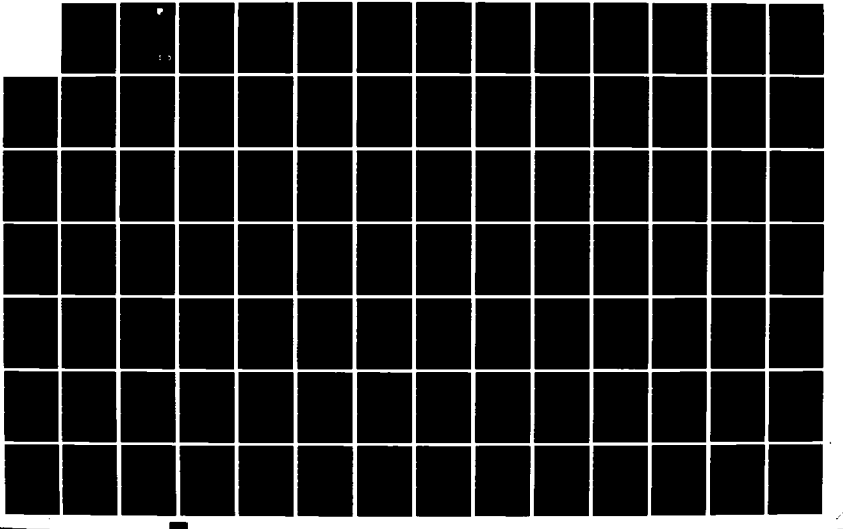
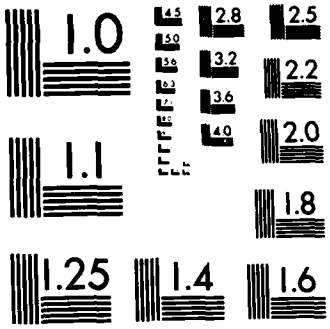
AD-A123 681 JOVIAL PROGRÁM VERIFIER RUGGED JOVIAL ENVIRONMENT(U) 1/3
SRI INTERNATIONAL MENLO PARK CA B ELSPAS ET AL. OCT 82
RADC-TR-82-277 F30602-78-C-0031

UNCLASSIFIED F/G 9/2 NL





RADC-TR-82-277
Final Technical Report
October 1982

JOVIAL PROGRAM VERIFIER RUGGED JOVIAL ENVIRONMENT

SRI International

Bernard Elspas, Dwight F. Hare, Karl N. Levitt and David L. Snyder

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) 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-82-277 has been reviewed and is approved for publication.

APPROVED: 

DONALD F. ROBERTS
Project Engineer

APPROVED: 

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command & Control Division

FOR THE COMMANDER: 

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 (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-277	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) JOVIAL PROGRAM VERIFIER RUGGED JOVIAL ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 14 Nov 77 - 15 Nov 81
		6. PERFORMING ORG. REPORT NUMBER Project 6977
7. AUTHOR(s) Bernard Elspas Karl N. Levitt Dwight F. Hare David L. Snyder		8. CONTRACT OR GRANT NUMBER(s) F30602-78-C-0031
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park CA 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25320205
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE October 1982 13. NUMBER OF PAGES 205
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		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

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

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

Program verification	Hierarchical design methodology
JOVIAL-J73 language	Automatic theorem proving
Proof of correctness	Inductive assertions
Verification condition generation	Rugged JOVIAL Environment

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

This report summarizes project work carried out at the Computer Science Laboratory of SRI International under Contract F30602-78-C-0031 with Rome Air Development Center during the period November 1977 - November 1981. The project has been concerned with the development of a programming environment ("Rugged Jovial Environment")--with verification tools as a principal component--for the specification, development, and formal verification of programs in JOVIAL-J73. The term verification here refers

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

to proof of correctness with respect to formal specification by the method of Floyd assertions.

The effort on this project has ranged from fairly abstract work on formal models of computation (in particular, regarding the semantics of JOVIAL constructs) to practical techniques for implementing these models. This has resulted in the development of a user-friendly system, written in INTERLISP, with the aid of which it is possible to verify JOVIAL software of significant complexity. In addition to our primary project work, we developed several very useful software tools of more general applicability. These tools are also described in the report.

Table of Contents

List of Figures

I. INTRODUCTION

This report summarizes project work carried out at the Computer Science Laboratory of SRI International under Contract F30602-78C-0031 with Rome Air Development Center during the period November 1977 - November 1981. The project has been concerned with the development of a programming environment ("Rugged Jovial Environment")--with verification tools as a principal component--for the specification, development, and formal verification of programs in JOVIAL-J73. The term verification here and elsewhere refers to proof of correctness with respect to formal specification by the method of Floyd assertions [17, 27, 33, 22, 26].[1]

The effort on this project has ranged from fairly abstract work on formal models of computation (in particular, regarding the semantics of JOVIAL constructs) to practical techniques for implementing these models. This has resulted in the development of a user-friendly system, written in INTERLISP, with the aid of which it is possible to verify JOVIAL software of significant complexity. In addition to our primary project work, we developed several very useful software tools of more general applicability. These tools are also described in the report.

The RJE system files have been transferred to the RADC-TOPS20 computer as part of the National Software Works, thereby making the system available to a large user community.

A. OVERALL OBJECTIVE

The overall objective of the RJE work was, as suggested above, to bring out of the laboratory and into practice a number of program development and verification tools integrated into a usable system. In 1977 such tools were available only in relatively imperfect forms--usable only by specialists--and were applicable only to small software modules written in highly subsetted languages (or in ad hoc languages concocted specifically for demonstration purposes). In particular, the best mechanical theorem-proving systems available in 1977 for carrying out the mathematical proofs needed in verifying software were slow, difficult to use, and limited in scope. Most of them were

---

[1] Numbered references appearing in square brackets are collected in the Reference section following the appendices.

either nonextendable or could be extended (with difficulty) only by specialists on those systems [27, 9]. User-interactive features were either entirely absent or quite primitive, and built-in record-keeping for the highly iterative proof processes was usually minimal.

Thus, to make this kind of program verification practical significant progress was needed on several fronts:

1. Development of a fast, powerful, and flexible mechanical theorem prover capable of proving formulas in a user-extendable assertion language.

2. Accommodating as much as possible of a large, complex programming language in wide current use.

3. Providing a user-friendly environment in which a relatively nonspecialized user could have some hope of overcoming the severe conceptual and practical problems involved in the specification, refinement, and verification of practical programs.

4. Combining the verification method based on Floyd assertions with a hierarchical program-development methodology (the SRI HDM approach).

B. REPORT OVERVIEW

In Chapter II of this report we summarize the background of our work over several preceding project efforts conducted at the SRI Computer Science Laboratory in relation to other versions of JOVIAL. These prior efforts were concerned, in particular, with the JOVIAL-J3 and J3-JOCIT languages.

Chapter III comprises a discussion of the several versions of JOVIAL-J73 that were considered with respect to verification during this project.

Chapter IV contains an overview of the structure of the RJE Program Verification System.

In Chapter V a description is given of the means provided in RJE for the parsing and transduction of JOVIAL-J73 programs into an internal representation.

Chapter VI covers the generation of verification conditions for JOVIAL programs that have been annotated with specifications. Thus, this chapter is concerned with the techniques we used for axiomatizing the semantics of JOVIAL-J73. As part of this phase of the work, a technique was developed for mechanizing the production of verification condition generators by means of an

approach called META-VCG.

The theorem-proving component of our system is described in Chapter VII of this report. This is the December 1980 version of the Boyer-Moore Theorem Prover for recursive functions.

Chapter VIII is concerned with the application of the SRI-HDM approach--a technique for the hierarchical specification, design, and verification of program modules--to the verification of suitably structured JOVIAL programs.

Chapter IX provides an overview of the user executive facilities provided in the RJE system. These facilities include on-line documentation of t system's commands, principles of operation, and file structures, as wel numerous bookkeeping features that enable the system user to keep track progress toward final verification of JOVIAL programs.

Chapter X presents some conclusions reached as a result of the project work.

The six appendices contain samples of the RJE system's output in verifying JOVIAL programs, as well as a formatted version of the modified BNF grammar that was developed for JOVIAL-J73A.

C. PROJECT PERSONNEL

The composition of the project team has, of course, varied in the course of the work. The following list includes the principal professional contributors over the past four years.

* Dr. Bernard Elspas, Staff Scientist (Project Leader)
* Mr. Jack Goldberg, Director CSL (Project Supervisor)
* Dr. Robert S. Boyer, Staff Scientist*
* Mr. Milton W. Green, Senior Computer Scientist
* Mr. Dwight F. Hare, Computer Scientist
* Dr. Karl N. Levitt, Associate Director CSL
* Dr. Mark S. Moriconi, Senior Computer Scientist
* Dr. J Strother Moore, Staff Scientist*
* Dr. Robert E. Shostak, Senior Computer Scientist

* Mr. Brad S. Silverberg, Computer Scientist*

* Mr. David L. Snyder, Computer Scientist*

* Dr. Jay M. Spitzen, Computer Scientist*

Those individuals whose titles are flagged with an asterisk are no longer with SRI.

D. ACKNOWLEDGMENTS

The authors of this report express their appreciation to Dorothy Berson and Mary Oakley for their patient assistance in its preparation.

The report was prepared with the aid of the SCRIBE document production system developed by Brian K. Reid [46].

II. PROJECT BACKGROUND

A. INTRODUCTION

Prior to initiation of the RJE project in November 1977, the SRI Computer Science Laboratory worked on two successive efforts closely related to the RJE work. These projects were also concerned with the verification of JOVIAL software, but for versions of JOVIAL other than JOVIAL-J73. The goals and accomplishments of those projects are summarized in this chapter.

B. THE RPE/1 PROJECT

In September 1974 we began work on a verification system for JOVIAL-J3 under Air Force Contract F30602-75-C-0042, entitled "Rugged Programming Environment-RPE/1," supported by Rome Air Development Center (Capt. John M. Ives, ISIS, Project Engineer). This was our first effort to apply formal program-verification methods to a JOVIAL dialect. As was the case for the subsequent projects, the ultimate goal was to make program verification (by means of Floyd assertions [17, 33]) a practical technique for a real programming language, in this case, the J3 dialect of JOVIAL.

During that one-year pilot effort we succeeded in axiomatizing a subset of JOVIAL-J3, and produced a parser/transducer and a verification condition generator for this subset. Two theorem provers were developed in the course of this project--one an INTERLISP implementation of a subgoaling system based on an earlier QA4-QLISP system developed by Waldinger, Levitt and Elspas, and the second a completely new theorem prover (the so-called Tableaux system), created mainly by Spitzen, for proving the generated verification conditions. The JOVIAL-J3 subset handled by the RPE/1 system did not include all the numeric data types of the language, only the signed and unsigned integers. Another restriction of the subset language was that such statement types as the "alternative," "case," and "return" statements were omitted.

It was found, too, that the Tableaux deductive system was rather cumbersome to use for all but the simplest verification conditions. (Its poor speed was due in part to the presence of a rather elaborate proof checker as an option of the Tableaux system. In practice, we rarely made use of the proof checker.) Because of the sluggishness of the Tableaux system (even when the proof checker was omitted), a Presburger decision mechanism was added to the Tableaux system to speed up and simplify the proofs of a subclass of formulas

(the so-called Presburger formulas) that encompasses linear arithmetic over the signed integers. Even with this enhancement the efficacy of the deductive system still left much to be desired.

The major conclusions of this effort were that more work was needed on deductive mechanisms (in particular, on more highly automatic deduction) and in extending the subset of JOVIAL that could be handled. It was also perceived that there was a serious need to mechanize portions of the task of inventing inductive assertions (required as an essential step in the Floyd method for proving programs). The work of the RPE/1 project was described fully in a final technical report issued by RADC in March 1976 [12].

C. THE RPE/2 PROJECT

In April 1976 we began work on the second phase of our verification work related to JOVIAL, entitled "Rugged Programming Environment-RPE/1," and supported by Rome Air Development Center (Capt. John M. Ives, ISIS, Project Engineer) under Air Force Contract F30602-76-C-0204. In this second one-year effort on JOVIAL verification, we were concerned with a distinct dialect of JOVIAL-J3, i.e., the so-called JOCIT dialect [6].

The shortcomings of the original Tableaux deductive system were, in large measure, overcome by translating it from INTERLISP into MACLISP, resulting in a speed-up of about 3 to 1. The Presburger decision algorithm was also greatly improved. However, Tableaux was still rather cumbersome because there was an excessive burden on the user to guide the direction of the proof. Tableaux developed huge goal trees for relatively simple formulas; even for a skilled user it soon became impossible to keep track of all the portions of these trees without resorting to pencil and (large sheets of) paper.

The RPE/2 system was still limited to a subset of a real language, albeit a comparatively richer and fuller subset than the JOVIAL-J3 subset treated by RPE/1. Notably, the RPE/2 system was enabled to handle procedure and function calls (but only in the absence of aliasing), something that was lacking in the RPE/1 system. Although the arithmetic of the mathematical real numbers could be handled after a fashion in the RPE/2 deductive system, this treatment did not include the finite precision aspects of machine arithmetic (i.e., rounding and truncation) for machine fixed-point and floating-point numbers.

One major conclusion of the RPE/2 project was that much additional work

was needed to develop a user-friendly interface that would handle much of the bookkeeping entailed in the highly iterative processes of debugging the program and its assertions and in proving the verification conditions. It was clear, too, that a much more automatic theorem prover was needed. The independent work of Boyer and Moore on automatic deduction was at that time showing considerable promise in this direction. The need to handle the rich variety of data types present in JOVIAL (and other current high-level languages) was also still manifest. The RPE/2 effort was described in a Final Technical Report issued by Rome Air Development Center in June 1977 [13].

III. JOVIAL-J73 VERSIONS

A. INTRODUCTION

During our four years of work, we were involved in a succession of related versions of JOVIAL-J73. This aspect of our effort is summarized in the rest of the chapter. In each case considerable effort (more than we had anticipated) was entailed in becoming acquainted with the new version, analyzing crucial differences, and constructing a new parser/transducer to handle it. The mere alteration of language syntax was troublesome in itself, but more serious difficulties were introduced by subtle changes in the semantics.

B. JOVIAL-J73/I

Shortly after work began on RJE it was mutually agreed that we would build the projected verification system for the Level-I subset of JOVIAL-J73, or JOVIAL-J73/I. This subset is also known as the "avionics subset." The first reference guide we used for this dialect was [58], which also covered two, progressively more extensive subsets, J73/II and J73/III. Subsequently, we received the actual language standard for JOVIAL-J73/I, viz., MIL-STD-1589 (USAF), 28 February 1977 [59]. In August 1980, further documentation for J73/I was obtained from RADC in the form of a "Computer Programming Manual," written by Computer Sciences Corporation [7]. The latter was particularly useful for the numerous sample programs it contained. We were able to produce a parser/transducer for the J73/I subset in fairly short order, since J73/I was a relatively severely subsetted language. It lacked fixed-point numbers, status constants, ordinary tables, rounding on assignment, and a number of other features present in one or the other of the higher subsets, not to mention those features (about 35 in all) included only in the full J73.

The three subsets, J73/I, J73/II, and J73/III, were designed to support the production of different kinds of software systems, and they differ considerably in the sizes and speeds of the machines that can host them. Level-I J73 was intended primarily for avionics systems, executives and operating systems, communication and other real-time control systems. It requires no more than 32K bytes of main memory for compilation.

C. J73A VERSIONS

When our efforts on capturing the semantics of J73/I in terms of a verification condition generator were about a year under way, our work was redirected by RADC toward a new JOVIAL-J73 standard then emerging. The reference manual for this new dialect then existed only in draft form from SofTech, Inc. [57]. Although we were now dealing with the full J73 language (not just the avionics subset) this phase was completed without much difficulty about September 1979 by making use of the INTERPG parser-generating system [54].

The SofTech draft was superseded by Military Standard JOVIAL (J73), MIL-STD-1589A (USAF), March 1979 [60], as the new compiler standard for JOVIAL-J73. A careful comparison was made between the SofTech draft and this new compiler standard. The results indicated a surprisingly large number of differences (about 40), many of them relevant to our work. These changes required further revisions to our parser/transducer. In all, about one month of extra effort was necessitated by the changes that appeared between the SofTech draft and the Military Standard for JOVIAL-J73A.

D. JOVIAL-J73B

In the fall of 1980, a new revision [61] appeared, and the revised language was now known as JOVIAL-J73B. This time we were also given fairly extensive documentation for the differences between the J73A and J73B versions. Our analysis indicated that some of the changes were, in fact, such as to facilitate the semantic analysis required by our verification condition generator. This was true, in particular, for the modifications in the conventions regarding the inclusion of BYREF or BYVAL with formal procedure parameters in J73B. Most of the other changes affected only details of the BNF grammar.

It was necessary to revise the BNF grammar and its transduction augments (see Chap. V, Sec. C) in order to update our parser to the J73B language. In fact, the grammar and augment changes were carried out shortly after we received MIL-STD-1589B [61] (the fall of 1980).

Subsequently, when we turned our attention to remaking the parser for the revised (J73B) grammar, a peculiar bug was discovered in the INTERPG system (actually, in an updated version called PGS, which had been adapted from the

original INTERPG to run entirely on PDP-10 hardware). The original INTERPG [54] was designed to perform the parser table-construction phase on a PDP-11. The replacement of INTERPG by PGS eliminated the annoying necessity of working on two separate machines and moving grammar tables back to the PDP-10. The symptom of the bug was that some array space was being overwritten in the table-generation phase of parser construction for the large J73 grammar. Nevertheless, these useful software tools continued to produce reliable results for smaller grammars.

Various attempts were made to deal with this problem, including (a) trying to discover its source, (b) rebuilding the PGS executive files by reallocating array sizes, (c) shortening the J73B grammar, and (d) partitioning the J73B grammar into two separate portions each of manageable size. Nevertheless, despite our best efforts, none of these expedients were productive. Indeed, it has not been possible to rerun either of the parser generators (INTERPG or PGS) successfully on what appear to be minor variations of the J73A grammar.

The upshot of this stumbling block is that we have a parser for JOVIAL-J73A, but none for J73B. Fortunately, we have not had to modify the JOVIAL grammar for J73A beyond the operative version. Certain transduction augments had to be modified, but it was possible to do this entirely in the INTERLISP phase of construction or, more precisely, by loading modified versions of the augment functions on top of the machine-constructed parser/transducer.

IV. PROGRAM VERIFIER STRUCTURE

A. SYSTEM ENVIRONMENTS

RJE, a program verification system for JOVIAL-J73, consists of several program packages written in INTERLISP-10 [25] and composed of two main environments:

* A front end environment that performs the operations of parsing JOVIAL programs, transducing them to an internal form, and computing verification conditions (VC). This environment also contains a user executive subsystem, described below.

* A theorem prover environment that performs mathematical proofs of correctness of the VCs. The theorem prover is the Theorem Prover for Recursive Functions (1980 version) developed by Boyer and Moore. In addition to the theorem prover proper, this environment also contains its own user executive with an on-line documentation facility.

Both of the above environments can call upon other facilities of the host computer, a text editor being the only other component required for RJE. Although our system is designed to make optimum use of the EMACS text editor, other text editors such as TECO or TVEDIT may be employed instead. The user executive facilities of these two environments also provide for easy passage between the two environments and to/from an EMACS text editor. Communication of results among these environments (including the EMACS, or some other, text editor) is carried out by means of the host computer's file system.

B. AN OVERVIEW OF THE VERIFICATION PROCESS

Verification of correctness consists in establishing that a program text is consistent with a set of formal specifications. By consistency is meant a mathematical proof that whenever the program is executed (in a specified hardware/software environment) with any input data meeting the input specifications, the execution will (a) terminate, i.e, not go into a loop or halt because of some error condition, and (b) upon termination the values of program variables will meet the output specifications. Inclusion of Condition (a) refers to so-called total correctness (the adjective "total" is, however, often omitted). Satisfaction of Condition (b), i.e., consistency without proof of termination, is often termed partial correctness. In this report we shall usually mean the latter when we use the term "correctness." Where the stronger meaning is intended this will be clear from the context. It should be noted that separate proofs of termination are usually easy to carry out,

e.g., by the introduction of "ghost" variables (ones not involved in the original algorithm) [26]. The exclusion of error conditions (and other sources of "unclean" termination) can be guaranteed by methods described in Sec. VI. and Sec. VIII.

The process of verifying correctness for a JOVIAL program consists of the following main steps, whose roles should be understood in general terms before the reader proceeds to the details described in the rest of this report:

1. Annotating the JOVIAL program with specifications in the form of assertions that attempt to characterize the behavior of the program in a manner independent of the actual code.

2. Subjecting the annotated program to the operations of parsing, transduction, and generation of VCs.

3. Proving these VCs (which express in a mathematical form the conditions for consistency between the program code and the assertions) by means of the theorem prover.

Step 1 may be carried out entirely outside the RJE system, but it is most readily performed in the front end environment, with appropriate forking to a text editor (e.g., EMACS) to do the actual annotation.

Step 2 is carried out entirely within the front end environment, which provides numerous aids to the user in the form of on-line assistance and a data base manager that keeps track of the current JOVIAL program's state of verification. The latter feature, in particular, is heavily based on ideas developed by Moriconi [40].

Step 3 is performed entirely within the theorem prover environment. Although this usually entails considerable communication with special files supplementing the Prover's knowledge base, such communication is invisible to the user.

Further details on the the ways in which the RJE system may be used to verify JOVIAL-J73 programs will become apparent in the ensuing chapters. The reader's attention is also directed to the sample verifications appearing in several of the appendices, as well as to the user manual [15].

V. JOVIAL SYNTAX--PARSING AND TRANSDUCTION

A. INTRODUCTION

In this chapter we discuss the first main component of the front end subsystem of the RJE, its parser/transducer, and how it was constructed. Our discussion is partitioned into the actual parsing function and the conceptually distinct (but actually closely interleaved) operation of transducing the source program into an internal form. The second portion of the front end operations, i.e., the generation of verification conditions, is described in the next chapter.

The parser for JOVIAL-J73 described here captures the context-independent portion of the syntax of the language, and therefore represents our axiomatization of those syntactic features. The context-dependent part of the syntax of JOVIAL-J73, as described in the "Constraints" sections of [60], are encapsulated in our verification condition generator.

B. JOVIAL-J73 PARSER

Our parser for JOVIAL-J73 is a machine-generated program. It was constructed by the parser-generating program INTERPG [54] from the following kinds of input specifications:

* A modified BNF grammar (described in Appendix A.)
* Certain auxiliary information--precedence rules, descriptions of pseudoterminals of the language, and lexical-analysis rules.

We describe first the general structure of our parser, emphasizing those features common to all INTERPG-generated parsers. Aspects of the parser particular to JOVIAL are determined largely by the BNF grammar, but also by the auxiliary specifications mentioned above. These JOVIAL-related features will be discussed later.

1. The Parse Function

The top-level function of the JOVIAL parser is JP. (This name will, of course, be different for different parsers, and is supplied to INTERPG by the user when the parser is constructed).

The parse function JP expects its first argument to be the name of a file containing the JOVIAL source code to be parsed. The second and third arguments are optional. If a non-NIL second argument is supplied (e.g., T) the

action of the parser is written out to a trace file (PARSETRACE) in the user's directory. (This file is automatically opened and closed by JP; the user need not first create the file, open it, and so forth). The third argument to JP is another flag, CURRENTPOSFLG, which is used by the function SETUPPARSE (to be described below) to leave the source file open when CURRENTPOSFLG is non-NIL.

The first thing that JP does is set a global variable PARSEFN (used freely by other functions of the parser) to the name of the file supplied as the main (i.e., first) argument to JP. This file name is later used for the various READ operations during parsing and finally, for closing the source file.

The next operation performed by JP is execution of the function JPSETINITIALSTATE(). The code for this operation appears next:

```
(JPSETINITIALSTATE
  (LAMBDA NIL
    (SETQ TERMINALPROP (QUOTE JPTERMINAL))
    (SETQ READTABLE JPREADTABLE)
    (SETQ PPRFLG NIL)
    (SETQ PARSEFN (SETUPPARSE PARSEFN TRACEFLG CURRENTPOSFLG))
    (SETQ CURRENTSTATE JPINITIALSTATE)
    (SETQ PDS JPINITIALPDS)
    (PUSHSTACK CURRENTSTATE NIL)
    (JPREADTOKEN)))
```

Briefly, JPSETINITIALSTATE initializes the various variables and names used by the parser. It makes JPREADTABLE the readtable to be used by the parser; this readtable will differ from the normal INTERLISP readtable in that certain characters and combinations of characters are treated specially (e.g., in JOVIAL it is convenient to read such character strings as '3B1234567' or '1.23E-3' as single special tokens). The JPREADTABLE will have been constructed by INTERPG to facilitate these lexical matters in accordance with instructions supplied by the designer via an INTERLISP variable INTERLISP-SETSYNTAXDOUBLETS (see below) when the parser was created. The variable TERMINALPROP is defined to be JPTERMINAL; this is a property list key used for the terminal symbols of the JOVIAL grammar. Thus, such terminal symbols as IF, AND, THEN, +, -, /, MOD, and the like, receive their appropriate "meanings" (under the key JPTERMINAL) when the parser is initialized. Most of the reserved words of JOVIAL, such as IF, FOR, and the like, receive as their "meanings" the actual reserved word. For

pseudoterminals such as LOGOP, the process is slightly different in that several terminal symbols (in the case of LOGOP they are the logical operators AND, OR, XOR, and EQV) are subsumed under a single pseudoterminal. In these cases the terminals AND, OR, XOR, and EQV receive the meaning LOGOP, and LOGOP itself receives the special meaning NIL (which designates it as a pseudoterminal). The pseudoterminal INEQOP is treated similarly in that the inequality operators <, >, <=, and >= get the meaning INEQOP, while INEQOP receives NIL as its meaning. The other pseudoterminals and their definitions are

```
TYPEOFLIST = ((PLUSMINUS + -)
              (LPARSTAR %(*)
              (STARRPAR *%))
              (SU S U)
              (NMD N M D)
              (OTHERLETTER G H I J K L O Q X Y Z)
              (LOGOP AND OR XOR EQV)
              (INEQOP < > <= >=)
              (LISTOPT LISTEXP LISTINV LISTBOTH)
              (ASSERTKEY ASSERT ASSUME PROVE))
```

The variable TYPEOFLIST must be supplied by the INTERPG user, along with the grammar (GRAMMAR), precedence rules (PRECRULELST), and lexical information (the variable INTERLISP-SETSYNTAXDOUBLETS and the function NEXTTYPE).

Other variables involved in the mechanics of parsing (e.g., CURRENTSTATE) are next initialized, the variable PDS (the pushdown stack used by the parser) receives the initial value JPINITIALPDS, the stack is pushed once (by calling PUSHSTACK on the current state, and finally a token is read from the source file by JPREADTOKEN. This completes the analysis of JPSETINITIALSTATE.

2. Precedence Rules

Operator precedence for JOVIAL was built into the INTERPG-generated parses by specifying the value of the variable PRECRULELST as follows:

```
PRECRULELIST = ((nonassoc NOT)
                (left AND OR EQV XOR)
                (nonassoc < > = <> <= >=)
                (left + -)
                (left * / MOD)
                (left **)))
```

The convention followed by INTERPG in analyzing and using these rules to generate the parser gives NOT the highest, and ** the lowest precedence. It also makes the logical operators AND, OR, EQV, and XOR associative to the left (e.g., 'AA AND BB AND CC' is parsed as '(AA AND BB) AND CC'). The other

left-associative operators +, -, *, /, MOD, and ** are affected likewise. It was not necessary to prescribe operator precedence for assignment (=) and @, subscripting, or function calls, because these are provided directly within the grammar.

3. Lexical Analysis

INTERPG also generates a lexical analyzer for any parser it produces. The input specifications used for this purpose are twofold--a variable, INTERLISP-SETSYNTAXDOUBLETS, and a function NEXTTYPE. We now describe these specifications briefly for the JOVIAL parser JP.

We note first that the reserved words of JOVIAL are among the terminal symbols of the grammar, and will be recognized as such by the parser. Certain character strings other than the reserved words in JOVIAL (as in most high-level languages) are also to be interpreted invariably as atomic symbols, even though they are formed from two or more successive characters. Examples of such atomic symbols in JOVIAL are the following two-character tokens:

(* *) <= >= ** <>

The function that reads a JOVIAL source file into the parser packs character strings into such tokens, i.e., reserved words and the above doublets. On the other hand, some characters, such as +, must always produce a "break" when the parser reads a source file. For example, the character string AA+BB must be interpreted in the same way as AA+ BB, AA +BB, and AA + BB, independent of the presence of spaces.

The variable INTERLISP-SETSYNTAXDOUBLETS of INTERPG specifies which character strings (apart from reserved words) are to be "gobbled up" as tokens of the grammar. The definition used for this variable in producing the JOVIAL grammar is

```
INTERLISP-SETSYNTAXDOUBLETS =
       ((%" (MACRO COMMENTGOBBLE))
        (# (MACRO NOESC NONIMMEDIATE FIRST (LAMBDA
                    NIL
                    (SETQ SAVEFN (CONS PARSEFN SAVEFN))
                    (SETQ PARSEFN
                          (SETUPPARSE (READ PARSEFN
                                            FILERDTBL)))
                    (READ PARSEFN READTABLE))))
        (%% (MACRO PERCENTGOBBLE))
        (@ BREAK)
        (' (MACRO STRINGGOBBLE))
        (%( (MACRO LPARGOBBLE))
        (%) BREAK)
```

```
(* (MACRO STARGOBBLE))
(+ BREAK)
(, BREAK)
(- BREAK)
(/ BREAK)
(; BREAK)
(< (MACRO LESSTHANGOBBLE))
(> (MACRO GREATERTHANGOBBLE))
(= BREAK)
(? BREAK)
(%[ BREAK)
(%] BREAK)
(^ BREAK)
(` (MACRO NOESC NONIMMED ALWAYS (LAMBDA
            NIL
            (PROG1 (LIST (QUOTE `)
                         (READC PARSEFN))
                   (READC PARSEFN)))))
({ BREAK)
(} BREAK)
(~ BREAK)
(! (MACRO DIRECTIVEGOBBLE)))
```

For each initial character of a doublet, e.g., *, which initiates the doublet **, a macro is named (in this instance, STARGOBBLE) that packs just the right combinations. For characters such as the following four:

. , + -

the situation is simpler and we needed only to specify them as BREAK characters. (Note that certain characters above are preceded by a percent sign % because the variable INTERLISP-SETSYNTAXDOUBLETS must be read by INTERLISP). The character % is always treated as an "escape" character in that language and, because parentheses () and square brackets [ ] play special roles as list delimiters, these characters must be supplied as %(, %), %[, %], and %%, respectively.

4. Parsing

The parsing action described in a preceding subsection, "The Parse Function," is applied to the tokens outputted from the lexical analyzer under control of a pushdown stack automaton. The stack is controlled by an INTERLISP list of reduce/shift actions that were derived (by the INTERLISP portion of INTERPG [54]) from parse tables constructed from the BNF grammar for JOVIAL-J73. The BNF grammar shown in Appendix A. was developed from a succession of source documents (principally [57] and [60]). A "raw" grammar was first written directly from the syntax equations of the reference documentation. Next, certain simplifications were carried out by transferring

purely lexical matters (such as the productions for <letter> ::= A | B | ... | Z, and the like) out of the grammar and into the lexical analysis phase described above.

It should be noted that since INTERPG requires an LR(1) grammar, considerable effort had to be devoted to manipulating the grammar into LR(1) form. This is a rather complex process involving much merging of nonterminals and rearranging of production groups. After each such modification, the trial grammar was supplied to INTERPG for checking. In attempting to produce parse tables, INTERPG generates a list of conflicts discovered in the grammar. These are of two types: shift/reduce conflicts and reduce/reduce conflicts. The latter indicate a fatal flaw that must be removed from the grammar, but shift/reduce conflicts are resolved by the parser in favor of the shift action. Thus, if a manual analysis indicates that the action SHIFT is appropriate for each of the shift/reduce conflicts, the parser produced by INTERPG is acceptable. After much effort, we managed in this way to produce a JOVIAL-J73A grammar with no reduce/reduce conflicts and 38 safe shift/reduce conflicts. (This was accomplished after generating several hundred conflicts of both types in the early versions.)

A portion of the statistics reported for this grammar by INTERPG follows.

```
97/400 terminals, 203/500 nonterminals
433/750 grammar rules, 746/2000 states
38 shift/reduce, 0 reduce/reduce conflicts reported
```

C. TRANSDUCTION

The GRAMMAR variable that the INTERPG parser-generator accepts as part of its input contains specifications for the transduced forms to be produced by the parser/transducer. These specifications, called transduction augments, are in addition to the BNF syntax equations that define the grammar itself. Both the transduction augments and the syntax equations appear as components of one large LISP S-expression (the variable GRAMMAR). Syntactic details of the structure of this S-expression are given in [54]. Suffice it to say here that attached to each production of the grammar there must be provided one or more subexpressions in the form of executable LISP functions that take as their arguments the syntactic elements of the right-hand side of the production.

By way of example, the production for the nonterminal

<bit_function_variable> (see production /* 26*/ in A.) is represented in GRAMMAR as

```
(bit_function_variable ((BIT %( variable ,
                          numeric_formula ,
                          numeric_formula %))
                        (LIST 1 3 5 7)))
```

where the component (BIT %(variable , numeric_formula , numeric_formula %)) represents the right-hand side of the BNF equation, and the component (LIST 1 3 5 7) is the transduction augment for this particular production. This specification tells the parser/transducer that, when a concrete bit_function_variable, say 'BIT(BB, II, 100)' is parsed, the desired transduced form should be the list expression (BIT BB II 100). Observe that the numerals 1, 3, 5, and 7 in the transduction augment refer to the first, third, fifth, and seventh elements (i.e., to BIT, BB, II, and 100), respectively. The second, fourth, sixth, and eighth elements (commas and parentheses) are simply discarded in making the transduction.

In the above example (as in the great majority of the augments for our grammar) the only LISP functions used are the list constructors LIST, CONS, APPEND, and the like. However, there is nothing to prevent the use of much more complicated functions, and this was indeed done in some instances where it was necessary to produce internal forms bearing little resemblance to the JOVIAL syntax. In many cases we had to substitute a different key word for the first element of the transduction to permit the VCG to distinguish easily among different usages for a JOVIAL key word. In the example just shown, the JOVIAL key word BIT could be used without ambiguity. In still other cases, a key word had to be invented simply because none was provided in JOVIAL for that nonterminal. An illustration of the last type is provided by the production 'lower-bound : upper-bound' for <case_index>. Here, for example, the concrete case_index '1:100' is transduced to (CASE_IND_BNDS 1 100) by virtue of the augment (LIST (QUOTE CASE_IND_BNDS) 1 3).

One instance in which it was deemed advisable to perform relatively complex processing through the augment function is furnished by the production for the nonterminal <bit_literal>. This production reads

<bit_literal> ::= <bead-size> B <bead-string>

The transduction augment for this case is (LIST 1 (UNPACK.BITSTRING 3)). It constructs a list of length two whose first element is the <bead-size> (a number from 1 to 5) and whose second element is the result of applying the

function UNPACK.BITSTRING to the third component <bead-string> of the <bit-literal>. The component <bead-string> is a character string composed of one or more characters ("beads") selected from the digits 0 through 9 and the letters A through V. (These letters actually represent bead "values" 10 through 31 in JOVIAL). Our parser relaxes this requirement somewhat by accepting any character string for <bead>. However, while expanding the <bead-string> into a LISP list of integers (the bead values), the function UNPACK.BITSTRING also checks for legality of the string as a JOVIAL <bead-string>. If, perchance, an illegal character string is parsed as a <bead-string> the message "Warning: Bad BEAD element" is printed at the user's terminal by the transducer.

VI. JOVIAL SEMANTICS--GENERATION OF VERIFICATION CONDITIONS

A. INTRODUCTION

In this chapter we describe our work in axiomatizing the semantics of JOVIAL-J73. This is, of course, a necessary step in building a program verifier--i.e., a system that can determine whether a JOVIAL program will execute as intended. As described in Sec. IV-B, verification consists in establishing consistency between the text of a program and the programmer's intentions for its behavior. Axiomatization of language semantics is needed so that the system can accurately predict the effects of executing a program with arbitrary input data.

The designer's "intentions" for a program must be determined by writing formal specifications for the program--either as Floyd assertions or in the form of HDM specifications.

There is another, very basic problem here: How does one establish consistency between such formal specifications and the "real" needs of a program user? This problem appears unsolvable in any formal sense. We know of no way to bridge the gap between the informally structured desires in a user's mind with respect to program behavior, on the one hand, and some kind of formal specification. Nor is one likely to be found unless advances in neurophysiology enable us somehow to "get inside" a person's brain. Until then a program designer will simply have to refine his notions of what a program is supposed to do up to the point at which he can express them precisely in a formal specification language. Some will argue that programming languages themselves are that kind of medium, that, "after all, the program expresses the user's intentions directly and precisely." We regard this view as fallacious. The fallacy here is that a program is an explicit set of directions (in fact, too explicit!) that tell the machine exactly what to do and how to do it, rather than what result the program is to accomplish. However, because this basic problem is formally unsolvable does not mean that it is unsolvable in a practical sense. In our opinion, as increasing numbers of program designers and programmers become familiar with formal specification techniques (e.g., through such languages as SPECIAL [56] or OBJ [19], or simply through predicate calculus), the gap between their mental models of intended behavior and the formal specifications will be narrowed considerably.

Ideally, it would be most natural to verify programs written in a language that has been defined in terms of a formal definition, preferably an axiomatic one. With an axiomatization in hand, the effects of all constructs of the language would then be "known" in precisely the form needed to build a verification condition generator. Moreover, conformity of, say, a compiler for the language could be judged against an axiomatic definition, perhaps as readily as against an operational one.

The importance of formal language definition was recognized in the requirements for the preliminary design phase of the IRONMAN development, which culminated in Ada. These requirements included an analysis of the feasibility of carrying out just such a formal language definition. (See [16] as an example one such analysis.)

Unfortunately, no large, rich programming language has ever been completely axiomatized prior to implementation in ways demanded by formal verification of programs written in that language. The closest approach to this ideal is, perhaps, the axiomatization of a subset of Euclid [34], a language designed with this feature in mind. Certainly, languages such as FORTRAN, ALGOL, and JOVIAL--all existing in a variety of dialects and compiler implementations--were not designed with formal semantic specification in mind. The very existence of many divergent and incompatible implementations speaks for the absence of rigorous standards for these languages. Nor do any alternative approaches, e.g., operational (interpretive) semantics or denotational semantics, which have since been applied to such languages, adequately meet the needs of program verification for a strictly axiomatic semantics.

