</td>
<td>Integer variable</td>
</tr>
<tr>
<td></td>
<td>Hollerith data</td>
</tr>
<tr>
<td>Set Names</td>
<td>Integer variables</td>
</tr>
<tr>
<td></td>
<td>Hollerith data</td>
</tr>
<tr>
<td>Usage indicator</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates one item from</td>
<td></td>
</tr>
<tr>
<td>each of the following</td>
<td></td>
</tr>
<tr>
<td>groups:</td>
<td></td>
</tr>
<tr>
<td>1. PROTECTED, EXCLUSIVE,</td>
<td></td>
</tr>
<tr>
<td>CONCURRENT</td>
<td></td>
</tr>
<tr>
<td>2. RETRIEVAL, UPDATE</td>
<td></td>
</tr>
<tr>
<td>Statement number or subroutine name for</td>
<td>Integer variable</td>
</tr>
<tr>
<td>error handling</td>
<td>Hollerith data</td>
</tr>
<tr>
<td>Record Name</td>
<td>Integer variables</td>
</tr>
<tr>
<td></td>
<td>Hollerith data</td>
</tr>
<tr>
<td>Subschema name</td>
<td>(cf. FORTREV, Section 4.8.1)</td>
</tr>
<tr>
<td>Privacy Key (assoc. with PRIVACY</td>
<td>(cf. FORTREV, Section 6.2)</td>
</tr>
<tr>
<td>statement)</td>
<td></td>
</tr>
<tr>
<td>DML Command Indicator</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates reference by one of the</td>
<td></td>
</tr>
<tr>
<td>following DML commands:</td>
<td></td>
</tr>
<tr>
<td>READY, FINISH, FIND, FETCH, OR PRIVACY</td>
<td></td>
</tr>
<tr>
<td>Line Number</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates line number of DML command in</td>
<td></td>
</tr>
<tr>
<td>listing</td>
<td></td>
</tr>
<tr>
<td>Item Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Set Name</td>
<td>Integer variable Hollerith data</td>
</tr>
<tr>
<td>Record Names</td>
<td>Integer variable Hollerith data</td>
</tr>
<tr>
<td>Modification Indicator</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td>Privacy Key (assoc. with PRIVACY statement)</td>
<td>(cf. FORTREV, Section 6.2)</td>
</tr>
<tr>
<td>DML Command Indicator</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td></td>
<td>Indicates reference by one of the following DML commands:</td>
</tr>
<tr>
<td></td>
<td>CONNECT, DISCONNECT, MODIFY, FIND, FETCH, ERASE, or PRIVACY</td>
</tr>
<tr>
<td>Statement Number or</td>
<td>Integer variable Hollerith data</td>
</tr>
<tr>
<td>Subroutine Name for</td>
<td></td>
</tr>
<tr>
<td>Error Handling</td>
<td></td>
</tr>
<tr>
<td>Record Delete Indicator</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td>Subschema Name</td>
<td>Integer variable Hollerith data</td>
</tr>
<tr>
<td>Line Number</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td></td>
<td>Indicates line number of DML command in listing.</td>
</tr>
<tr>
<td>Item Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Record Name</td>
<td>Integer variable Hollerith data</td>
</tr>
<tr>
<td>Set Name</td>
<td>Integer variables Hollerith data</td>
</tr>
<tr>
<td>Realm Name</td>
<td>Integer variables Hollerith data</td>
</tr>
<tr>
<td>Subschema Name</td>
<td>(cf. FORTREV, Section 4.8.1)</td>
</tr>
<tr>
<td>Modification Indicator</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td></td>
<td>Indicates that set relationship has been</td>
</tr>
<tr>
<td></td>
<td>changed within the program.</td>
</tr>
<tr>
<td>Statement Number or Subroutine Name</td>
<td>Integer variables Hollerith data</td>
</tr>
<tr>
<td>Data Base Key Name</td>
<td>Integer variables Integer data</td>
</tr>
<tr>
<td>Data Item Names</td>
<td>Integer variables Integer data</td>
</tr>
<tr>
<td>Privacy Key (assoc. with PRIVACY statement)</td>
<td>(cf. FORTREV, Section 6.2)</td>
</tr>
<tr>
<td>DML Command Indicator</td>
<td>Integer constant Integer data</td>
</tr>
<tr>
<td></td>
<td>Indicates reference by one of the following</td>
</tr>
<tr>
<td></td>
<td>DML commands: FIND, GET, FETCH, STORE, MODIFY, ERASE, CONNECT, DISCONNECT, or PRIVACY</td>
</tr>
<tr>
<td>Item Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Line Number</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates line number of</td>
<td></td>
</tr>
<tr>
<td>DML command in listing</td>
<td></td>
</tr>
<tr>
<td>DML Statement Indicator</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates privacy access</td>
<td></td>
</tr>
<tr>
<td>by one of the following</td>
<td></td>
</tr>
<tr>
<td>DML commands:</td>
<td></td>
</tr>
<tr>
<td>INSERT, REMOVE, STORE,</td>
<td></td>
</tr>
<tr>
<td>MODIFY, FIND, GET,</td>
<td></td>
</tr>
<tr>
<td>FETCH, or ERASE</td>
<td></td>
</tr>
<tr>
<td>Record Delete Indicator</td>
<td>Integer constant</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>Indicates PERMANENT, SELECT</td>
<td></td>
</tr>
<tr>
<td>IVE, or ALL</td>
<td></td>
</tr>
<tr>
<td>record delete</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 3-11. ERROR STATUS TABLE

#### RECORD DESCRIPTION

<table>
<thead>
<tr>
<th>Item Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure Name</td>
<td>Integer variable</td>
</tr>
<tr>
<td></td>
<td>Hollerith data</td>
</tr>
<tr>
<td>ALL indicator</td>
<td>Integer variable</td>
</tr>
<tr>
<td></td>
<td>Hollerith data</td>
</tr>
<tr>
<td>OTHER indicator</td>
<td>Integer variable</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
<tr>
<td>STATUS indicator</td>
<td>Integer variable</td>
</tr>
<tr>
<td></td>
<td>Integer data</td>
</tr>
</tbody>
</table>
Problem Statement:

Design a data base verifier subsystem that analyzes FORTRAN Data Manipulation Language (DML) statements for the purpose of ensuring a consistent and valid data base evocation by the program. Print the results of this analysis in a form easily interpretable by the user.

SYSTEM SOFTWARE REQUIREMENTS DOCUMENT

Direction

Design a data base verifier subsystem that analyzes all FORTRAN DML statements and organizes the subschema, realm, set, record, privacy, and error information contained within the DML statements into appropriate output for the user.

Input

LINE_BUFFER: FORTRAN DML source card

Transductions

INITIALIZE_SYSTEM: Initialize DML command, character, and keyword tables, line number counter, etc.

CHECK_DML_COMMAND: Check first keyword for a match in the DML command (Table 3-3)

CHECK_DML_KEYWORD: Check keyword for a match in the keyword table (Table 3-4)

BUILD_KEYWORD_SEQ: Build keyword sequence for each DML statement

BUILD_IDENTIFIER_SEQ: Build identifier sequence for each statement

Figure 3-1. Subsystem Software Requirement Document for DBVS
BUILD_LIST_ITEM_SEQ: Build list item sequence for each statement

BUILD_DML_TABLE: Construct DML command table

BUILD_KEYWORD_TABLE: Construct keyword table

READ_LINE: Read a line of source code

EVOKE_MODULE: Evoke appropriate module after processing each DML statement or QUIT command.

Output

SUBSCHEMA_INFO: Subschema, realm, set, record, privacy, and error handling information in tabular form

Constraints

Input: The source code must be written in CODASYL extended FORTRAN which contains DML statements for program/data base interaction. The acceptable input formats are listed in Table 3-12.

