AD-A052 730 CHARLES STARK DRAPER LAB INC CAMBRIDGE MA F/G 9/2
JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD). VOLUME II.(U)
FEB 78 G GODDARD, M WHITWORTH, E STROVINK F30602-76-C-0408
UNCLASSIFIED R-1120-VOL-2 RADC-TR-78-9-VOL-2 NL
1 OF 1
AD
A052730
END
DATE
FILMED
5-78
DDC

AD A052730

RADC-TR-78-9, Vol II (of four)
Final Technical Report
February 1978

JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD)

G. Goddard
M. Whitworth
E. Strovink

The Charles Stark Draper Laboratory, Inc.

DDC FILE COPY

Approved for public release; distribution unlimited

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-9, Vol II (of four) has been reviewed and is approved for publication.

APPROVED: *Donald VanAlstine*

DONALD VANALSTINE
Project Engineer

APPROVED: *Wendall C. Bauman*

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: *John P. Huss*

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIM) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

18 RADC 19 TR-78-9-VOL-2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-78-9, Vol II (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD). Volume II.		5. TYPE OF REPORT & PERIOD COVERED 9 Final Technical Report. September 76 - October 77
		6. PERFORMING ORG. REPORT NUMBER 14 R-1120-VOL-2
7. AUTHOR(s) 10 G. Goddard, M. Whitworth, E. Strovink		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-76-C-0408
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 62702F J.O. 55811412
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE 11 February 1978
		13. NUMBER OF PAGES 65 12 68P.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) 16 5581 Same 17 14		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

DDC APR 17 1978 F

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald VanAlstine (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Structured Programming
Structured Design Diagram
Structured Extension
Parse
Parser Generator
Preprocessor
Flowcharter
JOVIAL J3
Invocation Diagram

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This final report, describing the implementation of the prototype JOVIAL Structured Design Diagrammer (JSDD) presents the techniques used in the design of the system and details the usage of Structured Design Diagrams (SDDs) and Invocation Diagrams. The utility of the JSDD output is that it provides a graphic portrayal of the nested logical sequences that define the structure of a computer program. The JSDD should be integrated into a comprehensive analysis and documentation system in order for it to realize its full potential. The JSDD is implemented in JOVIAL J3 as a three program system designed to run on a Honeywell

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408 386 JOB

6180 computer under the GCOS Operating System Version 1/G. It was developed on the GCOS Encapsulator under the MULTICS operating system. The first program in the system is a LALR(k) parsing technique formalized by DeRemer (and implemented by Lalonde). However, rather than outputting object code, the program creates a data base from which the other programs create SDDs and Invocation Diagrams. The JSDD system processes programs written in JOVIAL J3 with or without the structured extensions (which are described in Section 6).

FINAL REPORT

This document was produced to satisfy the requirements of contract number F30602-76-C-0408 with the Rome Air Development Center. It is one of four companion volumes:

* JOVIAL Structured Design Diagrammer (JSDD) Report Summary

  This document is a summary of the contents of the JSDD Final Report.

* JOVIAL Structured Design Diagrammer (JSDD) Final Report

  This volume presents the design techniques for implementing the JSDD and describes the use of Structured Design Diagrams.

* JOVIAL Structured Design Diagrammer (JSDD) Program Description

  This volume presents a detailed description of the program implementation for purposes of maintaining and/or modifying the JSDD.

* JOVIAL Structured Design Diagrammer (JSDD) User's Manual

  This volume presents the user's view of the JSDD along with user options and other information about running the program.

ACCESSION for
NTIS
DOC B.ff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
SPECIAL
A

Acknowledgement

This report was prepared by The Charles Stark Draper Laboratory, Inc., under Contract F30602-76-C-0408 with the Rome Air Development Center at Griffis Air Force Base.

Especial credit is due Margaret Hamilton, who pioneered principles of Structured Programming at Draper Laboratory. Saydean Zeldin originally suggested the symbology implemented in the output of the JOVIAL Structured Design Diagrammer. Thanks should go also to William Daly, who created the Structured Design Diagrammer for the HAL language (currently being used on the NASA Space Shuttle project). The authors are indebted to Victor Voydock for his invaluable assistance in implementing a complete MULTICS user interface which was used successfully for the duration of the JSDD implementation. The authors are also grateful to J. Barton DeWolf whose many suggestions were of great assistance throughout this effort.

TABLE OF CONTENTS

Section	Page
1. Introduction	6
2. Structured Design Diagram Description	8
3. Invocation Diagram Description	14
4. Design of the JOVIAL Structured Design Diagrammer	17
5. Defining the JOVIAL J3 Syntax	20
6. The Structured Extensions to JOVIAL J3	37
7. Conclusions and Recommendations	42
8. References	45
Appendix A. The Deliverables	

1. Introduction

In recent years, the digital computer software industry has directed considerable effort toward the development of design and implementation methodologies to ensure the sufficiency, reliability, and maintainability of software systems. The most widely known product of this effort is the loosely defined set of design and programming practices called "Structured Programming" (see references 1, 2, and 3).

Structured Programming does not constitute a complete software development methodology. Rather, it is a collection of general guidelines for use by software designers and implementors. As such, it provides no uniform approach to system design and offers no method of evaluating system sufficiency with respect to requirements or design. Despite these shortcomings, adherence to Structured Programming principles can be of great assistance in producing software systems which are reliable and intellectually manageable.

The techniques of Structured Programming are sufficiently general to allow system developers a tremendous amount of stylistic freedom. However, the generality of the techniques has made the development of a standard approach to software analysis extremely difficult. The prototype JOVIAL Structured Design Diagrammer (JSDD) is the first component of an integrated software analysis and documentation system which will address itself to this task.

The JSDD is an automated analysis and documentation system which produces two types of diagrams: Structured Design Diagrams (SDDs) and Invocation Diagrams. SDDs provide a graphic display of program control logic. Invocation Diagrams are a display of a software system's functional (calling) structure.

The JSDD processes digital computer programs written in either JOVIAL J3 or Extended JOVIAL J3. Extended JOVIAL J3 is standard JOVIAL J3 as specified in reference 4 with the addition of structured extensions (see Section 6) which are based upon reference 5.

The symbology employed in the Structured Design Diagrams emerged as a result of research at Draper Laboratory directed toward the development of a software system

specification methodology. This symbology is inherently suited to portray the nested logical sequences in a modern structured computer language. Because the symbology is tailored to structured programs, its utility diminishes when it is used to diagram a program that makes liberal use of Goto statements or other unstructured language constructs. Since JOVIAL J3 lacks certain structured programming mechanisms which eliminate the need for Goto statements, these mechanisms have been added as allowable programming features in programs submitted to the JSDD. It is assumed that such programs would have to be subjected to a preprocessor before submission to a JOVIAL J3 compiler. Such a preprocessor has been supplied as a deliverable item with the JSDD. Sections 5 and 6 discuss the JOVIAL J3 syntax, the JOVIAL J3 structured extension syntax, and the preprocessor.

The JSDD is implemented in JOVIAL J3 as a three program system that is designed to run on a Honeywell Information Systems, Inc., Series 6000 computer supporting GCOS Version I/G. The implementation work was conducted on the Rome Air Development Center's MULTICS computing facility via the ARPA Network. Development work was performed on the GCOS Encapsulator which is available under the MULTICS operating system. The MULTICS environment provided most of the software tools that were employed during the implementation. Appendix A contains a list of the deliverable items produced under this contract.

The Final Report is organized as follows: Section 2 discusses the use of Structured Design Diagrams (SDDs); Section 3 describes the utility of Invocation Diagrams; Section 4 details the design of the JSDD; Section 5 defines the JOVIAL J3 BNF grammar; Section 6 discusses the structured extensions and the preprocessor; and Section 7 contains conclusions and recommendations for further work.

2. Structured Design Diagram Description

Structured Design Diagrams (SDDs) provide a graphic two dimensional display of the nested logical sequences that define the structure of a computer program. SDDs for JOVIAL J3 are constructed from the two basic structural elements shown in figure 2-1.

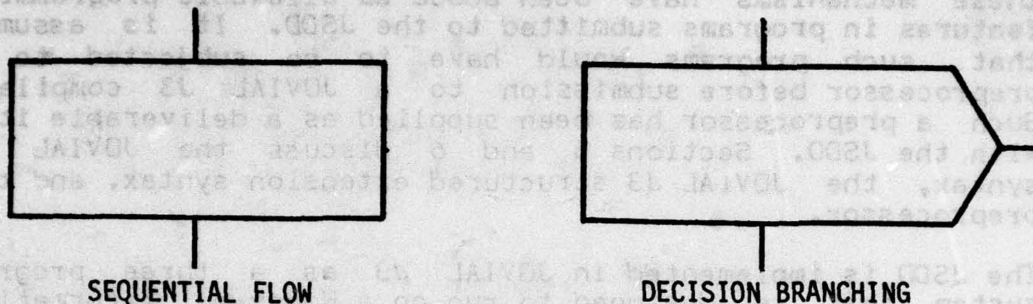

Figure 2-1. SDD Primitives

The rectangular box is used to contain elements that are executed in sequence. Control enters from the top or left side of the rectangle. Each element in a rectangle is executed in sequence and control flows through the bottom of the box.

The pentagonal box is used to contain two types of JOVIAL constructs: module heads and decision making elements. A module head is a <PROGRAM HEAD>, <PROC DESCRIPTOR> or <CLOSE HEAD> (see the syntax definition of JOVIAL in Section 5). Control passes through the bottom of a module head's pentagonal box to the module's code body. Figures 2-2, 2-3 and 2-4 illustrate SDD representations of module heads.

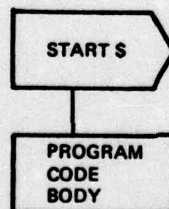

Figure 2-2. SDD representation of <PROGRAM HEAD>

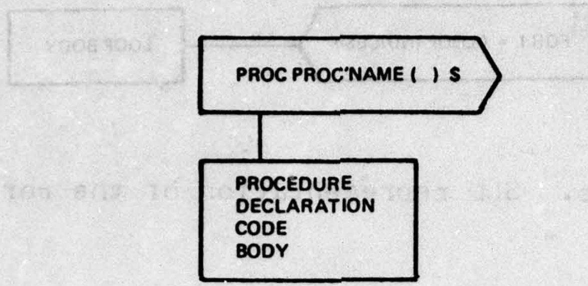

Figure 2-3. SDD representation of <PROC DESCRIPTOR>

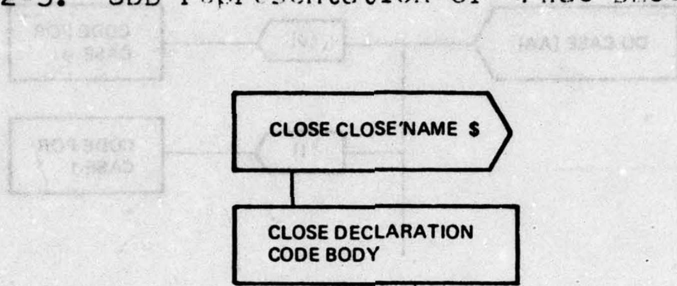

Figure 2-4. SDD representation of <CLOSE HEAD>

A decision making element is a JOVIAL construct which directs the flow of control to one of two paths. Evaluation of the contents of the pentagonal box determines the path to which control is passed. Figures 2-5, 2-6, 2-7 and 2-8 illustrate the SDD representaions of JOVIAL's decision making elements. Non-standard decision making elements have been introduced as structured extensions to JOVIAL J3. The structured extensions are the Do While Loop, the Do Until Loop and the Case Statement. Full descriptions of these new constructs are available in Section 6.