One possible exception to this last statement is the remarkable development by Boyer and Moore of a formal interpreter model (also implemented as a verification condition generator) for a subset of two dialects of ANSI FORTRAN (FORTRAN 66 and FORTRAN 77) [5]. This VCG does not operate on FORTRAN program text, but rather on a fully parenthesized transliteration written in what we would call an "internal form" of the program--one that corresponds to Boyer and Moore's formal model for the language subset. Consequently, if a user wants to apply this VCG in its present version, he must first rewrite a FORTRAN program manually in an unfamiliar form before it can be verified. However, that system includes several unusual features, among them a syntax

checker that enforces all the syntactic restrictions of the FORTRAN subset.

B. LANGUAGE REFERENCE DOCUMENTATION

1. Textual Documentation

As already stated above, JOVIAL-J73 (like all other "real-world" programming languages) lacks any precise formal specification that is directly usable for implementing a verification condition generator. An earlier version of JOVIAL had been specified in terms of an interpreter model called SEMANOL. However, this model depended heavily on informal constituents in the form of narrative-style language qualifications and restrictions that appeared to make it undesirable for us as a semantic model for the VCG.

Our basis for the axiomatization of JOVIAL-J73A was provided by MIL-STD-1589A [60]. This document also served, of course, as the reference for the syntactic description of JOVIAL-73A discussed in Sec. V. The specification of the context-independent part of JOVIAL-J73 syntax is couched in BNF syntax equations, a thoroughly precise and formal medium. Unfortunately, no correspondingly formal specification is given in [60] for either the context-dependent portion of the syntax or the semantics of JOVIAL-J73. The context-dependent syntax is expressed there in terms of "constraints" appended as narrative English text. The semantics appears, also in the form of narrative description, under the heading of "Semantics" for most constructs in [60].

It was necessary, therefore, in axiomatizing JOVIAL semantics, to take account of the above informal descriptions to the extent possible. In many cases where we felt that the reference documentation left some doubt as to its precise meaning, or where there were other ambiguities, we had recourse to a specific JOVIAL-J73 compiler. This is a compiler for J73 on DEC-20 hosts that was developed for the Wright-Patterson AFB Avionics Laboratory. We made extensive use of this AFAL compiler on SRI's DEC-1090T TOPS-20 machine (ARPANET host SRI-KL) to resolve such ambiguities and conflicts whenever necessary. In the next paragraphs we discuss some instances in which recourse to the AFAL compiler was crucial. This compiler was also utilized to check the legality of test programs, including all of the many sample programs that we succeeded in verifying.

2. AFAL Compiler Tests

While attempting to pin down the semantics of JOVIAL-J73, we ran into a number of instances where the textual documentation (e.g., [60]), left us with some uncertainty. We cite here only a few of the potential trouble spots--ones where there were significant changes either in the documentation for the series of military standards [59, 60, 61] or in the corresponding draft versions [7, 57].

1. The case statement, especially with respect to the DEFAULT and FALLTHRU options.

2. The FOR variety of the loop statement.

The above list is not complete, but it should suffice to illustrate the nature of the problems we faced and the ways in which they were resolved.

Case Statement Problems

The syntax of case statement in J73A is given by production /* 72*/ and its descendants in Appendix A. By combining these productions and using [...] for optional syntactic elements, we arrive at the following condensed syntactic form for case statement:

```
CASE formula ';'
   BEGIN ['(' DEFAULT ')' ':' statement [FALLTHRU] ]
   case_alternatives [labels]
   END
```

where case_alternatives is a list of elements, each having the form

```
case_index_group statement [FALLTHRU]
```

The details of case_index_group are irrelevant to this discussion, hence we do not show productions below that point in the grammar.

Analysis of the semantics of the J73A case statement is complicated by the presence of FALLTHRU inside a default-option. Observe that

1. The default-option ['(' DEFAULT ')' ':' statement [FALLTHRU]] appears ahead of the (main) case alternatives.

2. The option FALLTHRU can occur in any or none of the alternatives (even in the default-option).

The statement in the default-option is, of course, intended (according to Sec. 4.4, p. 65 of [60]) to be executed if the value of the case-selector-formula <formula> does not match any of the case index values (in the case_index_groups). However, it is also stated in the same reference that "if FALLTHRU is present after the selected <statement>, the <statement>

in the textually succeeding [our emphasis] <case-alternative> is executed." This poses the possibility of a peculiar kind of interaction between the two options, DEFAULT and FALLTHRU, i.e., if the default-option is selected and it contains a FALLTHRU. Since sub-alternatives in this case.
However strange this interpretation might appear, the documentation [60] seems to require it. Besides, there would otherwise be no point in permitting FALLTHRU to occur inside a default-option. Since the same effects could also be achieved by means of GOTOs in the <statement>s of the case-alternatives, there seems little reason for the presence of such bizarre semantics. Nevertheless, the question demanded resolution.

This dilemma was ultimately resolved only through tests on the AFAL compiler, whereupon we were somewhat surprised by the results. Before conducting these tests, we cross-checked other documentation. An earlier reference for J73A [57] was found to be entirely consistent with the military standard [60]. The documentation for J73/I [59, 7] was of no assistance here because J73/I's SWITCH statement (which corresponds to the CASE statement in J73A) does not contain a FALLTHRU option. However, we noticed that J73B's syntax for case statements differs from J73A's in that default-option is simply one of the two alternative productions for case-alternative. Moreover, a default-option may be placed anywhere in the list of case-alternatives. Our tests of many versions of CASE statements on the AFAL compiler indicated that FALLTHRU from the default-option into the next alternative statement was executed, much to our surprise.

Moreover, the test results provided another surprise in that the AFAL compiler accepted case statements regardless of where the default-option was placed relative to the case-alternatives. The AFAL compiler does not necessarily expect to see the default-option at the end of the case-alternatives in conformity with the J73A specifications. Thus, the AFAL compiler follows J73B syntax in this respect. The default-option can, of course, only "fall through" to a textually following case-alternative. Hence, if the default-option is placed last, any FALLTHRU it might contain is thus rendered ineffective. This is always the case for the last case-alternative (whether or not it is a default option).

During this investigation it was also noted that there is no possibility for effective modification of the case-selector-formula by statements in the case-alternatives. This is so because, once a test succeeds, no further tests are carried out (statements executed by virtue of FALLTHRU are executed unconditionally, i.e., without testing). This aspect was also checked during the compiler runs. Thus, within one execution of a CASE statement, all tests are carried out against the same value of the case-selector-formula, i.e., the value it had upon entry. Subsequent executions, of course, might involve a different value for this formula--for example, if the case statement is re-entered through a GOTO or is inside a loop statement.

We have already mentioned the possibility of GOTO jumps out of case-alternatives. The documentation is quite explicit about this possibility, and also about possible jumps into or between case-alternatives. We find that jumps out of a CASE statement are fairly reasonable, and our VCG provides for them. However, the flow of execution control does not appear to be well defined for jumps into a case-alternative from outside the case statement, nor for jumps between different case-alternatives. We say this because cont into case-alternatives in our axiomatization.

(FOR-Loop Statement Problems

The J73A loop statement exhibits two general classes of potential semantic difficulties. These are (1) behavior in the degenerate cases where either the by/then phrase or the while phrase, or both are absent, (2) differences in behavior under <letter> control-items (represented by a single letter) as contrasted with <control-variable> control-items (representing external data items).

It should be noted that J73/I (e.g., as documented in [7]), also permitted the "initial-phrase" (corresponding to initial-value in J73A or J73B) to be absent, providing a still greater number of peculiar cases to be analyzed. However, the documentation just cited at least provided a detailed table of instances with informal descriptions of the intended semantics for each case. In the J73A version, an initial-value is mandatory in all FOR-loops.

The J73A documentation [60] failed to give the same level of detail as [7]. It was reasonable to assume initially that the corresponding features of J73/I [59, 7] pertained to J73A as well. However, this was also checked on the AFAL compiler, just to be sure.

For the case in which the by/then phrase and the while-phrase are both absent, J73/I specifies that the initialization is carried out and then the controlled-statement is executed "indefinitely" (i.e., presumably until some internal jump, exit, or abort occurs).

In case a by/then phrase, but no while-phrase, is present, J73/I specifies that "the initialization is carried out, next the controlled-statement is executed once, and then the replacement (by-or-then) is carried out on the control-item once before each subsequent execution of the controlled-statement" (again "indefinitely").

Finally, if a while-phrase is present, but no by/then phrase, J73/I specifies that the initialization is performed, and the controlled-statement is then executed repeatedly only if the while-test is TRUE (the test is carried out before each execution, even the first time!).

The tentative assumption that J73A conformed to J73/I in these instances was confirmed by tests on the AFAL compiler. It was, in fact, necessary to test all three cases under both the <item-name> and <letter> assumptions. This was because of the second problem area (2) mentioned earlier, which raises an independent issue: the effects of "hiding" of the (local) letter-type control versus the intended external effects on a (global) <item>-type control item. This matter is discussed next.

The J73A documentation [60] is quite specific about the distinction between the semantics of FOR-loops with <letter> control items and those with <item-name> control items. In the latter case the value of the item-name after execution of the loop is the last value it received in the loop statement. Thus, there is an external global effect on that item. On the other hand, if the control item is a <letter> the scope of this letter is the loop statement. To quote from [60]: "Its value is inaccessible prior to the start of the loop-statement and after the loop-statement concludes."

This description, however, fails to clarify the case of nested loop

statements, where it is conceivable that the same control letter might be used in nested inner and outer loops. While the J73A documentation sheds no light on this question, the J73B documentation (see pp. 65-66 of [61]), expressly forbids a control letter in a loop statement from being the same as the control letter of any enclosing loop statement. However, the AFAL compiler gives a warning message (duplicate name) only when this constraint is violated. References to this control letter in the initial-value of the inner loop are interpeted by the compiler as values from the outer loop scope. Other references to this letter within the inner loop body are interpreted as belonging to the inner scope. Thus, if the only references to the letter within the inner loop body (apart from initialization) are meant to refer to the inner loop control variable, the compiled code then generated will run correctly.

In our axiomatization we have provided strict scoping for letter-type loop-control variables, essentially by a process of renaming. Thus, the VCG operates on an internal form of the loop statement, in which each letter-type loop-control variable receives a new, unique machine-generated name throughout its scope, but not including any nested inner loop scope (in which this same letter--if used as a loop control there as well--would receive another name). Appearances of a loop-control letter in an initial-value, however, refer to the next outermost scope's usage of that letter, in conformity with the AFAL compiler. All other occurrences of this letter (e.g., in the boolean-test, inductive assertion, or by/then phrase) refer to the value local to that inner loop.

In addition to the foregoing version, we also implemented a more restrictive version of VCG that does not permit the same letter-type control-item to appear in nested for-loops. This conforms to a strict reading of the constraints in the J73B documentation. Both item and letter control-items are handled. Only nonreserved letters are allowed as letter control-items.

One final observation must be made with respect to characters that may legally be employed as loop control letters. The letters A, B, C, D, E, F, M, N, P, R, S, T, U, V, and W are, in a sense, reserved "words" of JOVIAL-J73. Each of these 15 characters has a special significance in JOVIAL (e.g., A for fixed-type item-descriptions and conversions, W for words-per-entry in

specified table entries). The JOVIAL standards do not actually forbid using these letters in the context of loop control variables, and the AFAL compiler accepts such usage. Nevertheless, we found it convenient, for ease of implementation, to restrict the naming of loop control variables to the remaining "nonreserved" letters, G, H, I, J, K, L, O, Q, X, Y, and Z. Our parser will detect a violation of this additional restriction as a syntax error. The letters that are legal for this usage are known collectively to the parser as the psuedoterminal OTHERLETTER. (See Sec. V-B and Appendix A.). Observe that (apart from their special uses as "reserved words") single letters can only be used as loop control variables in JOVIAL. Normal data items must be named by JOVIAL names, i.e., by words containing two or more characters.

C. METHODS FOR AXIOMATIZING SEMANTICS

1. Fundamentals of Hoare Semantics

Before we proceed to the details of how we achieved an axiomatization for J73 semantics, it will be appropriate to explain the principles of verification condition generation, and its relation to Hoare semantics and Dijkstra's predicate transformers.

We present first an exposition of the basic ideas underlying the kind of axiomatic semantics that meshes most naturally with the generation of verification conditions for partial correctness proving, i.e., that formulated by Hoare [23], which has come to be called the Hoare calculus. The Hoare calculus is a formal language for talking about and deducing the effects of executing program statements or, more generally, "fragments" of a program. We shall use the term "fragment" to refer to any segment of a program that comprises an executable piece of code by itself. In particular, any single statement, declaration, or a sequence of statements or declarations is a fragment. Incomplete segments of program text, such as the following three J73 segments are not fragments:

```
IF XX<YY; GOTO LL; ELSE;

LL: XX=YY-XX

WHILE XX>YY; BEGIN
```

are not fragments. The second segment, for example, lacks only a final semicolon to make it a fragment in our sense.

The Hoare calculus seeks to characterize the state of a computation at an arbitrary point during execution by means of predicates over the program variables. These predicates are, in effect, Floyd assertions for the program. This calculus relates the states of a computation across a program fragment by relations among these predicates, e.g., predicates true before and after execution of a fragment. Thus, in addition to the notion of program fragment, a Hoare system must build on an underlying assertion language. We take this assertion language to be predicate calculus, but most of the details are irrelevant at this point, except that the language should include an unlimited set of predicate variables--for example, P, Q, R, P1, P2,..., various interpreted functions (e.g., equality and inequalities over program variables), and the usual logical connectives, AND, OR, NOT, and IMPLIES. Logical quantification is permissible, but not essential for our purposes.

Considered as a system of formal logic, a Hoare system consists of sentences (a notion to be defined below) and a set of axioms and rules for singling out certain such sentences as "theorems" of the system. It is important to observe that these sentences belong to a different logical domain from that of the underlying assertions (or predicates).

a. Hoare Sentences

A sentence in the Hoare calculus is a sequence of the form

P{ fragment }Q

where P and Q are logical formulas, i.e, predicates expressed in the assertion language, and "fragment" denotes any program fragment. P and Q will, in general, make reference to program variables. P and Q constrain the states of all program variables to which they refer by insisting that the values of those variables be such as to make P (or Q, respectively) true. In the sentence P{fragment}Q, P refers to the program state existing immediately before execution of fragment; Q refers to the state immediately after execution of fragment. P and Q are usually called the precondition and postcondition, respectively.

The interpretation (or meaning) of the above sentence is as follows:

"If execution of fragment is initiated with values
for the program variables such that P is true, and
if fragment completes execution, then Q must
be true after completion."

The underscoring of the phrase "if fragment completes execution" stresses the fact that we are dealing here with partial correctness. The Hoare calculus

cannot deal directly with most questions relating to program termination, nor with total-correctness assertions.

We first discuss some particularly trivial examples informally, i.e, in terms of the interpretation just given. In the next subsection we continue the formal development, where, it must be emphasized, we may not appeal to this interpretation. From a formal standpoint, a Hoare sentence is then to be considered valid if and only if it is either an axiom or can be deduced from axioms by the inference rules of the system.

Observe that either P or Q (or both) may be the constant predicates, T or F (i.e., the logical constants, true or false). Thus, according to the informal meaning, "T{fragment}Q" asserts that Q must be true after fragment has been executed, whatever the precondition state. This is an extremely strong assertion, one that we hardly expect to hold for any nontrivial Q; from it one can deduce (informally) that p{fragment}Q is also valid (for any precondition p). Later we shall see how to make such deductions formally.

Another special case of interest is P{fragment}T. This asserts that if P was true before, then T is true after executing fragment. But T is (per se) true. Hence, this sentence in no way constrains the semantics of fragment.

Consider also the two sentences F{fragment}Q and P{fragment}F. Like the sentence P{fragment}T, the first is trivially valid because the precondition F (for "false") can never hold. The second, however, says that after execution has been initiated with P true, if execution runs to completion, then F is true. Since F can never be true, this means that fragment cannot complete execution after initiation in a state with P true. Thus, T{fragment}F gives us a way to express the notion that no execution of fragment can run to completion. Unfortunately, though the Hoare calculus can express nontermination in this way, it cannot formalize termination. That is, ~[T{fragment}F] is not a sentence of the Hoare calculus. [Observe that (~T){fragment}F, which is a Hoare sentence, is equivalent to the sentence F{fragment}F, already perceived as trivially valid.]

b. Hoare Axioms

A Hoare axiom is simply a Hoare sentence, as already defined, except that we regard it as implicitly quantified universally over all free variables appearing in it. Hoare axioms may therefore be used to express constraints on the specific program construct appearing between the curly brackets {...}. However, some Hoare axioms must also be considered as axiom schemas--fundamental to any Hoare calculus. The following axiom schemas are basic:

Axiom A1. T{fragment}T, for any fragment.

Axiom A2. F{fragment}F, for any fragment.

Let us now consider a Hoare axiom that relates to a specific program construct. For this illustration we use assignment, since it is a notion fundamental to all program languages. In JOVIAL-J73 an assignment statement can have the following form:

variable = formula ;

We restrict our attention to the case of a simple program variable, say XX (assumed not to be aliased to any other variable), and a formula E whose evaluation produces no side effects. In this case, any property Q that is supposed to be true (with regard to XX and other variables) after execution of the assignment must also have been true for the formula E before execution. Let Q[XX/E] denote the result of substituting E for all (free) occurrences of XX in Q. Thus,

Q[XX/E]{XX = E;}Q

is an axiom that defines the semantics of simple variable, nonaliased assignment without side effects. Later we shall see how this can be broadened to remove at least some of the above restrictions.

Let us note the roles of free variables (both predicate symbols and program variables) in the way we interpret Hoare axioms. There is an implicit universal quantification over all such free variables. That is, in the axiom for restricted simple assignment shown above, it is intended that Q denote any predicate formula, that XX be any simple, nonaliased program variable, and that E refer to any JOVIAL expression without side effects. The above axiom may therefore be used to apply to any JOVIAL assignment statement (subject to the above restrictions).

c. Hoare Rules

Hoare rules are the inference rules for the Hoare calculus. A Hoare system of axioms and rules permits one to establish a set of Hoare sentences that are to be regarded as valid in that system. The axioms are valid (by assumption), and any Hoare sentence that can be shown to be a consequence of the axioms and rules is also valid in that calculus.

A Hoare rule has the form "hypothesis |- conclusion," which we shall also write as follows:

$$\frac{\text{hypotheses}}{\text{conclusion}}$$

Here conclusion is a Hoare sentence and hypotheses is a comma-separated list of logical formulas and Hoare sentences. This is to be read as "From hypotheses you may infer conclusion."

Like the basic axiom schemas listed above, some rules are really rule schemas, that would be included in any Hoare calculus, regardless of the particular program language.

One such rule schema is the following Rule of Consequence:

$$\frac{P \Rightarrow P1,\ Q1 \Rightarrow Q,\ P1\{\text{fragment}\}Q1}{P\{\text{fragment}\}Q}$$

Thus, if it can be shown that P implies P1, that Q1 implies Q, and that the hypothesis Hoare sentence P1{fragment}Q1 is valid, this inference rule states that one may therefore infer the validity of the conclusion P{fragment}Q. Note that "fragment" must refer to the same program fragment in both hypothesis and conclusion.

From the Rule of Consequence, the tautology P=>T, and Axiom A1, we have the following derived result:

A1*. P{fragment}T

Similarly, from the Rule of Consequence, the tautology F=>Q, and Axiom A2 we obtain

A2*. F{fragment}Q

Of course, we might just as well have taken A1* and A2* as axioms, but these derivations serve as useful illustrations of the basic ideas.

Two other inference rule schemas are needed (as axioms) to flesh out the desired interpretation for Hoare sentences. They are as follows:

Disjunction Rule:

$$\frac{P\{fragment\}R,\ Q\{fragment\}R}{(P \text{ or } Q)\{fragment\}R}$$

Conjunction Rule:

$$\frac{P\{fragment\}Q,\ P\{fragment\}R}{P\{fragment\}(Q \text{ and } R)}$$

We can justify these two rules by means of informal reasoning. The rationale for the Disjunction Rule is as follows. One of the two preconditions P, Q must hold for any initial state in which (P or Q) is satisfied. By the first hypothesis, execution of fragment from an initial state satisfying P guarantees that R holds in the final state, or that execution fails to complete. The second hypothesis guarantees the same thing for any initial state satisfying Q. Hence, it must be the case that, if (P or Q) holds initially, any terminating execution of fragment leads to a final state in which R is satisfied.

The corresponding rationale for the Conjunction Rule is simpler. Any terminating execution of fragment started from a state in which P is true must lead to a final state satisfying both Q and R, i.e., one satisfying the predicate (Q AND R).

2. The Hoare Calculus and Predicate Transformers

The generation of verification conditions in RJE is implemented in terms of predicate transformers rather than directly through Hoare rules. However, the notion predicate transformer (generally credited to Dijkstra [11]), is closely related to the Hoare calculus, as we shall see shortly.

Let fragment denote an arbitrary piece of executable program text, and let Q be any postcondition predicate for fragment, as in our earlier discussion. We now ask: "What is the weakest precondition P* for which the Hoare sentence P*{fragment}Q holds?" Clearly P* is the predicate that is true precisely for those initial states that guarantee that Q will hold after any terminating execution of fragment. Among all the predicates P for which P{fragment}Q is valid, P* is the unique one that is implied by any such P.

We use P* to define the predicate transformer function wlp as follows:

```
wlp: Frag x Pred --> Pred
```

where Frag is the class of fragments and Pred is the class of predicates in our assertion language.

That is, wlp(fragment, Q) is defined to return the predicate P* for an arbitrary fragment in Frag and any predicate Q in Pred. The notation "wlp" is meant to suggest "weakest liberal precondition," with "liberal" as a reminder that we are talking about partial correctness. That is, if execution of fragment is initiated with wlp(fragment, Q) = true, and if execution is completed, the state upon completion will satisfy postcondition Q; moreover, no initial state for which wlp(fragment, Q) is false will have this result.

Note that if all executions of fragment terminate and no initial state exists for which Q can be guaranteed to hold upon completion, then wlp(fragment, Q) = F. Likewise, if for every possible initial state, execution of fragment either leads to completion with Q true or fails to terminate, then wlp(fragment, Q) = T.

In particular, we have the following special relations:

wlp(fragment, T) = T, for any fragment.

wlp(fragment, F) = F, if all executions of fragment terminate.

The crucial relations between the Hoare and Dijkstra viewpoints may be summarized as follows:

1. wlp(fragment, q){fragment}q
2. If p{fragment}q then p implies wlp(fragment, q)

Thus, (1) states that wlp is a sufficient precondition for q across any execution of fragment, while (2) states that wlp is the weakest such precondition.

The predicate transformer wlp satisfies the following "monotonicity" relation with respect to implication by virtue of the Rule of Consequence:

If p => q then wlp(frag, p) => wlp(frag, q).

The rules of conjunction and disjunction discussed in the preceding subsection have their following counterparts in the wlp formalism:

wlp(frag, p AND q) = wlp(frag, p) AND wlp(frag, q)

wlp(frag, p) OR wlp(frag, q) => wlp(frag, p OR q)

These two rules are readily deduced from the above monotonicity law and the Rule of Consequence. It should be noted that the second rule (relating to disjunctions) is only a one-way implication when nondeterministic computations

are allowed, i.e., computations in which execution of the same fragment may lead to different final states. However, if we restrict ourselves to deterministic computations, the second law becomes an equivalence like the first law.

At least one alternative notion of predicate transformer deserves mention here. Most of Dijkstra's discussion of predicate transformers (see [11]) is in terms of the stronger notion of "weakest (nonliberal) precondition," or wp. For this notion termination is not assumed, but is guaranteed by the truth of wp(fragment, Q). We shall, however, make no further use of this concept.

The most striking difference between the Hoare and Dijkstra formalisms is that Hoare sentences are in a "higher-order" domain than the predicate domain for preconditions and postconditions. On the other hand, in the Dijkstra predicate transformer approach, all assertions are in the same predicate domain, i.e., the assertion language. This uniformity facilitates the direct implementation of verification condition generation in terms of predicate transformers. Nevertheless, because of the close correspondence between the two viewpoints, one can usually interpret predicate transformers as Hoare rules and vice versa.

D. VERIFICATION CONDITION GENERATION

1. Overview

As indicated in Sec. IV-B, the task of the verification condition generator (VCG) is to generate mathematical-logical formulas that express consistency conditions between the program text to be verified and its formal specifications, and to do this in the light of the semantics of the programming language (JOVIAL-J73A) and the constraints imposed by the machine environment. The semantic and machine-implementation-constraint questions were addressed in the two preceding sections.

Here we describe our VCG mechanisms in some detail and also how they were constructed from a partial axiomatization for our subset of JOVIAL-J73A.

The VCG must accept any program (legal in our subset) to which inductive assertions have been affixed (as formal specifications of behavior), while rejecting with either a warning or a fatal error any program that violates semantic or context-dependent syntax restrictions. If the VCG accepts the

program, it must generate a list of verification conditions (VC) for subsequent validation by the Theorem Prover.

Our VCG performs this task by applying a system of recursive LISP functions to the parsed internal form (PIF) produced by the parser/transducer as described in Sec. V-B. These recursive functions are implementations of the predicate transformers discussed in Sec. C-2 of the present chapter.

At the top level, a function called PROCESS is applied to the whole PIF. PIF, the transduced form of a JOVIAL complete-program, already contains transduced versions of the internal inductive assertions (through the inclusion of ASSERT statements in the program). PROCESS must then work its way down to the subprogram level, collecting symbol table information (declarations, presets, and the like) along the way for later use. A succession of subfunctions (e.g., PROCESS.COMPLETEPROGRAM, PROCESS.MODULE, PROCESS.PROCMODULE, PROCESS.MPMODULE1, PROCESS.MPMODULE2, and WP.SUBRS) is invoked on corresponding pieces of the PIF until the level of a single subroutine body is reached with a call to WP.SUBRBODY.

The function WP.SUBRBODY and those called subsequently all take two arguments, a piece of (transduced) code, and a postcondition. These functions too are predicate transformers in the sense defined in Sec. C-2. In general, there is a particular predicate transformer for each executable construct of the language. The outermost call to WP.SUBRBODY employs the postcondition T as its second (predicate) argument. It produces VCs through side effects and returns the weakest precondition T. Ultimately, a function named WP is reached.

The function WP is, in effect, the master predicate transformer function acting at the level of statements and declarations. It acts as a large switch to invoke the appropriate subfunctions, e.g., WP.ASSERT, WP.ASSIGN, WP.CASE, WP.COMPDSTAT, WP.DECL1, WP.DECL2, WP.DECLS, WP.EXIT, WP.FOR.AFAL, WP.FOR, WP.GOTO, WP.IFSTAT, WP.LOOPSTAT, WP.LSTAT, WP.MASSIGN, WP.PROC.CALL, WP.RHEXPR, WP.STATEMENT, WP.STATEMENTS, WP.STATS, WP.SUBRBODY, WP.SUBRS, and WP.WHILE. In general, each of these WP subunctions implements the predicate transformation appropriate to a particular type of JOVIAL statement or declaration. WP.LOOPSTAT, for example, hands its task over to either WP.FOR or WP.WHILE, depending on the kind of loop statement it sees. The switching

action is implemented through the INTERLISP pattern matcher in conjunction with a clever "production accessor" facility designed and built by David Snyder. This facility is table-driven by the same GRAMMAR variable (see Sec. V-B) used by INTERPG in building the parser/transducer.

The semantics of JOVIAL constructs has been given a virtual axiomatization through the implementation of their corresponding predicate transformers. In the next subsection we shall describe the details of some representative instances and relate them to the Hoare-type rules on which their implementation is based. In Sec. D-3 we outline results obtained on a much more formal approach, developed in part on this project, whereby the construction of such predicate transformer functions can be made largely automatic. In this Meta-VCG technique, only the Hoare rules need be written manually; the system then mechanically constructs the required predicate transformers that implement the Hoare rules.

2. Predicate Transformer Functions

In order to illustrate the action of predicate transformer functions in our VCG we have chosen the following interesting cases:

* The assertion "statement"
* The compound statement
* The conditional statement
* The assignment statement
* The GOTO statement
* WHILE and FOR statements.

a. Assert Statements

There is, of course, no such thing as an "assert statement" in JOVIAL. It constitutes an addition made to the JOVIAL grammar (in our parser) simply to enable RJE to perform verification by means of Floyd assertions. In its simplest form, the added <assert_statement> has the following syntax:

```
assert_statement ::= ASSERT formula ';'
```

However, when such assertions are inserted into a program intended for submission to a JOVIAL compiler, the assert_statement must be enclosed in %...% comment brackets. Thus, the assertion will be treated as a comment by that compiler. The bracketing percent signs are stripped by our parser during

the lexical-analysis phase. By this means we manage to preserve compatibility with JOVIAL compilers even though the assert_statement has been added to our parser's grammar.

The semantics of assertion is captured by the following Hoare rule:

$$\frac{\text{formula} \Rightarrow Q}{\text{formula}\{\text{ASSERT formula ;}\}Q}$$

Thus, the weakest precondition for an arbitrary postcondition Q and the assert_statement shown is the "asserted" formula. In addition, verification of the Hoare sentence "formula {ASSERT formula ;}Q" requires that the subsidiary hypothesis "formula => Q" be validated.

This Hoare rule corresponds exactly to the following predicate transformer:

wp("ASSERT formula ;" , Q) = formula

with an additional proviso, i.e., that execution of the function wp produce the side effect of adding the subsidiary clause, formula=>Q, to the list of VCs to be proved. A complete rationale for this equivalence is given, for example, in [14], pp. 36-37. The reader should not be confused by our use of "wp" here in place of the more appropriate "wlp." We shall actually be talking about weakest liberal preconditions from here on. (See Secs. C-1a and C-2).

An INTERLISP function that implements this rule is defined as follows:

and

```
(WP.ASSERT
  (LAMBDA (FORMULA Q)
    (NCONC1 VCS (MAKE.IMPLIES FORMULA Q))
    FORMULA))
```

Thus, the function returns the predicate argument Q as its value and also produces the side effect of appending (by means of NCONC1) the implication (IMPLIES FORMULA Q) to the variable VCS.

b. Compound Statement

The JOVIAL compound statement is a BEGIN ... END block of statements. The rule for such a block is the trivial one

$$\frac{P\{\text{statements}\}Q}{P\{\text{BEGIN statements END}\}Q}$$

This simply says that the BEGIN...END brackets have no semantic import besides

grouping the embedded list of statements into a single executable statement. Of course, we shall not know how to compute the precondition P until we also give a rule for P{statements}Q.

The required rule is most easily stated in a recursive form for a list consisting of a single statement followed by more statements:

$$\frac{r\{\text{statements}\}Q,\ p\{\text{statement}\}r}{p\{\text{statement statements}\}Q}$$

where we have written the auxiliary predicates p and r in lowercase characters to emphasize a new situation. These auxiliary predicates must be computed from the postcondition Q and recursive applications of the predicate transformer wp on the two auxiliary constructs appearing as hypothesis clauses. (See also Sec. D-3).

The corresponding predicate transformer definition is provided by the following recursion:

```
wp(statement statements, Q) =
        wp[statement, wp(statements, Q)]

wp(empty-statement-list, Q) = Q
```

The LISP code that implements this definition is simply

```
(WP.COMPDSTAT
    (LAMBDA (BLOCK Q)
      (COND
        ((NULL BLOCK)
          Q)
        (T (WP (CAR BLOCK)
               (WP.COMPDSTAT (CDR BLOCK)
                             Q))))))
```

This shows clearly the direct correspondence between the recursive version of wp for a statement list and the LISP code. Observe that (CAR BLOCK) is the initial statement of the BLOCK and that (CDR BLOCK) is the rest of the list.

c. Conditional Statements

There are really two types of conditional statements in JOVIAL-J73 as shown in the BNF grammar of Appendix A., productions numbered /*188*/ and /*189*/.

```
IF boolean_formula ';' statement else_clause

IF boolean_formula ';' statement
```

However, the parser/transducer is arranged to transduce both to the same internal form (with NIL appearing for the missing else_clause of the simple

IF). Consequently we can subsume both into the following single Hoare rule:

$$\frac{p\{\text{statement}\}Q,\ r\{\text{else\_statement}\}Q}{(\text{boolean} \Rightarrow p)\ \&\ (\sim\text{boolean} \Rightarrow r)\ \{\text{IF boolean ; statement else\_clause}\}Q}$$

where else_clause is ELSE else_statement, or is NIL.

It is easily seen (cf. [14], pp. 37-38) that the above rule for conditional statements is equivalent to the following predicate transformer:

```
wp(IF boolean ; statement else_clause, Q) =
        boolean=> wp(statement, Q) &
        ~boolean=> wp(else_statement, Q)
```

Here p = wp(statement, Q) and r = wp(else_statement, Q).

d. Assignment Statements

We have already discussed (in Sec. C-1b) the Hoare rule for assignment to simple, nonaliased variables when the righthand side produces no side effects.

The more general situation present in JOVIAL-J73A poses a number of additional difficulties, as follows:

1. A list of variables on the left side.
2. Structured variables on the left side.
3. Aliasing of a left-side variable to other program variables.
4. Side effects in the evaluation of the right hand side of the assignment.

The presence of several variables (as a comma-separated list) on the left side of a JOVIAL assignment appears at first to be a trivial generalization of the simple case. It might seem that the assignment "XX, YY, ZZ = expression;" is equivalent to the following sequence of assignments:

```
XX=expression; YY=expression; ZZ=expression;
```

However, first appearances are often deceptive! The reason is that expression may contain references to XX or YY, in which case the first sequential assignment modifies the value of expression before YY gets its value, and again modifies expression (through YY) before ZZ is assigned. As a result, XX, YY, and ZZ might acquire three different values! JOVIAL semantics is very clear on this point: the right-hand side expression is evaluated just once (after all the left-hand addresses have been evaluated); this value is saved and used to provide the same new value for all the left-side variables.

Thus, the proper paradigm for the assignment shown for XX, YY, and ZZ is

as follows:

```
TEMP = expression;
XX = TEMP;
YY = TEMP;
ZZ = TEMP;
```

In terms of a Hoare rule, we could write

$$\frac{p\{\text{BEGIN new=rhs; v1=new; ... vn=new; END}\}Q}{p\{v1,\ldots,vn = rhs;\}Q}$$

to capture this equivalence. The LISP implementation for this aspect of JOVIAL assignment is provided by the introduction, through the INTERLISP function GENSYM, of a unique, newly named variable ("new" above) for all multiple left-side assignments. The wp predicate transformer function for this case is called WP.MASSIGN. We do not exhibit it here because it also contains provisions for the proper handling of structured variables (i.e., table references) on the left side of the assignment. This is discussed below.

Table references appearing on the left side of an assignment pose an entirely different problem (but also one that interacts with the multiple left-hand-side problem). Our method of handling this situation is discussed in detail elsewhere in this report (see Sec. VII-C-2a). Suffice it to say at this point that assignment to a table is axiomatized by essentially the same functionalization originally suggested by John McCarthy, in terms of two functions, which he called "access" and "change." Here they are named SELECT and ALPHA. It may be noted that similar means for treating arrays have been employed by many other authors, both in papers and in the implementation of program verifiers (see [18], for example).

e. GOTO Statements

The use of GOTOs is often decried as an "unstructured" approach to programming. Indeed, GOTOs are somewhat troublesome in program verification, which may be indicative of the same phenomenon. However, provided one is willing to insert additional assertions at all statement labels actually addressed by a GOTO, they are not really much trouble. The whole point is to attach at each such label an invariant that captures the program state whenever control resides at that point (regardless of the path by which control reached it). This assertion can then be "reflected back" to each GOTO statement targeting that label.

Handling GOTOs in this way did not even require further modifications of JOVIAL syntax beyond the addition of assert_statement as one kind of statement. All that is necessary is to interpose an assert_statement between each label and the statement (or other construct) that is nominally so labeled. In other words, the interposed assertion becomes the labeled statement. Since assertions (enclosed in % brackets) are transparent to a JOVIAL compiler, this interposition does not affect the semantics of the program.

The usual Hoare rule for GOTOs has the form of an axiom invariant{GOTO label;}Q, where invariant is the assertion predicate of the assertion introduced between the target label and the labeled statement. Thus, the axiom may also be written as (GET.ASRTN.AT.LABEL label){GOTO label;}Q, where the function GET.ASRTN.AT.LABEL returns the required invariant assertion. This function simply retrieves the assertion from the property list of the atom label, where it was stored during parsing. Thus, the predicate transformer for GOTO statements, WP.GOTO(stat, Q) is defined simply as (GET.ASRTN.AT.LABEL (CADR stat)).

f. While Statement

The while statement of our JOVIAL grammar differs from that of [60] by virtue of the addition of an (optional) assertion form immediately after the key word WHILE.

```
WHILE [ASSERT formula ';']
      boolean ';' statement
```

The assertion form "ASSERT formula ;" provides the inductive invariant needed to capture the semantics of the loop, i.e., to provide a VC for passage around the loop. The form boolean is a <boolean_formula> that provides the test for exit from the loop; <statement> is the loop body. As is the case elsewhere for assertions in our system, the assertion form must be enclosed in %-type comment brackets for the WHILE statement to be accepted by an actual JOVIAL compiler. Thus,

```
WHILE %ASSERT formula;%  boolean; statement
```

is what a WHILE statement actually looks like in an executable program. The RJE lexical analyzer, of course, strips the %-comment brackets to make the assertion available to the parser.

The operative Hoare rule is as follows:

```
   formula & boolean{statement}formula,
   formula & ~boolean => Q
  ------------------------------------------------
  formula{WHILE ASSERT formula ; boolean statement}Q
```

Thus, wp for WHILE statements returns the inductive invariant <formula> as its value (the weakest liberal precondition). In addition, the two subsidiary hypotheses are appended to the list of VCs for the program being analyzed. The second hypothesis, "formula & boolean => Q," is the VC for exit from the loop statement. The first hypothesis entails a recursive call to wp that computes the weakest precondition wp(statement, Q), i.e., the assertion that must be shown as holding just before the loop body <statement> is executed. This results in construction of the VC for the path around the loop, i.e., the formula "formula & ~boolean => wp(statement, Q)."