Implications

BUILD_DML_TABLE ⊆ CHECK_DML_COMMAND

BUILD_KEYWORD_TABLE ⊆ CHECK_DML_KEYWORD

READ_LINE ⊆ CHECK_DML_COMMAND

READ_LINE ⊆ CHECK_DML_KEYWORD

READ_LINE ⊆ BUILD_KEYWORD_FILE

Figure 3-1. (Continued)
TABLE 3-12. INPUT FORMATS

1. `INVOKE(SUBSCHEMA=<char const*>* , SCHEMA=<char const*>* , PRIVACY = <char const> *)`  
2. `READY({ALL|SET = <db names>*}|{REALM = <db names>*}) , <usage>*[,<error>*])`  
3. `FINISH({ALL|SET = <db names>*}|{REALM = <db names>*}) [,<error>*])`  
4. `CONNECT({RECORD = <db name>*}|{ALL|SET = <db names>*}) , [,<error>*])`  
5. `DISCONNECT({RECORD = <db name>*}|{ALL |{SET = <db names>*}} , [,<error>*])`  
6. `ACCEPT(CURRENCY=<db key>[, <currency type> ]|[,<error>])`  
   `<currency type>::=({RECORD|SET}=<db name>|RUNUNIT|REALM=<db name>`  
   or  
   `ACCEPT(REALM NAME=<char var>[, <name type>]|,[,<error>])`  
   `<name type>::=({RECORD|SET}=<db name})|RUNUNIT|{KEY=<db key>)`  
7. `FIND({<rse>|[RSE=<char exp>*]} [,retain*]|[,<error>*])`  
   `<rse>::=<rse 1>|<rse 2>|<rse 3>|<rse 4>|<rse 5>|<rse 6>|<rse 7>`  
   `<rse 1>::= KEY=<db key>[,RECORD=<db name>]`  
   `<rse 2>::={(DUPLICATE|ANY),RECORD=<db name>}`  
   `<rse 3>::=DUPLICATE,SET=<db name>,USING=<id list>`  
   `<rse 4>::=<posi>([,]RECORD=<db name>`  
   `[,{REALM|SET}=<db name>]`  
   `<posi>::=(OFFSET=<int exp>)|FIRST|LAST|NEXT|PRIOR`  
   `<rse 5>::=CURRENT[,{RECORD=<db name}>]|{REALM|SET}=<db name>`  
   `<rse 6>::=OWNER,SET=<db name>`  
   `<rse 7>::=RECORD=<db name>,[CURRENT]SET=<db name> [,USING=<id list>]*`  
8. `GET({RECORD=<db name>}[<error>] |item list])`  
9. `FETCH({<rse>|[RSE=<char exp>*} [,retain*]|[,<error>]*`)  
   `[item list]*)`  
10. `STORE (RECORD=<db name>*[,retain*]|[,error]*)`  

* Meta symbols are explained in Table 3-13.
TABLE 3-12. (Continued)

11. MODIFY([RECORD=<db name>])
    ([,ALL][,SET=<db names>])[,ONLY][,<error>])
    [<id list>])

12. ERASE([RECORD=<db name>,*][,member type]
    [,MEMBERS=<db names>]*[,SET=<db names>]*
    [,<error>])
    <member type>::= PERMANENT|SELECTIVE|ALL

13. PRIVACY(SUBSCHEMA=<char con>*[,DISPLAY],[<priv key>]*
    [<priv key>::=PRIVACY=[<char exp>|<priv key proc>]
    <priv key proc>::= FORTRAN subroutine name
    PRIVACY([SUBSCHEMA=<name>,REALM=<db name>*
    ,<usage mode>*[,<priv key>]*
    <usage mode>::=
    [,{PROTECTED|EXCLUSIVE}][,RETRIEVAL|UPDATE]
    PRIVACY([SUBSCHEMA=<char con>*][,RECORD=<db names>]*
    [,<dml stmt>],[<priv key>])
    <dml stmt>::=
    {INSERT|REMOVE|STORE|<erase>|MODIFY|FIND|GET
    FETCH}
    <erase>::=ERASE{PERMANENT|SELECTIVE|ALL
    PRIVACY([SUBSCHEMA=<char con>*][,SET=<db names>]*
    [,<dml type>]*[,<priv key>]*
    <dml type>::=ORDER|CONNECT|DISCONNECT|<erase>|FIND|
    FETCH
    <erase>::= ERASE{PERMANENT|SELECTIVE|ALL
    PRIVACY([SUBSCHEMA=<char con>*][,item dml>,<priv key>]*
    [<id list>])
    <item dml>::=MODIFY{GET|FETCH}

* Meta symbols are explained in Table 3-13.
TABLE 3-12.  (Continued)

14. USE(PROCEDURE=<identifier>*[,{ALL|OTHER}|{STATUS=<status list>*}]})

15. ORDER({SET=<db name>*}[,LOCALLY][]{<sort spec>...}[,<error>])

   <sort spec>::={ASCENDING|DESCENDING}=(RECORD|KEY|<sort field>...)

   <sort field>::=ITEM=<id list>|KEY=<db names>|RECORD=<db names>

* Meta symbols are explained in Table 3-13.
<table>
<thead>
<tr>
<th>Meta-Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;char var&gt;</td>
<td>character variable - FORTREV variable of type character</td>
</tr>
<tr>
<td>&lt;char const&gt;</td>
<td>character constant - is an apostrophe followed by a non-empty string of characters followed by an apostrophe</td>
</tr>
<tr>
<td>&lt;char exp&gt;</td>
<td>character expression - is used to express a character string consisting of a character primary alone or concatenated with other character primaries. A character primary may be a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, character function reference, or character expression enclosed in parentheses. (See FORTREV 75-09-26, Section 4)</td>
</tr>
<tr>
<td>&lt;db key&gt;</td>
<td>data base key - integer variable (FORTREV's definition of variable excludes array elements)</td>
</tr>
<tr>
<td>&lt;db name&gt;</td>
<td>data base name - record name, set name, realm name, or character expression. (The first 3 are names in the subschema being used)</td>
</tr>
<tr>
<td>&lt;db names&gt;</td>
<td>data base names - data base name</td>
</tr>
<tr>
<td>&lt;error&gt;</td>
<td>error phrase - contains keyword ERROR and either statement number or subroutine (with arguments if applicable) name</td>
</tr>
<tr>
<td>&lt;id spec&gt;</td>
<td>item specification - input list item which must be one of the following: variable name, array element name, character substring name, array name, or array block item (See FORTREV 12.8.2.1)</td>
</tr>
</tbody>
</table>
TABLE 3-13. (Continued)

<table>
<thead>
<tr>
<th>Meta-Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;identifier&gt;</td>
<td>identifier - item specification</td>
</tr>
<tr>
<td>&lt;id list&gt;</td>
<td>identifier list - item specification(s)</td>
</tr>
<tr>
<td>&lt;item spec&gt;</td>
<td>identifier specification - item specification or implied-DO lists consist-</td>
</tr>
<tr>
<td></td>
<td>ting of one of the following: a variable name, an array (element name, a</td>
</tr>
<tr>
<td></td>
<td>character substring name, an array name or an array block item (12.8.2.3)</td>
</tr>
<tr>
<td>&lt;item list&gt;</td>
<td>item list - identifier specification</td>
</tr>
<tr>
<td>&lt;priv key proc&gt;</td>
<td>subroutine name - consists of one to six letters or digits, the first of</td>
</tr>
<tr>
<td></td>
<td>which must be a letter</td>
</tr>
<tr>
<td>&lt;retain&gt;</td>
<td>retain - contains keyword RETAINING and either 1) the keyword RECORD, REALM,</td>
</tr>
<tr>
<td></td>
<td>SET OR 2) keyword SET and appropriate data base names; or contains keywords</td>
</tr>
<tr>
<td></td>
<td>RETAINING and MULTIPLE</td>
</tr>
<tr>
<td>&lt;usage&gt;</td>
<td>usage - one of each of the following groups of keywords: 1) PROTECTED,</td>
</tr>
<tr>
<td></td>
<td>EXCLUSIVE, CONCURRENT 2) RETRIEVAL, UPDATE</td>
</tr>
<tr>
<td>&lt;rse&gt;</td>
<td>record selection expressions - are used to specify criteria whereby the data</td>
</tr>
<tr>
<td></td>
<td>base management system is to select a record in the data base. The various</td>
</tr>
<tr>
<td></td>
<td>record-selection expressions are used as follows:</td>
</tr>
<tr>
<td>&lt;rse 1&gt;</td>
<td>Format 1 - direct access</td>
</tr>
<tr>
<td>&lt;rse 2&gt;</td>
<td>Format 2 - calculate mode access</td>
</tr>
<tr>
<td>&lt;rse 3&gt;</td>
<td>Format 3 - set search access</td>
</tr>
<tr>
<td>&lt;rse 4&gt;</td>
<td>Format 4 - positional access</td>
</tr>
<tr>
<td>&lt;rse 5&gt;</td>
<td>Format 5 - currency indicator access</td>
</tr>
<tr>
<td>&lt;rse 6&gt;</td>
<td>Format 6 - set owner access</td>
</tr>
<tr>
<td>&lt;rse 7&gt;</td>
<td>Format 7 - set occurrence selection rules access</td>
</tr>
</tbody>
</table>

REPRODUCIBILITY OF THIS ORIGINAL PAGE IS PROTECTED.