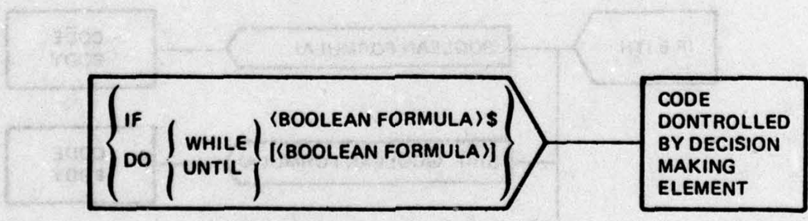

Figure 2-5. SDD representation of the If Statement. Do While Loop and Do Until Loop

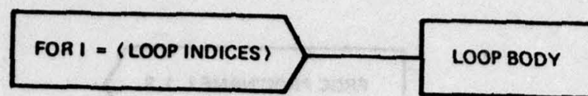

Figure 2-6. SDD representation of the For Loop

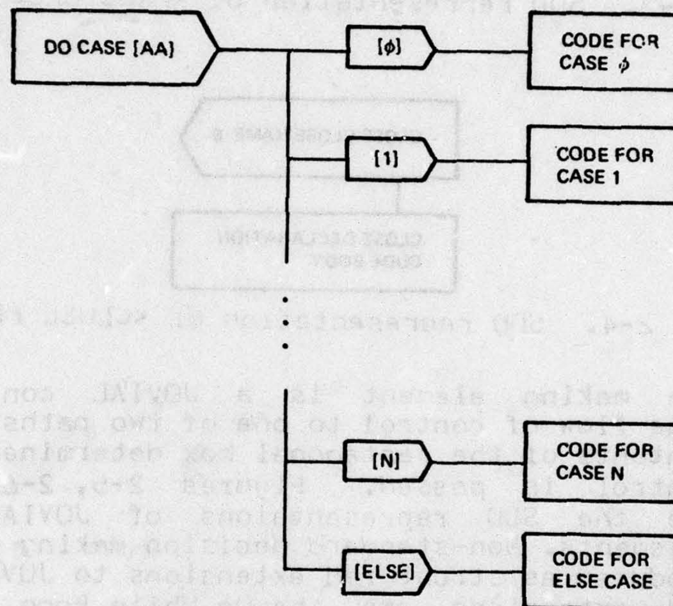

Figure 2-7. SDD representation of the Do Case Statement

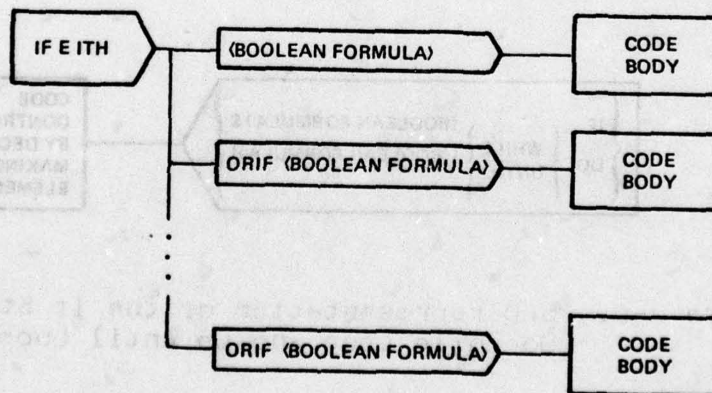

Figure 2-8. SDD representation of the Alternative Statement

Pentagonal boxes containing decision making elements are entered from the top or from the left. In general, they are exited by taking the horizontal path (to the right) or by taking the vertical path (through the bottom of the pentagon). The horizontal path is taken if the decision element is evaluated to be TRUE. Otherwise, the vertical path is taken. Note that the SDD representations of the Do Case Statement and Alternative Statement contain ''DO CASE'' and ''IFEITH'' decision making elements. These elements are always evaluated to be TRUE.

If a decision making element is evaluated to be false and its pentagon has no vertical path, then the SDD execution path must be retraced until a pentagon is found which has an unexecuted vertical path. If such a pentagon is found, then the vertical path must be followed. If no such pentagon is found, then the execution of the module has been completed.

It is important to note that Goto Statements appear in rectangular boxes and their effect upon a program's flow of control is not illustrated by an SDD. Restrained usage of the Goto statement will result in SDDs which will better illustrate a program's flow of control. The structured extensions to JOVIAL J3 (see Section 6) were introduced to minimize the JOVIAL programmer's reliance upon the Goto Statement.

Occasionally, the level of nesting (of decision making elements) in a program makes it impossible to display a code block in the available number of page columns. In such cases, it is necessary for the JSDD to create what is referred to as a stump. A stump is a diagram continuation. When the width of a page is such that the display of a code block can not be accommodated, that code block's logical position in the SDD is filled with a stump reference display. The stump reference display consists of a stump reference number by which the diagram continuation can be located. If the HEADING option is on (see Section 5.2 of the JSDD User's Manual), then the stump reference number is the number of the page on which the continuation appears. Otherwise, the stump reference number is the stump's sequence number.

The JSDD recognizes three types of comments: in-line comments, type-1 (or same line) comments and type-2 (or C-type) comments. In-line comments are comments which are embedded in a JOVIAL statement. In-line comments are

displayed in their embedding statements in SDDs. A type-1 comment is a comment which begins on the same line as a JOVIAL statement in the input file. In SDDs, type-1 comments appear below the statements to which they refer. A type-2 comment is a comment which appears by itself in the input file. SSDs display type-2 comments next to the line which connects the code blocks which precede and succeed the comment.

Figure 2-9 is a sample page from a design diagram produced by the JSDD system.

```
C S DRAPER LABORATORY JOVIAL STRUCTURED DESIGN DIAGRAMMER                                    PAGE 151
DESIGN DIAGRAM OF TRANSFER'WRITE3

*IFEITH *---*NEW'FILE3 $ *----* ** TRANSFER FROM FILE3'2 TO
                                  FILE3'1 **
                                *OPEN INPUT FILE3'2 $ *
                                *OPEN OUTPUT FILE3'1 $ *
                                *FOR I = 0, 1, F3'EMPTY - 1 $ *---*IFEITH *---*I EQ F3'BLK $ *---* 153*
                                                                  *--*ORIF 1 $ *---*POS(FILE3'2) = I $
                                                                                   *INPUT FILE3'2 TMPI, TMP'BUF $ *
                                                                                   *OUTPUT FILE3'1 TMPI, TMP'BUF $*
                                *SHUT INPUT FILE3'2 $ *
                                *SHUT OUTPUT FILE3'1 $ *
             *--*ORIF 1 $ *----* ** TRANSFER FROM FILE3'1 TO
                               FILE3'2 **
                              *OPEN INPUT FILE3'1 $ *
                              *OPEN OUTPUT FILE3'2 $ *
                              *FOR I = 0, 1, F3'EMPTY - 1 $ *---*IFEITH *---*I EQ F3'BLK $ *---*OUTPUT FILE3'2 RECS'IN'BLK3,*
                                                                                             * F3'BUF $
                                                               *--*ORIF 1 $ *---*POS(FILE3'1) = I $
                                                                                *INPUT FILE3'1 TMPI, TMP'BUF $ *
                                                                                *OUTPUT FILE3'2 TMPI, TMP'BUF $*
                                                                                * OUTPUT FILE3'2 TMPI, TMP'BUF *
                              *SHUT INPUT FILE3'1 $ *
                              *SHUT OUTPUT FILE3'2 $ *
```

Figure 2-9. Sample page of SDD output

3. Invocation Diagram Description

The Invocation Diagrammer produces two different outputs: (1) a list of procedures that are members of one or more recursive invocation loops, and (2) the Invocation Diagram itself.

The first output, if it appears on the diagram, occurs before the actual diagram under the heading "ULTIMATELY SELF-RECURSIVE." Under it are listed all procedures that call themselves, either directly or indirectly. An example of a direct recursive call is a procedure which contains, as part of its code, a call to itself. Indirectly recursive calls are best illustrated, again, by an example. Suppose procedure A can call procedure B which can call procedure C. If, as part of its code, procedure C contains a call to procedure A, all three procedures (A, B, and C) can theoretically call themselves.

The Invocation Diagram supplies recursion information for two reasons. First, the diagrammer has to detect recursive procedures because its algorithm for producing the Invocation Diagram is itself recursive. A recursive procedure could thus cause the diagrammer to diagram forever. Therefore, recursive procedures are only expanded once in the diagram, and thereafter are simply printed and flagged. Since this affects the readability of the diagram, the recursion information ought to be summarized for the user.

Secondly, it can be argued that recursion has no place in JOVIAL programs (JOVIAL J3 does not support recursion), and thus should be banned. The recursion information can be thought of as a warning to the programmer that illegal recursion is a possibility in the submitted program structure. Note that the Invocation Diagrammer only reports the possibility of recursion - it is quite possible that the program in question avoids actually making a recursive call.

The second output is the Invocation Diagram itself. The diagram comes in two parts: a main procedure diagram and diagrams of continuations and independent routines. The diagram of the main procedure, if it occurs, occurs first. Invocation Diagrams are quite simple to read - all procedure names which are connected horizontally to a vertical line are called by the procedure whose name started the vertical line. If the main program exists, it is the top level of

the diagram. For example:

```
/
/--PROC1
/  /
/  /--PROC2+
/  /
/  /--PROC3
/  /  /
/  /  /--PROC4
/  /
/  /--PROC5*
/
```

In this example, PROC1 calls PROC2, PROC3, and PROC5. Additionally, we see that PROC3 calls PROC4, and some invisible procedure calls PROC1. That procedure is at the top level of this particular diagram, because PROC1 hangs off the leftmost vertical line possible. There is still more information here: we know PROC2 is an external procedure because it is flagged with a "+". PROC5 is recursive - it is flagged with a "*".

Main diagram continuations (caused by running off the right side of the page) and independent procedures (procedures not called by any of the procedures which are directly or indirectly called by the main program) occur at the end of the diagram, under the heading "CONTINUATIONS AND INDEPENDENT ROUTINES." Continuations consist of "stumps" which correspond to similar "stumps" in the main diagram. If confronted by:

```
/
/-------3
```

in the main program, the user can find the continuation below, which starts with:

```
-------3
  /
  /
 etc.
```

Stump continuations occur in numerical order, after independent procedure diagrams.

The aim of the diagrammer is to diagram all procedures,

regardless of whether they are called (directly or indirectly) by the main program. However, the fact that a procedure is not called directly or indirectly by the main program is not a guarantee that it will appear as an "independent" procedure. It is quite possible that a second "independent" procedure whose diagram is output before that of the first procedure may call the first. In that case, the first procedure's "independent" diagram is suppressed. Nevertheless, the first procedure's diagram will have been generated as part of the second procedure's diagram. No procedure will ever remain both uncalled and undiagrammed.