There is an additional complication produced by the need to handle EXIT statements occurring in the loop body. The mechanism that handles this complication is somewhat similar to that used for GOTOs. Whenever an exit_statement is encountered while wp(statement, Q) is being processed (inside a loop statement, either of the WHILE or FOR type) the postcondition Q for the whole WHILE statement is returned by wp(exit_statement, q). To do this, the VCG maintains a stack of LOOPEXITASRTNS, pushing the current postcondition Q onto the stack for each call made by wp on loop statements, and then popping this stack to return the most recent postcondition Q (as the value of wp(exit_statement, q)) whenever an EXIT statement is encountered. Thus, the stack discipline ensures that the returned postcondition corresponds to the Q of the innermost loop surrounding the EXIT statement. An error is generated if an EXIT is seen with no surrounding loop statement from which to exit.

It seems impossible to describe this control mechanism strictly in terms of the Hoare formalism, as is similarly the case for other control jumps that may interact with the other Hoare rules. Observe that our handling of GOTOs also represents an extension to strict Hoare rules in that the whole burden of the axiomatization is placed on the function that returns the assertion predicate associated with the targeted label.

The other kind of JOVIAL loop statement, i.e., the FOR statement, is handled by our VCG in much the same way as the WHILE statement. As may be apparent from the discussion of FOR statement semantics in Sec. B-2, the use of a local control variable (letter control) complicates matters considerably.

This is managed by renaming occurrences of the control-letter by a new and unique name (as was done for the temporary variable required for multiple left-side assignments in Sec. D-2). In this case we have taken greater precautions to guarantee uniqueness of the name by using a more elaborate renaming function than INTERLISP's GENSYM. Further complications in handling FOR loops are caused by the options of by-or-then clauses, which modify the control variable following each execution of the loop body. These features require that the loop invariant be "pushed back" through a virtual assignment statement involving the replacement or incrementation caused by the by-phrase or then-phrase, respectively. As with the WHILE statement, any EXITs from the body are handled by the LOOPEXITASRTN stack.

3. The Meta-VCG Approach

a. The Basic Idea

It may have occurred to the reader that the relationship between the Hoare formalism and its implementation in terms of predicate transformers could conceivably be mechanized. If a VCG is regarded as being a kind of compiler, this mechanization would then correspond to a metacompiler. With such a mechanization, a set of Hoare rules defining the semantics of a programming language could be transformed into a set of predicate transformer functions that implement a VCG in consistency with the Hoare rules.

Let us consider briefly what such a mechanization of the mapping

BNF Grammar x Hoare Rules --> Predicate Transformers

entails. The relationship discussed in Sec. C-2 requires that the Hoare rules be written in a somewhat restricted form. First of all, the conclusion of each rule must have a free predicate variable as its postcondition. This is so because the desired predicate transformer function wp(frag, q) is to be defined for an arbitrary postcondition q. Hence, "forward" rules, e.g., P{ASSUME A;}P&A, are inadmissible. In many cases, however, such rules can be transformed to equivalent ones in the required form. The forward rule for ASSUME, for example, also be written in the equivalent "backward" form (A=>Q){ASSUME A;}Q, as shown in [14].

A second restriction needed to ensure the mechanizability of a set of Hoare rules relates to the dependencies among subsidiary predicate variables. This restriction requires, roughly speaking, that any subsidiary hypotheses present in a rule must be capable of being ordered so that the bindings of

intermediate predicate variables can be computed in succession. This was seen, for example, in the way we arranged the hypotheses for the rules for the compound statement (in Sec. D-2b) and conditional statement (in Sec. D-2c). Note too that any Hoare sentences appearing as hypotheses must have free predicate variables as their preconditions. Although there are other restrictions besides the two cited, we must refer the interested reader to [39] for details, as a full discussion would involve highly technical matters quite beyond the scope of the present description.

The initial ideas that led us to pursue the possibility of mechanizing the production of the JOVIAL VCG along the above lines were suggested by Lawrence Flon (University of Southern California) during a visiting appointment at our laboratory in the summer of 1980. At that time Flon and Moriconi developed the first version of a Meta-VCG that could handle the transformation of a set of Hoare rules (suitably restricted) in conjunction with the syntactic specification of a language (by a BNF grammar).

This initial version seemed sufficiently promising to warrant subsequent refinement and elaboration of the approach by Moriconi and Schwartz, leading ultimately to a much more versatile system of this type, META-VCG. The underlying theory is given in their 1981 paper [39]. This implementation was considerably less restrictive in regard to the Hoare rule syntax it could accept. Moreover, it also included many technical improvements affecting the way nonterminal subfragments could be referenced and bound in the Hoare sentences of the rules.

b. Further Details

Let us consider the rule for while statements (Sec. D-2f) to examine in more detail what is involved in the transformation of a rule to a predicate transformer. Basically, what was done (manually) to transform that rule into a predicate transformer WP.WHILE(stat, Q) was to construct LISP code that performs the following functions:

1. Extract from the generic while-statement syntax (the construct appearing inside the Hoare {...} brackets of the conclusion) those components (e.g., the invariant, boolean-formula, and statement) that will be needed in forming the returned precondition and any subsidiary hypotheses.

2. Construct the subsidiary statement forms that appear in Hoare sentences of the hypothesis, and their pre- and postcondition expressions.

3. Cause invocation of the function wp to take place successively for any such subsidiary hypotheses, and bind the corresponding precondition expressions (which must be predicate variables!) to the returned values of the corresponding calls.

4. Construct the formulas for any (base logic) hypotheses (such as "formula & ~boolean => Q") that are not Hoare sentences with the appropriate bindings for their free variables, as determined from either the original matchings of Step 1, or from Step 3.

5. Output all the VCs that are generated from subsidiary hypotheses to a global list of VCs.

6. Return as the ultimate precondition value for this call to WP.WHILE the (bindings for the) precondition expression appearing in the conclusion of the rule.

This may sound rather involved, but it is actually relatively simple for rules like those for the WHILE and IF statements. This is because those rules were already in a canonical form permitting direct implementation (at least in the absence of complications such as loop EXITs and local variables). As already indicated, some rules require prior transformation into an admissible form before they can be used to generate a predicate transformer.

The actual implementation of a software system capable of mechanically transforming a wide variety of Hoare rule systems into predicate transformers over general programming language grammars is, however, a good deal more complex than the above discussion might indicate. One particular problem that arises in this connection is the need to provide a clean interface between the grammar for the programming language and that for Hoare rules. Flexibility demands that the rule grammar can remain fixed when the program language grammar is changed. Yet the former must have a construct (like the notion of "fragment" discussed earlier) that provides a multilevel bridge to the language grammar. Still other factors create a need to provide for recursive descent into the language grammar. For example, so that lower-level syntactic components of a fragment may be referenced elsewhere in the rule, the syntax of fragments appearing in rule conclusions and hypotheses may need to be spelled out to a finer level of detail than is present in a given production of the language grammar.

Moreover, the program language parser used in connection with this kind of mechanization must be able to accept fragment variables (i.e,, nonterminal symbols) as well as concrete fragments. Thus, the BNF grammar for the program

language must be modified or enhanced to permit such recursive parsing. The problems we have just hinted at were solved satisfactorily in the course of our work on this approach.

c. Attempts to Apply META-VCG to JOVIAL

An approach based on the above criteria was pursued for several months on this project in connection with the axiomatization of JOVIAL-J73A and the development of a VCG for that language. The potential benefit of this approach was considerable in that, if successful, we could have applied META-VCG directly to the JOVIAL grammar and Hoare rule axiomatization, resulting in the mechanical production of a VCG based on those inputs.

A series of pilot experiments was conducted in 1980-81 by using the then existing version of META-VCG to construct VCGs for several small subset languages. Most of these languages were subsets of JOVIAL or Pascal. The purpose of these experiments was to gain experience with the META-VCG technique and discover any limitations or shortcomings it might have. In the next few paragraphs we shall describe the process followed in these experiments and what was learned from them.

First, a general BNF grammar was constructed for Hoare rules. The INTERPG parser-generator [54] was employed to build a parser/transducer for Hoare rule files. It was sufficiently general to permit its use for a wide variety of specific sets of rules. Next, this Hoare rule parser was applied to a number of sets of Hoare rules that had been written for the language subsets. The result of such parsing was a parse tree for the Hoare rule set. This parse tree is an S-expression and is saved by META-VCG as the value of a LISP variable to be accessed during the VCG phase.

What we have described thus far, i.e., the construction of the Hoare rule parser, is essentially independent of the programming language (or subset) for which a VCG is to be constructed. Next a parser/transducer was constructed (again by means of INTERPG) for chosen language subsets. These subsets were typically very small subsets of JOVIAL at the subroutine level--comprising perhaps ten to twenty statement constructs. In some cases, larger subsets of Pascal were also attempted with some success. These Meta-VCG parsers differed from the parser/transducers described in Sec. V in that, instead of merely producing a parse tree (PIF) for the program to be verified, they invoke the VCG directly from their transduction augments.

The third step in the conduct of these experiments consisted of applying the parser for the subset language to a series of individual programs (subroutines, actually) annotated with Floyd assertions, and written in the subset language. At this juncture we should clarify a point on which our description of META-VCG thus far has deliberately been vague. The analogy with metacompilers at the beginning of this section is, in fact, somewhat inaccurate. META-VCG does not actually "compile" a set of Hoare rules for and a syntactic description of a language into a set of predicate transformers (i.e., a VCG) for that language. META-VCG is more accurately described as a set of fixed, built-in "core" routines that are table-driven by the parsed Hoare rules in conjunction with the BNF syntax equations for the language. The top-level "core" VCG function of META-VCG is very much like the top-level VCG function PROCESS described in Sec. D-1. It has built into it only a very general knowledge of verification condition generation; for particulars it turns to the parse tree for the Hoare rules. Thus, META-VCG is, in effect, a table-driven, general-purpose verification condition generator.

In these experiments we found that the META-VCG approach possessed some very attractive features, but also certain shortcomings (at least in the then existing version). The following features are among its advantages:

* META-VCG exhibits a simplicity and uniformity of treatment for language constructs at all levels, including declarations, expression evaluation, and statements.

* It permits the specification of language semantics to be expressed in terms of an axiomatization in Hoare rules, thus affording a clarity and perspicuity not achievable in other ways.

* It allows the VCG designer to decouple to a considerable extent the syntactic and semantic aspects of the language to be handled.

* A considerable portion of the VCG task is built into META-VCG as a fixed set of routines that do not need to be changed when the target language semantics or syntax has to be modified. Such modifications require only that the "tables" (i.e., the Hoare rules or BNF grammar, respectively) be updated.

We found the following shortcomings in the META-VCG approach:

* The BNF grammar for the target language required awkward modifications to make it amenable for parsing language fragments within the Hoare rules.

* The treatment of assignments to structured variables had to be

accomplished with the same techniques used for conventional VCGs. Moreover, the facilities for doing this had to be built into the fixed part of META-VCG, whereas this should logically be part of the tables describing the language.

* The features available for referencing different syntactic elements of the same type in language fragments of a Hoare rule (e.g., the two <statement> parts of a conditional statement) required that the rules be rewritten manually into somewhat awkward forms. Similar difficulties arose in connection with referencing syntactic components at different production levels of the grammar. Occasionally, this even necessitated rewriting the grammar itself.

Not all of these disadvantages appear to be inherent in this approach. Instead, some were deficiencies only of the existing version of META-VCG. This was true, in particular, for the first and third disadvantages, which have been alleviated in more recent versions. A more serious set of difficulties with the META-VCG approach seems to be inherent in any approach based on Hoare rules, at least for languages like JOVIAL. This aspect is discussed next.

Ultimately it was found that a combination of circumstances rendered this approach infeasible for JOVIAL, at least within our time constraints. First, in trying to capture the semantics of JOVIAL by means of Hoare rules it was determined that the language was replete with constructs, data types, and features that severely strained the Hoare formalism. Among these factors were unstructured control jumps (e.g., GOTOs, EXITs from loops, and RETURN or ABORT from subroutine calls), the presence of multiple possibilities for aliasing of variables (e.g., through pointers and table references), the passage of labels as procedure parameters and, in general, a language "flavor" that stresses the influence of the run-time dynamic environment on the semantics of program execution. We found that, to accommodate these aspects of JOVIAL to the Hoare calculus, it would be necessary to extend and enrich that formalism substantially by introducing the notion of an "environment" as an additional explicit "variable" (above and beyond the actual variables of the program). Such an "environment" would have to capture such dynamic aspects of program execution as aliasing among the program variables, symbol table information (including name scopes), and declaration scopes.

Concurrently with these attempts to write Hoare rules for JOVIAL, our work on META-VCG was also being broadened to include such "environment" factors. However, at the time when a working prototype of the extended

META-VCG was needed in connection with the JOVIAL effort, its development was still largely at the theoretical stage. In short, it did not appear that an operating version could be completed in time to mesh with work on the other aspects of JOVIAL. Research and development of an improved META-VCG are still being pursued, but under other sponsorship.

In view of the foregoing considerations, it was decided to continue the VCG phase of our work on JOVIAL along conventional lines, i.e., without the aid of META-VCG. The experience we gained during the above META-VCG tests, however, proved extremely useful in simplifying the construction of our VCG.

In addition to the attempts just described to apply the Meta-VCG technique to full JOVIAL, we should also mention its much more successful application to a JOVIAL subset in connection with the generation of verification conditions for use with HDM. This phase of the work is covered in Sec. VIII-F of the report.

VII. THEOREM-PROVING COMPONENTS

A. INTRODUCTION

The deductive system used in the RJE program verifier is the December 1980 version of the Boyer-Moore Theorem Prover for Recursive Functions. This deductive system is the result of a long collaboration between Boyer and Moore carried out while its authors were at several university and industrial research laboratories. Among these research centers are the Machine Intelligence Laboratory at the University of Edinburgh, the Xerox Palo Alto Research Center, and (between 1974 and August 1981) the Computer Science Laboratory at SRI International. Further work on this tool is currently being done by Boyer and Moore at the University of Texas, Austin.

The development of the 1980 version at SRI International was supported in part by the present RJE contract, along with other contracts and grants from the Office of Naval Research and the National Science Foundation. The 1979 theorem prover version that preceded the 1980 release has been documented in a user manual [4]. The underlying theory on which this theorem prover is based is covered in great detail in the book [2] by Boyer and Moore. An extensive series of improvements in and additions to the 1979 theorem prover culminated in the December 1980 version described here. Some of these changes were documented in a 1979 SRI Computer Science Laboratory report by Boyer and Moore [3]. Other modifications present in the 1980 version are summarized below.

In addition to this work on the Boyer-Moore deductive system, some effort was devoted early in the project to other deductive tools. Among these were attempts to further improve the usability of the Tableaux deductive system, which was developed under the RPE/1 and RPE/2 projects (see Chap. II, Secs. B and C, respectively). We also devoted some effort to producing a faster and more versatile version of the Presburger decision mechanism, which had been combined with the Tableaux system in our earlier work. The latter work--carried out largely by R. E. Shostak--was also supported, in part, by an AFOSR contract [52]. Both the Tableaux and Presburger approaches were dropped in regard to RJE when it became apparent at that time (1979-80) that the Boyer-Moore system was considerably more advanced. In particular, the power of the linear arithmetic package added in the 1980 Boyer-Moore system was comparable to that of the Presburger approach, and it was adjudged as

interfacing more readily with the Boyer-Moore system.

It should be mentioned, though, that the Presburger decision algorithm has recently evolved into a powerful theorem-proving system, called STP [51], which has shown itself to be of great utility in proving a large number of verification conditions--many of them exceedingly difficult--in connection with our laboratory's SYSPROOF project [37] with NASA. This work concerns the verification of much of the PASCAL software for the SIFT fault-tolerant computer [36]. The STP theorem prover is distinguished from the Boyer-Moore system, among other ways, by being strongly typed. It contains numerous built-in type-checking features. Unfortunately, the STP system attained a usable state too late for its incorporation into RJE.

B. BASIC PRINCIPLES

1. An Overall View

The basic features of the theorem prover are summarized below. The reader is, however, referred to the theorem prover user manual [4] and the RJE user manual [15] for more complete descriptions.

The Boyer-Moore Theorem Prover (hereinafter referred to as the "theorem prover," or simply as the "Prover") is an automatic deductive system based on a formal logic for recursive function theory. This formal basis is fully covered in the first five chapters of the book [2].

Like most goal-driven systems for automatic deduction, the theorem prover functions basically through the reduction of formulas expressed in a formal calculus. The Prover attempts to reduce formulas obtained by its proving algorithms to (TRUE). The steps by which it proceeds are automatically documented for the user's inspection, accompanied by a system-generated narrative explanation (in English) justifying those steps.

When a formula has been reduced to (TRUE) by means of the system's proof procedures it will have been shown to be valid. Failure to achieve this reduction does not, however, necessarily mean that the formula is invalid--only that the proof of validity has not succeeded. One reason for this is that (when all else fails) the system may replace the current goal formula by a more general one and attempt to prove the latter, which may be invalid even when the original one is valid. Sometimes, however, a formula

may be reduced to (FALSE) in such a manner that it will be obvious (by examining the proof steps) that the formula is invalid (usually because the Prover has discovered a false instance).

Formulas of the Prover's formal system resemble LISP S-expressions syntactically. Functions, arithmetic operations, and predicates appear uniformly in LISP-like prefix form; in fact, no inherent distinction is made among function calls, arithmetic expressions, and relations. Despite the syntactic resemblance between the Prover's language and LISP, it is necessary to maintain a clear distinction between the two for reasons that will become clear below. Suffice it to say, for the moment, that, even where a Prover concept (e.g., a function) has the same name as an INTERLISP function [25], the semantics of the two may be quite different. The Prover language is, in fact, not a programming language in the usual sense, but a pure expression language. In particular, there are no "side effects."

One important proof procedure employed by the Prover is definitional expansion, whereby instances of a functional expression appearing in the formula are expanded by the substitution of actual arguments for the formal parameters appearing in the body of the function definition. When a recursively defined function is expanded in this manner, the Prover uses powerful built-in facilities for automatically carrying out complicated proofs by formal induction. This feature distinguishes the theorem prover from most other systems for mechanized deduction.

In addition to its induction mechanism, the Prover comprises powerful packages for expression simplification (using a pattern matcher and rewrite lemmas), a linear arithmetic package, a tautology checker, and expression normalizers for converting formulas into canonical forms. The 1980 version of the Prover, which is the deductive part of the RJE system, also has a facility for evaluating theorem prover expressions that contain only explicit constants (i.e., without free variables). This is made possible through the execution of efficient code that is automatically compiled at the time of function definition or shell creation (see Sec. C-3). This code is executed, whenever applicable, during proofs. It can also be called directly by the user, and it is then very useful, e.g., for checking the appropriateness of user-supplied function definitions. (See Sec. C-4).

The theorem prover is flexibly user-extendable in a numbe. of ways. The system's knowledge base can be augmented by the user through a number of powerful, uniform mechanisms. These mechanisms permit the user to

* Create user-defined datatypes (called "shells").

* Introduce new, user-defined functions in a logically consistent manner (enforced by the Prover itself).

* Introduce axioms whose validity can be vouched for independently by the user.

* Build incrementally a collection of lemmas (proved by the system) that may subsequently be used (e.g., as rewrite lemmas or induction lemmas) to aid the system in carrying out other proofs.

Finally, the deductive capacity of the system itself (as distinguished from its knowledge base) can be extended by proving and adding new proof procedures through a metatheorem capability [3]. This feature was first added to the Prover in its 1980 version.

Facilities are provided for saving, on a library file of moderate size, the state of the knowledge base accumulated during a session. This feature allows the user to restore the knowledge base for subsequent use without having to redo the proofs that led to that state. It is discussed in Sec. C-5b.

The system is implemented as a set of INTERLISP [25] programs executable on any system supporting that language. The Prover incorporates convenient error recovery features (see Sec. C-6) that are implemented through the editor facilities of INTERLISP-10. Implementations of the theorem prover exist for TOPS-20 and TENEX host computers. In particular, they have been tested extensively on the Arpanet hosts SRI-KL and SRI-CSL as well as on another F2 machine (running under TENEX) at SRI International. The theorem prover has also been mounted on Arpanet host RADC-TOPS20.

2. Theorem Prover Events

The primary data base for the theorem prover is concerned with "events"--a fundamental notion for the system. The act of introducing a new definition, declaration, data type, or theorem to the system is called an "event." Every event has a "name" and a "body"; properties associated with an event are accessed by the system through its name. (Such properties are stored on the INTERLISP property list of the name). All events must be supplied with distinct names by the user. In the case of functions, the name of the event that defines or declares a function is simply the function symbol itself.

The user-level functions employed to create new events are
ADD.AXIOM, ADD.SHELL, DCL, DEFN,
PROVE.LEMMA, and MOVE.LEMMA.

a. Legality of Events

Before an event is accepted by the system and incorporated into the data base, the theorem prover must first be satisfied as to its legality. When an attempt is made to enter an illegal event, an error condition is invoked (see Sec. C-6). Since the theorem prover operates by carrying out deductions based on prior events, all events (definitions in particular) are subjected to checks for consistency with the data base before they are accepted. The details of this checking depend on the kind of event. Declarations, e.g., are subjected only to syntactic checks. The following 82 characters are legal in a theorem prover name:

The uppercase and lowercase letters, A-Z, a-z
The numerals 0 through 9
The symbols ! # $ & + , - . / : ; < = > ? @ \ ^ _ ~

However, none of the characters + - . 0 1 2 3 4 5 6 7 8 9 may be used as the first character in a name.

All names, hence all events, are restricted by the above syntactic checks. Furthermore, the syntax of all formulas appearing in events is rigorously checked for conformity with the theorem prover's conventions. For example, functions of fewer than two arguments must be called with the correct number of actual arguments (i.e., the same number appearing in the formal parameter list used in the corresponding definition or declaration). For functions of two or more arguments the restriction is weaker, and is discussed in Sec. C-2a.

In addition, most events are also subjected to extensive semantic checks. First of all, no event may have the same name as any prior event. Second, and more importantly, functions referenced in a nonrecursive function definition must have been defined or declared in a prior (accepted) event. Thus, if a new function, say NEWFN, is introduced by a definition whose body contains a reference to some other function, say OLDFN, the latter must already have been defined (or declared) to the theorem prover. If NEWFN is accepted, the event NEWFN is then made "dependent" upon the event OLDFN (and perhaps upon other events as well). Other kinds of semantic checks are described in Sec. C-2.

Recursive function definitions--extremely important to the Prover--require more complicated checking. Naturally, references in the definition body to functions other than the one being defined must be to already defined (or declared) functions. just as for nonrecursive definitions. Since a recursive function definition always contains one or more references to the function being defined, OLDFN and NEWFN are the same in this case. Hence, OLDFN cannot, of course, have been defined before NEWFN. The soundness of a recursive definition depends on the notion of well-foundedness, and the theorem prover must first be satisfied as to the well-foundedness of each recursive definition before it is accepted. To be well-founded, a recursive definition must involve two things: (a) one or more determinate "base cases" (or stopping points) for which no recursion is involved, and (b) a demonstrable descent toward a base case for all inputs. Checking that Condition (a) is satisfied is always a simple syntactic check, i.e., for the presence of at least one nonrecursive exit in the definition body. For the most part, the process whereby the theorem prover establishes well-foundedness is automatic, since clever heuristics have been built into the Prover for this purpose. In complicated cases, however, the user may first have to prove an induction lemma to assist the Prover in establishing Condition (b) before a recursive function definition can be accepted. (See also the discussion in Sec. C-2).

b. Undoing and Editing of Events

As already mentioned, the system prevents the user from introducing more than one event with the same name. However, events can be undone (i.e., completely erased from the Prover's data base) by means of the function UNDO.NAME(name).

Events may also be edited by means of the functions EDITEV(name), EDITD(funcThe third, EDITC, may be used for editing the comment portion of an event without the overhead of actually redoing the body of the event itself. All three editing rrly, DEPENDENTS.OF(EVT) returns a list of the names of events that are dependent on EVT. If the event EVT were to be undone (by typing UNDO.NAME(EVT)) this action would also undo all its dependents. Likewise, editing EVT by means of EDITEV(EVT) throws the user into the INTERLISP editor [25] with the list of all dependent events as the editor's "current expression," giving the user an opportunity to edit these dependent events at the same time. Exiting from the editor normally (i.e., with OK) causes all dependent events to be redone along with the event EVT.

Certain types of events, notably "shell events," entail the creation of "satellite" events, e.g., other functions that are generated as subsidiaries of the main event. Satellites are dependent on the main event that resulted in their creation. The function ADD.SHELL is used to add new data types to the theorem prover data base, as described in Sec. C-3. For additional

details, see [2, 4].

Events also have a time of creation and, therefore, a chronological order known to the data base. The variable CHRONOLOGY is a list of event names in (reversed) chronological order, i.e., the first element (the INTERLISP "CAR") of CHRONOLOGY is the most recent event.

3. Functions for Proving Theorems

The basic theorem-proving function is PROVE, which takes as its single argument a formula in the theorem prover's syntax [2]. PROVE does not create any events. Thus, carrying out a proof by means of this function does not alter the system's knowledge base; to save the proved formula as an event one must use the function PROVE.LEMMA (as described below). PROVE invokes a rotating succession of the various proof strategies of the Prover---simplification, definitional expansion, tautology checking, induction, lemma application, introduction of new variables, generalization, and linear arithmetic.

An attempt at proving a formula will either succeed in an ultimate reduction to T--in which case the proof succeeds--or in a reduction to F, or it may go on indefinitely generating new subgoals in an infinite regression. Needless to say, the user can never be sure when this last situation is the one at hand. However, it is usually apparent fairly quickly from the printed output when the proof is, so to speak, "going off the tracks."

At least two typical such situations are easy to spot. The first occurs when some variable, say X, is replaced by a form such as (ADD1 Y), and then Y is replaced by (ADD1 Z), and so forth, leading to an infinite progression of fruitless replacements. A second situation that can occur is when some formula is "generalized" to a new variable, and the interrelationships among the free variables of the goal formula are thereby lost. Recognition of the second situation may take considerable insight on the part of the user, but in most cases it is best to err on the side of caution and abort the proof attempt (with CTRL-D) if there is any doubt. In such cases the burden is on the user to "guide" the next proof attempt. This can be done by noticing where the proof went astray and first proving some lemmas that might have aided the proof at that point.

This brings us to the second user-level proving function, called

PROVE.LEMMA. This function takes the four arguments name, lemma.types, formula, and comment, where <formula> is the theorem to be proved and the fourth argument is optional. Unlike PROVE, when PROVE.LEMMA succeeds it adds the proved lemma to the data base under the name supplied as the first argument. Since PROVE.LEMMA operates through PROVE the strategies invoked are exactly the same for both kinds of proof. The second argument, <lemma.types>, is either the empty list NIL or a list of one or more of REWRITE, ELIM, INDUCTION, and GENERALIZE. This argument tells the theorem prover under what circumstances the lemma should be invoked in future proofs (see Sec. C-2b).

C. THEOREM PROVER FEATURES

1. Primitive Data Types and their Functions

The built-in, primitive objects of the theorem prover are

* The logical constants (TRUE) and (FALSE)
* Quoted literal atoms, e.g., (QUOTE ABC)
* The nonnegative integers (Peano numbers)
* Lists.

The logical constants are the only objects of type "Boolean," i.e., they are the only primitive objects that satisfy the predicate BOOLEANP. (TRUE) is abbreviated as T, and (FALSE) as F. T and F are words reserved by the theorem prover for just these uses. Quoted literal atoms such as (QUOTE ABC) may be abbreviated as 'ABC, just as in INTERLISP.

Observe carefully that unquoted literal atoms (other than the abbreviations T and F) are interpreted as free variables by the theorem prover. Thus, theorem prover constants (with the exception of T, F, and the nonnegative integers) must be written as functions with no arguments.

The system must also permit the representation of quoted expressions other than literal atoms, i.e., expressions such as '(A B C) for explicit lists. The theorem prover's convention here is exactly the same as for quoted atoms. Thus, just as 'ABC is treated by the system as (QUOTE ABC), the expression '(A B C) is mapped into (QUOTE (A B C)). Similarly, '(1 2 3 4) becomes (QUOTE (1 2 3 4)). However, quoted numbers are simply the numbers themselves, i.e., '0 or (QUOTE 0) is just 0, '1 or (QUOTE 1) is just 1, and so forth. These conventions will be thoroughly familiar to users of

INTERLISP [25].

The theorem prover is a weakly typed system. More precisely, the Prover is able to distinguish among the primitive types listed above and any user-defined (shell) types, and it uses such information internally for simplifying certain kinds of deductions. For example, two variables (or expressions) x and y that are known to be of different types can never be equal. Hence which it can deduce such information at the time of function definition. The user may observe this when the Prover prints a message such as "Observe that (NUMBERP form) is a theorem" following the acceptance of a definition. This particular message will be printed when the the Prover is able to establish that the function just defined always returns a numeric value. Similar messages are printed for other result types.

Such result-type information is used by the Prover for subsequent deductions about the corresponding functional expressions. See also the discussion about type distinctions among user-defined types in Sec. C-3.

We say that the Prover is "weakly" typed because the arguments to functions are unrestricted as to their type. Arguments of arbitrary types may be supplied to any defined function without producing an error, and a definite result will always be determined by the definition. Moreover, the R-function feature mentioned above (see Sec. C-4) will compute this explicit value result correctly when R is applied to a function-call expression containing explicit arguments of any types.

Type information for variables and functional expressions is also utilized by the theorem prover in its expansion of definitions. For example, since the functions AND, OR, NOT, IMPLIES, IF, and EQUAL (discussed next) all have result-type BOOLEAN, the Prover is able quickly to determine the validity of such formulas as (BOOLEANP (AND P Q)) and (BOOLEANP (OR P Q)). This capability is even more useful in eliminating irrelevant branches of complicated IF expressions that result from the "opening up" (or expansion) of function definitions.

a. Logical Functions

The primitive logical functions--i.e., functions operating primarily on the logical constants T and F, and returning values of the same type--are AND, OR, NOT, IMPLIES, IF, and EQUAL. Most of these functions have the usual standard meanings. The function IF, however, may be unfamiliar. This function takes three arguments, i.e., (IF a b c) has the meaning "if a is not-F then b else c." Note that nonfalsity plays the role usually accorded to NIL in LISP conditional forms. The function IF is primitive among the logical functions in that AND and OR are defined in terms of IF, and all formula expressions manipulated by the Prover are put internally into a canonical IF form. Thus, (AND P Q) is actually (IF P (IF Q T F) F), while (OR P Q) is (IF P T (IF Q T F)). The general meaning of (NOT x) in the Prover is "x is F." Thus, NOT has the usual results (NOT F) = T and (NOean arguments we obtain, e.g., (NOT 0) = (NOT 1) = (NOT (PLUS X Y)) = F.

The function EQUAL is the usual equality predicate. When applied to logical formulas it has the meaning of logical equivalence (i.e., the connective "if-and-only-if"), since such formulas only can have T or F as their values.

b. Arithmetic Functions

The Peano numbers are axiomatized by the shell ADD1, which has (ZERO) as its "bottom object," the function NUMBERP as its "recognizer," and SUB1 as its "destructor function." The following "intrinsic" functions (created as satellites of the initializing operation BOOT.STRAP [4]) are among those available for manipulating Peano numbers and variables, or other expressions representing such numbers:

ZERO, SUB1, ADD1, NUMBERP, ZEROP, LESSP,
PLUS, TIMES, DIFFERENCE, QUOTIENT, and
REMAINDER.

The functions ZERO, SUB1, ADD1, and NUMBERP have already been discussed. (ZEROP x) is T if and only if x is either the number (ZERO) or any object of a nonnumeric type. LESSP is a hand-coded theorem prover function quite fundamental to the whole structure of the Prover (because it must be used to establish the well-foundedness of all recursive definitions submitted to the Prover, even of the other intrinsic functions). The functions SUB1, ADD1, PLUS, TIMES, DIFFERENCE, QUOTIENT and REMAINDER all have the effect of "coercing" any nonnumber arguments to 0. This ensures their totality and that

they will always return a numeric result, even when supplied with nonnumeric arguments. (DIFFERENCE x y) = 0 if (LESSP x y), since we are dealing here only with the Peano (nonnegative) integers. The QUOTIENT function is, of course, the integer quotient, e.g., (QUOTIENT 5 2) = 2. Observe, too, that (QUOTIENT x 0) = 0, which is quite different from the INTERLISP convention for IQUOTIENT, i.e., (IQUOTIENT x 0) = x.

Many other functions and predicates for handling Peano arithmetic besides those listed above are present in the Prover. For example, (GREATERP x y) is defined as (LESSP y x), (GREATEREQP x y) as (NOT (LESSP x y)), and so forth. Many of these subsidiary concepts are recursive. For example, exponentiation of Peano numbers (EXPT x y) is defined recursively in terms of TIMES and SUB1. The user may add to this store of functions and predicates at any time by defining new ones through use of the function DEFN (to be described in Sec. C-2).

c. Functions on Lists

The theorem prover's axiomatization of the list data type corresponds very closely to the LISP notion of "list." The only significant difference is that in the Prover, in contrast to LISP, NIL does not satisfy the predicate LISTP. The fundamental notion for Prover lists is the shell constructor CONS. CONS has the recognizer LISTP, the destructors CAR and CDR, but no bottom object. (See the discussion of shells in Sec. C-3).

In particular, the shell definition for CONS produces the following satellite rewrite lemmas:

```
CAR.CONS = (EQUAL (CAR (CONS X Y) X)

CDR.CONS = (EQUAL (CDR (CONS X Y) Y)

CONS.CAR.CDR = (IMPLIES (LISTP X)
                        (EQUAL (CONS (CAR X) (CDR X))
                               X))
```

The derived notions LENGTH, the predicates NLISTP (true for nonlist objects), MEMBER, and SUBSETP, and LAST are straightforward and correspond exactly to the corresponding LISP concepts. For example, (LENGTH LST) is defined by

```
(IF (LISTP LST)
    (ADD1 (LENGTH (CDR LST)))
    0).
```

(NLISTP X) is simply (NOT (LISTP X)).

(MEMBER X LST) is defined recursively as

```
(IF (NLISTP LST)
    F
    (IF (EQUAL X (CAR LST))
        T
        (MEMBER X (CDR LST)))),
```

(SUBSETP X Y) has the definition

```
(IF (NLISTP X)
    T
    (IF (MEMBER (CAR X) Y)
        (SUBSETP (CDR X) Y)
        F)),
```

while (LAST L) is defined (again recursively) as

```
(IF (LISTP L)
    (IF (LISTP (CDR L)) (LAST (CDR L)) L)
    L)))
```

The Prover also provides a special function LIST, which may take an arbitrary number of arguments. (LIST a b ... x) in effect behaves like (CONS a (CONS b ... (CONS x NIL))). To achieve this, LIST makes use of a convention about the behavior of extra arguments in function calls which is explained in Sec. C-2a.

2. Definition of Functions

Although we have just already given a number of specific function definitions as illustrations, we have deferred until now any discussion of the actual process whereby such events are introduced into the data base.

A function is defined through the operation DEFN, which takes four arguments, the last being an optional comment. The syntax for DEFN is

DEFN (name args formula comment)

Here <name> is the name of the function to be defined to the Prover. The second argument <args> is a LISP list of the formal parameters of the function. The third argument <formula> is the body of the definition; all functions referenced in the body except a possible recursive reference to <name> must already have been defined or declared to the system. The <comment> is an optional text string. If <name> appears in the body <formula>, the function being defined is recursive and its acceptance by the Prover will depend on whether the Prover is able to show the definition to be well-founded.

As discussed earlier (see Sec. B-2a), the proof that a recursive definition for a function RFN is well-founded depends on the Prover's being able to show that the recursion proceeds monotonically toward a base case (or cases). To do this the Prover tries to find some predefined numeric function (m x1 x2 ... xn) (called a "measure function") for which the inequality (m r1 r2 ... rn) < (m x1 x2 ... xn) holds, where r1, r2,..., rn are the argument expressions given to the function RFN in any of recursive calls appearing in the body <formula> for RFN. The Prover has some powerful built-in heuristics that aid it in discovering a suitable measure function m. One general measure function used by the Prover for this purpose is the function COUNT, which works for lists, numbers, and even for literal atoms. Suitable inequality lemmas (CDR.LESSP and SUB1.LESSP, for example) are available in the Prover's data base for showing the required monotonicity relations (for lists and numbers, in this case). When several of the variables x1,...,xn undergo simultaneous recursion, the process is more complicated. However, one built-in heuristic is to try a lexicographic ordering constructed (as a kind of direct product) from independent measure functions for the separate variables. (See [4] for details.) As mentioned earlier, if the Prover is unable to find a suitable measure function, the user must find one, define it, and prove one or more lemmas that establish the required monotonicity relations.

a. Function Argument Conventions

We stated above that the Prover enforces conformity between the number of formal arguments appearing in a function definition and the number of actual arguments present in a call on that function. This is not, however, strictly true. It is strictly true for functions defined to have one argument or none. For functions of two or more arguments, the number supplied must be at least equal to the number defined. When more actual arguments than the defined number are supplied in a call, the system accepts this by rewriting the call expression into a nested, right-associative expansion. For example, although the functions AND, OR, PLUS, etc. are defined as two-argument functions, the system rewrites (AND P Q R) into the form (AND P (AND Q R)), and similarly for OR, PLUS, etc. It will be seen that this is a perfectly natural convention that permits the user to write convenient abbreviations without significant loss of rigor or error checking. It is certainly appropriate for the the three functions just mentioned as well as many others.

This convention applies, in particular, to the two-argument function CONS. Thus, (CONS a b c d) is an abbreviation for (CONS a (CONS b (CONS c d))). Moreover, one can then type (CONS a b c d NIL) to the system to represent (CONS a (CONS b (CONS c (CONS d NIL)))). Since this last expression has the same meaning as the conventional LISP form (LIST a b c d), the excess-argument convention provides a way to axiomatize the function LIST.

b. Function Declarations and Axioms

It may occasionally happen that the user wishes to introduce a function for which it is unnecessary (perhaps difficult or impossible) to provide an explicit interpretation. Such uninterpreted functions can be declared to the system by means of the operation DCL. The syntax for DCL is "DCL (name args comment)." Here <name> is the name of a function (as yet unknown to the Prover) to be declared, and <args> is a LISP list of the formal arguments to <name>. The third argument <comment> is an optional text string.