3-22
3.4 FUNCTIONAL SPECIFICATIONS FOR THE DBVS

To facilitate the understanding of the DBVS functional specifications, the following documents should be used as references:

- the portion of the "CODASYL FORTRAN Data Base Facility Journal of Development" (November 25, 1975) which was printed in Appendix A of the January 1976 monthly progress report
- the special report, "SSL-A Software Specification Language," which was prepared for this contract
- the "draft proposed ANSI FORTRAN, BSR X3.9, X3J3/76" which appeared in the March 1976 issue of SIGPLAN Notices. Any references to FORTREV appearing in the specifications are synonymous with those of the aforementioned ANSI FORTRAN document.

In the special report on SSL, explicit descriptions of the preamble components as well as subsystem and module descriptions for designing a software system are presented. A simple outline of this document (Table 3-14) will serve as a guide to understanding the SSL description of the data base verifier. In accordance with the SSL report, the subsystem preamble will be presented first (Figure 3-2) and then the module descriptions (Figures 3-3 through 3-10). To accompany these specifications, a module structure chart for the DBVS (cf. Figure 3-11) and a table containing a summary of the module descriptions (cf. Table 3-15) are given.
TABLE 3-14. OUTLINE OF SSL COMPONENTS

I. Preamble Description
   1. Requirement Declaration
      a. Input and Output Parts
      b. Transduction Parts
      c. Constraint Declarations
   2. Data Type and Variable Declarations
      a. Simple Types
      b. Structured Types
      c. Pointer Types
   3. Constant Declarations

II. Module Description
   1. Module Statement
   2. Assumes and Satisfies Statements
   3. Fulfills Statement
   4. Accesses Statement
   5. Receives and Transmits Statements
   6. Creates, Modifies, and Uses Statements
   7. Execute Statement

III. Subsystem Description
   1. Subsystem Preamble
   2. Module Description
/* preamble for DBVS subsystem*/

Requirement

Input line_buffer; /*FORTRAN dml source card*/

Transductions

initialize_system; /*initialize dml command, character, and keyword tables, line number counter.*

check_dml_command; /*check first keyword for a match in the dml command table (Table 3-3) including terminate keyword quit*/

check_dml_keyword; /*check keyword for a match in the keyword table (Table 3-4)*/

build_keyword_seq; /*build keyword sequence for each dml statement*/

build_identifier_seq; /*build identifier sequence for each statement*/

build_list_item_seq; /*build list item sequence for each statement*/

build_dml_table_in_check_dml_command; /*construct dml command table*/

build_keyword_table_in_check_dml_keyword; /*construct keyword table*/

read_line_in_check_dml_command, check_dml_keyword, build_keyword_seq; /*read a line of source code*/

evoke_module; /*evoke appropriate module after processing each dml statement or quit command*/

Figure 3-2. DBVS Subsystem Preamble
save_subschema_info;/*save subschema, schema or privacy key names, and the dml command indicator in subschema table (Table 3-7)*/

save_realm_info;/*store set or realm information and usage information in realm table (Table 3-8)*/

save_set_info;/*store record (if specified) and set information in record and set tables, (Table 3-9 and 3-10, respectively)

save_record_info;/*store record name information, and if specified, item dml, record items, and privacy key information for record in record table (Table 3-10) and pertinent record information in set table (Table 3-9)*/

save_subschema-privacy in save_subschema_info; /*store subschema name and privacy key information for subschema in subschema table*/

save_realm_privacy in save_realm_info; /*store realm name, subschema name (if specified), usage indicator, and privacy key information in realm table*/

save_set_privacy in save_set_info; /*store set name, subschema name (if specified) dml type indicator, and privacy key information for set in set table (dml type indicator reflects one of the following commands; order, connect, disconnect, find, fetch, permanent, selective, or all)*/

Figure 3-2. (Continued)
save_record_privacy_in save_record_info;/*store record name and if specified, subschema name, dml statement (indicating insert, remove, store, erase (permanent, selective, all), modify, find, get, fetch) and privacy key information for record in record table*/

save_item_privacy_in save_record_info;/*store item dml (modify, get, fetch) and privacy key information and if specified, subschema name and item names in record table*/

save_error_table_info;/*store error information in error table (Table 3-11 )*/

print_tables;/*read and write information from subschema, realm, set, record, and error tables and write to printer*/

Output subschema_info;/*subschema, realm, set, record, privacy, and error handling information in tabular form*/

end;

/* beginning of data description within the preamble*/

variable dml_intrinsic; array [1..5] of text;/* contains current dml command for check_dml_command*/

for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, save_subschema_info, save_subschema_privacy, save_realm_info, save_realm_privacy, save_set_info, save_set_privacy, save_record_info, save_record_privacy, save_item_privacy;

Figure 3-2. (Continued)
variable keyword_table: array [1..39, 1..5] of text; /*Table 3-4(allowable keywords within dml statements)* /

for build_keyword_table, check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, save_subschema_info, save_subschema_privacy, save_realm_info, save_realm_privacy, save_set_info, save_set_privacy, save_record_info, save_record_privacy, save_item_privacy, save_error_table_info, build_dml_table, initialize_system;

variable list_item: array [1..3] of text; /* contains user specified list item*/

for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, save_record_info, save_record_privacy, save_item_privacy;

variable identifier; array [1..3] of text; /* contains user specified identifier*/

for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, save_subschema_info, save_subschema_privacy, save_realm_info, save_realm_privacy, save_set_info, save_set_privacy, save_record_info, save_record_privacy, save_item_privacy, save_error_table_info;

type ident_seq=sequence of text
ident: array [1..3] of text/* identifier name*/

Figure 3-2. (Continued)
variable dml_command_table: array [1..16, 1..5] of text;/*contains commands of Table 3-3*/
for build_dml_table, check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, initialize_system, build_keyword_table;

variable keyword: array [1..5] of text;/* contains user specified keyword*/
for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, save_subschema_info, save_subschema_privacy, save_realm_info, save_realm_privacy, save_set_info, save_set_privacy, save_record_privacy, save_item_privacy, save_error_table_info, save_record_info;

type line_buffr = array [1..72] of char;/*reflection of program card*/

variable line_buffer: line_buffr;/*contains current source statement being analyzed*/
for read_line, check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module;

variable char_table; array [1..49] of char;/*
A..Z, 0..9, blank, =, +, *, /, (, ), ,, , ;, ::=, $*/:
for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, initialize_system, build_dml_table, build_keyword_table;

Figure 3-2. (Continued)
variable identifier_seq: ident_seq;/*sequence containing all identifiers associated with one statement*/
for build_identifier_seq, check_dml_command,
check_dml_keyword, build_keyword_seq, build_list_seq, evoke_module, read_line, save_subschema_info, save_subschema_privacy, save_realm_info, save_realm_privacy, save_set_info, save_set_privacy, save_record_info, save_record_privacy, save_item_privacy, save_error_table_info;

{type list_itm_seq = sequence of text
  parametr: array [1..3] of text;/*list_item_name*/
} end;

variable list_item_seq: list_itm_seq;/*sequence containing all list items associated with one statement*/
for build_list_item_seq, check_dml_command,
check_dml_keyword, build_keyword_seq, build_identifier_seq, evoke_module, read_line;

{type keywrd_seq = sequence of records
  keyword: array [1..3] of text;/*name of keyword*/
  identifier_counter; integer;/*contains number of identifiers (i.e., no. of times to read the identifier sequence associated with each keyword*/
} end;

variable keyword_seq: keywrd_seq;/*contains keyword number of identifier, associated with each keyword*/
for check_dml_command, check_dml_keyword, build_keyword_seq, build_identifier_seq, build_list_item_seq, evoke_module, read_line, build_keyword_seq, save_subschema_info, save_subschema_privacy, save_realm_info,
save_realm_privacy, save_set_info, save_set_privacy, save_record_info, save_record_privacy, save_item_privacy, save_error_table_info;

variable list_item_seq_ind: integer;/*indicates number of list items associated with a fetch, get, modify, or privacy statement, i.e., it indicates number of entries in the list item sequence*/

for check_dml_command, check_dml_keyword, build_item_seq, evoke_module, read_line, save_record_info, save_record_privacy, save_item_privacy, initialize_system, build_dml_table, build_keyword_table;

type subschema_table=file of records

  subschema_name: array [1..3] of text;
  schema_name: array [1..3] of text;
  privacy_key: array [1..3] of text;
  dml_intrinsic: array [1..5] of text;

  /*dml intrinsic represents either the invoke or privacy statement*/
  line_no: integer; /*line number of dml command*/

end;

variable subschema_table: subschema_table;/*the subschema table contains the subschema name and depending on the options exercised by the invoke and privacy statements, the schema name, and privacy key*/

for save_subschema_info, save_subschema_privacy, print_tables;

type realm_table = file of records

  realm_name: array [1..3] of text;
  set_name: array [1..3] of text;
  usage_indicator: integer;/*indicates either protected, exclusive, concurrent and either retrieval or update