4. Design of the JOVIAL Structured Design Diagrammer

Figure 4-1 depicts the (conceptual) two pass construction of the JSDD system. "Pass 1" (consisting of the Design Diagram Database Generator) performs the tasks of analyzing the syntax of an input program and creating a data base for use by the second pass. "Pass 2" (consisting of the Design Diagram Generator and the Invocation Diagrammer) uses the data base created by Pass 1 to construct Structured Design Diagrams (SDDs) and Invocation Diagrams. A two pass design is motivated by two factors. First, it is desirable to separate language dependent functions from language independent functions. Such a separation facilitates the adaptation of Pass 2 to target languages other than JOVIAL J3. Second, the two pass design provides a great deal of flexibility in the formatting of diagrams. Pass 2 of the JSDD can produce diagrams having a wide variety of formats from the data base produced by a single Pass 1 execution.

The language dependent first pass uses the powerful LALR(k) parsing technique formalized by DeRemer (see Reference 6) and implemented by Lalonde (see Reference 7). It is based on the compiler structure introduced by McKeeman, et. al. (see reference 8).

In parsing input source programs, Pass 1 acts as a table-driven deterministic pushdown automaton (DPDA). The tables which drive the Pass 1 parse are the product of an LALR(k) parser generator that accepts a syntactic description of a language as input and outputs parsing tables for the language (see Section 5).

As is the case in most modern compilers, Pass 1 generates output as the result of a parser decision to recognize a high level programming construct. The parser is guided in this decision by the tables described above. However, instead of object code, which is what a compiler would ordinarily produce, Pass 1 outputs a diagramming data base to be used by Pass 2.

The two Pass 2 programs (the Design Diagram Generator (DDG) and Invocation Diagrammer) interpret the Pass 1 generated data base, and create diagrams in accordance with the formatting specifications in the DDG options compool (see User's Manual). The DDG is implemented as a two part program. First it maps out the Structured Design Diagram and creates a temporary data base containing the mapping

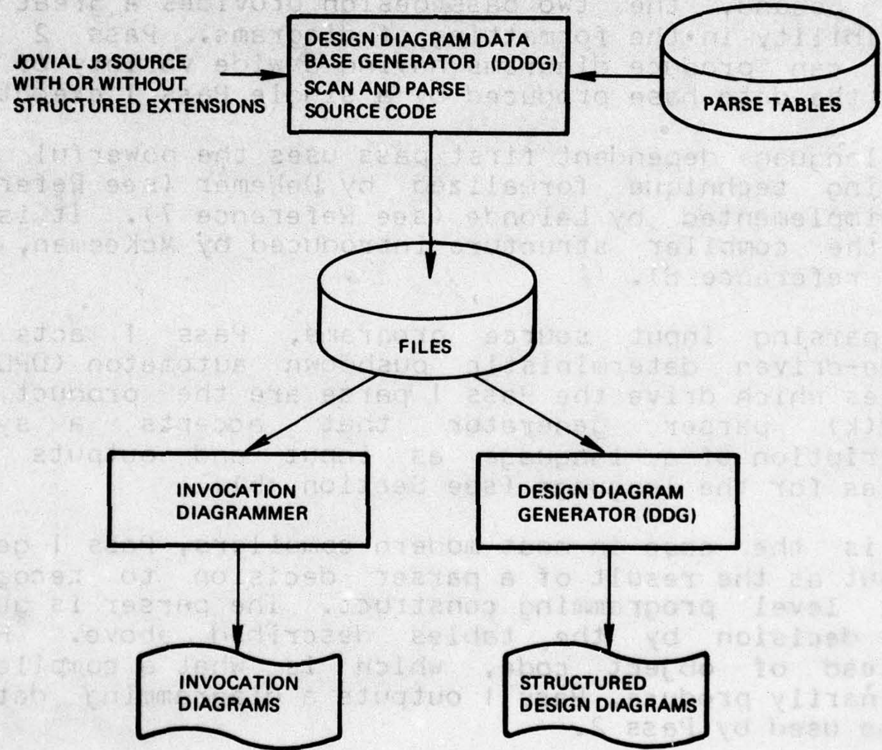

Figure 4-1. Two-Pass Construction of the JSDD

information. Only then does it produce the actual diagram. This strategy allows it to calculate forward and backward referencing information (in case a diagram overflows the page width) and a Table of Contents, without committing prematurely to any hard-copy output. If adjustments need to be made in the diagram, the temporary data base can be modified easily.

The Invocation Diagrammer uses a simple recursive strategy to create the Invocation Diagram. As the first procedure name is written out, the next procedure it calls is written beneath and to the right of it. The information for the first procedure is stacked, and the new procedure is treated as though it were the first. When no procedures are called by the current rightmost procedure name, the stack is popped back and the next rightmost procedure name is written out. When no more procedures are on the stack, the diagram is complete.

The Invocation Diagrammer and DDG have been implemented as separate programs to enhance system maintainability.

5. Defining the JOVIAL J3 Syntax

The syntactic analysis of JOVIAL J3 source programs is performed by a table driven parser in Phase I of the JSDD.

The tables are generated by an LALR(k) parser generator. The parser generator accepts the syntactic description of a language (in BACKUS-NAUR form) and produces the parsing tables for the language.

The parser generator can produce tables for any LALR(k) grammar where k is finite.

A (context-free) grammar can be defined loosely as an alphabet and a set of productions of the form:

A ::= xy

where A, x and y are members of the alphabet (the alphabet is the set of all symbols used to define the syntax of a language). The phrase ''can be replaced by'' can be substituted for the symbol ::=. Members of the alphabet which appear on the left hand sides of productions are called non-terminals and those that appear on only the right hand sides of productions are called terminals. In the set of non-terminals, there is one designated symbol called the goal symbol (in the case of JOVIAL J3's grammar, the goal symbol is <PROGRAM>).

A sentence is defined to be a string of terminal symbols generated by the grammar. A sentence can be generated by starting with the goal symbol and then, continually replacing each nonterminal with its definition (i.e., its right hand side) until no nonterminals appear in the string. The resulting string is a sentence.

The sequence of replacements involved in generating a sentence can be represented by a tree (called the parse tree) which is unique for each sentence generated by an unambiguous grammar.

There is no grammar processable by a left to right parser generator (having an finite upper bound on required lookahead) which generates all JOVIAL J3 programs and which generates no sentences which are not JOVIAL J3 programs.

Consider the following JOVIAL J3 subgrammar:

```
<PROCEDURE CALL> ::= <IDENTIFER> ( <FORMULA> ) $
<FORMULA> ::= <BOOLEAN FORMULA>
            | <NUMERIC FORMULA>
            | <ENTRY FORMULA>
<BOOLEAN FORMULA> ::= 0
<NUMERIC FORMULA> ::= 0
<ENTRY FORMULA> ::= 0
```

The sentence:

```
PROC'NAME (0) $
```

has three distinct parse trees. They are shown below.

```
1)                 <PROCEDURE CALL>
                          |
     ---------------------------------------------
     |                |       |         |        |
<IDENTIFIER>          (   <FORMULA>     )        $
                              |
                      <BOOLEAN FORMULA>
                              |
                              0

2)                 <PROCEDURE CALL>
                          |
     ---------------------------------------------
     |                |       |         |        |
<IDENTIFIER>          (   <FORMULA>     )        $
                              |
                      <NUMERIC FORMULA>
                              |
                              0

3)                 <PROCEDURE CALL>
                          |
     ---------------------------------------------
     |                |       |         |        |
<IDENTIFIER>          (   <FORMULA>     )        $
                              |
                       <ENTRY FORMULA>
                              |
                              0
```

In order to sidestep this problem (and several other similar problems), the JOVIAL J3 grammar has been "expanded" so that all JOVIAL J3 programs can be parsed. However, as a

result of this expansion, there are sentences which the grammar can generate which are not valid JOVIAL J3 programs.

The solution which has been adopted to the problem described above is to thread all of the formulae together. That is, redefine the grammar in this fashion:

```
<PROCEDURE CALL> ::= <IDENTIFIER> ( <FORMULA> ) $
<FORMULA> ::= <BOOLEAN FORMULA>
<BOOLEAN FORMULA> ::= <ENTRY FORMULA>
<ENTRY FORMULA> ::= <NUMERIC FORMULA>
<NUMERIC FORMULA> ::= 0
```

The obvious drawback to this redefinition is that every <NUMERIC FORMULA> can be accepted as an <ENTRY FORMULA> and as a <BOOLEAN FORMULA> (and so on). The advantage is that it will avoid an ambiguity in the grammar and allow the parser generator to produce parsing tables. (Many modern languages such as XPL and ALGOL use an approach similar to this in their syntax definitions.)

The syntax of JOVIAL J3 is relatively complex for a programming language. This complexity requires that JOVIAL's BNF description contain a very large number of productions. In attempting to generate parsing tables, it was found that the size of JOVIAL's BNF description exceeded an internal limit imposed upon the parser generator's input grammars. In order to avoid a costly investigation of the parser generator's limits, work on the JSDD was based on an early, slightly inaccurate version of the JOVIAL grammar. The Design Diagrammer Data Base Generator has been adjusted to enable it to successfully parse the full set of valid JOVIAL programs.