Thus, DCL permits the user to introduce and use uninterpreted functions in the Prover. The same argument conventions discussed in Sec. C-2a with respect to defined functions apply to declarations. One example of an uninterpreted function that is used in the RJE system is the function SELECT. It is used to axiomatize TABLE references (i.e., JOVIAL array references). SELECT has the declaration

```
(DCL SELECT
     (A I)
     (* Uninterpreted function whose meaning is
        the selection of the element of the
        array A at index I.))
```

Since the system knows nothing about an uninterpreted function other than its name and formal arguments, it will be unable to perform any real deductions about expressions involving it. The user must supply any such information by means of axioms. The axiom used by RJE to give an implicit interpretation to the function SELECT defines its interaction with another declared function ALPHA as follows:

```
(ADD.AXIOM SELECT.ALPHA
   (REWRITE)
   (EQUAL (SELECT (ALPHA A I V) J)
          (IF (EQUAL J I) V (SELECT A J))))
```

This is the only axiom we need to give SELECT and ALPHA their intended meanings--i.e., referencing the Jth element of (ALPHA A I V) yields V if I=J; otherwise it yields the contents of A at index J before the change produced by

ALPHA. The uninterpreted function ALPHA is declared as (DCL ALPHA (A I V)). The intended meaning of (ALPHA A I V) is the array A with the contents of the Ith element changed to V.

Observe that the second argument to ADD.AXIOM is a list (in the above instances a list of length one) of lemma types. The available lemma types are REWRITE, ELIM, GENERALIZE, and INDUCTION. They determine how the axiom is to be used by the system. The same four lemma types are employed in connection with the proof and the use of lemmas by the system.

REWRITE axioms and lemmas will typically have a body of the form (EQUAL expr1 expr2), or (IMPLIES hyp (EQUAL expr1 expr2)). In the first case (unconditional rewriting), if REWRITE is present as one of the lemma types in the axiom or lemma, the system will rewrite any expressions that match expr1 into new expressions in terms of expr2. (See also the discussion of conditional rewrites, below).

An ELIM lemma is used by the Prover to replace some variable in the theorem being proved by a new term, so as to allow rewrites to remove certain function symbols.

A GENERALIZE lemma mentioning an instance of the term t is used when the Prover decides to generalize a conjecture containing t by replacing it with a new variable v. The lemma is used to restrict v, but there are no constraints on the form of a GENERALIZE lemma. (See also [2]).

An INDUCTION lemma informs the Prover that a given measure decreases under the well-founded function LESSP. INDUCTION lemmas are used by the system to prove the well-foundedness of recursive function definitions. An INDUCTION lemma must have the form:

```
(IMPLIES hyp (LESSP (m t1 ... tn) (m x1 ... xn)))
```

where the xi are distinct variables, all the variables in hyp occur in the conclusion, and m (the measure function) is a special symbol. (See [2], Chaps. XIV and XV for details).

If the lemma-types argument is the empty list NIL, the Prover still stores the axiom under its user-given name, but it is unable to make use of the axiom in deductions. This is useful for record-keeping, and, once an axiom or lemma has been accepted, it is possible to change its list of lemma types by means of MOVE.LEMMA (see [4]).

In general, lemmas and axioms may also be conditionally applied by the Prover. For example, when the body of a REWRITE axiom (or REWRITE lemma) has the form (IMPLIES hyp (EQUAL expr1 expr2)), every expression matching expr1 encountered by the system will be tested to see if the hypothesis hyp is satisfied (for the variables associated with expr1). If so, expr1 will be rewritten in terms of expr2. The system does not expend a great deal of effort in trying to confirm the truth of hyp; only a "surface" deduction is applied to that end (e.g., using type information, tautology, immediately known hypotheses, and other easily obtained confirmation); otherwise the system might spend inordinate amounts of time and other resources in just finding out what axioms or lemmas to apply.

A general caveat must be issued with respect to the introduction of axioms into the Prover. It is possible thereby to introduce inconsistencies into the knowledge base since axioms per se are not checked for logical consistency with the rest of the data base. They are introduced by user fiat and the burden of guaranteeing consistency rests on the user's say-so. The only checks made when an axiom is introduced are for verifying that the functional forms of which its body is composed are known to the data base, that the argument conventions are satisfied, and that only legal names are used.

3. Introducing New Data Types

As mentioned above, the user can introduce new data types to the system by the operation ADD.SHELL, which we now describe in detail. The syntax for this operation is

```
ADD.SHELL( shell.name  btm.fn.symb  recognizer
                       destructor.tuples  comment)
```

Here <shell.name> is the name chosen by the user for the constructor. The name chosen by the user for the function that generates the bottom object for the new type is <btm.fn.symb>; if NIL is used, no default object is supplied with the type. The third argument <recognizer> is the name of the predicate that recognizes objects of this shell type. The fourth argument <destructor.tuples> is a list of n elements, each element being of the form (ac tr dv); the ac's are the accessors for components of a shell object of this type; the tr's are type restrictions on each component; the dv's are the corresponding default values.

For example, the following shell definition can be used to introduce a special data type for array indices of dimensionality 2 or more:

```
(ADD.SHELL INDEX2 NIL INDEX2P
           ((FIRST (NONE.OF) ZERO)
            (SECOND (NONE.OF) ZERO)))
```

This data type definition establishes the semantics for forms like (INDEX2 a b), (INDEX2 a b c), etc. No bottom object is provided; the predicate INDEX2P is (automatically) created as the recognizer for such forms. The functions FIRST and SECOND are the accessors (ac's) for the first and second components, a and b, respectively, of (INDEX2 a b). The type restrictions (tr's) on the components are vacuous in this example (i.e., an object of any type may appear as a component). (ZERO) is the default value (dv) to be returned by FIRST and SECOND when they are called on an object not of type INDEX2P. We could have used (ONE.OF NUMBERP) instead of (NONE.OF) as the tr's, had we wished to constrain the INDEX2 type to unsigned integer components. Observe that the extra-arguments convention (Sec. C-2a) permits INDEX2 to be used with more than two components, so that, e.g., (INDEX2 a b c) = (INDEX2 a (INDEX2 b c)). This allows INDEX2 to be used for array indices of any dimensionality greater than 1.

In addition to creating the satellite events corresponding to the bottom object (if any), the recognizer function, and the destructor function(s), the introduction of a new shell type provides a fast mechanism for deducing equality (or inequality) between shell objects of that type. Thus, any attempt to prove an equality like (EQUAL (INDEX2 a b) (INDEX2 x y)) will force the system into proving the subgoals (EQUAL a x) and (EQUAL b y).

Formally, therefore, INDEX2 is equivalent to the list constructor, except that the Prover's shell definition mechanism makes INDEX2 objects a separate type from lists. It will distinguish INDEX2 objects from lists (CONS shells) through the recognizer INDEX2P. This type distinction is very useful in those many situations in which one may wish to create special list-like constructs (e.g., rational numbers as pairs of integers, or complex numbers as pairs of rational numbers) that must nevertheless be treated as distinct types of data objects, not simply as lists. There is, however, nothing corresponding to the notion of a "subtype" in the Prover's data-type formalism.

4. The R-Compiler Facility

The 1979 version of the theorem prover lacked the means for dealing efficiently with proofs involving explicit expressions. In principle, it would be most effective to handle such formulas by actual evaluation. After all, if one has at hand a functional expression involving defined functions and explicit (constant) arguments, with no free variables, it seems wasteful to have to perform elaborate formal deductions as to the possible values of such an expression when it can simply be evaluated. This deficiency was remedied by adding the R-compiler feature to the 1980 Prover.

The R-compiler was implemented by the addition of code to the system to be invoked during the definition of new functions and data types to the Prover. This code is actually a kind of LISP compiler that processes the supplied theorem prover definition (and any satellite definitions generated by the introduction of new shells) into fast, compiled INTERLISP code. It creates for each Prover function FOO a compiled routine named 1FOO, executable by the user when he types (R '(FOO . args)). If R is called by the user on an expression containing either free variables or undefined (but declared) functions, R returns the expression (NOT REDUCIBLE). As with other attempts to manipulate unknown function objects, the Prover invokes an error if R is called on a functional expression whose function symbol is wholly unknown (i.e., neither defined nor declared), or on an expression with the wrong number of arguments. On the other hand, R does evaluate forms containing shell expressions. The results for shell expressions involve the functions QUOTE and 1QUOTE.

The function R and the compiled function 1FOO will also be called automatically whenever the theorem prover's simplifier package sees an expression of the form (FOO . args), where args is a list of explicit constant expressions. This occurs, e.g., when an attempt is made to prove a formula that contains an expression of this form.

An extremely useful byproduct of this feature is that the user can now debug definitions given to the theorem prover by testing them through direct evaluation on explicit cases. In the 1979 and earlier versions of the Prover, the user had to resort to rather indirect means for such testing, i.e., by actually trying to <u>prove</u> special cases of equalities such as (EQUAL (FOO a b) c), where a, b, and c are explicit values, with c equal to what the user

expected for (FOO a b). It is clear that the new evaluation capability affords the user a much more direct way to debug definitions. It is also considerably more efficient in its use of computational resources. While the PROVE technique is more complicated, it can yield more detailed information than mere evaluative testing, and, in any case, is still available when needed.

5. Input-Output Features

The theorem prover has a number of distinct mechanisms for saving and restoring the state of a session, as well as for writing proof transcript files. These features will be described briefly below. More detailed descriptions appear in the user manuals for the Theorem Prover [4] and the RJE [15].

a. Proof Transcripts]

The theorem prover prints an English-language narrative description of the proof process as it proceeds. The system provides two facilities for putting this output on a proof trace file:

* The simplest way uses the INTERLISP function DRIBBLE [25].

* There are special mechanisms integral to the Theorem Prover and using the variables PROVE.FILE and TTY:, which allow you to suppress printing at the terminal of all output, or all output except for any error messages that might be generated.

One can also arrange the system to run without printing to the terminal. This is useful when the theorem prover is to run in detached mode. Details of this feature are given in [4, 15].

b. Library Files

Library files contain theorem prover data base information (i.e., events) in a form that permits the Prover's state to be restored quickly in a later session. Only one library file may be loaded into an environment. The user writes library files by means of the function MAKE.LIB. The size of a library file is typically about 30 pages. Little overhead is entailed when it is loaded to restore the Prover to a previous state. Reloading is accomplished by means of the command NOTE.FILE(filename).

c. Event Files

Event files provide the system user with an alternative means for storing a number of theorem prover events--usually a small number--in a LISP file for later superposition upon a different Prover environment. They have two advantages over library files:

* They require only a relatively small amount of disk space (proportional to the number of events to be stored).

* Several event files may be loaded successively into the same environment.

However, event files suffer from the disadvantage that the events (or some subset thereof) must be re-executed in the theorem prover. There is, therefore, a time-space trade off in their use as compared with library files.

Strictly speaking, the event file-saving facility is not part of the Boyer-Moore system. It was added as one part of the user executive described elsewhere in this report.

Event files may be written by executing the command EVENT.SAVER(filename). They are loaded by means of the LISP command LOAD(filename), and the events contained in the event file may then be restored by executing the Prover command (REDO.UNDONE.EVENTS EVENTS.LIST). The events on EVENTS.LIST may, of course, be edited before redoing the events by calling the INTERLISP editing function EDITV on EVENTS.LIST.

6. Error Handling

Attempts to do anything "illegal"--such as defining an already defined function, defining a function in terms of another function that has not yet been defined, or violating the argument conventions for functions--will be intercepted by the checking mechanisms of the system and will produce an error condition.

Errors are grouped into three classes: warnings, soft errors, and fatal errors. Warnings occur when the system has detected something that is unusual, but not logically incorrect. For example, if, in defining a function where the definition body fails to refer to all of the formal parameters, such an omission invokes a warning error. This is something the user may wish to know about, but it is not necessarily wrong. After printing a warning message, the system resumes normal operation.

Soft errors are ones beyond which the system cannot continue until the user has repaired the error by editing a formula or changing a name. Typical examples of soft errors are

* Supplying too few arguments to a defined function.
* Omitting a needed argument to a theorem prover command, e.g., forgetting to supply a name for a lemma in a PROVE.LEMMA command.
* Misspelling a name in a command (so that the Prover thinks an undefined function is being referenced).
* Typing an extra left parenthesis in a formula.

After a soft error the INTERLISP editor is automatically invoked upon the offending command whenever possible. The user may use the editor to repair the error. Exiting from the editor normally (with OK) causes the (edited) command to be reinvoked. An abnormal exit (with STOP or CTRL-D) leaves the theorem prover in the same state it was in before the faulty command was issued.

Fatal errors occur when system resources (e.g., disk space) are exhausted or when internal checks indicate the presence of inconsistency in the data base or bugs in the theorem prover itself. It is not usually possible to proceed beyond a fatal error.

VIII. THE JOVIAL-HDM SYSTEM FOR VERIFYING HIERARCHICALLY STRUCTURED PROGRAMS[2]

A. INTRODUCTION

The previous chapters of this report have described an environment that provides support for the verification of individual JOVIAL programs. An additional feature of the environment allows the verification of JOVIAL programs that call other programs, the other programs being described in terms of their specifications. This technique for decomposing the development and verification of large programs into more manageable size units is known as procedure abstraction.

More powerful than procedure abstraction for development and verification is the technique of data abstraction. In data abstraction, procedures are collected into units called modules (other names such collections are: packages, forms, clusters, and classes). Ideally, all of the procedures of a module are concerned with the manipulations on the same kind of object. Examples of a well-conceived module are the following: File System, Directory Manager, Process Manager, Document Handler, Relational Dababase, and Device Manager. A module can be specified in a manner that is independent of its implementation, and for a well-conceived module the specification can be very concise. A system is, then, configured as a hierarchy of modules. The most common form of hierarchy is a partial ordering (representable by a loop-free graph). Modules in the hierarchy are linked by maps that represent the data structures of a module in terms of data structures in modules below the hierarchy, and implementations that are programs corresponding to each operation of a module.

This chapter describes the Hierarchical Development Methodology (HDM), an approach to data abstraction under development at SRI since 1973. A key component of HDM is the SPECIAL specification language, used to specify modules and maps. Our main contribution to HDM in this project has been to develop a verification system for a subset of SPECIAL used as the specification language and for a subset of JOVIAL as the language of

---

[2]This chapter, written by Karl N. Levitt, describes work carried out by him in conjunction with Dwight F. Hare and David L. Snyder.

implementation.

HDM is an attempt to provide languages, guidelines, and tools to permit a designer to manage the complexity found in the development of modern systems. HDM embodies the following concepts and capabilities:

* HDM structures the development process into a sequence of development decisions, each of which is described precisely in an appropriate language.

* HDM separates development decisions into three phases -- design, representation, and implementation.

* HDM structures a system design as a hierarchy of abstract machines. Each machine is specified independently other abstractions and independently of any ultimate implementation.

* HDM facilitates formal verification of designs as well as implementations. A change to a module will often involve either its implementation or its specification, thus making it easy to determine what part of a module must be reverified after a change.

* HDM is supported by tools that aid in all stages of development.

The remainder of this section summarizes the major software advances that have led to HDM, define the concepts of a "methodology", and justify the need for a suitable development methodology. The remaining sections of this chapter describe HDM, how it is linked with JOVIAL, and the verification environment that we we have developed. Section B provides a brief overview of HDM. In Section C we present a brief discussion of software development methodologies in general. Section D describes the languages of HDM (including our approach to using JOVIAL as the implementation language) in terms of a simple example: the module "stack" implemented by the module "array". Section E describes the verification process. The environment that implements this verification process is described in Section F.

B. ADVANCES IN SOFTWARE TECHNOLOGY

Recent advances in software technology may be described in terms of:

* new approaches to the design process.

* new approaches to the implementation process.

Many approaches to the implementation process have resulted in easily realized significant improvements in large systems. Among these are:

* Automatic flow-charters for high-level languages.

* Trace-and-interrupt packages in interactive execution.

* After-the-fact analysis packages that report on the various aspects of program response for test data, e.g., identifying statements that have not been executed.

* Programming without the "go to".

The last concept, expressed in the edict -- thou shall not use the "go to" -- is not a panacea in itself since, as noted by Knuth [28], it is possible to write structured programs with the "go to", and unstructured ones employing only the "modern" control constructs. However, the edict derives from a more powerful notion, namely that significant improvements can be realized by imposing restrictions on the design structure of the software system or on the rest of the development process. Thus, greater effort in the design process can pay off enormously by reducing the complexity of the implementation. This is the basis for most of the following design advances, all of which we view as fundamental.

Abstraction. The basic principle of abstraction in system development is that in order to solve an extremely difficult problem, it is useful to try to identify the details that are inessential in making a design decision at a particular system interface, and to hide them from the view at that interface. In programming, these hidden details typically relate to what we call data representation or implementation. This approach is made more precise by some of the following concepts.

Abstract Machines and Hierarchical Decomposition. Dijkstra [10] suggested the following paradigm for "realizing" a program P that is to execute on a machine M (the total number n of levels is the byproduct of the design process). If it is a difficult task to write such a P, then define a new machine Mn and a program Pn whose execution on Mn satisfies the intent of P executing on M. The machine Mn will provide operations that can be invoked in the execution of Pn with some data in structures that will be modified and referenced. These data structures will be abstract in the sense that their actual representation in terms of the concrete data structures of M is not apparent to the program using the machine. Consequently, Dijkstra viewed Mn as an abstract machine, and Pn as an abstract program since it executes on an abstract machine. Now it remains to implement Mn, which is accomplished by

viewing another abstract machine M(n-1) and a collection of abstract programs P(n-1), each of which implements an operation of Mn. This process continues until finally an abstract machine M1 is defined that is our target machine M. We denote M1 as the primitive abstract machine, "primitive" because it is the lowest-level machine under consideration, and "abstract" because it is not necessarily implemented in hardware; for example, it could be a standard well-defined language.

The following important notions can be observed from this paradigm:

1. The system appears to be built as a hierarchy (or a sequence) of abstract machines. Parnas [42] shows that, in any structure said to be a hierarchy, it is necessary to identify the components of the hierarchy and the relation that binds them. In Dijkstra's view, the abstract machines are the components and the relation "realizes" is the binding relation. The collection of abstract programs executing on M(i-1) realizes the operations of Mi.

2. An abstract program executing on Mi can refer only to the operations provided by Mi. This exhibits the principle of information hiding, namely, it is easier to control the system development if only a restricted collection of operations is available to a program. The modularity principle discussed below reinforces this point.

3. From the perspective of the abstract program that executes on it, an abstract machine can be viewed as maintaining abstract data structures that are modified and queried, using only operations of that abstract machine. These data structures are abstract since they are meaningful only with respect to the operations of the machine, as compared with, for example, the "concrete" data structures of a real machine. This notion of data abstraction is extremely important in designing large systems, and offers significant advantages over the more conventional approach of procedure abstraction. In procedure abstraction, which is the basis of several contemporary methodologies, a large system is decomposed into procedures, but no attempt is made to form subsets of procedures that logically constitute a data abstraction.

4. The presentation describes the system as if it were realized top down. It is perhaps convenient to observe a system a posteriori in this manner. However, actual developments tend to undergo modifications at different levels in orders that are not strictly top-down. HDM recognizes the realities of evolutionary development, and helps to organize it. (Note that Dijkstra suggests that when a particular abstract machine is conceived, the designer usually has in mind certain lower-level machines, presently available or yet-to-be designed, that will eventually serve to implement it. To some extent, this phenomenon guides the design.)

Modules. All large systems exhibit some form of modularity. Parnas [44] has attempted to define a module more specifically and to show what can be gained by decomposing a system into modules. His view is that the internal details of a module should not affect the functioning of any other module. This property is vital both for understanding what service the module supplies and for limiting the effects of changes to a module. Parnas suggested that a module should be a collection of operations and abstract data structures--like an abstract machine, but with each module typically supporting one (or a few) abstract data concepts. Examples of modules might be file systems, memory managers, and message handlers; for complex systems, these might advantageously be substructured into several modules. In our view, an abstract machine at a given level is a collection of one or more modules, but the reader may also visualize each level as a single module.

Abstract Data Type. An abstract data type is a collection of entities called objects and a set of operations defined on objects of the type. Thus, the only access to the objects of an abstract data type is via the operations of the abstract data type. A system can be "realized" as an inverted tree of abstract data types (with the root at the top). A type i residing above type j, type k, ..., implies that the objects of the latter types collectively represent the objects of type i. Similarly, the operations of type i are implemented as abstract programs in terms of the operations of the latter types.

Program Specification. We have previously indicated that a formal specification for a program can be given to a user to describe what the program does, and to an implementor to specify the desired behavior of the program. There are several attributes of a good specification, including precision and clarity. Floyd [17] suggested the use of first-order predicate calculus as a specification language, while others-- notably McCarthy [35]--have advocated recursive function theory. In any event, it has become clear that a specification language should be based on a mathematical theory. A specification can also be associated with an abstract machine, module, or abstract data type to portray the effect of invoking operations. Parnas [43] has described a technique (although not a language) for specifying modules, which indicates the effect of invoking each operation on the values of abstract data structures. Our approach to specification is based on that

of Parnas.

Another approach, due to Guttag [20] and Liskov and Zilles [32], and as applied to abstract data types, views the operations as mathematical functions. The specification consists of expressions in terms of these functions that describe the value of the functions for any sequence of function applications, i.e., any sequence of operations. A survey of several of the current specification techniques is given by Liskov and Berzins [31].

High-Level Programming Languages. The primary original benefit expected of high-level programming languages was the production of programs that could execute on many different machines. This is clearly of economic importance for application programs, and recently for system programs, as their prospects for portability have become enhanced. High-level programming languages also provide built-in powerful features (e.g., storage allocation), thus relieving the burden on the programmer. Recently, new features have been incorporated to aid the programmer in producing more error-free programs. Among these are (1) particular control constructs that often result in programs with cleaner structure, and (2) declarations of strongly typed variables that permit the detection of a large class of programming errors at compile-time. As the concept of data abstraction has become accepted, several recent languages [8, 30, 62, 24, 1] have provided facilities for abstract data types. We are in favor of many of these augmentations to the concept of a high-level programming language, but reject the view that programming languages should continue to become more complex in order to provide those features. Instead we advocate a methodology that provides several specialized languages for system development, one of which is a programming language. In particular, we show how an existing language, JOVIAL, that does not provide direct support for abstract data types, can be easily extended to support data abstraction.

C. SOFTWARE DEVELOPMENT METHODOLOGIES AND WHY THEY ARE NEEDED

There have been numerous advances in software technology, many of which have been applied to real system development with some success. For example, the concept of building systems as hierarchies of virtual systems is well represented in current thinking on communication protocols, e.g., ISO standards, and in operating systems research. However, despite this progress, we believe that the practice is still inadequate, intensified by the need for larger and less error-laden systems.

It is of interest to ask why the wisdom of Dijkstra, Hoare, and Parnas, and others has not been widely accepted and creatively applied by software application system designers. In our opinion, the reasons are as follows:

* The ideas represent an inherently new mode of thinking about systems that is not easy to understand or to apply routinely.

* The ideas have been illustrated only on particular, comparatively simple problems. Many system designers would experience difficulty in extrapolating to more complex problems, e.g., complex operating systems, message processing systems.

* No languages or formalisms have been provided to enable a designer to formulate decisions according to these ideas.

* There are gaps in the theory that prevent the application to complex systems.

The net result has been a misapplication of the basic ideas. Witness the intensity generated over "structured programming", a term coined by Dijkstra to denote the new approach to programming based on abstraction. The concept has been so trivialized by many of its current practitioners (some view it as just programming with single-input, single-output blocks or programming without the "go to") that Dijkstra has almost disavowed any connection with the term "structured programming."

In our view, "methodology" for a technical discipline should consist of notation, formalism, languages, procedures, and guidelines, all based on scientific principles. It should be supported by on-line tools, and illustrated by worked-out examples. In addition, a methodology should be sufficiently robust to allow incremental extension to cover newly discovered problems and advances.

The "scientific principles" for software methodology that are most widely accepted in computer science are the concepts of data abstraction and mathematically based programming. It is also well accepted that the "procedures" should be of the form that precludes the writing of randomly structured programs, and that requires the statement of decisions in a particular order.

Several other methodologies for software development are being pursued elsewhere. These include (1) Higher Order Software (HOS) [21], (2) Chief Programmer Team [38], (3) various approaches involving structured design, and

(4) an approach based on algebraic specifications [20]. We feel that (2) and (3) appear to be too informal and do not embody sufficient data abstraction. The others, although incorporating formalism and data abstraction, have yet to be tested on difficult real systems and do not yet have some of the important ancillary features of a methodology.

D. A SUMMARY OF HDM

This section provides a brief summary of the Hierarchical Development Methodology, as implemented in the HDM-JOVIAL verification system. HDM decomposes the design of a system into a hierarchy of abstract machines, linearly ordered with a different abstract machine at each level in the hierarchy. Each abstract machine in the hierarchy is dependent only on the functionality of lower-level machines. Each abstract machine provides all of the facilities (operations and abstract data structures) that are needed to realize (i.e., to implement operations of and to represent the data structures of) the machine at the next higher level. The facilities of the highest-level abstract machine, and only those of that machine, are visible to a user of the system. The lowest-level machine, denoted as the primitive machine, contains facilities that the designer deems as primitive, e.g., the hardware on which the system is running or a programming language. A machine is itself decomposed into modules, each module having operations and data structures which typically define a single abstract data concept. As in the Parnas module concept, the module is the programming unit of HDM; each of the modules may be independently implemented. The programs implementing a module can access the data structures of their own abstract machine, but not those of lower-level machines. Lower-level data structures may be modified only by the execution of lower-level operations. Thus the internal details of a module remain hidden from above the module.

In HDM there is a clear separation of the development of system realization into stages, as follows:

1. Conceptualization of the system.
2. Definition of the functions of the external interface and the structuring of those functions into a hierarchy of abstract machines, each consisting of one or more modules.
3. Adding further abstract machines to the structure of the entire system, including modules within the hierarchy that are not externally visible.

4. Formal specification of each module.

5. Formal representation of the data structures of each machine in terms of those of the modules at the next lower level.

6. Abstract implementation of the operations of each module, i.e., creating for each abstract machine an abstract program written in terms of the operations at the next lower level.

7. Coding, or transforming the abstract programs into efficient executable programs in the chosen implementation language.

Parnas [45] has characterized software development as a sequence of decisions, where it is likely that decision di is dependent on earlier decisions d1, ..., d(i-1). What Parnas recognized as vital is that there is a proper order for decisions, since the earlier decisions have the greater impact on the ultimate success of the system. Thus it is vital to identify the important decisions and to evaluate them critically. HDM has been designed to formalize this decision model.

Each of the stages of HDM involves the making of decisions, and HDM provides languages to express these decisions. Those decisions associated with stages (1) through (4) are generally considered as design. Those associated with stage (5) and with stages (6) and (7) involve representation and implementation, respectively. The decisions made from stage (1) to stage (7) are roughly in order of decreasing importance. For example, whether or not to use paging involves a design decision, and is clearly more important than the question of how to store the page table --which is a representation decision. The algorithm for page replacement is an implementation decision. This approach contrasts with the current approach to software realization in which the program itself is the only formal (i.e., machine processible) document and, hence, is often used to capture all of the decisions of design, representation, and implementation. In a system designed according to HDM, the four stages would largely be pursued in order. Thus, all of the design decisions should be made before the representation or implementation is attempted. However, backtracking is normally expected. In addition, it is not required that a designer first considers the highest abstract machine, then the next highest and so on, i.e., top-down design. We would expect that attention would be given to several abstract machines at a time, i.e. when a designer conceives of a particular abstract machine at a position in the hierarchy, he might also have in mind lower level abstract machines to

implement that machine. It is also possible for the design to be accomplished top-down while the implementation proceeds bottom-up.

Module specification (stage 4) involves the expression of the intent of a module, independent of its implementation. The language SPECIAL (SPECIfication and Assertion Language) [50, 47] is used for this purpose and enables the concise and formal description of a module. SPECIAL is also used for writing intermodule representations (stage 5), which we call mapping functions. The intermodule implementation programs (stage 6) are called abstract programs, since each can be viewed as running on an abstract machine whose operations they invoke. Abstract programs are intended to be directly compiled into executable code (stage 7). In this report we describe how JOVIAL can be used to write the implementation programs. One of the side benefits of the HDM approach is that much of the complexity of JOVIAL is not required here, since most of the complexity of the programs is embodied within the abstract machine operations invoked by the programs. To handle some of the features of HDM that are not part of JOVIAL (e.g., exceptions, data abstraction, initialization of modules) we impose constraints that are not enforced by the JOVIAL compiler; these constraints are enforced by a preprocessor that is part of the verification environment.

The first three stages of HDM are fundamental to the development. The decisions precisely formulated for these stages provide an early documentation of a system, which is created prior to implementation, and which, for large programs, is usually significantly more understandable than the implementation. They thus provide the basis for good implementation. The results of these stages also provide the assertions that define what correctness means for the system. Since each stage of HDM has an appropriate formal language for expressing the decisions made at that stage, machine checking is possible. Existing tools accomplish some types of machine checking for these stages.

The specifications for the highest-level abstract machine are a concise description of the system as seen by the user, but only in terms of those facilities that are relevant to the specifications, i.e., implementation details are omitted from the specification. In addition, the module specifications and mapping functions are used [48] to formulate assertions for the proof of the abstract programs. From the specifications of a module, pre-

and post-condtions are derived for each of the module's programs. Thus the programs of a module are individually verified. If these programs call programs of other modules, the pre- and post-conditions for these called programs are derived from the specifications of the called modules. The mapping functions provide a definition of the upper module data structures in terms of those of the lower level, thus allowing all specifications to be in terms of the same collection of data structures. Additional parts of a specification will indicate invariant properties that facilitate the verification, in addition to providing additional documentation of decisions.

HDM is a new synthesis of several approaches to software design. It has been developed to address deficiencies in the current software practice. It has been clearly influenced by the concepts of hierarchical programming and its extensions, in particular the important principles of hierarchical design, of doing design prior to implementation, of decomposing a system into small manageable pieces, and of carrying out a proof of correctness simultaneous with design. Although these principles are well-known, they are difficult to apply to real systems. The key to the effectiveness of HDM is that it offers a practical doctrine for constructing, manipulating, evolving, and maintaining formal program abstractions. This property is absent in current structured programming methodologies, and present only in primitive form in modern programming languages. Formal abstraction provides the mechanism for verification, separation of specifications and implementation, variations in the order of binding design decisions, family design, and other desiderata of modern system development. Finally, HDM is compatible with standard and modern programming languages, and can make good use of advanced language concepts.

At present, HDM is evolving and does not yet possess all of the on-line aids that would ease its routine use. For the immediate future (the next two years), it will see its greatest use in systems where correctness is of extreme concern. We anticipate that in the future, HDM-like methodologies will be an important approach to the design of general software.

This document is intended to serve as an overview of the HDM-JOVIAL verification system, describing in some detail most of the features needed to design and implement systems. Since this effort was largely an experiment to determine the feasibility of using an existing programming language with HDM,

some of the features of HDM have not been implemented. A more complete description of HDM can be found in the three-volume HDM Handbook [49, 55, 29].

E. THE LANGUAGES OF HDM

In this section, HDM is used to describe a complete -- although very simple -- system: a "stack" module implemented in terms of an "array" module. The discussion is organized into seven subsections: a review of HDM, and one subsection for each stage (except that of final coding) outlined in the preceding section.

In HDM, a system evolves from an initial concept to verified executable code as a sequence of "decisions". In each stage of the development process, the system developer makes a series of decisions. The stages are ordered so that improper decisions tend to be exposed early, and therefore can also be corrected early.

A primary concern is to illustrate the staged, decision-oriented development of a system using three languages -- HSL (the Hierarchy Specification Language), SPECIAL, and JOVIAL. Brief introductions to these languages are given to produce a reasonably self-contained description. However, the simplicity of the example does not properly illustrate many of the advantages of HDM as applied to complex systems. More details on HDM (SPECIAL, in particular) and a more complex example appear in [49, 55, 29].

1. Review of the Mechanisms of HDM

In HDM, a system is realized as a linear hierarchy (a sequence) of abstract machines, sometimes called levels. The top level is called the user-interface, while the bottom level is called the primitive machine. These two machines together are called the extreme machines. The remaining levels are called intermediate machines. Each machine provides operations, each of which has a unique name and arguments. An operation is invoked, similar to a subroutine call in a conventional programming language, by associating values for the operation's arguments. The invocation of an operation can return a value and/or modify the internal state (abbreviated as state) of the machine, as reflected by the values of the machine's abstract data structures. As discussed later, the "return" of an operation can be either a value or an "exception", the latter corresponding to one of a number of conditions that are defined for the module.

The "user-interface" provides the operations that are available to the user of the system. The operations of the "primitive machine" are typically constructs of a programming language and possibly some of the hardware operations.

A machine specification characterizes the value returned and the new state for each possible machine operation and each possible state of the machine. The specification describes the functional behavior of a system (returned values for all input combinations), but not necessarily the performance of the system or the resources consumed by its execution.

The realization of a machine (not the primitive machine, hereafter noted as machine i) is a two step process. First, the abstract data structures of a machine i (i not 1) are represented by those of the next lower-level machine i-1. Second, each of the operations of a machine i (i not 1) is implemented as a program in terms of the operations of machine i-1. The collection of implementations for all machines excluding the primitive machine constitutes the system implementation.

A machine is sometimes decomposed into simpler units called modules. For the purposes of this discussion, a module may itself be viewed as a machine; however, in reality a module's specification need not be self-contained, unlike that of a machine.

Clearly, system implementation is the desired end-product of the system development process. However, its emergence takes place only at stage 6. In the five previous stages, important decisions are made that logically progress toward the end product.

2. Stage 1 — Conceptualization

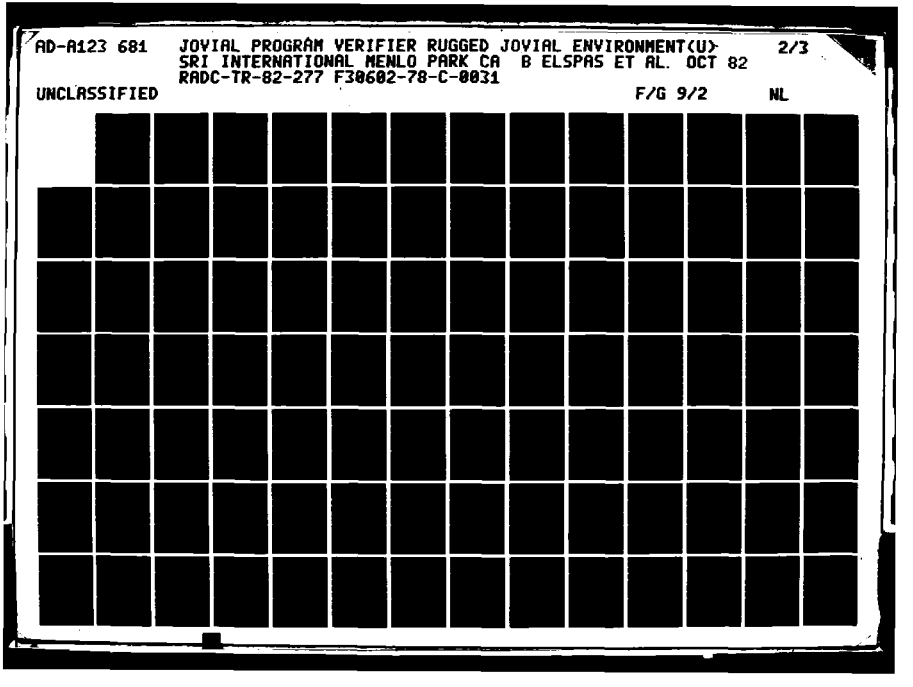
In stage 1, the problem to be solved is formulated in general terms. Typically, the statement is in terms of constraints imposed on the extreme machines, and of the performance expected from the system. Currently, English is employed as the description medium, although consideration is being given to a formal language for conceptualization. For our single example, we will utilize the Conceptualization stage to provide informal descriptions of the extreme machines.

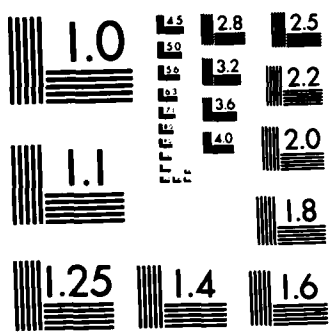
The user interface provides a collection of individually accessible

stacks, manipulatable by conventional stack operations. The primitive machine consists of a collection of individually accessible arrays, as provided by a conventional high-level programming language. This example is developed according to the stages of HDM. The completed example is presented in the following figures, whose content will be explained in the following discussions.