Figure 3-2. (Continued)
variable realm_table: realm_tabl;

for save_realm_info, save_realm_privacy,
    save_record_info, save_record_privacy,
    save_item_privacy, print_tables;

type set_tabl=file of records

set_name: array 1..3 of text;
record_name: array 1..3 of text;
mod_indicator: integer; /*indicates that set relationship has been changed within the program*/
privacy_key: array 1..3 of text;
dml_intrinsic: array 1..5 of text; /*dml intrinsic represents one of the following statements-connect, disconnect, modify, find, fetch, erase, or privacy*/
error_handling: array 1..3 of text;
record_delete_ind: integer; /*0-indicates no record deletions, 1-indicates normal deletion, 2-indicates permanent deletion 3-indicates selective deletion, 4-indicates all deletion*/
subschema_name: array 1..3 of text;
line_no: integer; /*line number of dml command in listing*/
end;

Figure 3-2. (Continued)
variable set_table: set_table;/* the set table
contains mainly set and record information
depending on the options exercised by the
connect, disconnect, modify, find, fetch, erase,
or privacy statements*/
for save_set_info, save_record_info, save_set_
privacy, save_record_privacy, save_item_
privacy, print_tables;
type record_tabl=file of records
record_name: array [1..3] of text;
set_name: array [1..3] of text;
realm_name: array [1..3] of text;
subschema_name: array [1..3] of text;
modification_ind: integer;/* indicates that
set relationship has been changed within
the program*/
error_handling: array [1..3] of text;
data_base_key_name: array [1..3] of text;
data_item_name: array [1..5, 1..3] of text;/*
contains first five names in a list of
data item names*/
privacy_key: array [1..3] of text;
dml_intrinsic: array [1..5] of text;/*
dml intrinsic represents one of the
following statements: find, get, fetch,
store, modify, erase, connect, disconnect,
or privacy*/
line_no: integer;/*line number of dml command
in listing*/
dml_stmt_ind: integer;/*indicates privacy access
by one of the following dml commands: insert,
remove, store, modify, find, get, fetch, or
erase*/
record_delete_ind: integer;/*indicates that
record is to be deleted*/
end;

Figure 3-2. (Continued)
variable record_table: record_tabl;/* the record
    table contains primarily record and set information
    depending on the options exercised by the
    find, get, fetch, store, modify, erase, connect,
    disconnect, or privacy statements*/
for save_record_info, save_set_info, save_set_
    privacy, save_record_privacy, save_item_
    privacy, print_tables;

type error_status_tabl=file of records
procedure_name: array [1..3] of text;
    /* subroutine procedure which is to be
    called in the event a data base exception
    condition is encountered*/
all-indicator: integer; /* the indicated pro-
    cedure will be called for all data base
    exception processing*/
other_indicator: integer; /* the indicated pro-
    cedure will be called for any data base
    exception conditions which are not previously
    established*/
status-indicator: integer; /* the indicated pro-
    cedure will be called whenever any of the
    conditions in the associated status list are
    called*/
line_no: integer; /* line number of dml command
    in listing */
end;
variable error_status_table; error_status_tabl; /*
 the error status table contains information con-
 cerning the options exercised in the use state-
 ment*/
for save_error_table_info, print_tables;
variable line-no-counter: integer; /* contains line
    number of statement as it appears in the
    listing */

Figure 3-2. (Continued)
for check_dml_command, check_dml_keyword, 
build_keyword_seq, build_identifier_seq. 
built_list_item_seq, evoke_module, read_ 
line, save_subschema_info, save_subschema_ 
privacy, save_realm_info, save_realm_ 
privacy, save_set_info, save_set_privacy, 
save_record_info, save_record_privacy, save_ 
item_privacy, save_error_table_info, 
initialize_system, build_dml_table, build_ 
keyword_table;

end;

/*end of data description within the preamble*/
/*end of preamble for DBVS subsystem*/
module dml_recognizer;

fulfills check_dml_command, check_dml_keyword,
  build_keyword_seq, build_identifier_seq,
  build_list_item_seq, evoke_module, read_line;

/*initialize system*/
creates line_buffer, dml_intrinsic, keyword, list_item, identifier, identifier_seq, list_item_seq, keyword_seq;

/*read card and fill line buffer*/
accesses card_reader; modify line_buffer;

/*determine dml command*/
modifies dml_intrinsic using line_buffer, char_table,
  dml_command_table;

/*isolate dml keyword*/
modifies keyword using line_buffer, char_table,
  keyword table;

/*construct keyword sequence*/
modifies keyword_seq using keyword;

/*isolate identifier*/
modifies identifier using line_buffer, char_table;

/*construct identifier sequence*/
modifies identifier_seq using identifier;

/*isolate list items for dml statements*/
modifies list_item using line_buffer, char_table;

/*construct list item sequence*/
modifies list_item_seq using list_item;

Figure 3-3. DML - RECOGNIZER Module Description
modifies list_item_seq_ind;

executes conditionally
    subschema_process, realm_process,
    set_process, record_process, error_process, output_summary;
executes initialize_system;

end;
module initialize_system;

fullfills initialize_system, build_dml_table, 
read_line, build_keyword_table;

create dml_command_table, char_table, keyword_table, 
line_no_counter, list_item_seq_ind;

/*construct dml command tables*/
modifies dml_command_table;

/*construct char_table*/
modifies char_table;

/*construct keyword_table*/
modifies keyword_table;

/*initialize line number counter*/
modifies line_no_counter;

/*initialize list item sequence indicator*/
modifies list_item_seq_ind;

end;

Figure 3-4. INITIALIZE-SYSTEM Module Description
module subschema_process;

fulfills save_subschema_info, save_subschema_privacy;

creates subschema_table;

/* retrieve subschema, schema, privacy, or display keywords */
modifies keyword using keyword_seq @[keyword, keyword_table];

/* retrieve subschema, schema or privacy key name */
modifies identifier using keyword_seq @[identifier_counter, identifier_seq];

/* construct record for subschema table */
modifies subschema_table @[subschema_name,
    subschema_table @[schema_name,
    subschema_table @[privacy_key,
    subschema_table @[dml_intrinsic,
    subschema_table @[line_no using keyword, identifier, dml_intrinsic, line_no_counter];

end;

Figure 3-5. SUBSCHEMA-PROCESS Module Description
/*module description for realm processing*/
module realm_process;

fulfills save_realn_info, save_ream_privacy;

creates realm_table;

/*retrieve all, set, realm, protected, concurrent,
exclusive, retrieval, update, error keywords*/
modifies keyword using keyword-seq @.keyword, keyword_table;

/*retrieve realm, set, subschema, error or
privacy key name*/
modifies identifier using keyword-seq @.identifier_counter, identifier_seq;

/*construct partial record for realm table*/
modifies realm_table @.realm_name,
realm_table @.set_name,
realm_table @.usage_indicator,
realm_table @.error_handling,
realm_table @.subschema_name,
realm_table @.privacy_key,
realm_table @.dml_intrinsic,
realm_table @.line-no, using keyword, identifier, dml_intrinsic, line_no_counter;

end;

Figure 3-6. REALM-PROCESS Module Description
module set-process;

fulfills save_set_info, save_set_privacy;
creates set_table, record_table;

/*retrieves record, all, set, error, only, subschema, order, connect, disconnect, permanent, selective, all, find, and fetch keywords*/
modifies keyword using keyword_seq @.keyword, keyword_table;

/*retrieve record, set, error, privacy, key, or subschema name*/
modifies identifier using keyword_seq @.identifier_counter, identifier_seq;

/*construct partial record for set table*/
modifies set_table @.set_name,
        set_table @.record_name,
        set_table @.mod_indicator,
        set_table @.privacy_key,
        set_table @.dml_intrinsic,
        set_table @.line-no,
        set_table @.error_handling,
        set_table @.subschema_name using keyword,
        identifier, dml_intrinsic, line_no_counter;

/*construct partial record for record table*/
modifies record_table @.record_name,
        record_table @.set_name,
        record_table @.modification_ind,
        record_table @.dml_intrinsic,
        record_table @.line_no using keyword, identifier, dml_intrinsic, line_no_counter;

Figure 3-7. SET-PROCESS Module Description

3-41
/*module description for record processing*/
module record_process;

fulfills save_record_info, save_record_privacy,
    save_item_privacy;

creates record_table, set_table, realm_table;

/*retrieve record, key, duplicate, any, set,
using,first, last, next, prior, realm, offset,
current, owner, rse, multiple, error, all, only,
members, permanent, selective, all, subschema,
insert, remove, store, modify, find, get, or
fetch keywords*/
modifies keyword using keyword_seq @.keyword, keyword_table;

/*retrieve record, key, set, using, offset,
real, rse, error, member or subschema name*/
modifies identifier using keyword_seq @.identifier_counter, identifier_seq;