The JOVIAL J3 grammar used by the JSDD is presented below.

```
1   <PROGRAM> ::= <PROGRAM HEAD> <ELEMENT LIST>
1                 <PROGRAM TAIL> _!_

2   <PROGRAM HEAD> ::= START $
3                    ! CLOSE <IDENTIFIER> $ START $
4                    ! START PROC <IDENTIFIER> $
5                    ! START PROC <IDENTIFIER>
5                      <FORMAL PARAMETER LIST> $

6   <PROGRAM TAIL> ::= TERM $
```

```
7                          | TERM <IDENTIFIER> $

8    <ELEMENT LIST> ::= <ELEMENT>
9                     | <ELEMENT LIST> <ELEMENT>

10   <ELEMENT> ::= <STATEMENT>
11               | <DECLARATION>
12               | <DIRECTIVE>

13   <DIRECTIVE> ::= <DEFINE DIRECTIVE>
14                 | <MODE DIRECTIVE>

15   <DEFINE DIRECTIVE> ::= <DEFINE HEAD> <″> <TEXT> ″ $

16   <DEFINE HEAD> ::= DEFINE <IDENTIFIER>

17   <″> ::= ″

18   <TEXT> ::= <CHARACTERS>
19            | <TEXT> <CHARACTERS>

20   <MODE DIRECTIVE> ::= MODE <ITEM DESCRIPTION> $
21                      | MODE <ITEM DESCRIPTION> P
21                        <CONSTANT> $

22   <STATEMENT> ::= <SIMPLE STATEMENT>
23                 | <LABEL> <SIMPLE STATEMENT>
24                 | <COMPLEX STATEMENT>
25                 | <LABEL> <COMPLEX STATEMENT>
26                 | <COMPOUND STATEMENT>
27                 | <LABEL> <COMPOUND STATEMENT>

28   <LABEL> ::= <IDENTIFIER> .
29             | <LABEL> <IDENTIFIER> .

30   <DECLARATION> ::= <DATA DECLARATION>
31                   | <PROCESSING DECLARATION>

32   <PROCESSING DECLARATION> ::= <SWITCH DECLARATION>
33                              | <PROGRAM DECLARATION>
34                              | <CLOSE DECLARATION>
35                              | <PROCEDURE DECLARATION>
36                              | <COMMON DECLARATION>

37   <COMMON DECLARATION> ::= <COMMON HEAD>
37                            <COMPOUND STATEMENT>
```

```
38   <COMMON HEAD> ::= COMMON <IDENTIFIER> $
39                   | COMMON $

40   <SWITCH DECLARATION> ::= <SWITCH HEAD> <ITEM TAIL>
41                          | <SWITCH HEAD> <INDEX TAIL>

42   <SWITCH HEAD> ::= SWITCH <IDENTIFIER>

43   <ITEM TAIL> ::= ( <IDENTIFIER> ) = ( <ITEM LIST> ) $

44   <ITEM LIST> ::= <CONSTANT> = <GOTO FORMULA>
45                 | <ITEM LIST> , <CONSTANT> =
45                   <GOTO FORMULA>

46   <CONSTANT> ::= <LITERAL CONSTANT>
47                | <STATUS CONSTANT>
48                | <NUMERIC CONSTANT>

49   <INDEX TAIL> ::= = ( ) $
50                  | = ( <GOTO LIST> ) $

51   <GOTO LIST> ::= <GOTO FORMULA>
52                 | <GOTO LIST> , <GOTO FORMULA>
53                 | ,
54                 | <GOTO LIST> ,

55   <GOTO FORMULA> ::= <IDENTIFIER>
56                    | <IDENTIFIER> ($ <INDEX LIST> $)

57   <PROGRAM DECLARATION> ::= PROGRAM <IDENTIFIER> $

58   <CLOSE DECLARATION> ::= <CLOSE HEAD>
58                           <COMPOUND STATEMENT>

59   <CLOSE HEAD> ::= CLOSE <IDENTIFIER> $

60   <PROCEDURE DECLARATION> ::= <BLOCK HEAD>
60                               <COMPOUND STATEMENT>

61   <BLOCK HEAD> ::= <PROC DESCRIPTOR>
62                  | <BLOCK HEAD> <DIRECTIVE>
63                  | <BLOCK HEAD> <DATA DECLARATION>
64                  | <BLOCK HEAD> <PROGRAM DECLARATION>

65   <PROC DESCRIPTOR> ::= <PROC HEAD> $
66                       | <PROC HEAD>
66                         <FORMAL PARAMETER LIST> $
```

```
67   <PROC HEAD> ::= PROC <IDENTIFIER>

68   <FORMAL PARAMETER LIST> ::= ( )
69                             | ( <PARAMETER LIST> )
70                             | ( <PARAMETER LIST> =
70                               <PARAMETER LIST> )
71                             | ( = <PARAMETER LIST> )

72   <PARAMETER LIST> ::= <IDENTIFIER>
73                      | <PARAMETER LIST> , <IDENTIFIER>
74                      | <IDENTIFIER> .
75                      | <PARAMETER LIST> , <IDENTIFIER> .

76   <COMPOUND STATEMENT> ::= <BEGIN> <END>
77                          | <BEGIN> <ELEMENT LIST> <END>

78   <CONSTANT LIST> ::= <CONSTANT>
79                     | <CONSTANT LIST> <CONSTANT>

80   <COMPLEX STATEMENT> ::= <CONDITIONAL STATEMENT>
81                         | <ALTERNATIVE STATEMENT>
82                         | <LOOP STATEMENT>
83                         | <DIRECT STATEMENT>
84                         | <STRUCTURED EXTENSION>

85   <CONDITIONAL STATEMENT> ::= <IF CLAUSE> <THEN CLAUSE>

86   <ALTERNATIVE STATEMENT> ::= <IFEITH CLAUSE>
86                               <THEN CLAUSE> <ALT LIST>
86                               <END>

87   <IF CLAUSE> ::= IF <BOOLEAN FORMULA> $

88   <THEN CLAUSE> ::= <SIMPLE STATEMENT>
89                   | <COMPOUND STATEMENT>

90   <IFEITH CLAUSE> ::= <IFEITH> <BOOLEAN FORMULA> $

91   <IFEITH> ::= IFEITH

92   <ALT LIST> ::= <ORIF PART>
93                | <ALT LIST> <ORIF PART>

94   <ORIF PART> ::= <ORIF CLAUSE> <THEN CLAUSE>

95   <ORIF CLAUSE> ::= ORIF <BOOLEAN FORMULA> $
```

```
96                    ¦ <LABEL> ORIF <BOOLEAN FORMULA> $

97    <LOOP STATEMENT> ::= <COMPLETE LOOP>
98                       ¦ <INCOMPLETE LOOP>

99    <INCOMPLETE LOOP> ::= <INCOMPLETE FOR>
99                          <INCOMPLETE BODY>

100   <INCOMPLETE FOR> ::= <1 FACTOR FOR CLAUSE>
101                      ¦ <2 FACTOR FOR CLAUSE>

102   <2 FACTOR FOR CLAUSE> ::= FOR <LOOP COUNTER> =
102                                 <2 INDEX LIST> $

103   <2 INDEX LIST> ::= <NUMERIC FORMULA> ,
103                      <NUMERIC FORMULA>

104   <1 FACTOR FOR CLAUSE> ::= FOR <LOOP COUNTER> =
104                                 <NUMERIC FORMULA> $

105   <INCOMPLETE BODY> ::= <SIMPLE STATEMENT>
106                       ¦ <COMPOUND STATEMENT>
107                       ¦ <SPECIAL COMPOUND STATEMENT>
108                       ¦ <LABEL>
108                         <SPECIAL COMPOUND STATEMENT>

109   <COMPLETE LOOP> ::= <FOR CLAUSE> <COMPLETE BODY>

110   <FOR CLAUSE> ::= <1 FACTOR PART> <3 FACTOR FOR CLAUSE>
111                  ¦ <3 FACTOR FOR CLAUSE>

112   <3 FACTOR FOR CLAUSE> ::= FOR <LOOP COUNTER> =
112                                 <3 INDEX LIST> $

113   <1 FACTOR PART> ::= <1 FACTOR FOR CLAUSE>
114                     ¦ <1 FACTOR PART>
114                       <1 FACTOR FOR CLAUSE>

115   <3 INDEX LIST> ::= <NUMERIC FORMULA> ,
115                      <NUMERIC FORMULA> ,
115                      <NUMERIC FORMULA>

116   <COMPLETE BODY> ::= <INCOMPLETE BODY>
117                     ¦ <INCOMPLETE LOOP>

118   <SPECIAL COMPOUND STATEMENT> ::= <BEGIN>
118                                    <SPECIAL PART> <END>
```

```
119                                                  | <BEGIN>
119                                                    <ELEMENT LIST>
119                                                    <SPECIAL PART> <END>

120   <SPECIAL PART> ::= IF <BOOLEAN FORMULA> $
121                    | <LABEL> IF <BOOLEAN FORMULA> $

122   <STRUCTURED EXTENSION> ::= <DO STATEMENT>
123                            | <CASE STATEMENT>

124   <DO STATEMENT> ::= <DO HEAD> <ELEMENT LIST> <END DO>

125   <DO HEAD> ::= DO WHILE ( <BOOLEAN FORMULA> )
126               | DO UNTIL ( <BOOLEAN FORMULA> )

127   <END DO> ::= END DO

128   <CASE STATEMENT> ::= <CASE HEAD> <CASE LIST>
128                        <END CASE>

129   <CASE HEAD> ::= DO CASE ( <NUMERIC FORMULA> )

130   <CASE LIST> ::= <CASE>
131                 | <CASE LIST> <CASE>

132   <CASE> ::= <INSTANCE> <ELEMENT LIST>
133            | <INSTANCE>

134   <INSTANCE> ::= ( <NUMBER> )

135   <END CASE> ::= END CASE

136   <SIMPLE STATEMENT> ::= <ASSIGNMENT STATEMENT>
137                        | <EXCHANGE STATEMENT>
138                        | <GOTO STATEMENT>
139                        | <RETURN STATEMENT>
140                        | <STOP STATEMENT>
141                        | <PROCEDURE CALL>
142                        | <IO STATEMENT>
143                        | <TEST STATEMENT>

144   <TEST STATEMENT> ::= TEST $
145                      | TEST <LOOP COUNTER> $

146   <IO STATEMENT> ::= <ACTION> <MODE> <IDENTIFIER>
146                      <DATA LIST> $
147                    | <ACTION> <MODE> <IDENTIFIER> $
```

```
148                          | <MODE> <IDENTIFIER> <DATA LIST> $

149   <ACTION> ::= OPEN
150              | SHUT

151   <MODE> ::= INPUT
152            | OUTPUT

153   <DATA LIST> ::= <FORMULA>
154                 | <IDENTIFIER> <RANGE>
155                 | <DATA LIST> , <FORMULA>
156                 | <DATA LIST> , <IDENTIFIER> <RANGE>

157   <RANGE> ::= ($ <NUMERIC FORMULA> <RANGE CLOSE>

158   <RANGE CLOSE> ::= ... <NUMERIC FORMULA> $)

159   <DIRECT STATEMENT> ::= <DIRECT> <TEXT> JOVIAL

160   <DIRECT> ::= DIRECT

161   <ASSIGNMENT STATEMENT> ::= <VARIABLE> = <FORMULA> $

162   <EXCHANGE STATEMENT> ::= <VARIABLE> == <VARIABLE> $

163   <GOTO STATEMENT> ::= GOTO <GOTO FORMULA> $

164   <RETURN STATEMENT> ::= RETURN $

165   <STOP STATEMENT> ::= STOP $
166                      | STOP <IDENTIFIER> $

167   <PROCEDURE CALL> ::= <PROC NAME> $
168                      | <PROC NAME>
168                        <ACTUAL PARAMETER LIST> $

169   <PROC NAME> ::= <IDENTIFIER>

170   <ACTUAL PARAMETER LIST> ::= ( )
171                             | ( <ACTUAL LIST> )
172                             | ( <ACTUAL LIST> =
172                               <ACTUAL LIST> )
173                             | ( = <ACTUAL LIST> )

174   <ACTUAL LIST> ::= <FORMULA>
175                   | <IDENTIFIER> .
176                   | <ACTUAL LIST> , <FORMULA>
```

```
177                       ! <ACTUAL LIST> , <IDENTIFIER> .

178   <DATA DECLARATION> ::= <SIMPLE ITEM DECLARATION>
179                        ! <ARRAY DECLARATION>
180                        ! <TABLE DECLARATION>
181                        ! <OVERLAY DECLARATION>
182                        ! <FILE DECLARATION>
183                        ! <MONITOR DECLARATION>

184   <MONITOR DECLARATION> ::= MONITOR <PARAMETER LIST> $
185                           ! MONITOR ( <BOOLEAN FORMULA>
185                             ) <PARAMETER LIST> $

186   <FILE DECLARATION> ::= <FILE HEAD> <FILE TAIL> $

187   <FILE HEAD> ::= FILE <IDENTIFIER>

188   <FILE TAIL> ::= <F TYPE> <NUMBER> <REC ORG> <NUMBER>
188                 <NUMBER>
189               ! <F TYPE> <NUMBER> <REC ORG> <NUMBER>
189                 <STATUS LIST> <NUMBER>

190   <F TYPE> ::= H
191              ! B

192   <REC ORG> ::= V
193               ! R

194   <STATUS LIST> ::= <STATUS CONSTANT>
195                   ! <STATUS LIST> <STATUS CONSTANT>

196   <SIMPLE ITEM DECLARATION> ::= ITEM <IDENTIFIER>