Figure VIII.-1: Specification of the STACK Module

```
MODULE stack

PARAMETERS INTEGER max_stack_size;

ASSERTIONS max_stack_size > 0 AND max_stack_size < maxint;

INVARIANTS ptr() >= 0 AND ptr() <= max_stack_size;

FUNCTIONS

        VFUN stack_val (INTEGER arg) -> INTEGER v;
        VFUN ptr () -> INTEGER v; initially v = 0;
        OFUN push (INTEGER v);
          EXCEPTIONS
            full_stack : ptr () >= max_stack_size-1;
          EFFECTS
            'ptr () = ptr () + 1;
            FORALL INTEGER j :
                'stack_val(j) = IF j = ptr () + 1 THEN v
                                ELSE stack_val(j) END_IF
                END_FORALL;
        OVFUN pop () -> INTEGER v;
          EXCEPTIONS
             empty_stack : ptr() = 0;
          EFFECTS
             v = stack_val(ptr());
             'ptr() = ptr() - 1;

END_MODULE
```


AD-A123 681
JOVIAL PROGRAM VERIFIER RUGGED JOVIAL ENVIRONMENT(U)
SRI INTERNATIONAL MENLO PARK CA B ELSPAS ET AL. OCT 82
RADC-TR-82-277 F30602-78-C-0031
2/3
UNCLASSIFIED
F/G 9/2
NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Figure VIII.-2: Specification of the ARRAY Module

```
MODULE array_mod

  PARAMETERS INTEGER Maxarraysize;

  ASSERTIONS Maxarraysize > 0 AND Maxarraysize < maxint;

  FUNCTIONS

    VFUN read (INTEGER arg) -> INTEGER v;

    OFUN write_op (INTEGER arg, val);
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          FORALL INTEGER j :
                'read (j) = IF j = arg THEN val ELSE read(j)
                   END_IF END_FORALL;

    OVFUN read_op (INTEGER arg) -> INTEGER v;
        ASSERTIONS
          arg >= 0 AND arg < Maxarraysize;
        EFFECTS
          v = read(arg);

END_MODULE
```

Figure VIII.-3: Mappings for STACK and ARRAY

```
MAP stack TO array_mod;

    MAPPINGS
        ptr () : read(0);
        stack_val (INTEGER arg) : IF arg > 0 THEN read(arg)
                                  ELSE 0 END_IF;
        max_stack_size = Maxarraysize - 1;

END_MAP
```

Figure VIII.-4: Abstract Implementation of the STACK Module

```
IMPLEMENTATION stack TO array_mod

START PROGRAM stack;
BEGIN ITEM DUMMYVAR U;
CONSTANT ITEM MaxStackSize U =10;
CONSTANT ITEM MaxArraySize U =11;
TYPE exckinds STATUS
        (V(normalreturn), V(fullstack), V(emptystack));
ITEM exc exckinds;

PROC readop(arg) U;
BEGIN ITEM arg U; END

PROC writeop(arg, val);
BEGIN ITEM arg U; ITEM val U; END
DUMMYVAR=0;      "Dummy statement to get parser to accept
                  code as a main_program_module"

PROC stackinit;  BEGIN writeop(0, 0); END
```

Figure VIII.-5: Abstract Implementation of the STACK Module, cont.

```
PROC push(vv);
BEGIN ITEM vv U; ITEM pointer U;
        pointer = readop(0);
  IF pointer >= maxstacksize-1;
        exc = V(fullstack);
        ELSE
          BEGIN writeop(0, pointer+1);
                writeop(pointer+1, vv);
                exc = V(normalreturn);
          END
END

PROC pop(: vv);
BEGIN ITEM vv U; ITEM pointer U;
  pointer = readop(0);
  IF pointer = 0; exc = V(emptystack);
    ELSE
      BEGIN
        vv = readop(pointer);
        writeop(0, pointer - 1);
        exc = V(normalreturn);
      END
END
END
TERM

END_IMPLEMENTATION
```

3. Stage 2 -- Extreme Machine Interface Design

In stage 2, more detail is developed for the two extreme machines, concerned primarily with the decomposition of these machines into modules and the selection of the operations of the constituent modules. An interface description is derived for each module, specifying the module's operations and providing supporting information. The interface description is sometimes [20] referred to as the "syntax" of a module, in contrast to the specification (stage 4) which is referred to as the "semantics".

For our example, each machine is a single module: "stack" for the "user-interface", and "array" for the "primitive machine". Hence we here refer to "stack" and "array" both as machines and as modules.

a. Interface Description for "stack"

```
MODULE stack

PARAMETERS INTEGER max_stack_size

OFUN push(INTEGER v )
OVFUN pop() -> INTEGER v
```

Some brief remarks about the syntax of SPECIAL are appropriate. First, all reserved words are in capital letters. Second, SPECIAL is a "typed" language in that a type, i.e., a description of allowed data values, is associated with each item when declared, thus permitting subsequent appearances of the items in a specification (see stage 4) to be checked for consistency with their declared type. For present purposes, a type is a set of values. The type INTEGER (a primitive type of SPECIAL) has as values all of the integers -- positive and negative (including zero). The type BOOLEAN (also a primitive type of SPECIAL) has as values TRUE and FALSE. Although not needed for this example, there are additional primitive types. New types, e.g., sets, vectors, structures (records), subtypes, may also be constructed out of existing types.

One or more types noted as designator types can be associated with a module, although our specification does not illustrate this feature. The values of these types, called designators, serve as names for abstract objects of the module. The interface description of a module lists all of its designator types. For example, the "stacks" module, an embellishment of the "stack" module, would have the designator type "stack_name", the values of

which would be the "names" of the stacks supported by the "stacks" module.

Following the designator types, the interface description lists the module's parameters. A parameter of a module is a symbolic constant that, upon initialization of the module, acquires a value which is not subsequently changed by any operation invocation. The parameter mechanism enables a module specification to have some generality. Often a module can appear in different machines in the hierarchy, with a different value for the parameters

Another reason for leaving the values of parameters unbound at specification time is that they are often dependent on the values of lower-level parameters, in a manner that is not decided until later stages.

The "stack" module has the single integer-value parameter "max_stack_size", whose value is the maximum number of elements that can be in a stack. The reader should observe that we have made the decision for this example that all stacks of the module are of the same fixed size.

Finally, the interface description lists the functions of the module, i.e., operations and abstract data structures.3

Depending on whether its invocation returns a value and/or causes a state change, an operation is declared to be one of the following three kinds:

* V-function (VFUN) -- returns a value, but causes no state change. V-functions are hidden, i.e., they cannot be called by the user of the module. The V-functions correspond to the abstract data structures of the module.

* O-function (OFUN) -- causes a state change, but does not return a value.

* OV-function (OVFUN) -- returns a value and can cause a state change.

The "stack" module has four functions, two operations and two functions corresponding to abstract data structures.

* "stack_val" -- returns the value v in position arg of the stack. Note that "stack_val" is a function that represents the stack; it is clearly not possible for the user to request the value at a position

---

3Consistent with Parnas' notation [43], we denote operations as "functions", even though they do not necessarily have the properties of mathematical functions.

of the stack other than the top.

* "ptr" -- returns the number of elements in the stack.

* "push" -- causes an integer v to be placed on top of stack s.[4]

* "pop" -- causes the integer value v on the top of the stack s to be removed and returned.

The reader should note that the decision to provide integer stack is manifested by declaring the argument of "push" and the returned value of "pop" to be of type INTEGER.

b. Interface Description for "array"

For "array", the interface description is as follows:

```
MODULE  array_mod

PARAMETERS INTEGER Maxarraysize

VFUN read(INTEGER arg) -> INTEGER v;
OFUN write_op(INTEGER arg, val);
OVFUN read_op(INGEGER arg) -> INTEGER v;
```

The array is of a given fixed size, namely the value given to the parameter "array_size". Two operations are provided:

* "write_op" -- causes the integer val to be written in position arg of the array.

* "read_op" -- returns the value v in position arg of the array

As with "stack", we have declared the values stored in array to be of type INTEGER.

4. Stage 3 -- Intermediate Machines and Interface Description

In stage 3 "intermediate machines" are selected to bridge the gap between the extreme machines. The choice of intermediate machines is one of the most creative aspects of the use of HDM. In general, relatively simple modules with relatively simple interdependencies are sought. As in stage 2, each intermediate machine is decomposed into modules, each of which is given an interface description. Also in stage 3, a hierarchy description of the system

---

[4]Hereafter we will refer to "stack s" as a shorthand for "the stack that corresponds to the designator s."

is produced in HSL (Hierarchy Specification Language), listing the modules assigned to each machine and the ordering of the machines in the hierarchy.

For the example, no intermediate machines are required. Thus the hierarchy description is

```
STACK EXAMPLE

(INTERFACE level1 stack)
(INTERFACE level0 array)

(HIERARCHY stack_example (level0 implements level1)
```

5. Stage 4 — Module Specification

In this stage, a specification is written (in SPECIAL) for each of the modules identified in the two previous stages. The specification for the modules that constitute a machine provide a complete description of that machine's functional behavior. Thus the specifications of the "user-interface" modules completely describe the functional behavior of the system.

SPECIAL specifications have been designed to facilitate communication of design decisions and to be machine processable for automatic consistency checking. The semantics of SPECIAL can be stated precisely.

a. Expressions in SPECIAL

In presenting a program using a conventional programming language, one produces a sequence of statements. On the other hand, a specification in SPECIAL is a collection of expressions. Each expression is of a particular type, characterizing the type of the values returned by the expression. Expressions are constructed using constants, variables declared in the specification, built-in functions and connectives of the language, functions (O, OV, and V) of the module being specified, and additional functions declared to produce a more readable specification. The following are examples of types of expressions supported by SPECIAL.

1. Arithmetic Expressions

The value returned by an arithmetic expression is of type INTEGER or REAL. An arithmetic expression is a single constant, a variable or a user-defined function of type INTEGER or REAL, or is built out of existing arithmetic expressions using the operations "+", "*", "-", "/".

2. Boolean Expressions

The value returned by a Boolean expression is of type BOOLEAN. The constants TRUE and FALSE are Boolean expressions, as are variables and functions declared to be of type BOOLEAN. The operations AND, OR, "~" (NOT) and "=>" (IMPLIES) are used to build up Boolean expressions from existing Boolean expressions.

3. Relational Expressions

Using the infix relational operators (namely "=", "<", "<=", ">", ">=", "~="), Boolean expressions are constructed from existing expressions. For "=" (or "~="), the resulting expression is of the form A = B (or A ~= B) where A and B are required to have the same type. For the other operators, each of the two component expressions is required to be of type INTEGER or REAL.

4. Conditional Expressions

A conditional expression is of the form IF P THEN Q ELSE R, where P is of type BOOLEAN, and Q and R are of the same arbitrary type. The type of the resulting expression is the type of Q (or R).

5. Quantified Expressions

To express properties relating to a large number of values, SPECIAL provides quantified expressions that are in the first-order predicate calculus. The universal quantified statement is written as

```
FORALL x | P(x): Q(x)
```

or

```
FORALL x: P(x)=>Q(x).
```

The meaning is "For all values of x such that P(x) is true, Q(x) is also true." Clearly, P(x) and Q(x) are of type BOOLEAN, as is the type of resulting expression. The variable x can be of any type, usually declared prior to its introduction in the specification.

The existentially quantified statement is written as

```
EXISTS x | P(x): Q(x),
```

which has the meaning "There exists a value x such that, if P(x) is true, then Q(x) is also true."

b. Role of "?" in SPECIAL

Full SPECIAL (as described in the HDM Handbook) provides the particular value UNDEFINED (abbreviated as "?") to stand for "no value". It is used in a specification where the designer wishes to associate the absence of a meaningful value with a data structure. (UNDEFINED should not be confused with "don't care", which stands for some value.) UNDEFINED is only used in a specification, not in an implementation; no operation can return "?" as a value. For purposes of establishing type-matching rules, however, "?" is assumed to be a value of every type. We have not yet implemented UNDEFINED in the HDM-JOVIAL verification system.

c. Specification of "stack"

Now we are ready to discuss the SPECIAL specification of the module "stack". This specification consists of four paragraphs: TYPES, PARAMETERS, and FUNCTIONS. More complex modules would require additional paragraphs.

1. TYPES paragraph

Here the types referred to in the specification are declared. It is required that all designator types be declared, but the declaration of other types can be deferred until the first appearance of an item of that type.

2. PARAMETERS paragraph

All of the parameters are listed as they appear in the interface description of the module.

3. ASSERTIONS paragraph

In this paragraph constraints on the module parameters are expressed as Boolean expressions. Since the values of parameters do not change, these assertions hold in all states of the module. In the stack example, the assertions indicate that "max_stack_size" is to be greater than 0, but less than "maxint", which is intended to be the maximum integer supported by the module.

4. INVARIANTS paragraph

These invariants (Boolean-valued expressions) express constraints on the V-functions that are to be true in any state of the module. The invariants are redundant with the specifications, in that they are derivable from the

specifications. Using the technique of generator induction our verification process requires that any invariants be proved from the specifications. From the viewpoint of the specifications conveying design decisions, the redundancy offered by the invariants is often desireable.

For the stack example, the invariants indicate that the value of "ptr" (the stack pointer) is always in the range 0 to max_stack_size.

3. **FUNCTIONS paragraph**

Most of the functionally interesting information in a module specification is embodied in the FUNCTIONS paragraph. Each of the operations of the module ("push" and "pop" for the module "stack") is listed and individually specified. In addition, other functions, typically V-functions corresponding to data structures, are introduced to assist in the specification of the operations. It is emphasized that, except for the primitive machine, the data structures serve only for purposes of specification.

We separately consider V-functions and O- and OV-functions.

a. **Specification of V-functions**

For purpose of specification, a V-function returns a value and never causes a state change.

The primitive V-functions -- "ptr" and "stack_val" for the "stack" module -- correspond to the module's data structures. Their specification requires the association of an initial value with each possible argument value. That is, all primitive functions are defined to be "total", although many argument values correspond to physically meaningless conditions. The expression following INITIALLY specifies the initial value. The primitive v-function "stack_val" returns the INTEGER v corresponding to the i-th location in the stack. By not providing an expression, we have decided that the initial value v of "stack-val" is to be "don't care." Note that in general the expression need not determine a unique initial value for a primitive V-function.

The other primitive V-function, "ptr" returns the value i of the stack pointer. The initial value of "ptr" is 0 for the stack, reflecting the decision that the stack is to be initially empty.

A hidden V-function cannot be called from outside the module, i.e., it is not an operation. Clearly, "stack_val" should be hidden since only the top element of the stack is to be accessible. However, some designs for a stack allow the pointer to be accessible.

b. Specification of O- and OV-functions

All O- and OV-functions are potentially state-changing operations, although an OV-function might return a value without changing the state. An operation can return one of n exceptions ex1, ex2, ..., exn (we use the descriptive term "raise" in referring to exceptional returns), or can return "normally". A state change can be specified to occur along with an exceptional return. (This is different from our previous version of SPECIAL wherein no state change occurs when an operation invocation raises an exception.) A value-returning operation (V- or OV-function) will return an actual value upon the NORMAL return; an O-function merely returns. Exception returns are a way of associating particular events with classes of states and values of the operation's arguments. In the specification of an operation, the specification of each exception condition consists of a name (typically a mnemonic for the condition) followed by one or two Boolean expression that (1) characterizes the condition, and (2) characterize the state change and returned value. The list of exception conditions follows the reserved word EXCEPTIONS.

The behavior of an operation that has n exception conditions is determined as follows: if the expression corresponding to ex1 evaluates to true, then the first exception is raised; if the expression corresponding to ex1 evaluates to false and the expression corresponding to ex2 evaluates to true, then the second exception is raised; ...; finally, if the expressions corresponding to ex1, ..., exn evaluate to false, the operation returns normally.

For the O-function "push", there is the single exception condition, specified as:

```
full_stack: ptr() = max_stack_size
```

The expression evaluates to true when the number of elements in the stack is equal to the maximum size of a stack. No state change is specified to occur for this exception.

Following the reserved word EFFECTS, the state changes that can occur as associated with O- and OV-functions, together with the value corresponding to the NORMAL return of an OV-function, are specified. The specification consists of a collection of Boolean expressions, each called an effect (in which the order of presentation is irrelevant). Semantically, the collection of effects should be read as a single expression which is the conjunction of the expressions corresponding to each of the effects. An effect can reference the following: arguments to the operation, values of primitive V-functions before the invocation ("old" values) of the operation, and value that primitive V-functions will obtain after the invocation ("new" values). In the specification, a single quote, "'" , preceding a primitive V-function indicates the value of the V-function after the invocation. The collection of effects defines the new value of each primitive V-function in terms of old values and argument values in the following way: the feasible new values for the primitive V- functions are those for which each of the effects evaluates to TRUE. Thus the specifications need not be deterministic, i.e., they need not define a unique new value for each primitive V-function argument list. However, the specifications for our simple example are deterministic (except, as mentioned above, the initial value of "stack-val" which is irrelevant to the decision that the stack is initially empty).

When a V-function is not referenced in the specification, it is assumed that the new value of that V-function for all all arguments is identical to the old value; not constraining the new value of the V-function for some arguments, means that the new value for these arguments is "don't care". This is different from the previous SPECIAL, in which not constraining the V-function value for some arguments means the new value of the V-function for these values is identical to the old value. The new approach leads to longer specifications, but less of a burden on the theorem prover.

```
For "push", the effects are:
   'ptr() = ptr() + 1;
   FORALL INTEGER j :
       'stack_val(j) = IF j = ptr() + 1 THEN v
                       ELSE stack_val(j) END_IF END_FORALL;
```

They constrain the new value of "ptr()" to be the old value incremented by one, and the new value of "stack_val" to be changed only for an argument corresponding to new top of the stack.

We will not burden the reader with a discussion of the effects for "pop", except for a few remarks. First, note that the returned value v is specified to be the INTEGER on the top of the stack in the old state. The pointer is decremented by 1. The specification does indicate a new value for "stack_val", which means it retains the value it had in the old state.

d. Specification of "array"

Since the specification of the module "array" is relatively straightforward, only a few clarifying remarks are necessary. The V-function "read" stores the values of the array. We have provided no "initially" section, which means the initial values in the array are to be "don't care". The ASSERTIONS paragraph for the module specifies a range for Maxarraysize.

The the two operations ("write_op" and "read_op") each have an ASSERTIONS section. The Boolean-valued expressions in an ASSERTIONS section for an operation, give pre-conditions for calling the operations. For these operations, the pre-conditions specify that the value of "arg" is to be between 0 and the size of the array.

6. Stage 5 -- Data Representation

a. Overview of Module Representation

In this stage, the primary concern is with representing the data structures of each machine (other than the primitive machine) in terms of the data structure of the next lower-level machine. The description of the representation of a machine m in SPECIAL is denoted as the "mapping". As with a module specification, a mapping can be checked for self-consistency, but also for consistency with the module specifications, interface description, and hierarchy description.

A mapping, similar to a module specification, does not act as executable code. Instead, a mapping is a formal description, serving as a record of the representation decisions and as an input to a verification system. Thus the representations are conveniently described using the SPECIAL expression mechanism.

Since the hierarchy for our example contains only two levels, only one mapping is required, for "stack". The mapping for "stack" contains only one paragraph (MAPPINGS); for more complicated systems additional paragraphs (PARAMETERS, ASSERTIONS, and INVARIANTS) might be required. Before discussing

the mapping in detail, it is appropriate to present informally the basic representation decisions.

b. Representation Decisions for "stack"

The stack of integers is represented as an integer array. The current value of the stack pointer for stack is the value in the 0-th location of the array. Each of the "defined" elements in the stack -- those in position 1, 2, ..., ptr() -- are in corresponding positions of the array. Thus the locations of the array starting with location ptr() + 1 hold values that are inconsequential to the "stack" module. Since all locations of the array except the 0-th are available to hold stack elements, the maximum stack size is the array size minus one.

c. MAPPINGS Paragraph

In this paragraph the representation decisions that were informally presented above are precisely formulated. Each upper-level data item is separately represented, that is, associated with an expression in terms of lower-level data items. The expression associated with an upper-level data item can be viewed as a definition of that item in terms of the data items at the next lower level.

The first of the mappings

```
ptr() : read_op(0)
```

captures the decision that the stack pointer is stored in the 0-th element of the array.

The second of the mappings

```
stack_val(INTEGER arg) : IF arg > 0 THEN read(arg)
                                    ELSE 0 END_IF
```

captures the decision that "defined" elements of the stack appear in corresponding elements of the array. For arg corresponding to a position outside the bounds of the stack, "read(arg)" returns 0.

The third of the mappings

```
max_stack_size:  Maxarraysize-1
```

captures the decision that the maximum number of stack elements is one less than the size of an array.

d. **INVARIANTS Paragraph**

This paragraph contains Boolean expressions (invariants) in terms of the lower level that are intended to be true after the execution of a program that implements an operation of the upper level machine. In effect, the invariants express constraints on the lower level state. It should be understood that the invariants are expected to be satisfied by any program referring to the operations of the lower-level machines. Generally, many invariants can be posed, but only those that assist in the verification, that are significant in the documentation of the system, or that simplify the implementation are included. Although not needed for verification, the following invariant expresses an interesting representation decision.

```
read(0) <= Maxarraysize-1
      AND
read(0) = 0
```

constrains the value in the 0-th location of the array to be bounded by 0 and array_size-1. Since the stack pointer is stored in the 0-th location of the corresponding array, this invariant indeed seems reasonable.

7. **Stage 6 -- Abstract Implementation**

In stage 6 each machine (other than the primitive machine) is implemented in terms of the machine at the next lower level. For machine i, the implementation consists of

* An initialization program whose execution causes the state of machine i-1 to become a state that maps (up) to the initial state of i; A program for each operation of i that satisfies its specifications.

All programs of the implementation of i reference operations of i-1.

For expressing the implementation programs, we developed the language ILPL (Intermediate Level Programming Language) to identify those features of a programming language that are well-suited to handling the issues of abstraction that dominate HDM. However, we believe almost any programming language could be used to express machine implementations. In this report, we indicate how JOVIAL can be used for this purpose. To motivate the way JOVIAL would be used in conjunction with HDM, we first present a brief overview of ILPL.

a. Overview of ILPL

ILPL is an extremely simple, imperative programming language that avoids many of the complex features of high-order programming languages (whose concepts are effectively achievable within the specification). The main purpose of ILPL is to describe a sequence of calls to operations. Some of the significant features of ILPL are the following:

* Simple argument-passing discipline: In ILPL, all arguments are passed by "value". Of the conventional schemes for passing arguments -- by "value", by "reference", and by "name" -- "call by value" is conceptually the simplest. It has several advantages in implementing secure systems, including the avoidance of a wide class of security bugs referred to as "time-of-creation to time-of-use" modifications [41].

* Limited built-in data structures: In HDM, most of the data structures are provided by specified modules. Thus, ILPL need provide only a few simple kinds of data types, namely integers, characters, booleans, vectors, and structures (records).

* Controlled side-effects: Since an ILPL program consists mainly of calls to operations of a machine, the only side-effects are changes to V-functions as portrayed in the specifications of modules.

* Simple storage allocation: The only allocation carried out in the execution of an ILPL program is for local variables. Any dynamic allocation of objects is reserved for the modules that maintain such objects.

* No design aids in the language: Since HDM separates design and implementation decisions into distinct stages, all descriptions relating to design are expressed in SPECIAL or HSL.

* Structured exception handling: The program implementing an operation has multiple return points, one corresponding to the normal return and the remainder corresponding to the exceptional returns. A program referencing an operation "handles" any of the possible returns -- exceptional or normal -- for that operation.

* Type compatibility with SPECIAL: ILPL provides only a subset of the types of SPECIAL, essentially those that are easily implemented. Among those omitted is the "set". However, ILPL does support designator types, enforcing the same protection rules for designators as SPECIAL.

b. Linking JOVIAL with HDM

Clearly, JOVIAL does not satisfy the above guidelines. However, a semantic subset of JOVIAL can be defined to satisfy these guidelines. We illustrate the semantic subset we have in mind with respect to the programs that implement "stack".

JOVIAL does not provide a module mechanism. However, we can use the PROGRAM declaration statement to cluster a set of procedures that we will interpret to be an HDM module. JOVIAL does not provide any direct encapsulation for modules; that is, the internal structures of a JOVIAL PROGRAM can be referenced from outside the PROGRAM. The only way to prevent such accesses in violation of the HDM model, is to provide a preprocessor to detect such accesses.

A JOVIAL PROGRAM links two or more modules -- the upper module and the lower modules that implement the upper module. Some of the module PARAMETERS can be mapped directly to program constants, and will be declared in the JOVIAL program to be constants -- for example, Maxstacksize and Maxarraysize in the "stack" example. HDM allows the declaration of constant functions, which in turn will be implemented by *JOVIAL procedures* that do not change any global variables.

JOVIAL does not provide support for structured exception handling, but by judicious use of the type mechanism and the allocation of global variables we can come close to what is provided in ILPL. Referring to the implementation "stack", we define a type called "exckinds" whose elements are the names of exceptions (including no exception) that can be returned by any of the PROGRAM's procedures -- here "normalreturn", "fullstack", and "emptystack". A single variable, global to the PROGRAM, is declared to have as value an element of the type "exckinds".

OV-functions can return a value, in contradiction to the JOVIAL procedure which does not return a value. Reconciliation is achieved by declaring all procedures corresponding to OV-functions (e.g., "pop") to have an output argument (i.e., "vv"); no read reference is permitted to such argument before it is written. Simliarly, HDM does not allow the input arguments to be modified; we enforce this restriction on all JOVIAL procedure input arguments.

The JOVIAL implementation of "stack" consists of three procedures:

stackinit, push, and pop.

The procedure "stackinit" is intended to initialize the array to a state that maps up to the initial state of the stack. All that is required here is to set the first element of the array to 0, the element in which the stack pointer is stored.

The procedure "push" returns an exception if the pointer (the first element of the array) is too large. Otherwise, it stores, as a side effect, the element vv at the appropriate position in the array. The pointer is also incremented.

The procedure "pop" returns an exception if the stack is empty. Otherwise it returns the topmost element of the stack and decrements the stack pointer.

Verification of the VCs for these procedures is shown in Appendix F.

We believe that the guidelines just presented are adequate and effective for writing JOVIAL programs to implement HDM specifications. It is essential that the programmer be fully acquainted with the viewpoint and techniques of HDM.

F. VERIFICATION CONDITION GENERATION

In this section we describe the process of verification condition generation for a collection of JOVIAL programs organized into modules according to the HDM discipline. The discussion is organized according to the key features of HDM (presented in the preceding section) and the constraints we impose on JOVIAL to link it with HDM. We have developed verification condition generators that produce formulas for the two theorem provers most recently developed at SRI: the Boyer-Moore theorem prover and the Shostak theorem prover (STP). The discussion that follows describes the process of generating verification conditions for STP.

There are three main semantic aspects of a JOVIAL program to be dealt with during verification condition generation. These are the control flow of the program, the change of state caused by operations, and the satisfaction of certain necessary conditions for proper execution and termination of the program. Since these are very different aspects of the program they are described and handled separately.

The flow of control through a JOVIAL program (i.e, the path analysis) is solely dependent on the semantics of JOVIAL statements that change the flow and are not dependent on the program specifications. This control flow is described using weakest precondition Hoare rules and is manipulated using the Meta-VCG facility [39]. The action of this Meta-VCG is described in the referenced paper and in Sec. VI-D. It produces verification conditions through a simple pattern-matching and substitution algorithm which uses only a JOVIAL program segment and the Hoare rules.

Various conditions are necessary to ensure the absence of execution time errors and proper execution. Possible execution time errors include overflow, underflow, array index out of bounds, and assignment out of range. Proper execution consists mainly of proving the preconditions to functions and procedures. These conditions are referred to in functional terms throughout the Hoare rules. Each function is expanded out into a Boolean expression before the verification condition is proven.

The state during the execution of a program is specified and changed as indicated by and according to the HDM model. Only two JOVIAL statements are allowed to change the state, the assignment and procedure call statements. The procedure call is the main point during verification condition generation where the semantics of JOVIAL and SPECIAL interact. There are references in the Hoare rules to functions which capture these semantics.

At this point, it is relevant to discuss the semantics of a SPECIAL module specification and the correlation between specifications and code. A module definition consists of types, parameters, definitions, assertions, invariants, and functions.

The types are as described in the HDM handbook except that the only types supported are INTEGER, BOOLEAN, enumerated types, VECTOR, STRUCT, and SET. there is no DESIGNATOR type or UNION types. These types are meant to correspond to the JOVIAL types INTEGER, BOOLEAN, scalar, ARRAY, and RECORD types. There is no correspondence to the SPECIAL SET type which can only be used for specification purposes. Whenever a correspondence between a JOVIAL and SPECIAL declaration is necessary, the above correspondence between types is required.

Parameters come in two flavors, symbolic and functional. Although

symbolic parameters are equivalent to constant functional parameters, they are distinguished because of the difference in their correspondence to the JOVIAL. Symbolic parameters are considered to be constant values and if they have an implementation, then the implemented value must be kept constant. The obvious correspondence of symbolic parameters is to JOVIAL CONST declarations. However, JOVIAL CONSTs are only allowed to be simple scalar types so it is also allowed to have a symbolic parameter correspond to a JOVIAL global variable with the restriction that the variable never be modified. Functional parameters are considered to be uninterpreted mathematical function symbols and never have an implemented correspondence. Functions that are meant to be implemented must be specified in the FUNCTIONS paragraph.

The ASSERTIONS paragraph of handbook HDM has been extended into two paragraphs, the ASSERTIONS and the INVARIANTS. The assertions are restrictions on the value space of the parameters only. Only constant values of SPECIAL including the symbolic and functional parameters of the module can be referenced. Symbolic parameters need not be restricted in the assertions since their value can be restricted in the mapping. However, there is little point in failing to restrict a functional parameter since without the restriction, the function can have any value and little can be proven about it. In code proofs, assertions can be assumed to be always true since if the assertions are initially true and the values are constant, then they must be always true. Assertions about implemented constants must be shown to be satisfied by the actual implemented value. Hence such assertions are guaranteed to be consistent.

It should be noted with some caution that assertions about uninterpreted symbolic parameters or parameter functions can be inconsistent in such a way as to render the proof trivial. Care must be taken by the user to ensure that there exists a mathematical function that satisfies all of the constraints. Mechanical assistance in this process is not currently available.

INVARIANTS are constraints on the values allowed to be taken on by the state functions (VFUNs). This might include value range restrictions or constraints on the relative values of a set of state functions. These invariants must be true in the initial state of the state functions and must be proved to be true after each invocation of a state-changing operation of the module. This is done by assuming the invariants to be true in the state

before the invocation of the operation and assuming that the effects of the operation imply the invariants in the post state. To help in the proof of the code, the invariants can be assumed to be true in the state at the beginning of the code. The invariants of the lower machine can be assumed to be true in all states of the upper machine.

The FUNCTIONS paragraph is used to specify the state and operation functions. The state functions capture the state of the module and intuitively correspond to the state of the machine during execution. This correspondence is never actually specified but is assumed for proof purposes. The operations of the module provide the only way to modify the state of the module. These operations must correspond directly to an operation in the implementation. An operation that is either not implemented or not specified cannot be proven.

A VFUN (value-function) is used to represent a state function of the module. It has optional parameters and represents a set of values depending on the state the module is in. Handbook HDM allows VFUNs to be either a state function or an operation and distinguishes between HIDDEN and VISIBLE, and between PRIMITIVE and DERIVED. As will be described later, all aspects of the state are HIDDEN. For the sake of simplicity, all VFUNs are PRIMITIVE with OVFUNs being used in place of DERIVED VFUNs. This means that all VFUNs are state functions. Besides formal parameters, VFUNs have an optional INITIALLY clause which is used to specify the initial value of the VFUN. The initialization of the implementation state is described later. These INITIALLYs are used to prove the INVARIANTS as described above.

The operations of a module are specified as OFUNs (operation functions) or OVFUNs (operation-value functions). They differ only in that an OVFUN returns a value while an OFUN does not. There are three clauses of an operation, the preconditions, the exception conditions, and the postconditions. The preconditions are the ASSERTIONS and are constraints on the allowed values of the input parameters and the state of the module. The EXCEPTIONS are a set of conditions under which the operation is aborted during execution. The postconditions are the EFFECTS and describe the effect the operation has on the state of the module.

The meaning of the operation specification is most easily described in

terms of its correlation with the corresponding JOVIAL operation. An operation in JOVIAL is represented as a PROCEDURE or a FUNCTION. Usually, an operation is implemented as a PROCEDURE though some simple operations can be FUNCTIONS. A strict correspondence is required between the specification and the code in order for the proof process to work. The correspondence in the header of an operation is a name correspondence. The name of the OFUN or OVFUN must be the same as the name of the PROCEDURE or FUNCTION. The input parameters must agree in number, name and type. The output parameter, if any, must agree in name and in type. The way a value is returned in an implementation depends on whether a FUNCTION or PROCEDURE is used. The value of a PROCEDURE is returned through an additional VAR parameter of the same name and type as the returned symbol in the OVFUN. In the implementation, the value is returned through this parameter. In JOVIAL, the type of the returned value of a FUNCTION given in the header must be the same type as the returned symbol in the OVFUN. The value to be returned is indicated by an assignment to the name of the FUNCTION as in normal JOVIAL. Because of restrictions inherent in the proof process, a JOVIAL FUNCTION is only allowed as the implementation for an OVFUN under very restrictive circumstances. Under some assumptions, an implemented expression can be treated as a purely mathematical expression. This is very convenient for proof purposes. Some of these assumptions are checked during verification, such as the absence of errors like overflow. Some assumptions such as the totality of functions and the commutivity of certain mathematical operations are ensured through restrictions on FUNCTIONs.

As for the restrictions under which a FUNCTION can be the implementation of an OVFUN: first, JOVIAL only allows a FUNCTION to return a simple scalar type, therefore the OVFUN being implemented must return such a simple type. Functions are not allowed to have side effects (and therefore cannot change the state of the module) for they could possibly invalidate the assumptions of the commutativity of certain mathematical operations. To keep them total, OVFUNs having a FUNCTION implementation cannot have exceptions and must therefore always return a value.

The ASSERTIONS of an operation is a set of preconditions that must be satisfied at the invocation of the operation. These preconditions must be proven at every invocation of this operation to ensure that the partial

function which the operation implements has a computable value. These preconditions are assumed to be true when proving the operation.

The EXCEPTIONS section allow the specification of abnormal returns from the operation. This usually occurs because the current state of the module prevents the completion of the operation. This differs from the ASSERTIONS in that preconditions are proved to be true while exceptions are proved to be handled correctly during execution. The preferred way for the programming language to handle the exception condition is by an automatic change in the flow of control as in ADA. JOVIAL does not support the handling of exceptions so the actual use appears to be more like a multi-return mechanism. The EXCEPTIONS section consists of a list of exception conditions, each condition having three parts, the name of the exception, the condition under which the exception is raised, and a postcondition describing the change of state associated with the exception return (note that this can include a specification of the returned value). The allowance for a change of state on an exception is a relaxation of the constraints on conditions described in handbook HDM. This postcondition is optional, and if missing, indicates that no state change occurred.

The method for associating the specified exceptions with the implemented program is rather arbitrary and unesthetic due to the complete lack of any such facility in JOVIAL. The approach chosen is meant to be flexible, simple, and not dependent on very much additional proof mechanism. To indicate and handle exceptions, it is necessary to communicate to the calling environment which exception, if any, has been raised. This is done through a special global variable EXC which must be declared of the correct type in the implementation. If the operation has exceptions, then the variable EXC is set to either NORMAL_RETURN in the case where there is no exception raised, or to the name of the raised exception, as indicated in the specification of the exception conditions. Therefore, the variable EXC must be declared as a scalar type consisting of the name NORMAL_RETURN and all of the exception names in any implemented operation. On return from an operation which may have raised an exception, the program may either test the EXC variable or may assume that no exception was raised. In the latter case, it will be necessary to prove that no exception occurred in order to benefit from the effects of the operation in the proof. The actual postcondition created during the

verification condition generation process is a combination of the EXCEPTIONS and EFFECTS sections and is described immediately below.

The postcondition of an operation is a specification of the effects the operation has on the state of the module. For the normal return this specification is captured in the EFFECTS section. The usual and most usable form is to specify the new state of each state function in terms of the old state function, universally quantified over the formal parameters. The actual postcondition is a combination of the exceptions and the effects. Each set of effects, the normal ones and each of the exception effects are guarded by the value of the special variable EXC in the form of an implication. For each possible change of state, there is assumed a clause of the form "EXC = name => effect". The values that EXC might take on under various conditions is specified in an IF expression. In the order that the EXCEPTIONS are listed (order is important), each exception condition is a conditional (IF) expression whose THEN part specifies the resulting value of EXC. The final ELSE part specifies that "EXC = NORMAL_RETURN". Hence the IF expression looks like:

```
IF cond1 THEN EXC = name1
   ELSE IF cond2 THEN EXC = name2
           ...  ELSE EXC = NORMAL_RETURN
```

If the normal return effects are necessary in a proof, they can be extracted from the implication described above by proving that EXC = NORMAL_RETURN. This can be done by either testing for the equality in the code or by proving that none of the exception conditions could have occurred, thereby reducing the above IF expression to the desired equality. Thus the code can efficiently ignore the possibility of an exception if it can be proven to be impossible.

Throughout these specifications of the change of state to VFUNs, it is implicit that no specification of the new value of a VFUN implies that no change occurred. This is supported in that the VCG automatically creates the expressions that explicitly state this fact.

A program is verified by proving tnat the accumulated state changes from each procedure call imply the desired change of state for the entire routine. Each procedure call describes its associated state change by specifying the new state of each state function in terms of the state just before the

procedure call. Hence a series of procedure calls describes a progression of ever-new states which are entered as a result of each procedure call. Intermediate and final assertions for the code still speak in terms of the new state and the old state. However, during execution of the code many states are entered. There is, therefore, a concept of current state in the verification condition generation process. The effects of an operation are considered to be universally quantified over the state and they may reference the current and previous states. When a procedure call is encountered, the current state is considered changed to a new state and the effects of the operation are instantiated to the current and previous states. Here, all quoted references to a state function are references to the current state and all unquoted references are to the previous state.

Quoted and unquoted references to a state function suffice to distinguish only two states even though many states may be encountered during a routine. For this reason, the notation for the state of a VFUN is changed during the verification condition generation process. The state of a VFUN is indicated by extra parameters to the VFUN. These extra parameters indicate the kind of state and the current "state" of the state. There are different states for each lower module and a different state for the path initiated at the beginning of a program. When a state change occurs, the state is modified by embedding it in a call to the function NEXT. Hence, if the state is "NEXT (STACK_MOD.STATE)" then the new state would be "NEXT (NEXT (STACK_MOD.STATE))." The state parameters possessed by each VFUN depend on the module containing that VFUN. VFUNs of the lower modules have one state, i.e., the state of that module. VFUNs of the upper module have a set of states, one for each lower module. This is because the state of the upper module is completely dependent on the state of the lower modules. Also, since definitions are allowed to reference VFUNs, they also have state parameters.

IX. USER EXECUTIVE FACILITIES

A. INTRODUCTION

The primary purpose of the RJE user executive facilities is to provide the system user with a "friendly" interface for communication with the verifier. Since the system is structured into two principal environments--the front end (parser and VCG) and the theorem prover--we have provided two separate user executives, one for each of these environments.

Some of the features of these user executives are identical and others are quite different because of the distinct needs of the user in the two environments. One feature common to the two user executives is the on-line documentation facility that makes it possible for the user to query and obtain printed responses at his terminal regarding the current status of the verification of some JOVIAL program on his files.

In the next two sections, we shall describe the principal features of the two user executive subsystems of RJE.

B. FRONT END USER EXECUTIVE

The user executive for the front end environment plays the following roles:

* It furnishes a simple set of single-word and two-word commands for the user to communicate with the front end environment, i.e., to initiate front end operations on a JOVIAL program to be submitted for verification, query the environment (and the file system) regarding the status of verification for that (and other) programs, and write associated files for subsequent use by the theorem-proving environment.

* It furnishes an on-line facility documenting the front end environment (including the user executive itself), whereby the user may ask questions about the facilities (commands, functions, variables, and file structures) of that environment and also receive assistance (should he reach a point at which he does not know how to proceed further).