/*construct partial record for record table*/
modifies record_table @.record_name,
    record_table @.set_name,
    record_table @.realm_name,
    record_table @.subschema_name,
    record_table @.modification_ind,
    record_table @.error_handling,
    record_table @.database_key_name,
    record_table @.data_item_name,
    record_table @.privacy_key,
    record_table @.dml_intrinsic,
    record_table @.line_no,

Figure 3-8 RECORD-PROCESS Module Description
record_table @.dml_stmt_ind,
record_table @.record_delete_ing using
keyword, identifier, dml_intrinsic, line_no_counter, list_item;

/*construct partial record for set table*/
modifies set_table @.set_name,
set_table @.record_name,
set_table @.mod_indicator,
set_table @.dml_intrinsic,
set_table @.line_no,
set_table @.record_delete_ind using keyword,
identifier, dml_intrinsic, line_no_counter;

/*construct partial record for realm table*/
modifies realm_table @.realm_name,
realm_table @.record_name,
realm_table @.subschema_name,
realm_table @.dml_intrinsic,
realm_table @.line_no using keyword, identifier, dml_intrinsic, line_no_counter;

/*retrieve list items (actually data item names) that have been saved as a result of a fetch, get, modify, or privacy statement*/
modifies list_item_seq_ind;

end;

Figure 3-8. (Continued)
/*module description for error processing*/
module error_process;

fulfills save_error_table_info;
creates error_status_table;

/*retrieve procedure, all, other, status keywords*/
modifies keyword using keyword_seq @.keyword, keyword_table,

/*retrieve procedure or status names*/
modifies identifier using keyword_seq @.identifier_counter, identifier_seq;

/*construct partial record for error status table*/
modifies error_status_table @.procedure_name,
    error_status_table @.all_indicator,
    error_status_table @.other_indicator,
    error_status_table @.status_indicator,
    error_status_table @.line_no using keyword,
    identifier, line_no_counter;

end;

Figure 3-9. ERROR-PROCESS Module Description
/*module description for print process*/
module output_summary;

fulfills print-tables:

    /*read and print subschema table records*/
    access line-printer using subschema_table;

    /*read and print realm table records*/
    access line_printer using realm_table;

    /*read and print set table records*/
    access line_printer using set table;

    /*read and print record table records*/
    access line_printer using record_table,

    /*read and print error status table records*/
    access line_printer using error_status_table,

customed;

/*end of main subsystem*/

Figure 3-10. OUTPUT-SUMMARY Module Description
Table 3-15. Module Descriptions for the DBVS

DML_RECOGNIZER

To isolate data manipulation language (DML) keywords and associated identifiers and list items. To construct appropriate keyword, identifier, and list item sequences. To evoke initialization or report modules, or to evoke appropriate DML statement processor modules.

INITIALIZE_SYSTEM

To initialize the line number counter, the list item sequence indicator, and the following tables: dml_command, character, and keyword.

SUBSCHEMA_PROCESS

To collect subschema information and enter it into the subschema table.

REALM_PROCESS

To collect realm information and enter it into the realm table.

SET_PROCESS

To collect set information and enter it into the set table.

RECORD_PROCESS

To collect record information and enter it into the record table.
Table 3-15. (Continued)

ERROR_PROCESS

To collect error information and enter it into the error status table.

OUTPUT_SUMMARY

To read and print the following tables: subschema, realm, set, record, and error status.
Figure 3-11. Module Structure Chart for DBVS
4. STATIC CODE ANALYSIS

In this section, we present the detailed design (build-to) specifications for the capabilities listed in Table 4-1 which will be incorporated into FACES. For each of the twelve capabilities, we provide a detailed Unit Module Description including a flowchart which is sufficient for coding. However, for complete understanding of this documentation, the following FACES documentation must be used:

- Version 2, Mod 0, Fortran Automatic Code Evaluation System SYSTEM DOCUMENTATION, September 1975, Browne and Ramamoorthy Inc.
- Version 2, Mod X FACES Program Listing

The unit module descriptions correspond to those set forth in NASA's "Guidelines For Software Detailed Design Specification (CODE-TO)," while the flowcharts follow ANSI FORTRAN flowcharting recommendations. The detailed specifications for the capabilities will appear in order according to Table 4-1. (New capabilities 5 and 6 and capabilities 7 and 8 are treated under the same unit module description.)
TABLE 4-1. NEW FACES CAPABILITIES

1. EQUIVALENCE and EXTERNAL statements are flagged.
2. COMMONs not named are flagged.
3. ALL COMMON BLOCK arrays must be dimensioned in COMMON BLOCK statements.
4. DIMENSION statement and variable which contain an adjustable (variable) dimension are flagged.
5. Constants, hollerith, or arithmetic expression arguments used in subroutine argument lists are flagged.
6. All occurrences where the same variable exists in multiple positions in an actual parameter list are flagged.
7. Arithmetic IFs are flagged.
8. Targets of branches should not be other branches, especially single GO TOs.
9. Variable which is I/O unit designator is flagged.
10. Statement labels must appear in increasing order.

11. Occurrences of error-prone FORTRAN statements such as ASSIGN statement, assigned GO TO, and PAUSE are flagged.

12. The appearance of the same COMMON variable in more than one DATA statement is flagged.
UNIT MODULE DESCRIPTION

IDENTIFICATION
AIR Modifications

STORAGE ALLOCATION
1K hexadecimal bytes

PURPOSE
Process new query numbers for new FACES capabilities.

DESCRIPTION
These modifications cause AIR to recognize the new query numbers and call the new subroutine or modified subroutine as it processes original query numbers.

HOW ENTERED
Called by LNKAIR

CALLING SEQUENCE
CALL AIR

UNIT MODULE OR OTHER ROUTINES CALLED

New Modules: ER230    ER275
ER240    ER280
ER250    ER290
ER280

Modified Modules: CONALC
COMBAL
MULBRA
SET/USE PARAMETERS

No new parameters are introduced NUMBER, the standard variable for query number is used.

SIGNIFICANT INTERNAL VARIABLES

No new variables are introduced.

LIMITATIONS AND RESTRICTIONS

All variables are set by IMPLICIT INTEGER (A-Z).

DETAILED FLOWCHART

Only the modifications are included on the following flowchart.
REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

4-6
FLAG VARIABLES USED AS NO DESIGNATORS

SUBPROGRAM DUMMY PARAMETERS THAT ARE NEVER USED ARE Flagged

STATEMENT NUMBERS NOT IN ASCENDING ORDER FLAGGED

VARIABLE IN PARAMETER LIST IF NO STATEMENT WHICH IS USED AS AN ADJUSTIBLE VARIABLE DIMENSION FOR ARRAY IS FLAGGED

ALL COMMON BLOCK ARRAYS MUST BE DIMENSIONED IN COMMON BLOCK STATEMENTS

DIMENSION STATEMENT AND VARIABLE WHICH CONTAIN AN ADJUSTABLE VARIABLE DIMENSION ARE FLAGGED

CONSTANTS, REALS SHOULD NOT BE USED IN SUBDIVISION CALLING ARGUMENT LISTS

QUERY 250? NUMBER, NE.250
YES ER 250

QUERY 250? NUMBER, NE.250
NO

QUERY 265? NUMBER, NE.255
YES ER 255

QUERY 265? NUMBER, NE.255
NO

QUERY 260? NUMBER, NE.260
YES ER 260

QUERY 260? NUMBER, NE.260
NO

QUERY 265? NUMBER, NE.265
YES ER 265

QUERY 265? NUMBER, NE.265
NO

QUERY 270? NUMBER, NE.270
YES ER 270

QUERY 270? NUMBER, NE.270
NO

QUERY 275? NUMBER, NE.275
YES ER 275

QUERY 275? NUMBER, NE.275
NO

QUERY 280? NUMBER, NE.280
YES ER 280

QUERY 280? NUMBER, NE.280
NO

C
UNIT MODULE DESCRIPTION

IDENTIFICATION

ER24Q -- Error 240 routine

STORAGE ALLOCATION REQUIREMENT (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag EQUIVALENCE and EXTERNAL statements

DESCRIPTION

The routine checks each module for statement type. If type is a 12 (EQUIVALENCE) or 40 (EXTERNAL) the routine flags the statement in Flag File.

HOW ENTERED

Called by AIR

CALLING SEQUENCE

Call ER240
No Arguments

ROUTINES CALLED

IE
GETE
GETL
POP

4-9
SET/USE PARAMETERS

SET

Global: COMMON/SPEREG/

   ER(10) -- Error registers
   NOTE: Only ER(3) and ER(4) are set.