196                                 <ITEM DESCRIPTION> $
197                               ! ITEM <IDENTIFIER>
197                                 <ITEM DESCRIPTION> P
197                                 <CONSTANT> $
198                               ! ITEM <IDENTIFIER>
198                                 <CONSTANT> $

199   <ITEM DESCRIPTION> ::= <INTEGER DESCRIPTION>
200                        ! <FIXED DESCRIPTION>
201                        ! <FLOATING DESCRIPTION>
202                        ! <LITERAL DESCRIPTION>
203                        ! <STATUS DESCRIPTION>
204                        ! <BOOLEAN DESCRIPTION>

205   <INTEGER DESCRIPTION> ::= <INT HEAD>
```

```
206                                     | <INT HEAD> <INT TAIL>
207                                     | <FIXED HEAD>
208                                     | <FIXED HEAD> <INT TAIL>

209    <INT HEAD> ::= I <NUMBER> <SIGNING>

210    <FIXED HEAD> ::= A <NUMBER> <SIGNING>

211    <SIGNING> ::= U
212                | S

213    <INT TAIL> ::= R
214                 | <INTEGER> ... <INTEGER>
215                 | R <INTEGER> ... <INTEGER>

216    <FIXED DESCRIPTION> ::= <FIXED HEAD> <FRAC BITS>
217                          | <FIXED HEAD> <FRAC BITS>
217                            <RIGHT PART>

218    <FRAC BITS> ::= <NUMBER>
219                  | <ADD OP> <NUMBER>

220    <RIGHT PART> ::= R
221                   | R <NUMERIC CONSTANT> ...
221                     <NUMERIC CONSTANT>
222                   | <NUMERIC CONSTANT> ...
222                     <NUMERIC CONSTANT>

223    <FLOATING DESCRIPTION> ::= F
224                             | F R

225    <LITERAL DESCRIPTION> ::= H <NUMBER>
226                            | T <NUMBER>

227    <STATUS DESCRIPTION> ::= S <STATUS LIST>
228                           | S <NUMBER> <STATUS LIST>

229    <BOOLEAN DESCRIPTION> ::= B

230    <ARRAY DECLARATION> ::= <ARRAY DESCRIPTION>
230                            <INIT LIST>
231                          | <ARRAY DESCRIPTION>

232    <ARRAY DESCRIPTION> ::= <ARRAY HEAD>
232                            <ITEM DESCRIPTION> $

233    <ARRAY HEAD> ::= ARRAY <IDENTIFIER> <DIM LIST>
```

```
234    <INIT LIST> ::= <BEGIN1> <CONSTANT LIST> END
235                  | <BEGIN1> <INIT SUB LIST> END

236    <INIT SUB LIST> ::= <INIT LIST>
237                      | <INIT SUB LIST> <INIT LIST>

238    <DIM LIST> ::= <NUMBER>
239                 | <DIM LIST> <NUMBER>

240    <TABLE DECLARATION> ::= <ORDINARY TABLE DEC>
241                          | <DEF TABLE DEC>
242                          | <LIKE TABLE DEC>

243    <ORDINARY TABLE DEC> ::= <ORD HEAD> <BEGIN2>
243                             <ORD ENTS> <END2>

244    <ORD HEAD> ::= <TABLE HEAD> $
245                 | <TABLE HEAD> <PACKING> $

246    <TABLE HEAD> ::= TABLE <TABLE SIZE>
247                   | TABLE <IDENTIFIER> <TABLE SIZE>
248                   | TABLE <TABLE SIZE> <TABLE STRUCTURE>
249                   | TABLE <IDENTIFIER> <TABLE SIZE>
249                     <TABLE STRUCTURE>

250    <ORD ENTS> ::= <ENTRY ITEM DEC>
251                 | <ORD ENTS> <ENTRY ITEM DEC>
252                 | <ORD ENTS> <SUBORDINATE OVERLAY DEC>

253    <ENTRY ITEM DEC> ::= ITEM <IDENTIFIER>
253                         <ITEM DESCRIPTION> $
254                       | ITEM <IDENTIFIER>
254                         <ITEM DESCRIPTION> $ BEGIN
254                         <CONSTANT LIST> END

255    <TABLE SIZE> ::= V <NUMBER>
256                   | R <NUMBER>

257    <TABLE STRUCTURE> ::= P
258                        | S

259    <PACKING> ::= N
260                | M
261                | D

262    <DEF TABLE DEC> ::= <DEF HEAD> <BEGIN2> <DEF ENTS>
```

```
262                               <END2>

263     <DEF HEAD> ::= <TABLE HEAD> <NUMBER> $

264     <DEF ENTS> ::= <DEF ITEM DEC>
265                  ! <STRING ITEM DEC>
266                  ! <DEF ENTS> <DEF ITEM DEC>
267                  ! <DEF ENTS> <STRING ITEM DEC>

268     <DEF ITEM DEC> ::= <DEF ITEM HEAD> $
269                      ! <DEF ITEM HEAD> <PACKING> $
270                      ! <DEF ITEM HEAD> $ BEGIN
270                        <CONSTANT LIST> END
271                      ! <DEF ITEM HEAD> <PACKING> $ BEGIN
271                        <CONSTANT LIST> END

272     <DEF ITEM HEAD> ::= ITEM <IDENTIFIER>
272                         <ITEM DESCRIPTION> <NUMBER>
272                         <NUMBER>

273     <STRING ITEM DEC> ::= <STRING HEAD> <NUMBER> <NUMBER>
273                           $
274                         ! <STRING HEAD> <PACKING> <NUMBER>
274                           <NUMBER> $
275                         ! <STRING HEAD> <NUMBER> <NUMBER>
275                           $ <2 DIM CONSTANT LIST>
276                         ! <STRING HEAD> <PACKING> <NUMBER>
276                           <NUMBER> $ <2 DIM CONSTANT LIST>

277     <STRING HEAD> ::= STRING <IDENTIFIER>
277                       <ITEM DESCRIPTION>

278     <2 DIM CONSTANT LIST> ::= BEGIN <COMPONENT LIST> END

279     <COMPONENT LIST> ::= BEGIN <CONSTANT LIST> END
280                        ! <COMPONENT LIST> BEGIN
280                          <CONSTANT LIST> END

281     <LIKE TABLE DEC> ::= <TABLE HEAD> L $
282                        ! <TABLE HEAD> <PACKING> L $

283     <OVERLAY DECLARATION> ::= <INDEPENDENT OVERLAY DEC>
284                             ! <SUBORDINATE OVERLAY DEC>

285     <INDEPENDENT OVERLAY DEC> ::= OVERLAY <ORIGIN>
285                                   <OVERLAY TAIL>
```

```
286   <ORIGIN> ::= <NUMBER>
287          | <OCTAL CONSTANT>

288   <SUBORDINATE OVERLAY DEC> ::= OVERLAY <OVERLAY TAIL>

289   <OVERLAY TAIL> ::= <IDENTIFIER LIST> $
290                    | <IDENTIFIER LIST> <OVERLAY LIST> $

291   <IDENTIFIER LIST> ::= <IDENTIFIER>
292                       | <IDENTIFIER LIST> , <IDENTIFIER>

293   <OVERLAY LIST> ::= = <IDENTIFIER LIST>
294                    | <OVERLAY LIST> = <IDENTIFIER LIST>

295   <FORMULA> ::= <BOOLEAN FORMULA>

296   <BOOLEAN FORMULA> ::= <BOOLEAN TERM>
297                       | <BOOLEAN FORMULA> OR
297                         <BOOLEAN TERM>

298   <BOOLEAN TERM> ::= <BOOLEAN PRIME>
299                    | <BOOLEAN TERM> AND <BOOLEAN PRIME>

300   <BOOLEAN PRIME> ::= <LITERAL FORMULA>
301                     | NOT ( <BOOLEAN PRIME> )
302                     | <RELATIONAL EXPRESSION>

303   <RELATIONAL EXPRESSION> ::= <LITERAL FORMULA> <REL OP>
303                               <LITERAL FORMULA>
304                             | <RELATIONAL EXPRESSION>
304                               <REL OP> <LITERAL FORMULA>

305   <REL OP> ::= EQ
306              | NQ
307              | GR
308              | GQ
309              | LS
310              | LQ

311   <LITERAL FORMULA> ::= <STATUS FORMULA>
312                       | <LITERAL CONSTANT>

313   <LITERAL CONSTANT> ::= <LITERAL HEAD> ( <CHARACTERS> )

314   <LITERAL HEAD> ::= <NUMBER> H
315                    | <NUMBER> T
```

```
316   <STATUS FORMULA> ::= <STATUS CONSTANT>
317                      | <NUMERIC FORMULA>

318   <STATUS CONSTANT> ::= <STATUS HEAD> <CHARACTERS> )

319   <STATUS HEAD> ::= V (

320   <NUMERIC FORMULA> ::= <EXPRESSION>

321   <EXPRESSION> ::= <EXPRESSION> <ADD OP> <TERM>
322                  | <TERM>

323   <TERM> ::= <TERM> <MULT OP> <FACTOR>
324            | <FACTOR>

325   <FACTOR> ::= <PRIME> ** <FACTOR>
326              | <PRIME> (* <FACTOR> *)
327              | <ADD OP> <PRIME>
328              | <PRIME>

329   <PRIME> ::= <VARIABLE>
330             | <FUNCTION>
331             | <NUMERIC CONSTANT>
332             | ( <BOOLEAN FORMULA> )
333             | (/ <NUMERIC FORMULA> /)

334   <FUNCTION> ::= <IDENTIFIER> ( )
335                | <IDENTIFIER> ( <ACTUAL LIST> )

336   <ADD OP> ::= +
337              | -

338   <MULT OP> ::= *
339               | /

340   <NUMERIC CONSTANT> ::= <INTEGER>
341                        | <FLOATING CONSTANT>
342                        | <FIXED CONSTANT>
343                        | <OCTAL CONSTANT>

344   <INTEGER> ::= <NUMBER>
345               | <NUMBER> E <NUMBER>

346   <FLOATING CONSTANT> ::= <MANTISSA>
347                         | <MANTISSA> <CHARACTERISTIC>

348   <CHARACTERISTIC> ::= E <NUMBER>
```

```
349                           ; E <ADD OP> <NUMBER>

350    <FIXED CONSTANT> ::= <FLOATING CONSTANT> A <NUMBER>
351                         ; <FLOATING CONSTANT> A <ADD OP>
351                           <NUMBER>

352    <OCTAL CONSTANT> ::= O ( <NUMBER> )

353    <VARIABLE> ::= <IDENTIFIER>
354                 ; <LOOP COUNTER>
355                 ; <IDENTIFIER> ($ <INDEX LIST> $)
356                 ; <MODIFIED VARIABLE>

357    <INDEX LIST> ::= <NUMERIC FORMULA>
358                   ; <INDEX LIST> , <NUMERIC FORMULA>

359    <LOOP COUNTER> ::= A
360                     ; B
361                     ; C
362                     ; D
363                     ; E
364                     ; F
365                     ; G
366                     ; H
367                     ; I
368                     ; J
369                     ; K
370                     ; L
371                     ; M
372                     ; N
373                     ; O
374                     ; P
375                     ; Q
376                     ; R
377                     ; S
378                     ; T
379                     ; U
380                     ; V
381                     ; W
382                     ; X
383                     ; Y
384                     ; Z

385    <MODIFIED VARIABLE> ::= CHAR ( <VARIABLE> )
386                          ; BIT ($ <NUMERIC FORMULA> $) (
386                            <VARIABLE> )
387                          ; BIT ($ <NUMERIC FORMULA> ,
```

```
387                                  <NUMERIC FORMULA> $) (
387                                  <VARIABLE> )
388                                ! MANT ( <VARIABLE> )
389                                ! NENT ( <VARIABLE> )
390                                ! POS ( <IDENTIFIER> )
391                                ! ODD ( <VARIABLE> )
392                                ! BYTE ($ <NUMERIC FORMULA> $) (
392                                  <VARIABLE> )
393                                ! BYTE ($ <NUMERIC FORMULA> ,
393                                  <NUMERIC FORMULA> $) (
393                                  <VARIABLE> )
394                                ! ENTRY ( <VARIABLE> )
395                                ! ENT ( <VARIABLE> )

396     <BEGIN> ::= BEGIN

397     <END> ::= END

398     <BEGIN2> ::= BEGIN

399     <END2> ::= END
```