1. Command Structure

One possible command structure for the front end user executive would have been to design a command language and construct a separate command parser for it. This approach has both advantages and disadvantages. Among its advantages is an ability to handle arbitrarily complicated command strings, built-in prompting, and command completion, as well as to provide a well-integrated error recovery mechanism. The main disadvantage of the command parser approach lies in its complexity and the consequent relative difficulty of making design changes. Thus, every addition, deletion, or change in the command language would have required a regeneration of the command parser. The design of our command language underwent several months of changes as a result of extensive experimentation with the system. The effort entailed in having to reconstruct the command parser for each set of changes would have been quite prohibitive, even though the parser would have been much smaller and simpler than, e.g., the JOVIAL parser (see Sec. B.).

The command language structure finally adopted for the user executive comprises only about two dozen command verbs, of which ten are concerned with operations of the front end proper, three invoke the on-line documentation facility, and the remaining twelve or so are concerned with inherently TOPS-20 executive commands, such as fork control and directory operations. About half of these commands are one-word commands (intransitive verbs), while the remainder take a single optional argument (i.e., an object acted upon by the command verb).

It should be clear that this is a fairly simple user language, one that hardly warrants a separate parser for its implementation. Our decision, therefore, was to forego the design of a separate command parser for the interpretation and execution of user commands (for both user executive subsystems) and simply use the LISPX read facilities of INTERLISP-10 [25] for this purpose. Thus, when typing commands to the user executive the user is simply communicating with the top level of INTERLISP, i.e., he is typing to the INTERLISP prompt "_". To avoid the necessity of typing parentheses around command arguments we made use of the LISPXMACRO feature of INTERLISP. Thus, each command verb of the user executive is given definition in the subsystem as a macro, and the arguments (if any) are automatically supplied as arguments to the corresponding INTERLISP function that implements the macro.

The decision to use INTERLISP macros for the user interface also has the attendant advantage that no discontinuity is introduced between the system with which the user communicates normally (i.e., the user executive), and the one (i.e., INTERLISP) which he is detoured into in the event of an unforeseen error (e.g., a resource error such as a missing file, a human typing error, or simply a system error at the LISP level). In particular, some interactive features have been designed to permit the user to employ the BREAK feature of LISP to recover from certain errors.

The typical user dialog with the RJE user executive is fairly interactive. This is true even more for the theorem prover environment. In the implementation of the RJE user executive (and even more in the corresponding executive for the prover environment), extensive use was made of INTERLISP's [25] ASKUSER function. When this function is invoked, the user is presented with a question, and the nature of his on-line response determines the system's next action. In such ASKUSER interactions, the user response "?" always provides a list of the options available. However, in other cases in which only a yes or no response by the user is appropriate, we employed a simpler version of ASKUSER (our own function "ASK?") to implement binary choices without incurring the overhead and programming of all the unneeded ASKUSER options.

The commands (command verbs) proper to the front end user executive are START, STATUS, SHOW, NEXT?, VCS?, PROGRAM?, PARSE?, PROOF?, and WRITEVCFILE.

Details of their uses are fully explained in the RJE User Manual [15]. However, a brief outline is given here for a few of the commands.

START is a no-argument command used to initialize the front end data base and begin the user's interaction with that subsystem.

Most of the other commands just listed summarize various aspects of the current state of that interaction. For example, STATUS yields a global summary (for the program currently under verification, if no argument is typed, or for a selected program supplied as argument to STATUS).

SHOW may take an optional argument. If the argument is omitted, SHOW causes a list of options to be printed first. Depending on the option selected, it permits the user to examine either the current JOVIAL program

source file, the parsed internal version of that program, the VCs already computed by the front end, or a previously written proof trace file.

WRITEVCFILE is a no-argument command that causes the current verification conditions (along with the parsed internal form and certain other information) to be written out to a so-called VC file. Incidentally, if an appropriately named VC file is already found on the user's directory and this file contains the same VCs in the data base of the current environment, no new file will be written. This file is accessed during later operations in the theorem prover environment to transfer the theorems (VCs) to be proved to that environment. Among the other information stored in the VC file is the name of the source file from which the VCs were constructed. To guarantee consistency among versions, this item is checked before the actual theorem-proving operations begin.

The commands DOC, HELP, and COMMANDS? are concerned with interaction with the on-line documentation facility. DOC usually takes a single argument KEYWORD for which information is desired. This causes a special documentation file (called PVCG.DOC, on the user's directory) to be searched for the keyword (prefaced by an asterisk), and the relevant information section following that occurrence is then printed at the user's terminal. For example, typing "DOC DOC" causes the information contained in PVCG.DOC about the keyword DOC to be printed. (Actually, a second, optional, file name argument may be included in the typed command string if the user wants to search a different file. This argument defaults to PVCG.DOC when it is omitted.) As a convenience to the user, the command verb ? may be used in place of DOC.

The HELP command (with no arguments) causes a special HELP file to be printed. This file simply contains directions for using the on-line documentation command DOC.

The command COMMANDS? (also with no arguments) causes a list of the user executive commands (in fact, the value of the system variable PVCG.CMDS) to be printed.

The remaining commands:

EMACS, LAST.EMACS, EXEC, PROVER, CONTIN, CONN,
TY, DEL, UND, DIR, NDIR, SY, DSK, and LOGOUT,

are largely concerned with interactions from within INTERLISP with the TOPS-20 executive or the file system. The reader is again referred to [15] for

further details.

C. THEOREM PROVER USER EXECUTIVE

The degree of interaction required in running the theorem prover is considerably more complex than for the (largely automatic) operations of the RJE front end. Some of these interactions are provided for within the Theorem Prover itself, i.e., as part of the Boyer-Moore system. However, a user interface on top of the Prover itself was judged to be necessary for the following reasons:

* To furnish the user with an on-line documentation facility that he could query for explanations of the commands available to him, for guidance about the procedures involved, and even for general information about mechanical theorem proving.

* To establish a user profile that would adapt the system to behave somewhat differently for three classes of users: novices, those with some experience, and experts.

* To carry out additional bookkeeping with respect to the program (or programs) being verified, their associated files, and their state of verification.

The on-line documentation feature was implemented through the same means already described for the front end user executive, the only difference being in the documentation file accessed by the function DOC. The file in this case is called PROVER.DOC. Naturally, the keywords (arguments presented to DOC) are different for the Prover, but a request for information about the commands, for example, is obtained by typing "? COMMANDS", just as in the front end executive. Even more general information is obtainable by typing "? INFO"; this presents to the user a list of general categories of information contained in the documentation file. This file is about 30 pages in length, as compared with a mere six pages in the documentation file PVCG.DOC for the front end, which may convey some idea of the relative complexity of the subsystems described by these two files.

When a user enters a fresh theorem prover environment, the system asks the user if he has used the system before. The options available for his reply (which are printed if he responds with "?") are: Yes, Slightly, and No. Incidentally, he need type only the first letter (i.e., Y, S, or N) in reply, since ASKUSER provides command completion. Depending on the user's response, two Boolean flags, AUTOFLG and VERBOSEFLG, are set to either T or NIL to

establish a user profile. This profile puts the user into one of the following modes:

AUTOMATIC/VERBOSE	(for first-time users)
AUTOMATIC/CONCISE	(for intermediate-level users)
MANUAL/CONCISE	(for advanced users)
MANUAL/VERBOSE	(obtainable manually only)

The mode status can be read from the terminal by typing (GET.USER.MODE).

It is seen that the user modes differ in two orthogonal dimensions: (1) the degree of interaction available (AUTOMATIC vs. MANUAL), and (2) the amount of printout displayed to the user (VERBOSE vs. CONCISE). The user is occasionally asked if he wishes to change his mode of interaction, and he is also able to toggle either of the two mode flags manually by typing, e.g., AUTOMATIC, AUTOMATIC OFF, VERBOSE, or VERBOSE OFF. An additional aspect of the user modes is that a system parameter RJE.USER.WAIT receives different values, depending on the mode in use. This parameter determines how long (in CPU-seconds) the system waits before supplying a default response if the user hesitates too long in responding to ASKUSER.

The bookkeeping functions of the theorem prover's user executive are concerned with the following items:

* The name of the program currently being verified.

* The existence (on the connected directory) of files associated with this program. These are the JOVIAL source file, a VC file, an events file (which stores theorem prover definitions, lemmas, and declarations associated with a proof or a partial proof), and a proof file (which contains a trace of previous attempts at a proof).

* The names of the verification conditions whose proof is being attempted, which of these have been proved, which have failed, and which have been subjected to (tentative) editing in ad hoc attempts to achieve a proof. (In the latter case, the user is reminded of this fact and is admonished to redo the proofs in legitimate fashion).

* The date and time when the proof was completed, or when it was last attempted.

It must be emphasized that a large additional amount of record-keeping is embedded in the Boyer-Moore Theorem Prover itself, apart from the user executive features just described. The Prover keeps track of all events (i.e., declarations, definitions, axioms, lemmas, etc.) which have been entered into it. It also keeps a list of conjectures (i.e., alleged theorems)

whose proofs have failed, i.e., as the binding of the global system variable FAILED.THMS. The variable CHRONOLOGY is a list of names of successful events in reverse chronological order. It is possible to undo such events, i.e., by means of the function UNDO.NAME(event-name), and also to redo a selected list of (undone or never done) events with (REDO.UNDONE.EVENTS event-list). The latter function is used by RJE, e.g., in incorporating the events from an events file into the Theorem Prover knowledge base prior to attempting the proofs of VCs. These matters are discussed in more detail in [4].

X. CONCLUSIONS

A. INTRODUCTION

In this final chapter we attempt to summarize our general conclusions drawn from four years of research and development on the RJE effort. We also give some indications as to the most promising directions for future work in the general areas of software specification, development, and verification.

B. PROJECT AREAS

The distinct areas which the RJE project work encompassed have been listed in Chap. I, but they are repeated here for convenient reference.

1. Development of a fast, powerful, and flexible mechanical theorem prover capable of proving formulas in a user-extendable assertion language.

2. Accommodating as much as possible of a large, complex programming language in wide current use.

3. Providing a user-friendly environment where a relatively nonspecialized user could have some hope of overcoming the severe conceptual and practical problems involved in the specification, refinement, and verification of practical programs.

4. Combining the verification method based on Floyd assertions with a hierarchical program development methodology (the SRI HDM approach).

A fifth task (added in a contract modification dated 2 January 1981) was concerned with the development of formal models for machine arithmetic of numerical data (fixed and floating point numbers). It addresses certain difficulties occasioned in verification of correctness by the presence of round-off, truncation, and coercion of numeric data. The objective of the added task was, in part, to find a more satisfactory means for handling these difficulties, and to incorporate an associated axiomatization for fixed-point and floating-point operations into the RJE verifier.

We comment briefly in the following paragraphs on the degree of success with which we feel these objectives have been advanced by our work.

1. Theorem Prover

The results of the theorem prover work have been almost uniformly encouraging, thanks to the dedicated efforts of Robert S. Boyer and J Strother Moore, who deserve the full credit for its creation. During the past four years they have not only greatly extended the power and versatility of their Theorem Prover, but have also (with support from some contracts[5] in addition to this RADC contract) completed a number of papers, reports, and a book (e.g., [4, 5, 2]) describing their work.

We feel that their contributions, not only to furthering the goals of this project, but also to automatic deduction in general have been crucial. We could not have envisaged completing verification for some of the many programs we worked on without the availability of the Boyer-Moore system in its present form. This is not to say that no further advances are possible in automatic deductive systems. On the contrary, recent work by Shostak, et. al., [53, 52, 51], Gerhart, et. al., [18], and Goguen, et. al., [19] all suggest that there are other ways to pursue the problems of automatic deduction, particularly in relation to more sophisticated specification languages.

2. Jovial Axiomatization

We are less than satisfied with our achievements in attempting to capture in a rigorous fashion the semantics (and syntax) of a large, rich real-world programming language. In part, this dissatisfaction stems from the present ill-defined state of affairs with respect to JOVIAL (and other high-level languages, for JOVIAL is by no means unique in this respect). Nevertheless, we cannot place all the blame for our failure to "axiomatize" all of JOVIAL on the absence of a rigorous language standard. Another contributing factor resides in the shifting directions our work necessarily followed under the impetus of changes in the target version of JOVIAL, and in the language standards, as discussed in Chap. III.

In retrospect, a potentially better approach to this axiomatization, albeit one fraught with its own peculiar sorts of difficulties, might have been along the lines of Boyer and Moore's work on the "axiomatization" of a

---

[5]Contract No. N00014-75-C-0816 with the Office of Naval Research, and Grant No. MCS-7684125 from the National Science Foundation.

subset of FORTRAN [5]. This formidable achievement really amounts to an operational definition for that language, together with a VCG implementation for it. The approach they used is, in particular, better suited to the precise modeling of finite machine arithmetic. It should be noted, though, that even here the FORTRAN subset did not include fixed-point numbers.

3. User Environment

Our plans for this phase of the work envisioned (see [40]) an extremely versatile, user-friendly environment that could greatly aid the RJE user keep track of all aspects of software specification, design, implementation, and verification.

The user interfaces we designed and implemented for RJE achieved considerable improvement over our previous efforts [12, 13]. The RJE verifier provides numerous features that take over much of the burdens of record-keeping, guiding the user as to how to proceed, and the like. These aids are certainly essential to the highly iterative (often cut-and-try!) processes entailed in specification, design, implementation, and verification of software.

However, the RJE user interfaces we developed do not permit the user to see directly how the effects of changes in one domain propagate to other domains, where by "domain" we mean any one of the following: program implementation, program specifications, and verification conditions. It was originally envisioned that it might be possible, e.g., for the user to "point" to part of a verification condition and have the user interface display that portion of source code giving rise to that verification condition, or to exhibit the effects of a tentative change in code in terms of changes in the VCs. In particular, we wanted to be able to display parts of the JOVIAL source code in bona fide JOVIAL syntax for such purposes rather than as internal, transduced forms. To achieve that objective would probably have required that we implement a rather sophisticated "unparser", although more mundane alternatives were also possible (at some cost in transparency and utility). In the end, we decided that none of these alternatives could be achieved with the resources available. Instead, we opted for providing a great deal of on-line documentation and guidance for the user, automating some of the file interactions, facilitating his passage among the different environments (editor, front end, and prover) and letting him suffer the

inconvenience of discovering the indirect relations among parts of the data base. While this was probably a reasonable compromise, the more ambitious goals of Moriconi's "A Designer/Verifier's Assistant" [40] still await realization.

4. JOVIAL-HDM Verification

The linkage of our verification tools to the SRI HDM approach, described in Sec. VIII-E, cannot be claimed to be complete. As discussed in that chapter, this is due in part to what may charitably be called "mismatches" between JOVIAL-J73 and the HDM specification language SPECIAL. Less charitably viewed, such nonfeatures as (1) the absence of a module "hiding" mechanism, the absence of structured support for exception handling, and (3) the complete lack of any data abstraction features are really shortcomings of JOVIAL. It should be noted that these problems have not arisen in connection with applications of HDM to other, more modern, high-level languages such as PASCAL.

Despite these innate difficulties we feel that our work on linking HDM to JOVIAL presents a reasonable approach to that task. It would possible with a modest additional effort to further "flesh out" the tools needed to provide a cleaner interface between JOVIAL and HDM. We believe that in future versions of the systems, the Shostak STP Theorem Prover would permit a more rational approach in that it incorporates its own type checking facilities and can handle quantified assertions directly. Unfortunately, STP became available too late in this project for use in the RJE verification system.

5. Numerical Analysis Aspects

Our work on this phase was concerned principally with the exploitation of an analogy (a mathematical homomorphism) between an original (JOVIAL) program P (with numeric variables, say, x1, x2,...) and a corresponding program P* which performs analogous operations on the 'error' quantities e1, e2,... associated with x1, x2,... it was discovered during the course of the last year's work that this homomorphism exists for the subclass of programs where the flow of control is not influenced by the presence of round-off, truncation, etc.

It was found possible to mechanically transform any program of this class (plus assertions about the variables x1,x2,..) into an asserted program P*

(including assertions about the error quantities e1, e2,...). This transformation, of course, makes use of an axiomatization of the machine arithmetic for numeric data.

We tested this idea successfully on a fairly complex, but thoroughly representative, JOVIAL program for computing the exponential function EXP(x), where x is a floating point (or fixed point) number. This test bed program made use of a version of the King-Floyd algorithm for the rapid computation of the exponential for the integer part of x, in combination with the exponential series (to a fixed number of terms) for the exponential of the fractional part of x.

On the negative side, not all of these ideas were incorporated into the RJE verifier. We did, however, manage to complete facilities for handling floating-point numeric constants in the verifier (i.e., by mapping them into lists of integers). Rational numbers (as well as the negative integers) were also modelled as theorem prover "shell" datatypes, so that deductions can be carried out in terms of these kinds of JOVIAL items. While no provision was completed for carrying out deductions about round-off or truncation, data items can be checked for representability in terms of maximum machine word sizes.

Clearly, there is much left to be done before it can be claimed that the semantics of real machine arithmetic has been treated in proper fashion. We still believe that this is thoroughly feasible, and we would encourage its support on the basis of our pilot effort.

APPENDICES

A. A GRAMMAR FOR JOVIAL-J73A

1. Grammar Tokens--Reserved Words and Pseudo-Terminals

A ABORT ASSERTKEY B BEGIN BIT BLOCK BY BYTE C CASE
COMPOOL CONSTANT DEF DEFAULT E ELSE END EXIT F FALLTHRU
FALSE FOR GOTO IF INEQOP INLINE INSTANCE ITEM LABEL LIKE
LISTOPT LOGOP LPARSTAR MOD NAMECOLON NMD NOT NULL
NUMBERNOTONETHRUFIVE ONETHRUFIVE OTHERLETTER P PARALLEL
PLUSMINUS POS POST PRE PROC PROGRAM R REC REF RENT REP
RETURN STARRPAR START STATIC STATUS STOP STRING SU
SYMBOL T TABLE TERM THEN TRUE TYPE V W WHILE

2. Pseudo-Terminals--TYPEOFLIST tokens

Pseudo-Terminal	Meanings
PLUSMINUS:	+ -
LPARSTAR:	(*
STARRPAR:	*)
SU:	S U
NMD:	N M D
OTHERLETTER:	G H I J K L O Q X Y Z
LOGOP:	AND OR XOR EQV
INEQOP:	< > <= >=
LISTOPT:	LISTEXP LISTINV LISTBOTH
ASSERTKEY:	ASSERT ASSUME PROVE

3. Precedence Rules

Assoc-type	Operator
nonassoc	NOT
left	AND OR EQV XOR
nonassoc	'<' '>' '=' '<>' '<=' '>='
left	'+' '-'
left	'*' '/' MOD
left	'**'

4. Modified BNF Grammar

```
/*  1*/ complete_program        : module
                                ¦ module complete_program ;

/*  3*/ abort_phraseq           :
                                ¦ ABORT name ;

/*  5*/ abort_statement         : ABORT ';' ;

/*  6*/ absolute_addressq       :
                                ¦ POS '(' numeric_formula ')'
                                    ':' ;

/*  8*/ actual_input_parametersc    : indicesc ;

/*  9*/ actual_input_parameterscq   :
                                    ¦ actual_input_parametersc ;

/* 11*/ actual_output_parametersc   : variable_list ;

/* 12*/ actual_parameter_list       : '('
                                      actual_input_parameterscq
                                     ':'
                                     actual_output_parametersc
                                     ')'
                                    ¦ '('
                                      actual_input_parameterscq
                                      ')' ;

/* 14*/ actual_parameter_listq      :
                                    ¦ actual_parameter_list ;

/* 16*/ allocation_specifierq       :
                                    ¦ STATIC ;

/* 18*/ assert_statement            : ASSERTKEY formula
                                      nameq ';' ;

/* 19*/ assertingq                  :
                                    ¦ ASSERTKEY formula
                                      nameq ';' ;

/* 21*/ assignment_statement        : variable_list '='
                                      formula ';' ;

/* 22*/ bit_conversion              : LPARSTAR B
                                      numeric_formula STARRPAR
                                    ¦ LPARSTAR B STARRPAR
                                    ¦ B
                                    ¦ REP ;

/* 26*/ bit_function_variable       : BIT '(' variable ','
                                      numeric_formula ','
                                      numeric_formula ')' ;
```

```
/* 27*/ bit_literal                : ONETHRUFIVE B STRING ;

/* 28*/ bit_primary                : bit_literal
                                   ¦ boolean_literal
                                   ¦ '(' boolean_formula ')'
                                   ¦ bit_conversion '('
                                       formula ')' ;

/* 32*/ block_body_options         : data_declaration
                                   ¦ null_declaration ;

/* 34*/ block_body_options_list    : block_body_options
                                   ¦ block_body_options
                                     block_body_options_list ;

/* 36*/ block_body_part            : null_declaration
                                   ¦ data_declaration
                                   ¦ BEGIN
                                       block_body_options_list
                                       END ;

/* 39*/ block_declaration          : BLOCK name
                                       allocation_specifierq ';'
                                       block_body_part
                                   ¦ BLOCK name
                                       allocation_specifierq
                                       name
                                       block_presetq ';' ;

/* 41*/ block_preset               : '=' block_preset_list ;

/* 42*/ block_preset_list          : block_preset_values_option
                                   ¦ block_preset_values_option
                                       ','
                                        block_preset_list ;

/* 44*/ block_preset_values_option : preset_values_option
                                   ¦ '(' table_preset_list ')'
                                   ¦ '(' block_preset_list ')' ;

/* 47*/ block_presetq              :
                                   ¦ block_preset ;

/* 49*/ block_type_declaration     : TYPE name BLOCK
                                       block_body_part ;


/* 50*/ boolean_formula            : logical_operand
                                   ¦ logical_operand LOGOP
                                       logical_operand
                                   ¦ NOT logical_operand ;
```

```
/* 53*/ boolean_literal          : TRUE
                                 | FALSE ;

/* 55*/ bound                    : numeric_formula
                                 | status_formula ;

/* 57*/ by_or_then_phrase        : BY numeric_formula
                                 | THEN formula ;

/* 59*/ by_or_then_phraseq       :
                                 | by_or_then_phrase ;

/* 61*/ byte_function_variable   : BYTE '(' variable ','
                                     numeric_formula ','
                                     numeric_formula ')' ;

/* 62*/ case_alternative         : case_index_group statement
                                     fallthruq ;

/* 63*/ case_alternatives        : case_alternative
                                 | case_alternative
                                     case_alternatives ;

/* 65*/ case_body                : default_option
                                     case_alternatives
                                 | case_alternatives ;

/* 67*/ case_index               : formula
                                 | bound ':' bound ;

/* 69*/ case_index_group         : '(' case_indicesc ')'
                                     ':' ;

/* 70*/ case_indicesc            : case_index
                                 | case_index ','
                                     case_indicesc ;

/* 72*/ case_statement           : CASE formula ';'
                                     BEGIN case_body
                                     labelsq END ;

/* 73*/ character_conversion     : LPARSTAR C numeric_formula
                                     STARRPAR
                                 | LPARSTAR C STARRPAR
                                 | C ;


/* 76*/ character_formula        : character_literal
                                 | '(' character_formula ')'
                                 | character_conversion
                                     '(' formula ')' ;

/* 79*/ character_literal        : STRING ;
```

```
/* 80*/ compool_declaration          : external_declaration
                                     ¦ constant_declaration
                                     ¦ type_declaration
                                     ¦ null_declaration
                                     ¦ BEGIN
                                       compool_declarations
                                       END ;

/* 85*/ compool_declarations         : compool_declaration
                                     ¦ compool_declaration
                                       compool_declarations ;

/* 87*/ compool_declarationsq        :
                                     ¦ compool_declarations ;

/* 89*/ compool_declared_name        : name
                                     ¦ '(' name ')' ;

/* 91*/ compool_declared_namesc      : compool_declared_name
                                     ¦ compool_declared_name ','
                                       compool_declared_namesc ;

/* 93*/ compool_module               : START COMPOOL name ';'
                                       compool_declarationsq
                                       TERM ;

/* 94*/ compound_declaration         : BEGIN declarations END ;

/* 95*/ compound_def                 : DEF BEGIN
                                       def_specification_choices
                                       END ;

/* 96*/ compound_ref                 : REF BEGIN
                                       ref_specification_choices
                                       END ;

/* 97*/ compound_statement           : BEGIN statements
                                       dollar_labelsq END ;

/* 98*/ constant_declaration         : CONSTANT ITEM name
                                       item_type_description
                                       item_preset ';'
                                     ¦ CONSTANT TABLE name
                                       dimension_listq
                                       table_description ;

/*100*/ constant_index               : numeric_formula
                                     ¦ status_formula ;

/*102*/ constant_indicesc            : constant_index
                                     ¦ constant_index ','
                                       constant_indicesc ;

/*104*/ continuation                 : by_or_then_phrase
```

```
                                    while_phraseq
                                  | while_phrase
                                    by_or_then_phraseq ;

/*106*/ control_clause             : formula
                                  | formula continuation ;

/*108*/ control_item               : name
                                  | letter ;

/*110*/ data_declaration           : item_declaration
                                  | table_declaration
                                  | constant_declaration
                                  | block_declaration ;

/*114*/ declaration                : declaration_not_null
                                  | null_declaration ;

/*116*/ declaration_not_null : declaration_not_null_not_compound
                               | compound_declaration ;

/*118*/ declaration_not_null_not_compound
                                  : data_declaration
                                  | type_declaration
                                  | subroutine_defn_or_decl
                                  | statement_name_declaration
                                  | external_declaration
                                  | inline_declaration ;

/*124*/ declarations               : declaration
                                  | declaration declarations ;

/*126*/ declarationsq              :
                                  | declarations ;

/*128*/ def_block_instantiation    : BLOCK INSTANCE name ';' ;

/*129*/ def_specification          : simple_def
                                  | compound_def ;


/*131*/ def_specification_choice   : null_declaration
                                  | data_declaration
                                  | def_block_instantiation
                                  | statement_name_declaration
                                   ;

/*135*/ def_specification_choices  : def_specification_choice
                                  | def_specification_choice
                                    def_specification_choices
                                   ;

/*137*/ default_option             : '(' DEFAULT ')' ':'
                                      statement fallthruq ;
```

```
/*138*/ default_preset_sublistcq    :
                                    | block_preset_list ',' ;

/*140*/ default_sublist             : status_constantsc ;

/*141*/ default_sublistcq           :
                                    | default_sublist ',' ;

/*143*/ dereference                 : '@' name
                                    | '@' '(' pointer_formula ')'
                                    | '@' '(' numeric_primary ')'
                                    ;

/*146*/ dimension                   : bound
                                    | bound ':' bound
                                    | '*' ;

/*149*/ dimension_list              : '(' dimensionsc ')' ;

/*150*/ dimension_listq             :
                                    | dimension_list ;

/*152*/ dimensionsc                 : dimension
                                    | dimension ',' dimensionsc ;

/*154*/ dollar_labelsq              :
                                    | '$' labels ;

/*156*/ else_clause                 : ELSE statement ;

/*157*/ entry_specifier             : ordinary_entry_specifier
                                    | specified_entry_specifier ;

/*159*/ equal_or_not_equal_operator : '='
                                    | '<>' ;

/*161*/ exit_statement              : EXIT ';' ;

/*162*/ extended_variable           : variable
                                    | function_call
                                    | LPARSTAR name STARRPAR
                                      '(' formula ')' ;

/*165*/ external_declaration        : def_specification
                                    | ref_specification ;

/*167*/ fallthruq                   :
                                    | FALLTHRU ;

/*169*/ fixed_conversion            : LPARSTAR A rtq
                                      numeric_formula
                                      fraction_specifierq
                                      STARRPAR ;
```

```
/*170*/ floating_conversion        : LPARSTAR F rt
                                     numeric_formula
                                     STARRPAR
                                   | LPARSTAR F numeric_formula
                                     STARRPAR
                                   | LPARSTAR F rt STARRPAR
                                   | LPARSTAR F STARRPAR
                                   | F ;

/*175*/ formal_parameter_list      : '(' namescq ':' namesc ')'
                                   | '(' namesc ')' ;

/*177*/ formal_parameter_listq     :
                                   | formal_parameter_list ;

/*179*/ formula                    : boolean_formula
                                   | character_formula
                                   | status_formula
                                   | pointer_formula ;

/*183*/ fraction_specifierq        :
                                   | ',' numeric_formula ;

/*185*/ function_call              : function_name
                                     actual_parameter_list ;

/*186*/ function_name              : name ;

/*187*/ goto_statement             : GOTO name ';' ;

/*188*/ if_statement               : IF boolean_formula ';'
                                     statement else_clause
                                   | IF boolean_formula ';'
                                     statement ;

/*190*/ index                      : formula ;

/*191*/ indicesc                   : index
                                   | index ',' indicesc ;

/*193*/ inline_declaration         : INLINE namesc ';' ;

/*194*/ integer_conversion         : LPARSTAR SU rt
                                     numeric_formula STARRPAR
                                   | LPARSTAR SU rt STARRPAR
                                   | LPARSTAR SU numeric_formula
                                     STARRPAR
                                   | LPARSTAR SU STARRPAR
                                   | SU ;

/*199*/ item_declaration           : ITEM name
                                     allocation_specifierq
                                     item_type_description
```

```
                                    item_presetq ';' ;

/*200*/ item_preset                 : '=' formula ;

/*201*/ item_presetq                :
                                    | item_preset ;

/*203*/ item_type_declaration       : TYPE name
                                      item_type_description
                                      ';' ;

/*204*/ item_type_description       : name
                                    | SU rt numeric_formula
                                    | SU rt
                                    | SU numeric_formula
                                    | SU
                                    | F rt numeric_formulaq
                                    | F numeric_formulaq
                                    | A rtq numeric_formula
                                       fraction_specifierq
                                    | B numeric_formulaq
                                    | C numeric_formulaq
                                    | STATUS numeric_formulaq
                                      '(' status_list ')'
                                    | pointer_conversion ;

/*216*/ label                       : NAMECOLON ;

/*217*/ labels                      : labels label
                                    | label ;

/*219*/ labelsq                     :
                                    | labels ;

/*221*/ letter                      : A
                                    | B
                                    | C
                                    | E
                                    | F
                                    | P
                                    | R
                                    | T
                                    | V
                                    | W
                                    | SU
                                    | NMD
                                    | OTHERLETTER ;

/*234*/ like_optionq                :
                                    | LIKE name ;

/*236*/ list_optionq                :
                                    | LISTOPT ;
```

```
/*238*/ location_specifier          : starting_bit ','
                                      numeric_formula ;

/*239*/ logical_operand             : bit_primary
                                    | relational_expression ;

/*241*/ loop_statement              : loop_type statement ;

/*242*/ loop_type                   : WHILE assertingq
                                      boolean_formula ';'
                                    | FOR control_item ':'
                                      assertingq
                                      control_clause ';' ;

/*244*/ main_program_module  : START PROGRAM name ';'
                                 subroutine_or_program_body
                                 non_nested_subroutines TERM
                               | START PROGRAM name ';'
                                 subroutine_or_program_body
                                 TERM ;

/*246*/ module                      : compool_module
                                    | procedure_module
                                    | main_program_module ;

/*249*/ multiply_divide_or_mod      : multiply_or_divide
                                    | MOD ;

/*251*/ multiply_or_divide          : '*'
                                    | '/' ;

/*253*/ name                        : SYMBOL ;

/*254*/ named_variable              : name subscript
                                    | name dereference
                                    | name subscript dereference
                                    | dereference subscript
                                    | name
                                    | dereference
                                    | OTHERLETTER ;

/*261*/ nameq                       :
                                    | name ;

/*263*/ names                       : name
                                    | name names ;

/*265*/ namesc                      : name
                                    | name ',' namesc ;

/*267*/ namescq                     :
                                    | namesc ;

/*269*/ non_nested_subroutine       : subroutine_defn_or_decl ;
```

```
/*270*/ non_nested_subroutines      : non_nested_subroutine
                                    | non_nested_subroutine
                                        non_nested_subroutines ;

/*272*/ null_declaration            : ';' ;

/*273*/ null_statement              : ';' ;

/*274*/ numeric_factor              : numeric_primary
                                    | numeric_factor '**'
                                        numeric_primary ;

/*276*/ numeric_formula             : numeric_term
                                    | PLUSMINUS numeric_term
                                    | numeric_formula PLUSMINUS
                                        numeric_term ;

/*279*/ numeric_formulaq            :
                                    | numeric_formula ;

/*281*/ numeric_primary             : NUMBERNOTONETHRUFIVE
                                    | ONETHRUFIVE
                                    | extended_variable
                                    | '(' numeric_formula ')'
                                    | integer_conversion '('
                                        numeric_formula ')'
                                    | floating_conversion '('
                                        numeric_formula ')'
                                    | fixed_conversion '('
                                        numeric_term ')' ;

/*288*/ numeric_term                : numeric_factor
                                    | numeric_term
                                      multiply_divide_or_mod
                                      numeric_factor ;

/*290*/ ordinary_entry_specifier    : packing_specifierq
                                      item_type_description
                                      table_presetq ';'
                                    | packing_specifierq
                                      table_presetq ';'
                                      ordinary_table_body ;

/*292*/ ordinary_table_body  : ordinary_table_item_declaration
                             | BEGIN ordinary_table_options_list
                                 END ;

/*294*/ ordinary_table_item_declaration
                             : ITEM name item_type_description
                                 packing_specifierq
                                 table_presetq ';' ;

/*295*/ ordinary_table_options
```

```
                              : ordinary_table_item_declaration
                              | null_declaration ;

/*297*/ ordinary_table_options_list
                              : ordinary_table_options
                              | ordinary_table_options
                                  ordinary_table_options_list ;

/*299*/ packing_specifierq      :
                              | NMD ;

/*301*/ pointer_conversion      : LPARSTAR P nameq STARRPAR
                              | P nameq ;

/*303*/ pointer_formula         : NULL
                              | '(' pointer_formula ')'
                              | pointer_conversion
                                  '(' formula ')' ;

/*306*/ postcondition           : POST formula ;

/*307*/ postconditionq          :
                              | postcondition ;

/*309*/ precondition            : PRE formula ;

/*310*/ preconditionq           :
                              | precondition ;

/*312*/ preset_index_specifier  : POS '('
                                  constant_indicesc
                                  ')' ':' ;

/*313*/ preset_values_option    :
                              | formula
                              | numeric_formula '('
                                  block_preset_list ')' ;

/*316*/ procedure_call_statement : name actual_parameter_listq
                                  abort_phraseq ';' ;

/*317*/ procedure_module        : START declarationsq TERM ;

/*318*/ procedure_or_function_defn_or_decl
                          : procedure_or_function_heading ';'
                              declaration
                          | procedure_or_function_heading ';'
                              subroutine_or_program_body ;

/*320*/ procedure_or_function_heading
                          : PROC name subroutine_attributeq
                              formal_parameter_listq
                              item_type_description
                              preconditionq
```

```
/*270*/ non_nested_subroutines      : non_nested_subroutine
                                    | non_nested_subroutine
                                        non_nested_subroutines ;

/*272*/ null_declaration            : ';' ;

/*273*/ null_statement              : ';' ;

/*274*/ numeric_factor              : numeric_primary
                                    | numeric_factor '**'
                                      numeric_primary ;

/*276*/ numeric_formula             : numeric_term
                                    | PLUSMINUS numeric_term
                                    | numeric_formula PLUSMINUS
                                      numeric_term ;

/*279*/ numeric_formulaq            :
                                    | numeric_formula ;

/*281*/ numeric_primary             : NUMBERNOTONETHRUFIVE
                                    | ONETHRUFIVE
                                    | extended_variable
                                    | '(' numeric_formula ')'
                                    | integer_conversion '('
                                        numeric_formula ')'
                                    | floating_conversion '('
                                        numeric_formula ')'
                                    | fixed_conversion '('
                                        numeric_term ')' ;

/*288*/ numeric_term                : numeric_factor
                                    | numeric_term
                                      multiply_divide_or_mod
                                      numeric_factor ;

/*290*/ ordinary_entry_specifier    : packing_specifierq
                                      item_type_description
                                      table_presetq ';'
                                    | packing_specifierq
                                      table_presetq ';'
                                      ordinary_table_body ;

/*292*/ ordinary_table_body  : ordinary_table_item_declaration
                             | BEGIN ordinary_table_options_list
                                 END ;

/*294*/ ordinary_table_item_declaration
                             : ITEM name item_type_description
                                 packing_specifierq
                                 table_presetq ';' ;

/*295*/ ordinary_table_options
```

```
                              : ordinary_table_item_declaration
                              | null_declaration ;

/*297*/ ordinary_table_options_list
                              : ordinary_table_options
                              | ordinary_table_options
                                  ordinary_table_options_list ;

/*299*/ packing_specifierq      :
                              | NMD ;

/*301*/ pointer_conversion      : LPARSTAR P nameq STARRPAR
                              | P nameq ;

/*303*/ pointer_formula         : NULL
                              | '(' pointer_formula ')'
                              | pointer_conversion
                                  '(' formula ')' ;

/*306*/ postcondition           : POST formula ;

/*307*/ postconditionq          :
                              | postcondition ;

/*309*/ precondition            : PRE formula ;

/*310*/ preconditionq           :
                              | precondition ;

/*312*/ preset_index_specifier  : POS '('
                                  constant_indicesc
                                  ')' ':' ;

/*313*/ preset_values_option    :
                              | formula
                              | numeric_formula '('
                                  block_preset_list ')' ;

/*316*/ procedure_call_statement : name actual_parameter_listq
                                  abort_phraseq ';' ;

/*317*/ procedure_module        : START declarationsq TERM ;

/*318*/ procedure_or_function_defn_or_decl
                              : procedure_or_function_heading ';'
                                  declaration
                              | procedure_or_function_heading ';'
                                  subroutine_or_program_body ;

/*320*/ procedure_or_function_heading
                              : PROC name subroutine_attributeq
                                  formal_parameter_listq
                                  item_type_description
                                  preconditionq
```

```
                                    postconditionq
                                | PROC name subroutine_attributeq
                                    formal_parameter_listq
                                    preconditionq
                                    postconditionq ;

/*322*/ ref_specification           : simple_ref
                                    | compound_ref ;

/*324*/ ref_specification_choice    : null_declaration
                                    | data_declaration
                                    | subroutine_defn_or_decl ;

/*327*/ ref_specification_choices
                                  : ref_specification_choice
                                  | ref_specification_choice
                                      ref_specification_choices ;

/*329*/ relational_expression
                                  : numeric_formula
                                      relational_operator
                                      numeric_formula
                                  | character_formula
                                      relational_operator
                                      character_formula
                                  | status_formula
                                      relational_operator
                                      status_formula
                                  | bit_primary
                                      equal_or_not_equal_operator
                                      bit_primary
                                  | pointer_formula
                                      relational_operator
                                      pointer_formula
                                  | numeric_formula ;

/*335*/ relational_operator         : equal_or_not_equal_operator
                                    | INEQOP ;

/*337*/ rep_function_variable       : REP '('
                                        named_variable ')' ;

/*338*/ return_statement            : RETURN ';' ;

/*339*/ rt                          : ',' R
                                    | ',' T ;

/*341*/ rtq                         :
                                    | rt ;

/*343*/ simple_def                  : DEF
                                      def_specification_choice ;

/*344*/ simple_ref                  : REF
```

```
                                    ref_specification_choice ;

/*345*/ simple_statement            : assignment_statement
                                    | loop_statement
                                    | if_statement
                                    | case_statement
                                    | procedure_call_statement
                                    | return_statement
                                    | goto_statement
                                    | exit_statement
                                    | stop_statement
                                    | abort_statement
                                    | null_statement
                                    | assert_statement ;

/*357*/ specified_entry_specifier
                                  : words_per_entry
                                      specified_item_description
                                      table_presetq ';'
                                  | words_per_entry table_presetq
                                      ';' specified_table_body ;

/*359*/ specified_item_description
                                  : item_type_description POS
                                      '(' location_specifier ')' ;

/*360*/ specified_preset_sublist    : preset_index_specifier
                                        block_preset_list ;

/*361*/ specified_preset_sublistsc : specified_preset_sublist
                                   | specified_preset_sublist
                                     ','
                                     specified_preset_sublistsc
                                   ;

/*363*/ specified_sublist           : numeric_formula
                                      status_constantsc ;

/*364*/ specified_sublistsc         : specified_sublist
                                    | specified_sublist ','
                                      specified_sublistsc ;

/*366*/ specified_table_body  : specified_table_item_declaration
                              | BEGIN
                                specified_table_options_list
                                END ;

/*368*/ specified_table_item_declaration
                                  : ITEM name
                                    specified_item_description
                                    table_presetq ';' ;

/*369*/ specified_table_options
                                : specified_table_item_declaration
```

```
                                | null_declaration ;

/*371*/ specified_table_options_list
                                : specified_table_options
                                | specified_table_options
                                    specified_table_options_list
                                ;

/*373*/ starting_bit            : numeric_formula
                                | '*' ;

/*375*/ statement               : labelsq simple_statement
                                | compound_statement ;

/*377*/ statement_name_declaration  : LABEL namesc ';' ;

/*378*/ statements              : statement
                                | statement statements ;

/*380*/ status                  : name
                                | letter ;

/*382*/ status_constant         : V '(' status ')' ;

/*383*/ status_constantsc       : status_constant
                                | status_constant ','
                                    status_constantsc ;

/*385*/ status_conversion       : LPARSTAR STATUS
                                    numeric_formulaq '('
                                    status_list ')'
                                    STARRPAR ;

/*386*/ status_formula          : status_constant
                                | '(' status_formula ')'
                                | status_conversion
                                    '(' formula ')' ;

/*389*/ status_list             : default_sublist
                                | default_sublistcq
                                    specified_sublistsc ;

/*391*/ stop_statement          : STOP numeric_formulaq ';' ;

/*392*/ structure_specifier     : PARALLEL
                                | T numeric_formulaq ;

/*394*/ structure_specifierq    :
                                | structure_specifier ;

/*396*/ subroutine_attributeq   :
                                | REC
                                | RENT ;
```

```
/*399*/ subroutine_defn_or_decl
                          : procedure_or_function_defn_or_decl ;

/*400*/ subroutine_defn_or_decls    : subroutine_defn_or_decl
                                    | subroutine_defn_or_decl
                                      subroutine_defn_or_decls ;

/*402*/ subroutine_defn_or_declsq   :
                                    | subroutine_defn_or_decls ;

/*404*/ subroutine_or_program_body  : labelsq simple_statement
                                    | BEGIN
                                      declarationsq statements
                                      subroutine_defn_or_declsq
                                      dollar_labelsq END ;

/*406*/ subscript                   : '(' indicesc ')' ;

/*407*/ table_declaration           : TABLE name
                                        allocation_specifierq
                                        dimension_listq
                                        table_description ;

/*408*/ table_description           : structure_specifierq
                                        entry_specifier
                                    | name table_presetq ';' ;

/*410*/ table_preset                : '=' table_preset_list ;

/*411*/ table_preset_list       : block_preset_list
                                | default_preset_sublistcq
                                    specified_preset_sublistsc ;

/*413*/ table_presetq               :
                                    | table_preset ;

/*415*/ table_type_declaration      : TYPE name TABLE
                                      table_type_specifier ;

/*416*/ table_type_specifier        : dimension_listq
                                      structure_specifierq
                                      like_optionq
                                      entry_specifier
                                    | dimension_listq name ';' ;

/*418*/ type_declaration            : item_type_declaration
                                    | table_type_declaration
                                    | block_type_declaration ;

/*421*/ variable                    : named_variable
                                    | bit_function_variable
                                    | byte_function_variable
                                    | rep_function_variable ;
```

```
/*425*/ variable_list          : variable
                               | variable ','
                                   variable_list ;

/*427*/ while_phrase           : WHILE boolean_formula ;

/*428*/ while_phraseq          :
                               | while_phrase ;

/*430*/ words_per_entry        : W numeric_formulaq
                               | V ;
```