   FBR -- forward/backward register

USE

Global: COMMON/FLAG/

   FLAGFL -- input/output designator

   COMMON/H/

   HB -- hollerith B
   HF -- hollerith F

   COMMON/TABLE/

   HDIR -- directory table
   HUSE1 -- use table
   HSYM -- symbol table
   HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

   IO, I1, I3, I4 -- integer constants
   K -- error flag
   SCIND -- source code indicator
   FSTAT -- first statement number
   LSTAT -- last statement number
   BF -- general purpose flag
   STATYP -- statement type storage
LIMITATIONS AND RESTRICTIONS

All variables are set IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached
IMPLICIT INTEGER (A-Z)

LABELLED COMMON/FLAG/,/H/, ISREG/,/TABLE/
THIS ROUTINE FLAGS EXTERNAL AND EQUIVALENCE STATEMENTS
STATYPE 40, 12

INITIALIZE INTEGER VARIABLES 10, 11, 13, 14 - 0, 1, 3, 4

INITIALIZE ERROR NUMBER K'240

SET FORWARD/ BACKWARD FLAG TO FORWARD FBR = IF

IF (HDIR, BF) SELECT A MODULE TO EXAMINE

ARE ALL EXAMINED IF, EQ.2;

YES

RETURN

NO

GETE (HDIR, 13) NEXT ELEMENT IN MODULE

END OF THIS MODULE EQ.0)

YES

SET FBR = HB FOR BACKWARD FLOW

NO

GETL (BF) FETCH LINE OF CODE

GETE (HDIR, 14) GET SOURCE CODE INDEX

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR.
UNIT MODULE DESCRIPTION

IDENTIFICATION

COMBAL addition

STORAGE ALLOCATION (estimate)

Additional 250 hexadecimal bytes

PURPOSE

Flag unlabelled COMMON

DESCRIPTION

The routine contains a check on the Common Block name. If it is blank the statement is flagged.

NOTE: There are no new variables and subroutines; there is only a new error message and an additional check.

DETAILED FLOWCHART

See attachments
EXAMINE NEXT COMMON BLOCK

COMMONS NOT NAMED ARE FLAGGED (210)

IS THIS LABELLED (NAME(1), NAME(2), NAME/HBL) YES EXAMINE LIST OF MODULES THAT CONTAIN THIS BLOCK

NO

GETE (HDIR, 14) FETCH AND STORE SOURCE CODE INDEX

SCIND(1) = ER(3) STORE INDEX

GETE (HNOD, 17) FETCH FIRST STATEMENT NO.

STORE FIRST STATEMENT NO. FSTAT(1) = ER(13)

GETE (HNOD, 18) FETCH LAST STATEMENT NO.

STORE LAST STATEMENT NO. LSTAT(1) = ER(13)

SET ERROR NO. K = 210

WRITE ERROR MESSAGE TO FILE FLAG
UNIT MODULE DESCRIPTION

IDENTIFICATION

Modification to CONALC

STORAGE ALLOCATION (estimate)

Additional 500 hexadecimal bytes.

PURPOSE

Flag Common Block arrays that are not dimensioned in Common Block statements.

DESCRIPTION

The modification locates the source and writes a message to FLAG FILE to indicate that an array in a COMMON BLOCK is dimensioned elsewhere.

HOW ENTERED

The code occurs at the point where a Common Block variable is dimensioned other than in a Common Block.

DETAILED FLOWCHART

See attachments.
ARRAY IN COMMON BLOCK DIMENSIONED ELSEWHERE

GETL
(RF)
LOCATE LINE OF CODE

GETE
(HNOD' 16)
FETCH SOURCE CODE INDEX

SET SOURCE CODE INDEX
SCIND(1) = ER(13)

GETE
(HNOD' 17)
GET LOCATION OF STATEMENT

SET FIRST STATEMENT NO.
FSTAT(1) = ER(13)

GETE
(HNOD' 16)
GET LAST STATEMENT NO.

SET LAST STATEMENT NO.
LSTAT(1) = ER(13)

SET ERROR CODE
K = 270

WRITE MESSAGE TO FLAGFL

CONTINUE WITH PROGRAM

REPRODUCIBILITY OF ORIGINAL PAGE IS FCOR

SAI 0461
UNIT MODULE DESCRIPTION

IDENTIFICATION

ER275

STORAGE ALLOCATION (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag DIMENSION statement and variable which contains an adjustable (variable) dimension.

DESCRIPTION

This subroutine searches a module for statement type 28 (DIMENSION) then examines each array element for a use code 15 (Array dimension). When this condition exists the error is recorded. At the end of the statement search each array is flagged that has a variable dimension.

HOW ENTERED

Called by AIR subroutine if query is selected. There are no arguments.

CALLING SEQUENCE

Call ER275

OTHER ROUTINES CALLED

IE
GETE
GETL
TT
SET/USE PARAMETERS

USE

COMMON/FLAG/
FLAGFL -- I/O designator

COMMON/H/
HB -- hollerith B
HF -- hollerith F

COMMON/TABLE/
HDIR -- directory
HUSE2 -- linked Test Statement
HUSE1 -- linked List Use
HNOD -- Node Table

COMMON/SPEREG/
ER(10) -- table information turn array
FBR -- program flow register

SIGNIFICANT INTERNAL VARIABLES

I0, I1, I2, I3, I4, I5, I6, I7, I8, I9 = 0 - 9
K -- error flag number
SCIND -- module indicator
FSTAT -- first statement number of error
LSTAT -- last statement number of error
NUMOCC -- counter for error

RESTRICTIONS

All variables are IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached
UNIT MODULE DESCRIPTION

IDENTIFICATION

ER280

STORAGE ALLOCATION (estimate)

PURPOSE

Flags subroutine calls with constants and hollerith arguments and flags all occurrences where the same variable exists in multiple positions.

DESCRIPTION

This routine searches the parameter list of each CALL statement (statement type 34) for either equal variable symbols or constants (symbol class 8) and holleriths (symbol type 6) used as an entire argument (USE table use code 19 and 17).

HOW ENTERED

Called by AIR subroutine

CALLING SEQUENCE

Call ER280 (QNUM)
QNUM is the query number = 280 or 285

OTHER ROUTINES CALLED

IE
GETE
GETL
CONALP
POP
SET/USE PARAMETERS

USE

GLOBAL: COMMON/ALINFO/

COMMON/FLAG/

FLAGFL -- input/output designator

COMMON /H/

HB -- hollerith B
HF -- hollerith F

COMMON/SPEREG/

ER(10) -- Error registers
NOTE: Only ER(3) and ER(4) are set.
FBR -- forward/backward register

COMMON/TABLE/

HDIR -- directory table
HUSE1 -- use table
HSYM -- symbol table
HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

OVFLG -- Output variable from CONALP; table overflow
ERFLG -- Output variable from CONALP; error flag
PTR -- Input variable to CONALP; pointer to statement
I0, I1, ..., I9 -- represent integers 1 - 9
K -- error flag
SCIND -- source code indicator
FSTAT -- first statement number
LSTAT -- last statement number
BF -- general purpose flag
LIMITATIONS AND RESTRICTIONS

All variables are set by IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached
IF ICII INFORMATION FOR CONSTRUCTING PARAMETER LIST IS CONSTRUCTIONS PARAMETER LIST IN THE USE TABLE TO PASS TO CONALP

SET FORWARD FLOW FBR - HF

IF REG2, HF TRANSFER INTO USE 2 TABLE

IF NO ENTRY IF F02, TRY ANOTHER STATEMENT

IF I POINTER FBR - PR REGISTER

CONALP PTR1, 11, FSTA1, LSTA1, O/VFG, ERFG CONSTRUCT PARAMETER LIST FOR STATEMENT

WAS THERE AN ERROR IN CONALP? (ERFG = 1)

WRITE TO PRINTFL IF ROUTINE PARAMETER CHECK OVER

FBR - HB FLOW BACKWARDS

IS THERE AN OVERFLOW? (O/VFG = 1)

WRITE TO PRINTFL IF THAT CHECK MAY NOT BE VALID

CHECK FOR WHICH COMPARISON IF TO BE MADE, THEN SEARCH PARAMETER LIST FOR THAT SITUATION, ONLY ALIGNMENT LIST 1 IS USED.

IS THIR THE CONSTANT HOLLERTHT CHECK? YES 3

NO 4
UNIT MODULE DESCRIPTION

IDENTIFICATION

MULBRA additions

STORAGE ALLOCATION REQUIREMENT (estimate)

Total after additions 2K hexadecimal bytes

PURPOSE

Flag arithmetic IFs and flag GO TOs that are targets of other GO TOs.

DESCRIPTION

One addition to MULBRA will flag GO TO statements which are the targets of previous GO TO statements. The program will search for statement type 45 that has a use code of 9. When this occurs the statement is flagged with a message to the Flag File (220).