6. The Structured Extensions to JOVIAL J3

Three structured extensions to JOVIAL J3 have been introduced during the development of the prototype JSDD. The JSDD was written using the structured extensions, and it is capable of diagramming JOVIAL J3 programs which employ them. Section 6.1 describes the structured extensions and Section 6.2 describes their translation into standard JOVIAL J3.

6.1 STRUCTURED EXTENSION DEFINITIONS

The current JOVIAL J3 structured extensions are:

1) The DO WHILE LOOP
2) The DO UNTIL LOOP
3) The CASE STATEMENT

Their syntactic definitions follow:

<DO WHILE LOOP> ::= do while [<condition>] <statement list>
                    [end do <text>]

<DO UNTIL LOOP> ::= do until [<condition>] <statement list>
                    [end do <text>]

<CASE STATEMENT> ::= do case [<selector>] <case list>
                    [end case <text>]

<case list> ::= <case> <statement list>
              | <case>
              | <case list> <case> <statement list>
              | <case list> <case>

<case> ::= [<number> <text>]
         | [else <text>]

The symbol <text> appearing in the productions presented above is an optional feature which allows the user to place quoted or unquoted comments inside the square brackets.

The symbol <condition> can be any boolean formula allowed in a conditional statement in standard JOVIAL J3.

The symbol <selector> can be any JOVIAL J3 variable, constant or function call which has an integer value.

The cases in the CASE STATEMENT'S <case list> must be numbered sequentially starting with zero. One ''else case'' may appear anywhere in the <case list>.

The symbol <statement list> refers to any sequence of standard JOVIAL J3 statements and structured extensions.

The structured extensions may appear wherever a standard JOVIAL J3 complex statement can appear.

A brief description of the semantics associated with each of the structured extensions appears below. A better understanding of their semantics can be obtained by examining the translations presented in Section 6.2.

The DO WHILE LOOP causes repeated execution of its <statement list> while its <condition> is true. Evaluation of the <condition> is performed prior to the execution of the <statement list>. If the value of the <condition> is false, control is passed to the statement immediately following the [end do].

The DO UNTIL LOOP causes repeated execution of its <statement list> while the value of its <condition> is false. Evaluation of the <condition> is performed after each execution of the <statement list>. If the value of the <condition> is true, then control is passed to the statement immediately following the [end do].

The CASE STATEMENT evaluates its <selector> and transfers control to the <case> whose <number> corresponds to the value of the <selector>. If no corresponding <case> appears in the <case list>, then control is passed to the ''else case'' (if one is present) or to the statement immediately following the [end case] (if there is no ''else case'').

Transferring control to a <case> causes execution of the <statement list> associated with the <case>. After execution of the <statement list> has been completed, control is transferred to the statement immediately following the [end case].

6.2 Translating Structured Extensions

Programs incorporating structured extensions are translated into standard JOVIAL J3 programs by the JOVIAL Extended Structures Translator (JEST preprocessor). JEST is a PL/I program implemented on the RADC MULTICS system.

JEST accepts extended JOVIAL (standard JOVIAL plus structured extensions) programs with MULTICS path-names having the suffix ''.jovp''. The sructured extension keywords (i.e., do, while, until, case, else and end) must be lower case in the input program. JEST's output is a standard JOVIAL J3 translation of the input program. The output program has the same MULTICS path-name as the input program except that it has the suffix ''.jov''.

JEST is invoked by typing ''jp pgm'name'', or by invoking the jovial EXEC_COM (See Appendix A).

JEST translates only those blocks of code which it recognizes as structured extensions. Any code which is not recognized as being a structured extension is passed to the output file unchanged.

Structured extensions and their translations are presented below.

The DO WHILE LOOP:

```
     do while [condition]
                code block a
     [end do]
     code block b
```

translates into:

```
     while'label. if condition $
                begin
                code block a
                goto while'label $
     end
     code block b
```

The DO UNTIL LOOP:

```
     do until [condition]
                code block a
     [end do]
     code block b
```

is translated into:

```
     until'label.
                code block a
     if not condition $ goto until'label $
     code block b
```

JEST emits two types of translations of CASE STATEMENTs. The first translation is used on CASE STATEMENTs whose <case list>s contain 100 <case>s or less. The second translation is used for CASE STATEMENTs having more than 100 <case>s.

Consider the pseudo statement:

```
    do case [selector]
            [0] code block 0
            [1] code block 1
            [2] code block 2
             .
             .
             .
            [n] code block n+1
    [end case]
    code block n+2
```

If n is less than 100, the JEST translation of the CASE STAEMENT is:

```
    goto process'label $
    case'label'0.
            code block 0
            goto escape'label $
    case'label'1.
            code block 1
            goto escape'label $
     .
     .
     .
    case'label'n.
            code block n
            goto escape'label $
    else'label.
            code block n+1
            goto escape'label $

    switch case'control = (case'label'0,...,case'label'n) $
    process'label.
            goto case'control ($selector$) $
            goto else'label $
    escape'label.
            code block n+2
```

Unfortunately, the JOVIAL J3 compiler imposes a limit on the number of labels that may appear in a single SWITCH DECLARATION. The limit seems to be around 100. For this reason, a more complicated translation is necessary for CASE STATEMENTS having more than 100 cases. If, in the above example the value 200 is assigned to n, the translation of the pseudo statement takes the following form:

```
    goto process'label $
    case'label'0.
            code block 0
            goto escape'label $
    case'label'1.
            code block 1
            goto escape'label $
    .
    .
    .
    case'label'199.
            code block 199 $
            goto escape'label $
    else'label.
            code block 200
            goto escape'label $

    switch block'1 = (case'label'0,...,case'label'99) $
    switch block'2 = (case'label'100,...,case'label'199) $
    switch control'switch = (block'1'label,block'2'label) $
    item switch'block i 36 s $
    item switch'displacement i 36 s $
    process'label.
    remquo(selector,100=switch'block,switch'displacement) $
    goto control'switch ($switch'block$) $
    goto else'label $
    block'1'label.
    goto block'1 ($switch'displacement$) $
    block'2'label.
    goto block'2 ($switch'displacement$) $
    goto else'label $
    escape'label.
    code block 201
```

JEST generates labels of the form ''zzzn'' where n is a non-negative integer. The programmer using JEST should take care to avoid any labelling conflicts with the JEST-generated labels.

Throughout the JSDD development, the JEST preprocessor has been heavily used and has proven to be completely reliable. This reliability has been such that JSDD debugging activities have been focussed on the extended JOVIAL code rather than on the somewhat obtuse JEST-generated code.

7. Conclusions and Recommendations

The principal utility of Structured Design Diagrams (SDDs) and Invocation Diagrams is that they provide readable, graphic portrayals of the nested logical sequences that define the structure of a computer program or of a system design written in some higher order design language. The prototype JOVIAL Structured Design Diagrammer (JSDD) was implemented to process, as input, digital computer programs written in JOVIAL J3 (with or without structured extensions) and to produce, as output, SDDs and Invocation Diagrams. The JOVIAL J3 grammar processed by the JSDD was extracted from the definition in reference 4. Due to certain ambiguities in the JOVIAL J3 language (see Section 5), it was necessary to relax some constraints on the JSDD grammar so that the JSDD parses JOVIAL J3 programs as a subset of a wider grammar. The structured extensions are based upon the structured extensions to JOVIAL J3 as described in reference 5.

SDDs produced from source code that uses structured constructs convey better information about program structure than those produced from code that does not obey the precepts of Structured Programming; therefore, the use of the structured extensions supported by the JSDD is encouraged. Programs that make use of the structured extensions need a preprocessor to convert them into source code that is acceptable to the JOVIAL J3 compilers. Such a preprocessor has been supplied as a deliverable with the JSDD. See Appendix A for a complete list of deliverables. That preprocessor was used to convert the source code of the JSDD, which was written with structured extensions, into compilable source code. Because of the structured extensions, the JSDD is a structured program; therefore, it was possible to use the JSDD to provide excellent quality design diagrams of its own source code. The SDDs and Invocation Diagrams of the JSDD programs are incorporated in the JSDD Program Description, a companion volume to this report.

To get full utility from the Structured Design Diagrammer, one would have to integrate the JSDD programs into a comprehensive documentation system. The JSDD would require additional design features to achieve its full potential. A list of desirable additional features follows:

1. Optional facilities for page formatting and pen plotting should be implemented.

2. Concordance listings and data summaries should be optionally available to the JSDD user.

3. Statements on the SDDs should be annotated optionally with sequence numbers that agree with the JOVIAL compiler listing.

4. The JSDD Symbol Table capability should be enhanced and its content appropriately written to a data base. This data base should be accessible to interactive software tools that can aid the user in understanding the interaction of variables in the computer program.

The JSDD's Invocation Diagrammer is an example of a software tool which processes a subset of the JSDD symbol table information described in item 4 above.

From the implementor's point of view the JOVIAL J3 language was not a good choice of a programming language in which to implement the JSDD. For instance, the JOVIAL J3 compiler supplied for use on this contract was incapable of optimizing the JSDD programs. Even if this were not the case, the JSDD would still run more slowly than necessary, because much computer time is consumed performing operations for which the JOVIAL J3 compiler is not very efficient. For example, a major shortcoming of the compiler is that it does not support random access output operations on disk files. This induces the JSDD to consume great amounts of computer time doing double buffered I/O, and requires the inclusion of additional software modules in the JSDD computer programs.

Two additional aspects of the JOVIAL J3 language render it less than desirable for the JSDD application or for implementing compiler-like tools in general. First, the static nature of JOVIAL's data handling makes it difficult to do dynamic memory management. Second, JOVIAL does not contain string handling constructs that are naturally suited to compiler-like programming. The outstanding difficulty with the JOVIAL character string manipulation capability is

that the current string length and the "declared" maximum string length for a string variable are not available at execution time. In order to circumvent the string handling difficulties, the character string handling package is incorporated into each JSDD program. This package offers string operations such as substring and concatenation.

Despite the difficulties attendant to the implementation, the JSDD produces Structured Design Diagrams of high quality. The JOVIAL programmer should find these diagrams to be of great assistance during program design and implementation as well as being useful for documentation purposes. The Invocation Diagrammer produces an output that is extremely useful and probably should long have been a standard feature of commercial compilers. The prototype JSDD has been designed such that it can also serve as the nucleus of a comprehensive automated documentation system for JOVIAL programs, should the construction of such a system be desired at a later time.

8. References

1. Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, New York, 1972.

2. Hamilton, M., and Zeldin, S., Top-Down, Bottom-Up Structured Programming and Program Structuring, (Revision 1), Charles Stark Draper Laboratory, Inc., Cambridge, Ma. - E-2728, December 1972.