B. VERIFICATION OF AN INTEGER ARITHMETIC PROGRAM

1. The GCD Program

The JOVIAL program shown below computes the greatest common divisor GCD(X0,Y0) of two unsigned integers X0 and Y0 according to an iterative version of the following recursion:

```
GCD(X0, Y0) = if Y0=0 then X0
                else if X0>Y0 then GCD(X0-Y0, Y0)
                else GCD(X0, Y0-X0).
```

The text of the JOVIAL program verified here is as follows:

```
START PROGRAM TstGCD;
BEGIN ITEM X0 U; ITEM X1 U; ITEM Y0 U;
        ITEM Y1 U; ITEM TEMP U;
        % ASSERT (X0>0) AND
                 ((Y0>0) AND NUMBERP4(X0,X1,Y0,Y1));%
        X1=X0; Y1=Y0;
        WHILE    %ASSERT (GCD(X1,Y1)=GCD(X0,Y0)) AND
                          NUMBERP4(X0,X1,Y0,Y1);%
            Y1>0 ;
            IF X1>Y1;
                BEGIN TEMP=Y1; Y1=X1-Y1; X1=TEMP; END
            ELSE Y1=Y1-X1;
        %ASSERT X1=GCD(X0,Y0);%
  END
TERM
```

The reader may observe that we have deliberately reversed the X and Y arguments in what corresponds to the first recursive call to make the verification somewhat more interesting. The specification of the program's intent is in terms of a function GCD which will be defined to the theorem prover by means of a recursive definition like the informal one shown above, and we did not wish to have the program be an exact iterative counterpart to this recursive definition. Of course, the GCD function has the property that GCD(X,Y) = GCD(Y,X), but this fact must be proved before it can be used. The proof of correctness given below for the program will both prove this symmetry property and use it as a lemma in the program's verification.

Note also that the three embedded assertions (enclosed in %-signs) are expressed in terms of another function, NUMBERP4, of four arguments. This predicate function, like GCD, is not declared as part of the JOVIAL program, but it is in the verifier's assertion language. These two functions will be defined to the theorem prover in the course of the verification shown below.

2. Verification of the Program

We show only the theorem prover's portion of the verification since the front end processing (parsing and generation of VCs) is not particularly interesting. Prior to entering the theorem prover environment we had generated VCs and written them out to a file called JGCD.PRS-VCS. That file, as well as a previously written events file are loaded automatically during the proof shown in the following transcript.

(* Beginning of transcript)

```
45 STARTUP
This is the Automatic Theorem Prover for Recursive Functions
You are in user mode: (MANUAL CONCISE)
Do you want to change your mode of interaction with the system?
NO

The last J73 program worked on was: <JOVIAL>JGCD.;20
Shall we continue with this source file?
Yes, continue
 [confirm]

You are in MANUALMODE; do you wish to proceed in AUTOMATIC?
NO

Should I load the VC file: <JOVIAL>JGCD.PRS-VCS;2? YES

Loading previously computed VCs from file: <JOVIAL>JGCD.PRS-VCS;2
The number of separate VCs is: 2
Should I load the EVENTS.FILE: <JOVIAL>JGCD.EVENTS;1? Y

FILE CREATED 10-Dec-81 00:26:50
JGCDCOMS
What should be done if a proof fails?  Continue
("SRI-CSL Tenex 1.34.41, SRI-CSL" <LISP>LISP.SAV;133
<BOYER>BASIS.;3 <BOYER>CODE.;2 <BOYER>CODE1.;787
<BOYER>DATA.;2 <BOYER>DATA1.;7 <BOYER>SIMPBLOCK.;2)
6/2/82 12:50-PST


_DEFN(NUMBERP4 (X Y Z W)
               (AND (NUMBERP X)
                    (NUMBERP Y)
                    (NUMBERP Z)
                    (NUMBERP W)))

     Observe that (OR (FALSEP (NUMBERP4 X Y Z W))
                      (TRUEP (NUMBERP4 X Y Z W)))
is a theorem.

[1335 cns / 4.7 s + 0.0 gc + .5 io (= 9  1)]
NUMBERP4
```

```
_DEFN(GCD (X Y)
          (IF (ZEROP X)
              (FIX Y)
              (IF (ZEROP Y)
                  X
                  (IF (LESSP X Y)
                      (GCD X (DIFFERENCE Y X))
                      (GCD (DIFFERENCE X Y) Y))))
          (* This defines GCD to the Theorem Prover; note that
           this definition applies even when X or Y are not
           integers, namely GCD(nonint, int) = int,
           GCD(int, nonint) = int, and GCD(nonint1, nonint2) =
           0, since nonints are treated as zeros by FIX and
           ZEROP))
```

The lemma RECURSION.BY.DIFFERENCE informs us that:
(CONS (COUNT X) (COUNT Y))
goes down according to the well-founded lexicographic order induced by LESSP and LESSP in each recursive call. Hence, GCD is accepted under the principle of definition. Observe that (NUMBERP (GCD X Y)) is a theorem.

```
[5672 cns / 14.6 s + 15.1 gc + 1.6 io (= 72  1)]
GCD


_PROVE.LEMMA(GCD.IS.SYM (REWRITE)
                        (EQUAL (GCD X Y) (GCD Y X)))
Do you want to redo this event? Yes

        (* Comment: This is the symmetry property which was
         mentioned above; it will be used as a lemma in the
         main proof below)
```

Name the conjecture *1.

We will try to prove it by induction. The recursive terms in the conjecture suggest two inductions. However, they merge into one likely candidate induction. We will induct according to the following scheme:

```
(AND (IMPLIES (NOT (NUMBERP X)) (p X Y))
     (IMPLIES (NOT (NUMBERP Y)) (p X Y))
     (IMPLIES (EQUAL X 0) (p X Y))
     (IMPLIES (EQUAL Y 0) (p X Y))
     (IMPLIES (AND (NUMBERP X)
                   (NUMBERP Y)
                   (NOT (EQUAL X 0))
                   (NOT (EQUAL Y 0))
                   (p (DIFFERENCE X Y) Y)
                   (p X (DIFFERENCE Y X)))
              (p X Y))).
```

```
The inequality RECURSION.BY.DIFFERENCE establishes that the
measure:       (CONS (COUNT X) (COUNT Y))
decreases according to the well-founded lexicographic order
induced by LESSP and LESSP in the induction step of the scheme.
The above induction scheme generates five new goals:
Case 5. (IMPLIES (NOT (NUMBERP X))
                 (EQUAL (GCD X Y) (GCD Y X))),

  which we simplify, expanding the function GCD, to the following
  two new conjectures:

  Case 5.2.
         (IMPLIES (AND (NOT (NUMBERP X))
                       (NOT (EQUAL Y 0))
                       (NUMBERP Y))
                  (EQUAL Y Y)),

    which we again simplify, using linear arithmetic, to:

         T. (* Identity)

  Case 5.1.
         (IMPLIES (AND (NOT (NUMBERP X))
                       (NOT (NUMBERP Y)))
                  (EQUAL 0 0)).

    This again simplifies, using linear arithmetic, to:

         T. (* Identity)

Case 4. (IMPLIES (NOT (NUMBERP Y))
                 (EQUAL (GCD X Y) (GCD Y X))),

  which we simplify, unfolding GCD, to two new goals:



  Case 4.2.
         (IMPLIES (AND (NOT (NUMBERP Y))
                       (NOT (NUMBERP X)))
                  (EQUAL 0 0)).

    This again simplifies, using linear arithmetic, to:

         T. (* Identity)

  Case 4.1.
         (IMPLIES (AND (NOT (NUMBERP Y))
                       (NUMBERP X)
                       (NOT (EQUAL X 0)))
                  (EQUAL X X)).

    But this again simplifies, using linear arithmetic, to:
```

```
        T. (* Identity)

Case 3. (IMPLIES (EQUAL X 0)
                 (EQUAL (GCD X Y) (GCD Y X))),
```

which simplifies, unfolding the definitions of EQUAL, GCD, and NUMBERP, to the following two new goals:

```
Case 3.2.
        (IMPLIES (AND (NOT (EQUAL Y 0)) (NUMBERP Y))
                 (EQUAL Y Y)),
```

which we again simplify, using linear arithmetic, to:

```
        T. (* Identity)

Case 3.1.
        (IMPLIES (NOT (NUMBERP Y))
                 (EQUAL 0 0)).
```

This simplifies again, using linear arithmetic, to:

```
        T. (* Identity)

Case 2. (IMPLIES (EQUAL Y 0)
                 (EQUAL (GCD X Y) (GCD Y X))).
```

This simplifies, expanding EQUAL, NUMBERP, and GCD, to three new formulas:

```
Case 2.3.
        (IMPLIES (NOT (NUMBERP X))
                 (EQUAL 0 0)).
```

But this simplifies again, using linear arithmetic, to:

```
        T. (* Identity)

Case 2.2.
        (IMPLIES (AND (NUMBERP X) (NOT (EQUAL X 0)))
                 (EQUAL X X)).
```

But this again simplifies, using linear arithmetic, to:

```
        T. (* Identity)

Case 2.1.
        (IMPLIES (AND (NUMBERP X) (EQUAL X 0))
                 (EQUAL 0 X)).
```

This again simplifies, using linear arithmetic, to:

```
        T. (* Symmetry of equality)
```

```
Case 1. (IMPLIES (AND (NUMBERP X)
                      (NUMBERP Y)
                      (NOT (EQUAL X 0))
                      (NOT (EQUAL Y 0))
                      (EQUAL (GCD (DIFFERENCE X Y) Y)
                             (GCD Y (DIFFERENCE X Y)))
                      (EQUAL (GCD X (DIFFERENCE Y X))
                             (GCD (DIFFERENCE Y X) X)))
                 (EQUAL (GCD X Y) (GCD Y X))).
```

This simplifies, applying DIFFERENCE.0 and EQUAL.DIFFERENCE, and expanding the functions GCD and EQUAL, to the following two new conjectures:

Case 1.2.

```
        (IMPLIES (AND (NUMBERP X)
                      (NUMBERP Y)
                      (NOT (EQUAL X 0))
                      (NOT (EQUAL Y 0))
                      (NOT (LESSP Y X))
                      (NOT (LESSP X Y)))
                 (EQUAL (GCD (DIFFERENCE X Y) Y)
                        (GCD (DIFFERENCE Y X) X))).
```

However this again simplifies, using linear arithmetic, to:

```
        (IMPLIES (AND (NUMBERP X)
                      (NUMBERP X)
                      (NOT (EQUAL X 0))
                      (NOT (EQUAL X 0))
                      (NOT (LESSP X X))
                      (NOT (LESSP X X)))
                 (EQUAL (GCD (DIFFERENCE X X) X)
                        (GCD (DIFFERENCE X X) X))),
```

which again simplifies, using linear arithmetic, to:

```
        T. (* Identity)
```

Case 1.1.

```
        (IMPLIES (AND (NUMBERP X)
                      (NUMBERP Y)
                      (NOT (EQUAL X 0))
                      (NOT (EQUAL Y 0))
                      (LESSP Y X)
                      (LESSP X Y))
                 (EQUAL (GCD X (DIFFERENCE Y X))
                        (GCD Y (DIFFERENCE X Y)))),
```

which again simplifies, using linear arithmetic, to:

```
        T. (* Contradictory LESSP hypotheses)
```

That finishes the proof of *1. Q.E.D.

```
[10416 cns / 29.5 s + 14.4 gc + 20.6 io (= 100  1)]
GCD.IS.SYM

        (* 48.8 cpu-secs were spent in deduction for
           establishing the above definitions and lemma)

Do you wish to examine the VCs?  YES
JGCD.1

  (IMPLIES (AND (GREATERP X0 0)
                (AND (GREATERP Y0 0)
                     (NUMBERP4 X0 X1 Y0 Y1)))
           (AND (EQUAL (GCD X0 Y0)
                       (GCD X0 Y0))
                (NUMBERP4 X0 X0 Y0 Y0)))

Type OK to continue, or NO to interrupt. OK
JGCD.2

  (AND (IMPLIES (AND (AND (EQUAL (GCD X1 Y1)
                                  (GCD X0 Y0))
                          (NUMBERP4 X0 X1 Y0 Y1))
                     (GREATERP Y1 0))
                (AND (IMPLIES (GREATERP X1 Y1)
                              (AND (EQUAL (GCD Y1
                                               (DIFFERENCE X1 Y1))
                                          (GCD X0 Y0))
                                   (NUMBERP4 X0 Y1 Y0
                                             (DIFFERENCE X1 Y1))))
                     (IMPLIES (NOT (GREATERP X1 Y1))
                              (AND (EQUAL (GCD X1
                                               (DIFFERENCE Y1 X1))
                                          (GCD X0 Y0))
                                   (NUMBERP4
                                      X0 X1 Y0
                                      (DIFFERENCE Y1 X1))))))
       (IMPLIES (AND (AND (EQUAL (GCD X1 Y1)
                                 (GCD X0 Y0))
                          (NUMBERP4 X0 X1 Y0 Y1))
                     (NOT (GREATERP Y1 0)))
                (EQUAL X1 (GCD X0 Y0))))
Type OK to continue, or NO to interrupt. OK
What should be done if a proof fails?  Quit
("SRI-CSL Tenex 1.34.41, SRI-CSL" <LISP>LISP.SAV;133
<BOYER>BASIS.;3 <BOYER>CODE.;2 <BOYER>CODE1.;787
<BOYER>DATA.;2 <BOYER>DATA1.;7 <BOYER>SIMPBLOCK.;2)
6/2/82 12:54-PST

Starting proofs of VCs in PROVE mode...

Attempting the VC:
JGCD.1
```

```
(IMPLIES (AND (GREATERP X0 0)
              (AND (GREATERP Y0 0)
                   (NUMBERP4 X0 X1 Y0 Y1)))
         (AND (EQUAL (GCD X0 Y0) (GCD X0 Y0))
              (NUMBERP4 X0 X0 Y0 Y0)))
```

This formula can be simplified, using the abbreviations NUMBERP4, AND, IMPLIES, and GREATERP, to the goal:

```
      (IMPLIES (AND (LESSP 0 X0)
                    (LESSP 0 Y0)
                    (NUMBERP X0)
                    (NUMBERP X1)
                    (NUMBERP Y0)
                    (NUMBERP Y1))
               (AND (EQUAL (GCD X0 Y0) (GCD X0 Y0))
                    (NUMBERP4 X0 X0 Y0 Y0))).
```

This simplifies, opening up the functions NUMBERP4 and AND, to:

```
      T. (* Obvious!)
```

Q.E.D.

Attempting the VC:
JGCD.2

```
(AND (IMPLIES (AND (AND (EQUAL (GCD X1 Y1) (GCD X0 Y0))
                        (NUMBERP4 X0 X1 Y0 Y1))
                   (GREATERP Y1 0))
              (AND (IMPLIES (GREATERP X1 Y1)
                            (AND (EQUAL
                                   (GCD Y1
                                        (DIFFERENCE X1 Y1))
                                   (GCD X0 Y0))
                                 (NUMBERP4
                                   X0 Y1 Y0
                                   (DIFFERENCE X1 Y1))))
                   (IMPLIES (NOT (GREATERP X1 Y1))
                            (AND (EQUAL
                                   (GCD X1
                                        (DIFFERENCE Y1 X1))
                                   (GCD X0 Y0))
                                 (NUMBERP4 X0 X1 Y0
                                   (DIFFERENCE Y1 X1))))))
     (IMPLIES (AND (AND (EQUAL (GCD X1 Y1) (GCD X0 Y0))
                        (NUMBERP4 X0 X1 Y0 Y1))
                   (NOT (GREATERP Y1 0)))
              (EQUAL X1 (GCD X0 Y0))))
```

This conjecture can be simplified, using the abbreviations NOT, NUMBERP4, IMPLIES, AND, and GREATERP, to two new goals:

```
Case 2. (IMPLIES (AND (EQUAL (GCD X1 Y1) (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (NUMBERP Y1)
                      (LESSP 0 Y1))
                 (AND (IMPLIES (LESSP Y1 X1)
                               (AND (EQUAL
                                      (GCD Y1
                                           (DIFFERENCE X1 Y1))
                                      (GCD X0 Y0))
                                    (NUMBERP4 X0 Y1 Y0
                                      (DIFFERENCE X1 Y1))))
                      (IMPLIES (NOT (LESSP Y1 X1))
                               (AND (EQUAL
                                      (GCD X1
                                           (DIFFERENCE Y1 X1))
                                      (GCD X0 Y0))
                                    (NUMBERP4 X0 X1 Y0
                                      (DIFFERENCE Y1 X1)))))).
```

This simplifies, rewriting with DIFFERENCE.0, and expanding the functions NUMBERP4, AND, IMPLIES, NOT, EQUAL, LESSP, and GCD, to five new conjectures:

Case 2.5.

```
        (IMPLIES (AND (NOT (EQUAL X1 0))
                      (LESSP X1 Y1)
                      (EQUAL (GCD X1 (DIFFERENCE Y1 X1))
                             (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (NUMBERP Y1)
                      (NOT (EQUAL Y1 0))
                      (LESSP Y1 X1))
                 (EQUAL (GCD Y1 (DIFFERENCE X1 Y1))
                        (GCD X0 Y0))),
```

which again simplifies, using linear arithmetic, to:

```
        T. (* Contains contradictory LESSP hypotheses)
```

Case 2.4.

```
        (IMPLIES (AND (NOT (EQUAL X1 0))
                      (NOT (LESSP X1 Y1))
                      (EQUAL (GCD (DIFFERENCE X1 Y1) Y1)
                             (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (NUMBERP Y1)
                      (NOT (EQUAL Y1 0))
                      (LESSP Y1 X1))
                 (EQUAL (GCD Y1 (DIFFERENCE X1 Y1))
                        (GCD X0 Y0))),
```

which again simplifies, applying GCD.IS.SYM, to:

```
      T. (* Comment: The Theorem Prover made use here
            of the symmetry lemma.)
```

```
Case 2.3.
       (IMPLIES (AND (EQUAL X1 0)
                     (EQUAL Y1 (GCD X0 Y0))
                     (NUMBERP X0)
                     (NUMBERP X1)
                     (NUMBERP Y0)
                     (NUMBERP Y1)
                     (NOT (EQUAL Y1 0))
                     (LESSP Y1 X1))
                (EQUAL (GCD Y1 (DIFFERENCE X1 Y1))
                       (GCD X0 Y0))),
```

which we again simplify, using linear arithmetic, to:

```
      T. (* Since X1=0, (LESSP Y1 X1) is impossible
            when (NUMBERP Y1) holds.)
```

```
Case 2.2.
       (IMPLIES (AND (NOT (EQUAL X1 0))
                     (NOT (LESSP X1 Y1))
                     (EQUAL (GCD 0 Y1) (GCD X0 Y0))
                     (NUMBERP X0)
                     (NUMBERP X1)
                     (NUMBERP Y0)
                     (NUMBERP Y1)
                     (NOT (EQUAL Y1 0))
                     (NOT (LESSP Y1 X1)))
                (EQUAL (GCD X1 (DIFFERENCE Y1 X1))
                       (GCD X0 Y0))).
```

This again simplifies, using linear arithmetic, to the new formula:

```
      (IMPLIES (AND (NOT (EQUAL X1 0))
                    (NOT (LESSP X1 X1))
                    (EQUAL (GCD 0 X1) (GCD X0 Y0))
                    (NUMBERP X0)
                    (NUMBERP X1)
                    (NUMBERP Y0)
                    (NUMBERP X1)
                    (NOT (EQUAL X1 0))
                    (NOT (LESSP X1 X1)))
               (EQUAL (GCD X1 (DIFFERENCE X1 X1))
                      (GCD X0 Y0))).
```

But this simplifies again, rewriting with DIFFERENCE.0 and GCD.IS.SYM, and expanding the definitions of EQUAL and GCD, to:

```
        T. (* Observe that the lemma was used here again.
              (DIFFERENCE X1 X1)=0 and (GCD X1 0) =
              (GCD 0 X1) = (GCD X0 Y0), by hypothesis.)

Case 2.1.
        (IMPLIES (AND (EQUAL X1 0)
                      (EQUAL Y1 (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (NUMBERP Y1)
                      (NOT (EQUAL Y1 0))
                      (NOT (LESSP Y1 X1)))
                 (EQUAL (GCD X1 (DIFFERENCE Y1 X1))
                        (GCD X0 Y0))).
```

But this again simplifies, opening up EQUAL, DIFFERENCE, and GCD, to:

```
        T. (* Since X1=0, (GCD X1 (DIFFERENCE Y1 X1)) =
              (GCD 0 Y1) = Y1 = (GCD X0 Y0))


Case 1. (IMPLIES (AND (EQUAL (GCD X1 Y1) (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (NUMBERP Y1)
                      (NOT (LESSP 0 Y1)))
                 (EQUAL X1 (GCD X0 Y0))).
```

This simplifies, unfolding the functions EQUAL, LESSP, and GCD, to:

```
        (IMPLIES (AND (EQUAL X1 0)
                      (EQUAL Y1 (GCD X0 Y0))
                      (NUMBERP X0)
                      (NUMBERP X1)
                      (NUMBERP Y0)
                      (EQUAL Y1 0))
                 (EQUAL X1 (GCD X0 Y0))),
```

which again simplifies, clearly, to:

```
        T. (* X1 = 0 = Y1 = (GCD X0 Y0))

Q.E.D.


The following VCs were proved: (JGCD.1 JGCD.2)
The following VCs were not proved: NIL

46 CHRONOLOGY
(GCD.IS.SYM GCD NUMBERP4 BEGINNING.OF.PROOF INBOUNDS
```

```
UNDEFINED ...

47_EVENTS.TO.SAVE
(NUMBERP4 GCD GCD.IS.SYM)

        (* These are the three events which were loaded from
         the EVENTS file JGCD.EVENTS.  They are the only events
         in CHRONOLOGY special to this particular verification)
```

C. INITIALIZING A ONE-DIMENSIONAL TABLE

1. The Jovial Program

The Jovial program shown below declares a one-dimensional table TAB of length NN with unsigned integer elements and initializes it to zero. The verification of this program serves to illustrate how the RJE verifier handles table semantics. It also illustrates the use of parameterized assertions.

```
START PROGRAM ZEROA;
       "Initializes contents of a table TAB to all zeros"
BEGIN ITEM NN U; TABLE TAB(1 : NN) U; ITEM II U;

%ASSERT ZEROA.IN(TAB, NN);%

II=1; TAB(II)=0;
WHILE %ASSERT ZEROA.LOOP(TAB, NN, II);%
      II<NN;
      BEGIN II=II+1; TAB(II)=0; END

%ASSERT ZEROA.OUT(TAB, NN);%

END
TERM
```

2. Assertions and VCs for the Program

There are three Floyd assertions--an entry assertion ZEROA.IN(TAB, NN), an exit assertion ZEROA.OUT(TAB, NN), and a loop assertion ZEROA.LOOP(TAB, NN, II). The loop assertion, or inductive invariant, involves the loop index II as well as the table length NN and table name TAB. The computation of VCs for this program does not require that these three assertions be specified in detail--only the variables appearing in the assertions.

The RJE verification condition generator computes two VCs for the program and names them ZEROA.1, ZEROA.2. ZEROA.1 is as follows:

```
(IMPLIES (ZEROA.IN TAB NN)
         (ZEROA.LOOP (ALPHA TAB (INDEX 1) 0)
                     NN 1))
```

The definitions for the assertions will be discussed below. Observe, at this point, that the arguments to the loop assertion appearing in ZEROA.1 are TAB = (ALPHA TAB (INDEX 1) 0) and II = 1. This is so because II was initialized to 1 and TAB(II) to 0 ahead of the loop statement. The construct (ALPHA <name> <index> <value>) represents the "changed" array, i.e., the table <name> with the element at index <index> altered to contain <value>.

The second VC, ZEROA.2, is computed by the VCG to be as follows:

```
(AND (IMPLIES (AND (ZEROA.LOOP TAB NN II)
                   (LESSP II NN))
              (ZEROA.LOOP (ALPHA TAB
                                 (INDEX (PLUS II 1))
                                 0)
                          NN
                          (PLUS II 1)))
     (IMPLIES (AND (ZEROA.LOOP TAB NN II)
                   (NOT (LESSP II NN)))
              (ZEROA.OUT TAB NN))))
```

This VC is a conjunction of two path conditions, viz., the path from the loop assertion around the loop, and the path leading from the loop assertion to the exit assertion. The first path condition includes the hypothesis (LESSP II NN) because the body of the loop statement is executed only if II<NN holds. Likewise, the second path condition in ZEROA.2 contains the hypothesis clause, (NOT (LESSP II NN)), which is the condition for termination of the loop.

3. Formal Proof of Correctness

Next, we examine the action of the Theorem Prover on these VCs.

The user has typed STARTUP to initiate interaction with the Theorem Prover's User Executive. The proof transcript shown here has been edited only to the extent of altering indentation to fit the pages in this report, and by adding underscores to flag text typed by the user.

```
                (* Beginning of transcript)
99 STARTUP
This is the Automatic Theorem Prover for Recursive Functions
You are in user mode: (AUTOMATIC VERBOSE)
Do you want to change your mode of interaction with the system?
NO
The last J73 program worked on was: <ELSPAS>ZEROA.;2
Shall we continue with this source file?
Yes, continue
 [confirm]

The most recent VC file for your program has already been loaded.
The number of separate VCs is: 2
Starting proofs of VCs in PROVE mode...

Attempting the VC:
ZEROA.1

(IMPLIES (ZEROA.IN TAB NN)
         (ZEROA.LOOP (ALPHA TAB (INDEX 1) 0)
                     NN 1))

This conjecture can be simplified, using the abbreviations
IMPLIES, INDEX, and ZEROA.IN, to:

      (IMPLIES (LESSP 0 NN)
               (ZEROA.LOOP (ALPHA TAB 1 0) NN 1)),

which simplifies, rewriting with the lemma ZEROA.LEMMA.4, and
expanding the functions ZEROA, LESSEQP, LESSP, EQUAL, and
ZEROA.LOOP, to the new conjecture:

      (IMPLIES (AND (NOT (EQUAL NN 0)) (NUMBERP NN))
               (NOT (LESSP NN 1))).

But this again simplifies, using linear arithmetic, to:

      T.

Q.E.D.
                (* Transcript interrupted here)
```

We have interrupted the display of the transcript at this point to call attention to several matters. Observe that the proof of ZEROA.1 was almost immediate, involving only the expansion of the definitions for ZEROA.IN and ZEROA.LOOP, plus one previously proved lemma ZEROA.LEMMA.4. We shall discuss this lemma (and others used in the remainder of the proof) along with the assertion predicates in the next subsection.

The system continues (in AUTOMATIC mode) with the proof of the second VC.

```
                    (* Resuming the transcript)
Attempting the VC:
ZEROA.2

(AND (IMPLIES (AND (ZEROA.LOOP TAB NN II)
                   (LESSP II NN))
              (ZEROA.LOOP (ALPHA TAB
                                 (INDEX (PLUS II 1)) 0)
                                 NN
                          (PLUS II 1)))
     (IMPLIES (AND (ZEROA.LOOP TAB NN II)
                   (NOT (LESSP II NN)))
              (ZEROA.OUT TAB NN)))

This conjecture can be simplified, using the abbreviations
NOT, LESSEQP, ZEROA.LOOP, IMPLIES, AND, ZEROA.OUT, and INDEX,
to two new conjectures:

Case 2. (IMPLIES (AND (LESSP 0 NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (LESSP II NN))
                 (ZEROA.LOOP (ALPHA TAB (PLUS II 1) 0)
                             NN
                             (PLUS II 1))),

  which simplifies, applying COMMUTATIVITY.OF.PLUS, SELECT.ALPHA,
  and SUB1.ADD1, and expanding the definitions of SUB1, NUMBERP,
  EQUAL, PLUS, ZEROA, LESSEQP, LESSP, and ZEROA.LOOP, to two new
  conjectures:

  Case 2.2.
         (IMPLIES (AND (NOT (EQUAL NN 0))
                       (NUMBERP NN)
                       (NOT (LESSP NN II))
                       (ZEROA TAB II)
                       (LESSP II NN))
                  (NOT (LESSP (SUB1 NN) (PLUS 0 II)))),

    which again simplifies, using linear arithmetic, to:

         T.
```

```
Case 2.1.
        (IMPLIES (AND (NOT (EQUAL NN 0))
                      (NUMBERP NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (LESSP II NN))
                 (ZEROA (ALPHA TAB (ADD1 (PLUS 0 II)) 0)
                        (PLUS 0 II))).
```

But this simplifies again, expanding the definitions of EQUAL, PLUS, LESSP, and ZEROA, to the following two new goals:

```
Case 2.1.2.
        (IMPLIES (AND (NOT (EQUAL NN 0))
                      (NUMBERP NN)
                      (NOT (NUMBERP II)))
                 (ZEROA (ALPHA TAB 1 0) 0)).
```

This simplifies again, rewriting with the lemma ZEROA.LEMMA.3, and expanding ZEROA and EQUAL, to:

```
        T.
```

```
Case 2.1.1.
        (IMPLIES (AND (NOT (EQUAL NN 0))
                      (NUMBERP NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (LESSP II NN)
                      (NUMBERP II))
                 (ZEROA (ALPHA TAB (ADD1 II) 0) II)),
```

which we again simplify, rewriting with ZEROA.LEMMA.3, to:

```
        T.
```

```
Case 1. (IMPLIES (AND (LESSP 0 NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (NOT (LESSP II NN)))
                 (ZEROA TAB NN)).
```

This simplifies, using linear arithmetic, to three new formulas:

```
Case 1.3.
        (IMPLIES (AND (NOT (NUMBERP NN))
                      (LESSP 0 NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (NOT (LESSP II NN)))
                 (ZEROA TAB NN)).
```

This again simplifies, opening up ZEROA, to:

```
        T.
```

```
Case 1.2.
        (IMPLIES (AND (NOT (NUMBERP II))
                      (LESSP 0 NN)
                      (NOT (LESSP NN II))
                      (ZEROA TAB II)
                      (NOT (LESSP II NN)))
                 (ZEROA TAB NN)),
```

which again simplifies, unfolding the definitions of LESSP and ZEROA, to:

```
        T.
```

```
Case 1.1.
        (IMPLIES (AND (NUMBERP II)
                      (NUMBERP NN)
                      (LESSP 0 II)
                      (NOT (LESSP II II))
                      (ZEROA TAB II)
                      (NOT (LESSP II II)))
                 (ZEROA TAB II)).
```

This simplifies again, trivially, to:

```
        T.
```

Q.E.D.

```
The following VCs were proved: (ZEROA.1 ZEROA.2)
All VCs were proved for your program: <ELSPAS>ZEROA.;2

                (* End of proof transcript)
```

This completes the actual proof transcript (as written out to a DRIBBLE file).

4. Discussion of the Proof

To understand the above proof transcript the reader must first be aware of the precise definitions used for the three assertion predicates. These definitions were as follows:

```
(DEFN ZEROA.IN (TAB NN) (LESSP 0 NN) (* The entry assertion))

(DEFN ZEROA.LOOP
      (TAB NN II)
      (AND (LESSP 0 NN)
           (LESSEQP II NN)
           (ZEROA TAB II))
      (* The loop assertion, or inductive invariant))

(DEFN ZEROA.OUT
      (TAB NN)
      (ZEROA TAB NN)
      (* The output, or exit assertion))
```

Thus, the entry assertion demands merely that the table length NN be a positive integer. The output assertion ZEROA.OUT and the loop predicate ZEROA.LOOP are both defined in terms of a recursive predicate ZEROA(A, N) which captures the notion of an array containing all zero elements from index 1 to N.

The recursive definition for ZEROA is

```
(DEFN ZEROA
      (A N)
      (IF (ZEROP N)
          T
          (AND (EQUAL (SELECT A N) 0)
               (ZEROA A (SUB1 N)))))
```

Thus, for nonzero N, (ZEROA A N) is true if and only if the Nth element of A, i.e., A[N], is zero and (ZEROA A N-1) is true. When N=0, (ZEROA A 0) is defined to be true for any table A. Thus, (ZEROA A N) means that A[1] = A[2] = ... = A[N] = 0. This definition is seen to be suitable only for tables whose lower index bound is 1. In general, we would need to define a modified "ZEROA" with an additional lower bound argument, such as ZEROA*(A, LB, UB). For multidimensional tables, still further generalizations would be needed.

The inductive assertion (or loop invariant) ZEROA.LOOP required for this program states that 0<NN, II<=NN, and ZEROA(AA, II). The last clause captures the principal inductive notion that when control is at the head of the WHILE loop the current state of the table AA is such that all elements from 1 to II

are zero.