The second addition to MULBRA flags Arithmetic IFs. In this case an IF statement (statement type 10) with more than 2 branch targets is flagged by an error message to the flag file (220).

HOW ENTERED

Called by AIR upon query request.

CALLING SEQUENCE

CALL MULBRA (Number)
Number is query number 150, 200, or 220; all are processed in this subroutine.
OTHER ROUTINES CALLED

No additional routines are called.

SET/USE PARAMETERS

There are no additional parameters used or set.

SIGNIFICANT INTERNAL VARIABLES

USECD -- use code variable
NSUC -- number of targets
BFLAG -- branch flag

DETAILED FLOWCHART

Attached
INITIATIVI
CONSTANT ANI DIRECITION

INITIALIZE FOR ENTITY TO DIRECTORY

HAVE ALL MODULES BEEN EXAMINED?

NO

BRING MODULE INTO MAIN MEMORY

FIND SOURCE CODE INDEX

BRING IN A STATEMENT

ARE ALL EXAMINED?

NO

SET STATEMENT TYPE STATYP

SET BRANCH FLAG TO ZERO UFLAG = 0

SET TARGET COUNT I LAG TO ZERO NSUC = 0

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR
NOTE: ONLY THE ADDITIONS ARE "DIFF TO" DETAIL.
GETE
(HNOD, I2)
GET AN ELEMENT
FROM NODE TABLE

FELUSE
(I2, ER(3), A)

USE TRANSFER
THROUGH A VARIABLE VALUE
A(I3).EQ.23?

YES → 2

NO

1

C

D

SAI-0469
IF NOT ARITHMETIC IF (NUSC.LT.3), AND (STATYPE.EQ.10)

IS BRANCH FLAG SET

K = 200

WRITE MESSAGE TO FLAG FILE

K = 150

C

1

E

2

3

M

YES

NO

YES

NO

1

C

SAI:0473
UNIT MODULE DESCRIPTION

IDENTIFICATION
ER250 -- Error 250 Routine

STORAGE ALLOCATION (estimate)
2K (hexadecimal bytes)

PURPOSE
To flag variables used as I/O designators.

DESCRIPTION
This routine examines each line of code in the source decks for variables with the classification "scalar" (class code = 6) which are also used as I/O designators (use code = 26). Having found one which satisfies both requirements, an applicable message is written to the flag file.

MATHEMATICAL EQUATIONS/DEFINITIONS
None

HOW ENTERED
Called by AIR

CALLING SEQUENCE
Subroutine call, no arguments. Call ER250.

UNIT MODULE OR OTHER Routines CALLED
IE
GETE
GETL
TT
SET/USE PARAMETERS

SET

Global Common: COMMON/SPEREG/

ER(10) -- Error registers
NOTE: Only ER(3) and ER(4) are set.

FBR -- forward/backward register

USE

Global Common: COMMON/FLAG/

FLAGFL -- input/output designator

Global Common: COMMON/H/

HB -- hollerith B
HF -- hollerith F

Global Common: COMMON/TABLE/

HDIR -- directory table
HUSEL -- use table
HSYM -- symbol table
HNOD -- node table

SIGNIFICANT INTERNAL VARIABLES

I0, I1, ..., I9 -- represent integers 1 - 9
K -- error flag
SCIND -- source code indicator
FSTAT -- first statement number
LSTAT -- last statement number
BF -- general purpose flag
CONSTRANTS
All variables are set to integer by the IMPLICIT statement.

COMPUTATIONAL ACCURACY AND RANGE OF INPUT/OUTPUT VARIABLES
Non-Applicable

ERROR MESSAGES AND SUMMARY
None

DETAILED FLOWCHART
See following pages.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS FOR
GETL (BF)
BRING IN LINE
OF CODE TO
ANALYZE

GETE
(HDIR, 14)
FIND END OF
THIS LINE

SET END OF
LINE INDICATOR
SCIND = ER (13)

IE
(ISY, BF)
INITIAL ENTRY
TO SYMBOL
TABLE

SET FORWARD/
BACKWARD FLAG
FBR = HF

IF END
OF SYMBOL
TABLES?
DF.EQ.2

YES
SET FORWARD/
BACKWARD
FLAG
FBR = HB

NO
GETE
(ISY, 13)
GET A SYMBOL
TABLE ELEMENT

4

8

6

D

SAI 0433
END OF THIS LINE? ER(3) # 6

TT (HUSE1, BF)
GET USE CODE LOCATION FROM USE TABLE

END OF TABLE BF = 2

GET6 (HUSE1, 12)
GET ELEMENT FROM TABLE

IS THIS AN I/O UNIT NUMBER? ER(3) = 26

SET FBR = HB

SAI 0434
GETE (HNBO, 17)
GET BEGINNING OF NODE

FSTAT = ER (3)
FSTAT IS FIRST STATEMENT NUMBER

GETE (HNOD, 18)
GET END OF NODE

LSTAT = ER(3)
LSTAT IS LAST NODE STATEMENT NUMBER

FLAGFL
MESSAGE THAT VARIABLE IS I/O UNIT

1001 FORMAT (5(2X, 15), 2X, ER200-----',
3(2X, 15), 2X, 2A4)
I/O PARAMETER LIST:
I1, SCIND, FSTAT, LSTAT
K, IO, IO, IO, IO

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR.
UNIT MODULE DESCRIPTION

IDENTIFICATION

ER260 -- Error 260 routine

STORAGE ALLOCATION (estimate)

2K (hexadecimal bytes)

PURPOSE

Flag statement labels not in increasing numerical order.

DESCRIPTION

This routine locates and reconstructs each statement label in a module. If the current label is less than the previous label it is flagged.

HOW ENTERED

Called by AIR

CALLING SEQUENCE

Subroutine: Call ER260
There are no calling arguments

UNIT MODULE OR OTHER ROUTINES CALLED

SUBROUTINE: IE
GETE
GETL
CONVER

FUNCTION: FLD
### SET/USE PARAMETERS

<table>
<thead>
<tr>
<th>SET</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global:</strong> COMMON/SPEREG/</td>
<td><strong>GLOBAL:</strong> COMMON/FLAG/</td>
</tr>
<tr>
<td>ER(10)</td>
<td>Flagf1 -- I/O file</td>
</tr>
<tr>
<td>FBR</td>
<td></td>
</tr>
<tr>
<td>NOTE: Only ER(3) is set, the rest are not referenced.</td>
<td>COMMON/HOSTWD/</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMON/H/</td>
</tr>
<tr>
<td></td>
<td>HB -- hollerith B</td>
</tr>
<tr>
<td></td>
<td>HF -- hollerith F</td>
</tr>
<tr>
<td></td>
<td>COMMON/TABLE/</td>
</tr>
<tr>
<td></td>
<td>HDIR</td>
</tr>
<tr>
<td></td>
<td>HSYM</td>
</tr>
<tr>
<td></td>
<td>HNOD</td>
</tr>
</tbody>
</table>

### SIGNIFICANT INTERNAL VARIABLES

- IO, I1, ..., I9 -- integers 1 - 9
- LSAVE -- current label number
- K -- error number flag
- SCIND -- source code indicator
- FSTAT -- first statement number of error
- LSTAT -- last statement number of error
- BF -- condition flag; return argument
- LABEL -- next reconstructed label number
LIMITATIONS AND RESTRICTIONS

All variables are set to integer by the IMPLICIT statement.

DETAILED FLOWCHART

See Attached
FLAG STATEMENT LABELS NOT IN INCREASING NUMERICAL ORDER. SEARCH FOR STATEMENT LABEL. CLASS 5: CHANGE LABEL TO INTEGER;
USES LABELED COMMON/FLAG/,/H/,/HOSTWP/,/SAEREG/,/TABLE/
IMPLICIT INTEGER (A - Z)

SET STANDARD INTEGER VARIABLES IQ, 11, 12, 13, 14, 15, 16, 17, 18, 19 = 0, 1, ..., 9

SET LOCAL VARIABLE LSAVE = 0

SET FORWARD/BACKWARD FLAG FBR = HF

SET ERROR INDICATOR K = 260

IF (HDIR, RF)
BRING IN MODULE TO SEARCH

SET FORWARD/BACKWARD FLAG TO FORWARD FBR = HF

IF END OF MODULES LEAVE SUBROUTINE IF.EQ.127

NO

YES

RETURN

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR.
GETE (HDIR, 13)

END OF THIS MODULE ER(13), GT. 10?

SET FORWARD/BACKWARD FLAG FBR = HB

GETL (BF)
BRING IN A LINE OF CODE

GETE (HDIR, 14)
FIND SOURCE INDEX

SET SOURCE INDEX INDICATOR
SCIND = ER(13)

IE (HSYM, BF)
INITIAL ENTRY INTO SYMBOL TABLE

SET FORWARD/BACKWARD FLAG FBR = HF

END OF SYMBOL TABLE GF, EQ, 12?