3. McGowan, C. L., and Kelly, J. R., Top-Down Structured Programming Techniques, Petrocelli/Charter, New York, 1975.

4. Standard Computer Programming Language for Air Force Command and Control Systems, Short Title: CED 2400, Air Force Manual AFM 100-24, Reprint dated 21 April 1972.

5. Structured Programming Series, Volume I, Programming Languages Standards, Final Report, IBM Corporation, FSD 74-0288, March 15, 1975.

6. DeRemer, F. L., Practical Translators for LR(k) Languages, Ph.D. Thesis at the Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1969.

7. Lalonde, W. R., An Efficient LALR Parser Generator, Technical Report CSRG-2, M.Sc. Thesis, University of Toronto, Toronto, Ontario, 1970.

8. McKeeman, W. M., Horning, J. J., and Wortman, D. B., A Compiler Generator Implemented for the IBM System/360, Prentice Hall, 1970.

Appendix A. The Deliverables

In accordance with contract number F30602-76-C-0409 with the Rome Air Devleopment Center at Griffis Air Force Base, the Charles Stark Draper Laboratory has delivered the following three items.

1) CDRL Item A003, The JOVIAL Structured Design Diagrammer Final Report
2) CDRL Item A004, The JOVIAL Structured Design Diagrammer Program Description and The JOVIAL Structured Design Diagrammer User's Manual
3) CDRL Item B-6-3308, Annex 1 to SOW Item 1, Digital Computer Software

The listings of the source programs have been transmitted to RADC via the RADC MULTICS system. The digital computer software has been delivered via a RADC MULTICS directory hierarchy accessible to RADC contracts personnel. Each subdirectory within the hierarchy contains an explanatory segment named info. These explanatory segments are presented below.

The deliverables directory contains all software tools developed by the Charles Stark Draper Laboratory, Inc., Cambridge, Ma. under contract F30602-76-C0408 with the Rome Air Development Center at Griffis Air Force Base.

The Deliverables directory contains the five subdirectories listed below:

1) dddg Contains all Design Diagrammer Data Base Generator software.
2) ddg Contains all Design Diagrammer software.
3) invoc Contains all Invocation Diagrammer software.
4) jest Contains all preprocessor software.
5) util Contains support software.

The directory deliverables>ddg contains all Design Diagram Generator (DDG) software. It contains the four subdirectories listed below:

1) source Contains source code for the DDG.
2) compools Contains source versions of the DDG compools.
3) obj Contains object code for compiling and running

the DDG.

4) control   Contains the control segments for running and compiling the DDG.

The directory deliverables>ddg>source contains two source versions of the DDG: ddg.jovp and ddg.jov.

The segment ddg.jovp is the primary source version. It is written in extended JOVIAL (i.e., JOVIAL J3 plus structured extensions). All development work was performed on ddg.jovp.

The segment ddg.jov is the preprocessed version of ddg.jovp. It consists of standard JOVIAL J3 code. The segment ddg.jov was obtained by executing the JEST preprocessor with the input segment ddg.jovp.

The directory deliverables>ddg>compools contains the source versions of the compools used by the DDG. They are:

1) spool.cmp   Contains the string package global declarations.
2) debug.cmp   Contains the DDG debug switches.
3) opt.cmp   Contains the DDG options.

The directory deliverables>ddg>obj contains object segments and cmp_out segments (used for compiling DDG elements). They are:

1) spool.obj   Global string package declarations.
2) spool.cmp_out
3) debug.obj   DDG debugging switches.
4) debug.cmp_out
5) opt.obj   DDG options.
6) opt.cmp_out
7) ddg.obj   The DDG.

The directory deliverables>ddg>control contains the control segments needed to invoke the GCOS simulator for a

compilation or execution. The segments are:

1) ddg.jin   Compiles the DDG.
2) ddg.rin   Executes the DDG.
3) spool.cin   Compiles spool.cmp.
4) debug.cin   Compiles debug.cmp.
5) opt.cin   Compiles opt.cmp.

The directory deliverables>dddg contains all Design Diagrammer Data Base Generator (DDDG) software. It contains the four subdirectories listed below:

1) source   Contains source code for the DDDG.
2) compools Contains source versions of the DDDG compools.
3) obj   Contains object code for compiling and running the DDG.
4) control   Contains the control segments for running and compiling the DDG.

The directory deliverables>dddg>source contains two source versions of the DDDG: dddg.jovp and dddg.jov. Two source versions of the external procedure synth, synth.jovp and synth.jov, are also in the directory.

The segment dddg.jovp is the primary source version. It is written in extended JOVIAL (i.e., JOVIAL J3 plus structured extensions). All developmental work was performed on dddg.jovp.

The segment dddg.jov is the preprocessed version of dddg.jovp. It consists of standard JOVIAL J3 code. The segment dddg.jov was obtained by executing the JEST preprocessor with the input segment dddg.jovp.

The directory deliverables>dddg>compools contains the source versions of the compools used by the dddg. They are:

1) spool.cmp   Contains the string package global declarations.
2) data.cmp   Contains the dddg gloabal declarations.
3) ntables.cmp   Contains the parsing tables used by the dddg.

The directory deliverables>dddg>obj contains object segments and cmp_out segments used for compiling the dddg.

They are:

1) data.obj Global dddg declarations.
2) data.cmp_out
3) ntables.obj Parsing tables.
4) ntables.cmp_out
5) spool.obj String package global declarations.
6) spool.cmp_out
7) synth.obj The dddg's external procedure.
8) dddg.obj The dddg.

The directory deliverables>dddg>control contains the control card segments needed to invoke the GCOS simulator for a JOVIAL compilation or execution. The segments are:

1) dddg.jin Compiles the dddg.
2) dddg.rin Executes the dddg.
3) synth.jin Compiles synth.jov.
4) data.cin Compiles data.cmp.
5) ntables.cin Compiles ntables.cmp.
6) spool.cin Compiles spool.cmp.

The directory deliverables>invoc contains all Invocation Diagrammer software. It contains the four subdirectories listed below:

1) source Contains source code for the Invocation Diagrammer.
2) compools Contains source versions of the Invocation Diagrammer compools.
3) obj Contains object code for compiling and running the Invocation Diagrammer.
4) control Contains the control segments for running and compiling the Invocation Diagrammer.

The directory deliverables>invoc>source contains two source versions of the Invocation Diagrammer: invoc.jovp and invoc.jov.

The segment invoc.jovp is the primary source version. It is written in extended JOVIAL (i.e., JOVIAL J3 plus structured extensions). All development work was performed on invoc.jovp.

The segment invoc.jov is the preprocessed version of invoc.jovp. It consists of standard JOVIAL J3 code. The segment invoc.jov was obtained by executing the JEST preprocessor with the input segment invoc.jovp.

The directory deliverables>invoc>compools contains the source versions of the compools needed to run the Invocation Diagrammer.

They are:

1) opt.cmp The options compool.
2) spool.cmp The string package compool.

The directory deliverables>invoc>obj contains the object segments and cmp_out segments (used for compiling the Invocation Diagrammer) relating to the Invocation Diagrammer.

They are:

1) invoc.obj The Invocation Diagrammer.
2) spool.obj The string package declarations.
3) spool.cmp_out
4) opt.obj The Invocation Diagrammer options.
5) opt.cmp_out

The directory deliverables>invoc>control contains the GCOS control card segments used for running and compiling the Invocation Diagrammer. They are:

1) invoc.jin Compiles the Invocation Diagrammer.
2) invoc.rin Runs the Invocation Diagrammer.
3) spool.cin Compiles spool.cmp.
4) opt.cin Compiles opt.cmp.

The directory deliverables>jest contains the source and object versions of the JOVIAL Extended Structures Translator (JEST). JEST is a preprocessor which translates programs written in extended JOVIAL (i.e., JOVIAL J3 plus structured extensions) into standard JOVIAL J3 programs. JEST is a PL/I program developed to work in a MULTICS environment.

The directory deliverables>util contains the MULTICS system support software developed for the JSDD effort. It has two subdirectories, exec_coms and code. The exec_com subdirectory contains the MULTICS exec_coms developed by Victor Voydock which generate GCOS control cards. The subdirectory code contains utility software used throughout the project.

The directory deliverables>util>exec_coms contains the MULTICS exec_coms which were developed to automatically generate GCOS job control card decks. The following is a list of the exec_coms accompanied by a brief description.

**** gcos_setup.ec ****

A number of exec_com's have been created to make it easier to compile programs using translators which run under the GCOS environment simulator and to execute the resulting GCOS object programs.

This is a description of what must be done to set up a process environment in which these exec_com's can be used. The actual use of the individual exec_com's is described

below.

Once the following instructions are followed, the exec_coms may be used directly as commands as described below.

Table of Contents:

The following are available:

jovial	compile a Jocit JOVIAL program
compool	compile a compool to be used by other Jocit JOVIAL programs
gfort	compile a GCOS FORTRAN program
gmap	assemble a GMAP (GCOS assembly language) program
run	load and execute one or more GCOS object programs
prr	print edited version of program run output

Instructions:

To use the above exec_coms directly as commands do the following:

1) Using the "link" command, put the following link in your "search" directory: (this link may already exist; if so, the link command will ask you if you wish to delete the old link; you should answer yes) >udd>fc>vlv>Library>on

2) In the abbrev command lines listed below, you should replace ZZZ by your initials (in lower case). The rest of the line should be typed literally as shown below. For example,

[user name]

means type a left bracket followed by the word "user", followed by the word "name", followed by a right bracket.

**** install.ec ****

The install exec_com "installs" new versions of the jovial, compool, run, gfort, and gmap exec_coms in the directory >udd>fc>vlv>lib.

NOTE: THIS EXEC_COM ASSUMES THAT THE USER'S WORKING DIRECTORY IS:

```
>udd>flowchart>Voydock>Library
```

Syntax:

```
cwd >udd>fc>vlv>lib ec install name
```

Argument:

name is the name of the segment (minus the "ec" suffix) to be installed.

Notes:

The installation process is as follows:

1) The segment "name.ec" in >udd>Flowchart>Voydock>Library (hereinafter called "the library") is renamed to name.X.ec where X is the date and time the installation began.

2) All other names on the old name.ec are removed

3) The segment >udd>Flowchart>Voydock>gcos_exec_com_dev>name.ec is copied into the library.

4) The names of all exec_coms invoked with name.ec are converted from relative to absolute pathnames. This is necessary to avoid forcing the user to know the names of all the exec_coms invoked in name.ec and having to place links to all of them in any directory which will be his working directory when name.ec is invoked.

**** jovial.ec ****

The jovial command allows the user to easily invoke the Jocit JOVIAL compiler which runs under the Multics GCOS Environment Simulator to translate a segment containing a

JOVIAL source program into a GCOS format object segment which can be executed using the run command. A listing segment is optionally produced.

The following output line is always printed out by the Jocit compiler. There is no way to suppress it, and is not indicative of a major problem:

gcos_mm_info_: mme geinfo will not return a copy of the SSA.

Syntax:

jovial path -jocit_options-

Arguments:

path is the pathname of a JOVIAL source segment that is to be translated by the Jocit JOVIAL compiler. The actual entry name of the source segment must have the suffix "jov". If path does not have the suffix "jov" then one is assumed. That is, the following two command lines are identical:

```
jovial my_program.jov
jovial my_program
```

jocit_options are options accepted by the Jocit compiler. See the Jocit Compiler User's Manual for a list of acceptable options. If multiple options are specified, they must be separated by commas. No embedded blanks are allowed.