END OF THIS MODULE ER(13), GT. 10?

SET FORWARD/BACKWARD FLAG FBR = HB
GET (HSYM, i)
GET SYMBOL TABLE ELEMENT

NOT A STATEMENT LABEL
ER11(13).NE.15?

NO

INITIALIZE COUNTER, DO VARIABLE, AND ANSWER VARIABLE
J = 0, LABEL = 0, I = 1

GET (HSYM, i)
GET ELEMENT FROM SYMBOL TABLE

CONVERT (ARRAY, COUNT)
CONVERT STRING & DECIMALS TO POSITIVE NUMERIC INTEGER VALUE

IS THIS LABEL LAST LABEL?

YES

GETE (HNOD, 17)
GET 1st STATEMENT NUMBER

FSTAT II(3)
SET INTO OUTPUT VARIABLE

SAI 0562

4-48
GETE (HNOD, IB)
GET LAST STATEMENT NO.

SET INTO OUTPUT VARIABLE LSTAT = ER(I3)

FLAGFL MESSAGE THAT STATEMENT VARIABLE OUT OF ORDER

SAVE CURRENT STATEMENT NUMBER LSAVE = LABEL

I/O PARAMETER LIST: I1, SCIND, FSTAT, LSTAT, K, IO, IO, IO, IO
UNIT MODULE DESCRIPTION

IDENTIFICATION
ER230  Error 230 routine

STORAGE ALLOCATION (estimate)
3K (hexadecimal bytes)

PURPOSE
Flag ASSIGN, PAUSE and assigned GO TO statements

DESCRIPTION
This routine searches a module for statement 38 (ASSIGN), 50 (PAUSE), or 45 (GO TO). When the first two are found, the error is written to Flag File immediately. The third one is checked further for a use code of 23 (transfer through a variable value). If the code is 23 a message is written to the Flag File also.

HOW ENTERED
Called by AIR

CALLING SEQUENCE
CALL ER230
Note: No arguments are used.

UNIT MODULE OR OTHER ROUTINES CALLED
IE
GETE
GETL
FELUSE
SET/USE PARAMETERS

<table>
<thead>
<tr>
<th>SET</th>
<th>COMMON/SPEGREG/</th>
<th>GLOBAL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL:</td>
<td>ER(10) -- error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>NOTE:</td>
<td>Only ER(3) is used</td>
<td></td>
</tr>
<tr>
<td>FER--</td>
<td>forward/</td>
<td></td>
</tr>
<tr>
<td>backward register</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>USE</th>
<th>COMMON/FLAG/</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL:</td>
<td>FLAGFL -- I/O de-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>signator</td>
<td></td>
</tr>
<tr>
<td>COMMON/H/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HB -- hollerith B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HF -- hollerith F</td>
<td></td>
</tr>
<tr>
<td>COMMON/TABLE/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HDIR -- directory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HNOD -- node table</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HSYM -- symbol</td>
<td></td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HUSEL -- use table</td>
<td></td>
</tr>
</tbody>
</table>

SIGNIFICANT INTERNAL VARIABLES

IO, I1, ..., I9 -- represent integers 1 - 9
K -- error flag
SCIND -- source code indicator
FSTAT -- first statement number
LSTAT -- last statement number
BF -- general purpose flag
STATYP -- statement type
A(10) -- array for return arguments

CONSTRAINTS

All variables are set to integer by IMPLICIT statement

DETAILED FLOWCHART

See attached
UNIT MODULE DESCRIPTION

IDENTIFICATION

ER290

STORAGE ALLOCATION REQUIREMENTS (estimate)

4K hexadecimal bytes

PURPOSE

Flag COMMON variables multiply defined in DATA statements.

DESCRIPTION

This routine examines all COMMON blocks. Each module containing that COMMON block is examined for DATA statements. At this point the parameter in the COMMON block is compared against the parameters in the DATA statements. A record is kept only for each module of multiple definitions. Multiple definitions will be written on the FLAG FILE.

HOW ENTERED

AIR subroutine calls this subroutine when query 290 is requested.

CALLING SEQUENCE

CALL ER290

ROUTINES CALLED

CONALC IE
CONALP POP
EQUIVL PUSH
GETE TT
SET/USE PARAMETERS

SET
GLOBAL: COMMON/SPEREG/
FBR -- program flow flag

COMMON/ALINFO/
NAME(2) -- name save variable
MNAME(2,2) -- module name save variable
SCIND(2) -- source code index
FSTAT(2) -- first statement number of source code modules
LSTAT(2) -- last statement number of source code modules
NUMOCC -- error counter

USE
GLOBAL: COMMON/ALI/
ALIGN (2, 300) -- alignment table
PLALI(2) -- alignment table indicators

COMMON/FLAG
FLAGFL -- I/O variable for error information

COMMON/H/
HB -- hollerith B
HF -- hollerith F

COMMON/LTS/
LISTAB(500) -- list of equivalenced variables
LLIS -- list table pointers
PLLIS -- list table pointers
MAP(6,20) -- list table map
PMap -- list table map pointer
PLMAP -- list table map pointer
COMMON/SPEREG/

ER(10) -- table information storage

USE

COMMON/TABLE/
HCOM -- hollerith COM for Common block table
HDIR -- hollerith DIR for Directory
HMAP -- hollerith MAP for Map table
HNOD -- hollerith NOD for Node table
HSYM -- hollerith SYM Symbol table
HUSE1 -- hollerith USE1 Use table
HUSE2 -- hollerith USE2 for table

SIGNIFICANT INTERNAL VARIABLES

AM -- Indicates which alignment table to be filled
OVFLAG -- overflow flag for tables
ERFLAG -- irretrievable error flag
ERINC -- counts 290 error occurrences

LIMITATIONS AND RESTRICTIONS

All variables are set with IMPLICIT INTEGER (A-Z)

DETAILED FLOWCHART

Attached

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR.
ER290

FLAGS COMMON VARIABLE THAT IS MULTIPLY DEFINED IN DATA STATEMENT – GLOBAL CHECK
IMPLICIT INTEGER (A-Z)
LABELLED COMMON/ALINFO/,/ALI/,/FLAG/,
/H/,/LTEMP/LIS/,/PRNT/TABLE

INITIALIZE INTEGER
CONSTANTS
i0, i1, ..., i9 = 0,
1, ..., 9

K = 290
SET ERROR
FLAG

FUR = HF
SET FLOW
FORWARD

IF
(HCOM, BFI

INTERNAL ENTRY
COMMON
BLOCK TABLE

RESET FLOW
TO FORWARD
FBR = HF

B. BLOCKS
EXHAUSTED
RETURN

NO

GETE
(HCOM, 11)
GET BLOCK
NAME

SAVE THIS
BLOCKNAME
NAME(1) = ERI(13)
NAME(2) = ERI(14)

SET ALIGN
TABLE INDICATOR
to 1
AM = 1

COMMON BLOCK
PARAMETER TABLE
GOES IN FIRST
PARAMETER, TABLE

SET FLOW
FLAG TO
FORWARD
FBR = HF

SAI-504

4-57
(HDIR, I(F)
TRANSITION TO DIRECTORY

SET FLOW FLAG FOR HF

BF.EQ.2
CAN'T LOCATE MODULE IN DIRECTORY?

GETE (HDIR, I4)
GET MODULE SOURCE NO.

SET SOURCE INDEX CODE
SCIND(2) = EP (I3)

GETC (HDIR, I3)
SET TO BRING MODULE INTO MEMORY

GETL (BF)
BRING MODULE INTO MEMORY

IF (HNOD, I(F)
INITIALIZE ENTRY TO NODE

ALL STATEMENTS CHECKED
BF.EQ.2

YES

NO

SAI 0487
K

THIS SAME AS PARAMETER_NAME?

YES

NO

ERCNT = ERCNT + 1 INCREASE COUNTER

ANY DUPS FOR THIS PARAMETER? ERCNT

YES

WRITE TO FLAGFL COMMON BLOCK, VARIABLE, MODULE STATEMENT NO. DEFINED

NO

SET FLOW TO BACK

I

6

I

6

K

IS THERE ANY IN EQUIVL THAT EQ?

YES

NO

NO 6 NO

I

IS THERE ANY IN EQUIVL THAT EQ?

YES

NO

FORMAT (5(2x, 15), 'ER290.', 3(2x, 15), 1A4)
IOLIST1 12, 10, 10, K 10, 10, 10, NAME IO LIST 2 12, FSTAT(2), LSTAT(2), K, ERCNT, 12, 10, VNAME
5. REFERENCES