Output Segments:

The following output segments are produced in a subdirectory of the user's working directory named jovwrk (which is created if it doesn't exist) during the compilation of the segment myprog.jov:

myprog.obj
GCOS format object program which is the translation of myprog.jov. The program may be executed by the run command.

myprog.list

Listing segment produced by the compiler in response to user specified options.

myprog.compile.list
Output of the GCOS environment simulator. This is usually of no interest to the user.

myprog.jin
Card images of GCOS control cards produced by the jovial command to cause the GCOS enivronment simulator to perform the translation. Usually of no interest to the user.

Notes:

1) If a "jin" segment already exists when the jovial command is invoked, the user is asked if he wants to use it. If he answers "yes" then the old segment is used without change. If he answers "no" then the old segment is deleted and a new "jin" segment is generated.

2) If a "jin" segment is being generated, the command asks the user for the names of the compool segments to be used during this compilation. The names should be all typed on the same line separated by at least one space. The names should be typed without the "cmp_out" suffix. The compools must previously have been compiled using the compool command. If no compools are to be used, a blank line should be typed.

3) The jovial command automatically compiles all programs with the compool >udd>Flowchart>Voydock>Library>utilities.cmp_out which contains various useful external procedure declarations.

4) The jovial command automatically adds the following options to the list of user specified options:

```
xref
map
ncomdk
name/xxxxxx/
```

where xxxxxx is the first six characters of the name of the program being compiled. The ncomdk option suppresses

the production of a compressed source card image optput file which is of no interest to the user. The Jocit manual claims ncomdk is the default but this is not the case.

5) Since the "jin" and "compile.list" segments are not of general interest and tend to clutter up one's directory, the user may wish to use abbrev or exec_com to automatically delete these segments after translation is completed. This feature may later be added as a control argument to the jovial command.

Example:

The following command line will translate the source program >any_dir>some_prog.jov using the xref, map, and lstou compiler options and store the output segments in the user's current working directory:

```
jovial >any_dir>some_prog lstou
```

**** compool.ec ****

The compool command allows the user to easily invoke the Jocit JOVIAL compiler which runs under the Multics GCOS Environment Simulator to translate a segment containing the source code of a JOVIAL compool.

Syntax:

```
compool path -jocit_options-
```

Arguments:

path is the pathname of a JOVIAL compool source segment to be translated by the Jocit JOVIAL compiler. The actual entry name of the source segment must have the suffix "cmp". If path does not have the suffix "cmp" then one is assumed. That is, the following two command lines are identical:

```
compool my_program.cmp
```

compool my_program

jocit_options are options accepted by the Jocit compiler. See the Jocit Compiler User's Manual for a list of acceptable options. If multiple options are specified, they must be separated by commas. No embedded blanks are allowed.

Output Segments:

The following output segments are produced in a subdirectory of the working directory called jovwrk during the compilation of the segment myprog.cmp:

myprog.cmp_out
Contains the translated form of the procedure and data declarations of the source compool which do not actually generate any storage. That is, procedure declarations and non compool common data declarations.

myprog.obj
Contains storage for compool common data declared in the source compool. In theory, if there are no compool common data, this segment should not be produced. However, the compiler seems to always produce this segment. In the case when there are no compool common data, it seems from empirical evidence to be safe to ignore the "obj" segment.

myprog.list
Listing segment produced by the compiler in response to user specified options.

myprog.compile.list
Output of the GCOS environment simulator. This is usually of no interest to the user.

myprog.cin
Card images of GCOS control cards produced by the jovial command to cause the GCOS enivronment simulator to perform the translation. Usually of no interest to the user.

Notes:

1) If a "cin" segment already exists when the compool command is invoked, the user is asked if he wants to use it. If he answers "yes" then the old segment is used without change. If he answers "no" then the old segment is deleted and a new "cin" segment is generated.

2) The compool command automatically adds the following options to the list of user specified options:

```
ncomdk
name/xxxxxx/
```

where xxxxxx is the first six characters of the name of the compool being translated. The ncomdk option suppresses the production of a compressed source card image optput file which is of no interest to the user. The Jocit manual claims ncomdk is the default but this is not the case.

3) Since the "cin" and "compile.list" segments are not of general interest and tend to clutter up one's directory, the user may wish to use abbrev or exec_com to automatically delete these segments after translation is completed. This feature may later be added as a control argument to the compool command.

Example:

The following command line will translate the source program >any_dir>some_prog.cmp using the xref, map, and lstou compiler options and store the output segments in the user's current working directory:

```
compool >any_dir>some_prog xref,map,lstou
```

**** run.ec ****

The run command allows the user to execute one or more GCOS format object segments using the GCOS environment simulator. A current implementation restriction allows only object segments produced by the JOVIAL compiler to be run.

Syntax:

run path1 ... pathn

Arguments:

path1 ... pathn are the pathnames of the object segments to be run. The actual entry names of the object segments must have the suffix "obj". BUT the pathnames given in the command line must not specify the "obj" suffix. That is, the command line:

run first_prog my_prog

is legal. Whereas the command line:

run first_prog.obj my_prog.obj

is not.

Output Segments:

The following output segments are produced in a subdirectory (jovwrk) of the user's working directory during the execution of the run command:

first_prog.run.list
Where "first_prog" is the first object segment specified in the command line. This segment contains two types of output. First it contains output produced by the GCOS simulator during the execution. This is usually of little interest to the user. Then it contains all program output directed to logical unit number 06 and all output produced by monitor statements.

The prr command allows the user to print the contents of this segment ignoring the GCOS simulator output. There is no way to divert output directed to logical unit number 06 to the terminal. To print output on the terminal use the trmout and tout procedures.

There currently is no way to divert monitor output to the terminal. This is a problem in the Jocit runtime library. It is being looked at by the compiler implementors.

first_prog.rin

This segment contains card images of GCOS control cards produced by the run command to cause the GCOS enviroment simulator to execute the programs. Usually it is of no interest to the user.

Notes:

1) If a "rin" segment already exists when the run command is invoked, the user is asked if he wants to use it. If he answers "yes" then the old segment is used without change. If he answers "no" then the old segment is deleted and a new "rin" segment is generated.

2) If a "jin" segment is being generated, the command asks the user for the names of the files which will be referenced during the execution of the object programs. The names should all be typed on the same line separated by at least one space. Output files need not exist when the run command is invoked: they are automatically created when needed. The files are assigned logical unit numbers starting with 11 (that is, the first file specified is assigned 11, the second 12, etc.). The file declarations in the object programs must, of course, assign logical unit numbers accordingly.

3) The logical unit number 08 is reserved for use by the trmin procedure, the unit 09 is reserved for use by the trmout procedure, unit 10 is reserved for possible special use. None of these unit numbers should be used by user programs.

4) The run command automatically loads the trmin, trmout, tout and frstnb programs for every execution.

**** prr.ec ****

The prr command prints the results of a run stored in a "run.list" segment starting with the line containing the string "EXECUTION PROGRAM ENTERED AT" That is, it ignores all of the output of the simulator (load map, etc.) which is usually of no interest to the user.

Syntax:

prr path

Argument:

path is the pathname of the segment containing the output to be printed. The actual name of the segment must have the suffix "run.list". However this suffix must not be specified in path. That is,

    prr my_prog

is legal: whereas

    prr my_prog.run.list

is not.

**** on.ec ****

Function: on is a condition-handling program for commands.

Syntax: on condname handler_com_line options subject_com_line

Arguments: "subject_com_line" will be executed. if condition "condname" is raised, then the handler for it will call cu_$cp to execute the line "handler_com_line".

Control arguments: -brief (-bf) to suppress the comment when condition is raised -retry to return to point of fault after handler -long (-lg) to print machine conditions when condition raised -cl to call cu_$cl after handler (command level, preserves machine conditions)

Notes:
Normal action after handler is to do a non-local goto and continue to the next command after "on".

"subject_com_line" begins with first non-option and need not be enclosed in quotes ("handler_com_line" must be though). If "condname" is "*", then an any_other handler is set up In this case, "alrm", "cput", "quit", and "program_interrupt" will not activate the handler.

Examples: on * "ec error" -lg v2pl| foobar on quit ts on * ts myprogram$myprogram | on command_error db -rt -bf foobar on accessviolation "sa &| rewa" -rt print &|

The directory deliverables>util>code contains the JSDD software support code. The segments contained in the directory are listed below accompanied by brief descriptions of their purpose and use.

utilities.cmp
utilities.cmp_out
utilities.obj

The compool >udd>Flowchart>Voydock>Library>utilities.cmp_out is automatically used by the jovial command for every compilation. It contains the proper declarations for the following items.

External procedures:

trmout	print a character string on the terminal suppressing leading blanks
tout	print a portion of a character string on the terminal
trmin	read a line typed at the terminal into a character string
frstnb	return the index of the first nonblank character in a string

Currently useless:

The following two declarations will eventually allow the user to divert the output of "monitor" statements to a segment other than the "run.list" segment. Since the j'file

procedure described in the Jocit manual does not currently work (any program which invokes it blows up), these items are not of current interest. They are included for completeness.

```
file      mon'output h 1 r 132 v(null) 10$
proc      j'file(mon'output'file)$
```

trmout.gmap trmout.obj

The procedure trmout allows a JOVIAL program to print a character string of any length on the terminal. Leading blanks, if any, are suppressed. Since the procedure is declared in utilities.cmp_out, it can (and must) be used in a JOVIAL program without being declared.

Syntax:

trmout(char'string, str'lng)$

Arguments:

char'string — is any character string known to the program in which the call to trmout is located.

str'lng — is the length of the string in characters. It should either be a constant or declared with the following declaration:

```
item str'lng i 35 u$
```

trmin.gfort trmin.obj

The procedure trmin allows a JOVIAL program to read a line typed at the terminal into a character string. This character string may be of any length. If the line typed is shorter than the string, it is stored right adjusted with leading blanks. Since the procedure is declared in utilities.cmp_out, it can (and must) be used in a JOVIAL

program without being declared.

Syntax:

trmin(char'string, str'lng)$

Arguments:

char'string   is any character string known to the program in which the call to trmin is located.

str'lng   is the length of the string in characters. It should either be a constant or declared with the following declaration:

item str'lng i 35 u$

tout.gfort tout.obj

The procedure tout allows a JOVIAL program to print part of a character string on the terminal. The character string may be of any length. Since the procedure is declared in utilities.cmp_out, it can (and must) be used in a JOVIAL program without being declared.

Syntax:

tout(char'string, first, str'lng)$

Arguments:

char'string   is any character string known to the program in which the call to tout is located.

first   the rightmost str'lng-first+1 characters of char'string are printed. First should be declared as follows:

item first i 35 u$

str'lng is the length of the string in characters. It should either be a constant or declared with the following declaration:

```
item str'lng i 35 u$
```

frstnb.gmap frstnb.obj

The function frstnb returns the index of the first nonblank character in a specified character string. This string may be of any length. Since the procedure is declared in utilities.cmp_out, it can (and must) be used in a JOVIAL program without being declared.

Syntax:

```
index=frstnb(char'string, str'lng)$
```

Arguments:

index is the index of the first nonblank character in char'string. The count is from the left; the index of the left most character is one. If the string contains no leading blanks then index is set to zero. Index should be declared as follows:

```
item index i 35 u$
```

char'string is any character string known to the program in which the call to frstnb is located.

str'lng is the length of the string in characters. It should either be a constant or declared with the following declaration:

```
item str'lng i 35 u$
```

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.