The proofs shown above in the transcript made use of several lemmas, two declarations, and one axiom, none of which have yet been discussed. These events are shown next.

```
(PROVE.LEMMA ZEROA.LEMMA.4
             (REWRITE)
             (IMPLIES (ZEROA A N)
                      (ZEROA (ALPHA A (ADD1 N) 0)
                             (ADD1 N))))

(PROVE.LEMMA ZEROA.LEMMA.3
             (REWRITE)
             (IMPLIES (ZEROA A N)
                      (ZEROA (ALPHA A M 0) N)))

(PROVE.LEMMA ZEROA.LEMMA.2
             (REWRITE)
             (IMPLIES (AND (ZEROA A N)
                           (EQUAL (SELECT A (ADD1 N)) 0))
                      (ZEROA A (ADD1 N))))


(ADD.AXIOM SELECT.ALPHA
           (REWRITE)
           (EQUAL (SELECT (ALPHA A I V) J)
                  (IF (EQUAL J I) V (SELECT A J)))
           (* The only axiom we need to give SELECT and ALPHA
            their intended meanings; i.e., selecting the Jth
            element of (ALPHA A I V) gives V if I=J, otherwise
            whatever A had at J before the change in A produced
            by the operation ALPHA.))

(DCL ALPHA
     (A I V)
     (* Uninterpreted function whose meaning is the array A
        with the Ith element changed to V; I does not have to
        be an integer, i.e., for 2-dim arrays I might be
        (INDEX2 a b)))

(DCL SELECT
     (A I)
     (* Uninterpreted function whose meaning is the selection
        of the Ith element of the array A))
```

These theorem prover events appear in reversed chronological order (i.e., the order in which they occur on the variable CHRONOLOGY). Let us, therefore, consider them starting with the declarations, which occurred first in the order of execution. The two declarations introduce (SELECT A I) and (ALPHA A I V) as undefined notions. The intent is for (SELECT A I) to represent the

table reference A[I], i.e., the Ith element of the table A. Indeed, the Jovial parser transduces table references such as AA[II] to the internal form (SELECT AA (INDEX II)). (Note that there is no need for theorem prover definitions to conform to Jovial naming conventions; in particular, the formal arguments A and I are single-character "names"). Likewise, the notation (ALPHA A I V) represents the result of changing the Ith element in table A to V. It is an abstract denotation for the resulting table object. In order for the theorem prover to be able to manipulate such ALPHA constructs, it needs an axiom to express the relation between elements of a changed table and those of the original (unchanged) table. The axiom SELECT.ALPHA accomplishes this. As can be seen above, this axiom states that referencing (ALPHA A I V) at index J yields V if J=I, and otherwise yields (SELECT A J), i.e., the value of the Jth element of the original table A.

The function INDEX(X) is defined as returning simply X, to accommodate the transduction (INDEX II) for one-dimensional table indices. (For tables of dimensionality two or higher the function INDEX2(X, Y) plays a corresponding role).

The three lemmas are considered next. All three were introduced to shorten the proofs of the VCs. Although the proofs of validity of the VCs would have succeeded without first proving these lemmas, they would have been reproved several times in the process. By separately proving these lemmas (by means of PROVE.LEMMA) the theorem prover is able to store these facts and use them in the VC proofs (as is clear from the theorem prover's explanatory output shown in the above transcript).

The lemma ZEROA.LEMMA.2 is simply a slight restatement of the definition of the functional predicate ZEROA, one which facilitates the induction in the above proof. As an additional feature, the user also requested a summary of the status of the verification of program ZEROA, shown below. Use of the theorem prover function IMMEDIATE.DEPENDENTS.OF (not displayed here) showed that ZEROA.LEMMA.2 was employed in the proof of another lemma, ZEROA.LEMMA.4, which in turn was used to prove the first VC.

The next lemma, ZEROA.LEMMA.3, states simply that changing any element of a table that already satisfies ZEROA to zero does not alter that property. This lemma was also used in the proof (not shown above) of ZEROA.LEMMA.4, and

appeared independently in the proof of the second VC.

The last lemma, ZEROA.LEMMA.4, states that if a table A has zero elements from index 1 to N, and the N+1th element is changed to zero, then the new table (ALPHA A N+1 0) contains zeros over the index range 1 to N+1. As already noted, this lemma was used in the proof of the first VC.

```
100_(PROOF.STATUS)
The Jovial program ZEROA
File version: <ELSPAS>ZEROA.;2
has the following associated files:

(VCFILE <ELSPAS>ZEROA.PRS-VCS;2)
(EVENTS.FILE <ELSPAS>ZEROA.EVENTS;1)

Proof file exists: <ELSPAS>ZEROA.PRF;2

The status of verification is:
(PROVED.VCS (ZEROA.1 ZEROA.2))
(FAILED.VCS NIL)
(VCS.TO.REDO NIL)

Last attempted on: "16-Feb-82 10:59:09"
Proof completed on: "16-Feb-82 10:59:09"
```

By this means the system user may apprise himself of the current status of verification for his program. The last line (i.e., "Proof completed...") is printed only when all VCs for the program have been verified. The indications for FAILED.VCS and VCS.TO.REDO (which are NIL in our example) serve as general reminders of what remains to be accomplished before the verification can be considered complete. Observe, in particular, that the mere existence of a proof file is no guarantee of the completeness of the proof. The indication of VCS.TO.REDO serves to remind the user of VCs that may have been "edited" by him as a temporary expedient in attempting to force through a proof. The names of any such VCs are saved on a variable called EDITED.VCS. If any of the current VCs are present on this list (it may also retain names of VCs from other programs attempted during the same session) the "Proof completed..." message will not be printed. Nor, of course, will this message be printed if any of the proofs actually failed.

D. VERIFICATION OF AN INTEGER SQUARE ROOT PROGRAM

This exercise illustrates the use of the RJE program verifier in the verification of an iterative Jovial program which computes the integer square root (i.e., the integer part of the square root) for an arbitrary integer NN.

The Jovial program is called ISQRT. Its text is as follows:

```
START PROGRAM IntSqrRt;
BEGIN ITEM NN U; ITEM XX U; ITEM YY U; ITEM ACC U;
                        %ASSERT NN>=0; %
   ACC=0; YY=1; XX=0;
 LP: %ASSERT (((XX=0) OR (XX-1)*(XX-1)<=NN) AND
        ((XX>=0) AND
        ((ACC=XX*XX) AND (YY=2*XX+1))))  ; %
   IF ACC<=NN; BEGIN ACC=ACC+YY; YY=YY+2; XX=XX+1;
        GOTO LP; END
                        %ASSERT ISQRTP(XX-1,NN);%
 END
TERM
```

The reader will observe that the three embedded assertions--for input, output, and the loop invariant--are written in expanded Jovial syntax (as Boolean formulas). However, the output assertion is only trivially of that form, since it is simply a predicate expression ISQRTP(XX-1,NN), whereas the other assertions contain fairly densely nested ANDs and ORs. The semantics of this output assertion are therefore not present in the program itself. Instead, they were incorporated at the theorem prover level by means of the following definition (i.e., a DEFN):

```
(DEFN ISQRTP
      (X N)
      (AND (LESSEQP (TIMES X X) N)
           (LESSP N (TIMES (ADD1 X) (ADD1 X)))))
```

Thus, ISQRTP characterizes X as the integer square root of N by means of the relation X*X <= N < (X+1)*(X+1); in words, X is the largest integer such that X*X does not exceed N.

This program also illustrates the handling of Jovial GOTOs in the RJE system. The program could equally well have been written (in a more structured form) through the use of a loop statement (in either the WHILE or FOR loop versions). Observe that the loop invariant is placed immediately following the statement label LP: addressed by the GOTO.

We show only the Theorem Prover's portion of the verification, the VCs for the program having already been generated (in the RJE front end subsystem)

prior to the theorem-proving phase. The Theorem Prover retrieves these VCs (which are shown below) from the .PRS-VCS file that was created by the front end operations.

```
        (*  Beginning of Theorem Prover transcript)

57 STARTUP
This is the Automatic Theorem Prover for Recursive Functions
You are in user mode: (MANUAL CONCISE)
Do you want to change your mode of interaction with the system?
N
The last J73 program worked on was: <JOVIAL>ISQRT.;3
Shall we continue with this source file?
Yes, continue
 [confirm]

You are in MANUALMODE; do you wish to proceed in AUTOMATIC?
N
Should I load the VC file: <JOVIAL>ISQRT.PRS-VCS;3? Y
The most recent VC file for your program has already been loaded.
The number of separate VCs is: 2
Should I load the EVENTS.FILE: <JOVIAL>ISQRT.EVENTS;1? N
Do you wish to examine the VCs?  Y
ISQRT.1          (* This is the entry VC; its proof amounts
                    to showing that the loop invariant, at LP:,
                    is satisfied by the initial values.)

  (IMPLIES (GREATEREQP NN 0)
           (AND (OR (EQUAL 0 0)
                    (LESSEQP (TIMES (DIFFERENCE 0 1)
                                    (DIFFERENCE 0 1))
                             NN))
                (AND (GREATEREQP 0 0)
                     (AND (EQUAL 0 (TIMES 0 0))
                          (EQUAL 1 (PLUS (TIMES 2 0)
                                         1)))))))

Type OK to continue, or NO to interrupt. OK
```

```
ISQRT.2          (* This is the VC for the paths leading from
                    the point LP:  (a) around the loop and back
                    to LP: , and (b) to the exit assertion.)

  (IMPLIES (AND (OR (EQUAL XX 0)
                    (LESSEQP (TIMES (DIFFERENCE XX 1)
                                    (DIFFERENCE XX 1))
                             NN))
                (AND (GREATEREQP XX 0)
                     (AND (EQUAL ACC (TIMES XX XX))
                          (EQUAL YY (PLUS (TIMES 2 XX)
                                          1)))))
           (AND (IMPLIES (LESSEQP ACC NN)
                         (AND (OR (EQUAL (PLUS XX 1)
                                         0)
                                  (LESSEQP
                                     (TIMES
                                        (DIFFERENCE (PLUS XX 1)
                                                    1)
                                        (DIFFERENCE (PLUS XX 1)
                                                    1))
                                     NN))
                              (AND (GREATEREQP (PLUS XX 1) 0)
                                   (AND
                                      (EQUAL (PLUS ACC YY)
                                             (TIMES (PLUS XX 1)
                                                    (PLUS XX 1)))
                                      (EQUAL
                                        (PLUS YY 2)
                                        (PLUS
                                           (TIMES 2 (PLUS XX 1))
                                           1))))))
                (IMPLIES (NOT (LESSEQP ACC NN))
                         (ISQRTP (DIFFERENCE XX 1)
                                 NN))))

Type OK to continue, or NO to interrupt. OK
What should be done if a proof fails? Quit
("SRI-CSL Tenex 1.34.41, SRI-CSL" <LISP>LISP.SAV;133
<BOYER>BASIS.;3 <BOYER>CODE.;2 <BOYER>CODE1.;787
<BOYER>DATA.;2 <BOYER>DATA1.;7 <BOYER>SIMPBLOCK.;2)
28/2/82 16:48-PST
Starting proofs of VCs in PROVE mode...
```

```
Attempting the VC:
ISQRT.1

(IMPLIES (GREATEREQP NN 0)
         (AND (OR (EQUAL 0 0)
                  (LESSEQP (TIMES (DIFFERENCE 0 1)
                                  (DIFFERENCE 0 1))
                           NN))
              (AND (GREATEREQP 0 0)
                   (AND (EQUAL 0 (TIMES 0 0))
                        (EQUAL 1 (PLUS (TIMES 2 0) 1))))))

This formula can be simplified, using the abbreviations
GREATEREQP, IMPLIES, and TIMES.ZERO, to:

        (* Observe that the only simplification achieved here
         is in replacing GREATEREQP by (NOT (LESSP ...)) and
         in evaluating (TIMES 2 0) to 0.)

      (IMPLIES (NOT (LESSP NN 0))
               (AND (OR (EQUAL 0 0)
                        (LESSEQP (TIMES (DIFFERENCE 0 1)
                                        (DIFFERENCE 0 1))
                                 NN))
                    (AND (GREATEREQP 0 0)
                         (AND (EQUAL 0 0)
                              (EQUAL 1 (PLUS 0 1)))))),

which simplifies, expanding the definitions of EQUAL,
DIFFERENCE, TIMES, LESSP, LESSEQP, OR, GREATEREQP, PLUS,
and AND, to:

      T.

Q.E.D.
```

Attempting the VC:
ISQRT.2

```
(IMPLIES (AND (OR (EQUAL XX 0)
                  (LESSEQP (TIMES (DIFFERENCE XX 1)
                                  (DIFFERENCE XX 1))
                           NN))
              (AND (GREATEREQP XX 0)
                   (AND (EQUAL ACC (TIMES XX XX))
                        (EQUAL YY (PLUS (TIMES 2 XX) 1)))))
         (AND (IMPLIES (LESSEQP ACC NN)
                       (AND (OR (EQUAL (PLUS XX 1) 0)
                                (LESSEQP (TIMES
                                           (DIFFERENCE
                                             (PLUS XX 1) 1)
                                             (DIFFERENCE
                                                (PLUS XX 1)
                                                 1))
                                         NN))
                            (AND (GREATEREQP (PLUS XX 1) 0)
                                 (AND (EQUAL
                                        (PLUS ACC YY)
                                        (TIMES
                                          (PLUS XX 1)
                                          (PLUS XX 1)))
                                      (EQUAL
                                        (PLUS YY 2)
                                        (PLUS (TIMES
                                                2
                                                (PLUS XX 1))
                                              1))))))
              (IMPLIES (NOT (LESSEQP ACC NN))
                       (ISQRTP (DIFFERENCE XX 1) NN))))
```

This formula can be simplified, using the abbreviations GREATEREQP, AND, and IMPLIES, to:

```
(IMPLIES
 (AND (OR (EQUAL XX 0)
          (LESSEQP (TIMES (DIFFERENCE XX 1)
                          (DIFFERENCE XX 1))
                   NN))
      (NOT (LESSP XX 0))
      (EQUAL ACC (TIMES XX XX))
      (EQUAL YY (PLUS (TIMES 2 XX) 1)))
 (AND
     (IMPLIES (LESSEQP ACC NN)
              (AND (OR (EQUAL (PLUS XX 1) 0)
                       (LESSEQP
                          (TIMES
                             (DIFFERENCE
                                  (PLUS XX 1) 1)
                                  (DIFFERENCE
                                     (PLUS XX 1)
                                     1))
                          NN))
                   (AND (GREATEREQP (PLUS XX 1) 0)
                        (AND (EQUAL (PLUS ACC YY)
                                    (TIMES
                                      (PLUS XX 1)
                                      (PLUS XX 1)))
                             (EQUAL (PLUS YY 2)
                                    (PLUS
                                      (TIMES 2
                                             (PLUS XX 1))
                                      1))))))
     (IMPLIES (NOT (LESSEQP ACC NN))
              (ISQRTP (DIFFERENCE XX 1) NN)))),
```

which simplifies, using linear arithmetic, applying COMMUTATIVITY.OF.PLUS, DIFFERENCE.PLUS1, SUB1.ADD1, ASSOCIATIVITY.OF.PLUS, COMMUTATIVITY2.OF.PLUS, TIMES.ADD1, COMMUTATIVITY.OF.TIMES, DISTRIBUTIVITY.OF.TIMES.OVER.PLUS, PLUS.ADD1, ADD1.DIFFERENCE, LESSP.DIFFERENCE2, PLUS.DIFFERENCE1, EQUAL.TIMES.0, PLUS.EQUAL.0, PLUS.DIFFERENCE2, TIMES.DIFFERENCE, DIFFERENCE.0, LESSP.DIFFERENCE1, and EQUAL.DIFFERENCE, and opening up the definitions of LESSEQP, SUB1, NUMBERP, EQUAL, PLUS, DIFFERENCE, OR, LESSP, GREATEREQP, TIMES, AND, IMPLIES, NOT, and ISQRTP, to 41 new formulas:

(* We have deleted the proofs of all but the first and last from this transcript since, like Cases 41 and 1, they are all trivial applications of linear arithmetic. This surprisingly large number of cases results from two factors: (a) the lack of predicates characterizing the types of the program variables as NUMBERP, and (b) the appearance of the function DIFFERENCE in the VCs. The latter function must, upon each expansion, produce a case split depending on the size relation of its arguments.)

```
Case 41.(IMPLIES (AND (LESSP (TIMES XX XX)
                             (PLUS XX (TIMES 0 XX)))
                      (NOT (LESSP NN (TIMES XX XX))))
                 (EQUAL (PLUS 1
                              (PLUS 2
                                    (PLUS XX
                                          (TIMES 1 XX))))
                        (PLUS 0
                           (PLUS 1
                                 (PLUS 2
                                       (PLUS XX
                                          (TIMES 1 XX)))))))).
```

But this simplifies again, using linear arithmetic, to:

T.

(* The proofs of Cases 2 through 40 have been deleted
from this transcript as already explained.)

```
Case 1. (IMPLIES (AND (EQUAL (TIMES XX XX)
                             (PLUS XX (TIMES 0 XX)))
                      (NOT (LESSP NN 0))
                      (LESSP NN (TIMES XX XX))
                      (LESSP (PLUS XX (TIMES 0 XX)) 1)
                      (NOT (LESSP (TIMES XX XX)
                                  (PLUS XX (TIMES 0 XX))))
                      (NUMBERP NN))
                 (NOT (LESSP (PLUS XX NN) (TIMES XX XX)))).
```

This simplifies again, using linear arithmetic, to:

T.

Q.E.D.

```
The following VCs were proved: (ISQRT.1 ISQRT.2)
All VCs were proved for your program: <JOVIAL>ISQRT.;3
```

The only Theorem Prover definition special to this proof was that of ISQRTP, which has already been discussed.

E. VERIFICATION WITH MACHINE-REPRESENTATION CONSTRAINTS

We show here a sample verification where machine representation constraints have been placed on the sizes of numeric (integer) data manipulated by a simple integer program. The program itself computes the integer quotient QUO and remainder REM for two integers NUM and DEN by repeated subtraction. (This example--but without the size constraints--is used as a running example in the RJE User Manual [15]. The VCs shown below were generated with the MAXINT option turned on in order to generate VCs with the integer-size constraints.

The entry assertion constrains the inputs NUM and DEN not to exceed the quantity MAXINT(10), where MAXINT(SS) is an implementation parameter defined (in Sec. 1.4 of MIL-STD-1589A [60]) as "the maximum integer value representable in SS+1 bits (including sign bit)." Thus, MAXINT(SS) is 2**SS-1. We modify the meaning slightly here to make our function MAX.INT(SS) the smallest nonrepresentable integer value, i.e., 2**SS. Thus, MAX.INT(10) = 1024, and all integer items in this program declared to be of size 10 (bits) are constrained to be strictly less than 1024.

```
START PROGRAM JREM.with.MAXINTs;
BEGIN ITEM NUM U 10; ITEM DEN U 10; ITEM REM U 10;
        ITEM QUO U 10;
        % ASSERT (DEN>0) AND ((DEN < MAXINT(10)) AND
                        (NUM < MAXINT(10))); %
        REM=NUM; QUO=0;
        WHILE % ASSERT LOOP.REM(NUM,DEN,REM,QUO); %
              NOT(REM<DEN);
                BEGIN
                  REM=REM-DEN;
                  QUO=QUO+1;
                END
        % ASSERT OUT.REM(NUM,DEN,REM,QUO); %
  END
TERM
```

The VCG has computed two VCs, MAXINT.1 and MAXINT.2, which are shown below. These VCs, of course, reflect the size constraints on the program variables, though part of the constraint is hidden in VCG-generated clauses of the form (LEGAL.J73.INTEGERP I). This function is later defined to the Theorem Prover to mean (AND (NUMBERP I) (LESSP I (MAX.INT 10))), i.e., that I is an unsigned integer less than (MAX.INT 10) = 2**10 = 1024, as already explained. The (as yet undefined) loop invariant LOOP.REM includes occurrences of the predicate LEGAL.J73.INTEGERP, which thereby impose the

legality restriction on all of the program variables. Thus, the verification of the VCs will prove, among other things, that the active variables REM and QUO never exceed the bound 1023 at any time during (any) execution of the program (provided that NUM and DEN satisfy the bounds at entry).

The definitions of those predicates special to this program which were used by the Theorem Prover are

```
_PPE(OUT.REM)   (* The output assertion defined here
                 asserts only that 0<=D, R<D, and
                 N=R+D*Q. Size constraint clauses
                 could, however, also have been
                 included.)
(DEFN OUT.REM
      (N D R Q)
      (AND (NOT (LESSP D 0))
           (LESSP R D)
           (EQUAL N (PLUS R (TIMES D Q)))))

_PPE(LOOP.REM)          (* The loop invariant assertion)
(DEFN LOOP.REM
      (N D R Q)
      (AND (LESSP 0 D)
           (EQUAL N (PLUS R (TIMES Q D)))
           (LEGAL.J73.INTEGERP N)
           (LEGAL.J73.INTEGERP D)
           (LEGAL.J73.INTEGERP R)
           (LEGAL.J73.INTEGERP Q)))

_PPE(MAX.INT)
(DEFN MAX.INT (S) (EXPT 2 S))   (* Defines (MAX.INT S) to be
                                 2**S)

_PPE(LEGAL.J73.INTEGERP)        (* Defines what it means for
                                 I to be a legal J73 unsigned
                                 integer.)
(DEFN LEGAL.J73.INTEGERP
      (I)
      (AND (NUMBERP I) (LESSP I (MAX.INT 10))))
```

The VCs are

```
_VClist
(MAXINT.1 MAXINT.2)
_PP MAXINT.1    (* User asks to see MAXINT.1; the other
                 VC, MAXINT.2, is shown after the proof
                 of MAXINT.1)

(IMPLIES (AND (LEGAL.J73.INTEGERP NUM)
              (LEGAL.J73.INTEGERP DEN)
              (LEGAL.J73.INTEGERP REM)
              (LEGAL.J73.INTEGERP QUO))
              (IMPLIES (AND (GREATERP DEN 0)
                            (LEGAL.J73.INTEGERP DEN)
```

```
                        (LEGAL.J73.INTEGERP NUM))
                   (AND (LESSP NUM (MAX.INT 10))
                        (LESSP REM (MAX.INT 10))
                        (LESSP 0 (MAX.INT 10))
                        (LESSP QUO (MAX.INT 10))
                        (LOOP.REM NUM DEN NUM 0))))
MAXINT.1

_(PROVE MAXINT.1)       (* Calling for proof of the
                                first VC.)
```

This formula can be simplified, using the abbreviations LEGAL.J73.INTEGERP, AND, IMPLIES, MAX.INT, and GREATERP, to:

```
      (IMPLIES (AND (NUMBERP NUM)
                    (LESSP NUM (MAX.INT 10))
                    (NUMBERP DEN)
                    (LESSP DEN (MAX.INT 10))
                    (NUMBERP REM)
                    (LESSP REM (MAX.INT 10))
                    (NUMBERP QUO)
                    (LESSP QUO (MAX.INT 10))
                    (LESSP 0 DEN))
               (AND (LESSP NUM (EXPT 2 10))
                    (LESSP REM (EXPT 2 10))
                    (LESSP 0 (EXPT 2 10))
                    (LESSP QUO (EXPT 2 10))
                    (LOOP.REM NUM DEN NUM 0))),
```

which simplifies, applying COMMUTATIVITY.OF.PLUS, and opening up the functions EXPT, LESSP, LEGAL.J73.INTEGERP, MAX.INT, PLUS, TIMES, EQUAL, LOOP.REM, and AND, to:

```
      T.

Q.E.D.

PROVED

_PP MAXINT.2

  (AND (IMPLIES (LOOP.REM NUM DEN REM QUO)
                (AND T (LEGAL.J73.INTEGERP REM)
                     (LEGAL.J73.INTEGERP DEN)))
       (IMPLIES (AND (LOOP.REM NUM DEN REM QUO)
                     (NOT (LESSP REM DEN)))
                (AND (LESSP REM (MAX.INT 10))
                     (LESSP DEN (MAX.INT 10))
                     (LESSP REM (MAX.INT 10))
                     (LESSP QUO (MAX.INT 10))
                     (LESSP 1 (MAX.INT 10))
                     (LESSP QUO (MAX.INT 10))
```

```
                    (LOOP.REM NUM DEN
                              (DIFFERENCE REM DEN)
                              (PLUS QUO 1))))
       (IMPLIES (AND (LOOP.REM NUM DEN REM QUO)
                     (LESSP REM DEN))
                (OUT.REM NUM DEN REM QUO)))
MAXINT.2
_(PROVE MAXINT.2)        (* Calling for proof of the
                                second VC.)
```

This conjecture can be simplified, using the abbreviations NOT, LEGAL.J73.INTEGERP, LOOP.REM, IMPLIES, AND, and MAX.INT, to three new conjectures:

```
Case 3. (IMPLIES (AND (LESSP 0 DEN)
                      (EQUAL NUM
                             (PLUS REM (TIMES QUO DEN)))
                      (NUMBERP NUM)
                      (LESSP NUM (MAX.INT 10))
                      (NUMBERP DEN)
                      (LESSP DEN (MAX.INT 10))
                      (NUMBERP REM)
                      (LESSP REM (MAX.INT 10))
                      (NUMBERP QUO)
                      (LESSP QUO (MAX.INT 10)))
                 (AND T
                      (AND (LEGAL.J73.INTEGERP REM)
                           (LEGAL.J73.INTEGERP DEN)))),
```

which we simplify, expanding the functions MAX.INT, LEGAL.J73.INTEGERP, and AND, to:

```
       T.

Case 2. (IMPLIES (AND (LESSP 0 DEN)
                      (EQUAL NUM
                             (PLUS REM (TIMES QUO DEN)))
                      (NUMBERP NUM)
                      (LESSP NUM (MAX.INT 10))
                      (NUMBERP DEN)
                      (LESSP DEN (MAX.INT 10))
                      (NUMBERP REM)
                      (LESSP REM (MAX.INT 10))
                      (NUMBERP QUO)
                      (LESSP QUO (MAX.INT 10))
                      (NOT (LESSP REM DEN)))
                 (AND (LESSP REM (EXPT 2 10))
                      (LESSP DEN (EXPT 2 10))
                      (LESSP REM (EXPT 2 10))
                      (LESSP QUO (EXPT 2 10))
                      (LESSP 1 (EXPT 2 10))
                      (LESSP QUO (EXPT 2 10))
                      (LOOP.REM NUM DEN
                                (DIFFERENCE REM DEN)
```

```
                                (PLUS QUO 1)))),
```

This simplifies, appealing to the lemmas COMMUTATIVITY.OF.TIMES, COMMUTATIVITY.OF.PLUS, SUB1.ADD1, LESSP.DIFFERENCE1, DIFFERENCE.PLUS1, ASSOCIATIVITY.OF.PLUS, PLUS.DIFFERENCE2, DISTRIBUTIVITY.OF.TIMES.OVER.PLUS, and TIMES.ADD1, and expanding the definitions of EXPT, AND, LESSP, SUB1, NUMBERP, EQUAL, PLUS, LEGAL.J73.INTEGERP, MAX.INT, TIMES, and LOOP.REM, to the following two new goals:

Case 2.2.

```
        (IMPLIES (AND (NOT (EQUAL DEN 0))
                      (LESSP (PLUS REM (TIMES DEN QUO))
                             1024)
                      (NUMBERP DEN)
                      (LESSP DEN 1024)
                      (NUMBERP REM)
                      (LESSP REM 1024)
                      (NUMBERP QUO)
                      (LESSP QUO 1024)
                      (NOT (LESSP REM DEN)))
                 (LESSP REM (PLUS 1024 DEN))).
```

This simplifies again, using linear arithmetic, to:

```
        T.
```

Case 2.1.

```
        (IMPLIES (AND (NOT (EQUAL DEN 0))
                      (LESSP (PLUS REM (TIMES DEN QUO))
                             1024)
                      (NUMBERP DEN)
                      (LESSP DEN 1024)
                      (NUMBERP REM)
                      (LESSP REM 1024)
                      (NUMBERP QUO)
                      (LESSP QUO 1024)
                      (NOT (LESSP REM DEN)))
                 (LESSP (PLUS 0 QUO) 1023)),
```

which again simplifies, using linear arithmetic, applying LESSEQP.TIMES, and unfolding the function ZEROP, to:

```
        T.
```

```
Case 1. (IMPLIES (AND (LESSP 0 DEN)
                      (EQUAL NUM (PLUS REM (TIMES QUO DEN)))
                      (NUMBERP NUM)
                      (LESSP NUM (MAX.INT 10))
                      (NUMBERP DEN)
                      (LESSP DEN (MAX.INT 10))
                      (NUMBERP REM)
```

```
                (LESSP REM (MAX.INT 10))
                (NUMBERP QUO)
                (LESSP QUO (MAX.INT 10))
                (LESSP REM DEN))
           (OUT.REM NUM DEN REM QUO)),
```

which simplifies, rewriting with COMMUTATIVITY.OF.TIMES, and opening up the functions LESSP, EQUAL, and OUT.REM, to:

```
      T.

Q.E.D.

PROVED
 CHRONOLOGY
(OUT.REM LOOP.REM LEGAL.J73.INTEGERP MAX.INT BEGINNING.OF.PROOF
INBOUNDS UNDEFINED STATE BOOLEANP EQUIVALENCE NEXT DOT INDEX
NOTEQUAL....<printing truncated>
```

It is worth noting how much "expression clutter" is produced in this example by the size-constraint clauses of the input and loop assertions. Individually, these clauses add little burden to the overall proof. However, because of their number it is possible that their presence adds appreciably to the time spent in proving the algorithmic properties of "real" interest, i.e., 0<=DEN, REM<DEN, and NUM = REM + DEN*QUO. This proof could have been shortened by declaring the input quantities NUM and DEN as J73 constants instead of (variable) integer data ITEMs. In that case, the corresponding legality clauses would have been automatically inserted by the VCG as hypotheses wherever required. Observe also that in most cases there appear to be "redundant" occurrences of the size-constraint clauses. This happens because for every numeric expression seen by the VCG, e.g., on the righthand side of an assignment statement, size-constraint clauses are generated for each subexpression of that expression. The variables themselves, of course, occur many times as such expressions, hence the multiple occurrences of these clauses. A little thought will show that these multiple occurrences are, in fact, necessary.

We wish to call attention to a subtlety in the above proof that is not apparent without closer inspection. In the proof of Case 2 of MAXINT.2 (the VC traversing the while-loop) one of the conclusions to be verified is that QUO stays within the size constraint after QUO is incremented in the loop body, i.e., that (LEGAL.J73.INTEGERP (PLUS 1 QUO)) is implied by the

hypotheses. This in turn requires proving that QUO is less than 1023, which fact is proved in Case 2.1. The only obvious upper bound on QUO here is supplied by the hypothesis (LESSP QUO 1024) of Case 2.1, and this is obviously insufficient for QUO might be equal to 1023. However, a closer look indicates that part of the loop invariant assertion requires that NUM = REM + DEN*QUO is less than 1024, and also that 0<DEN. Moreover, the path condition for traversing the loop provides the clause REM>=DEN. Thus, REM and DEN are both at least 1, and since QUO is nonnegative it follows that

```
1024 > NUM = REM + DEN*QUO
           >=  1 + 1*QUO
            = 1 + QUO
```

The Theorem Prover is quite clever at linear arithmetic. Here, however, the deduction required involves nonlinear, multiplicative expressions where two variables are multiplied together. For this deduction a lemma is needed. In fact, the proof of Case 2.1 is credited to "linear arithmetic" and a REWRITE lemma LESSEQP.TIMES. (Observe that we needed this lemma in the informal hand proof shown above when we implicitly used the fact that DEN >= 1 and QUO >=0 imply DEN*QUO >= QUO). The statement of the lemma LESSEQP.TIMES is

```
(PROVE.LEMMA LESSEQP.TIMES
             (REWRITE)
             (IMPLIES (NOT (ZEROP J))
                      (NOT (LESSP (TIMES J X) X))))
```

Observe, too, that the lemma's precondition (NOT (ZEROP J)) is crucial, for if J were zero J*X >= X would be false. In the application of the lemma to deducing DEN*QUO >= QUO, this precondition is satisfied because (NOT (EQUAL DEN 0)) is a hypothesis.

F. HDM VERIFICATION OF THREE PROCEDURES

We display here transcripts of STP proofs of the VCs for the procedures "stack-init", "push", and "pop" of the module "stack" which was discussed in Sec. VIII-E.

1. Verification of "stack-init"

```
1.     (DEFINEQ (PR*(NLAMBDA ARGS
                     (COND
                       ((EQ (QUOTE Y)
                            (ASKUSER NIL NIL
                                     "Attempt proof now? "
                                     (QUOTE ((Y "es")
                                             (N "o")))
                                     T NIL
                                     (QUOTE (CONFIRMFLG T))))
                        (APPLY (FUNCTION PR)
                               ARGS)
                        (OR (AND (LISTP EVENTS)
                                 (EQ (CAAR EVENTS)
                                     (QUOTE PR))
                                 (EQ (CADAR EVENTS)
                                     (CAR ARGS)))
                            (USEREXEC "!")))
                       (T (USEREXEC "!"))))))

       "Beginning of the actual proof"

1.     (DT STATE)
2.     (DT STATE)
Redundant event- ignored
2.     (DS STATE NEXT (STATE))
3.     (DSV INTEGER ARG.1)
4.     (DSV INTEGER ARG.2)
5.     (DSV INTEGER J)
6.     (DSV INTEGER ARG)
7.     (DS STATE ARRAY_MOD.BEGIN.STATE)
8.     (DSV STATE FREE.ARRAY_MOD.STATE)
9.     (DS INTEGER MAXARRAYSIZE)
10.    (DS INTEGER MAXINT)
11.    (DS INTEGER READ (INTEGER STATE))
12.    (DD INTEGER IPLUS (ARG.1 ARG.2)
           (PLUS ARG.1 ARG.2))
13.    (DD INTEGER IDIFFERENCE (ARG.1 ARG.2)
           (DIFFERENCE ARG.1 ARG.2))
14.    (DD INTEGER ITIMES (ARG.1 ARG.2)
           (TIMES ARG.1 ARG.2))
15.    (DD INTEGER PTR (FREE.ARRAY_MOD.STATE)
           (READ 0 FREE.ARRAY_MOD.STATE))
16.    (DD INTEGER STACK_VAL
           (ARG FREE.ARRAY_MOD.STATE)
           (IF (GREATERP ARG 0)
```

```
              (READ ARG FREE.ARRAY_MOD.STATE)
              0))
17.    (DD INTEGER MAX_STACK_SIZE
           NIL
           (IDIFFERENCE (MAXARRAYSIZE)
                        1))
18.    ARRAY_MOD.ASSERTION#1
19.    STACK.ASSERTION#1
20.    VC.1
21.    (PR* VC.1 ARRAY_MOD.ASSERTION#1
           STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
601 conses
.5 seconds
Proved
22.    VC.2
23.    (PR* VC.2 ARRAY_MOD.ASSERTION#1
           STACK.ASSERTION#1)
Attempt proof now? Yes
Want instance for VC.2? Y
   J/ 0
----------Proving----------
780 conses
.45 seconds
Proved
```

2. Verification of "push"

```
40_LEF(PUSH.STP)
1.     (DEFINEQ (PR*(NLAMBDA ARGS
                     (COND
                       ((EQ (QUOTE Y)
                            (ASKUSER NIL NIL
                                     "Attempt proof now? "
                                     (QUOTE ((Y "es")
                                             (N "o")))
                                     T NIL
                                     (QUOTE (CONFIRMFLG T))))
                        (APPLY (FUNCTION PR)
                               ARGS)
                        (OR (AND (LISTP EVENTS)
                                 (EQ (CAAR EVENTS)
                                     (QUOTE PR))
                                 (EQ (CADAR EVENTS)
                                     (CAR ARGS)))
                            (USEREXEC "!")))
                       (T (USEREXEC "!"))))))

       "Beginning of the actual proof"

1.     (DT STATE)
2.     (DT STATE)
```

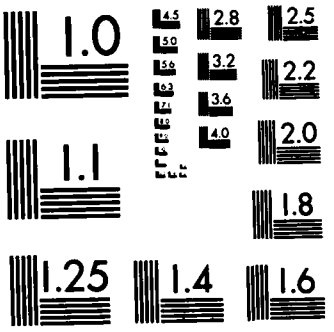
AD-A123 681 JOVIAL PROGRAM VERIFIER RUGGED JOVIAL ENVIRONMENT(U) 3/3
SRI INTERNATIONAL MENLO PARK CA B ELSPAS ET AL. OCT 82
RADC-TR-82-277 F30602-78-C-0031

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



```
Redundant event- ignored
2.      (DS STATE NEXT (STATE))
3.      (DSV INTEGER ARG.1)
4.      (DSV INTEGER ARG.2)
5.      (DSV INTEGER J)
6.      (DSV INTEGER ARG)
7.      (DS STATE ARRAY_MOD.BEGIN.STATE)
8.      (DSV STATE FREE.ARRAY_MOD.STATE)
9.      (DS INTEGER MAXARRAYSIZE)
10.     (DS INTEGER MAXINT)
11.     (DS INTEGER V)
12.     (DS INTEGER READ (INTEGER STATE))
13.     (DD INTEGER IPLUS (ARG.1 ARG.2)
            (PLUS ARG.1 ARG.2))
14.     (DD INTEGER IDIFFERENCE (ARG.1 ARG.2)
            (DIFFERENCE ARG.1 ARG.2))
15.     (DD INTEGER ITIMES (ARG.1 ARG.2)
            (TIMES ARG.1 ARG.2))
16.     (DD INTEGER PTR (FREE.ARRAY_MOD.STATE)
            (READ 0 FREE.ARRAY_MOD.STATE))
17.     (DD INTEGER STACK_VAL
            (ARG FREE.ARRAY_MOD.STATE)
            (IF (GREATERP ARG 0)
                (READ ARG FREE.ARRAY_MOD.STATE)
                0))
18.     (DD INTEGER MAX_STACK_SIZE
            NIL
            (IDIFFERENCE (MAXARRAYSIZE)
                         1))
19.     (DD INTEGER READ_OP
            (ARG FREE.ARRAY_MOD.STATE)
            (READ ARG FREE.ARRAY_MOD.STATE))
20.     ARRAY_MOD.ASSERTION#1
21.     STACK.ASSERTION#1
22.     VC.1.H1
23.     VC.1.H2
24.     VC.1.C
25.     (PR* VC.1.C VC.1.H1 VC.1.H2 ARRAY_MOD.ASSERTION#1
             STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
1140 conses
.9 seconds
Proved
26.     VC.2.H1
27.     VC.2.H2
28.     VC.2.H3
29.     VC.2.C.1
30.     (PR* VC.2.C.1 VC.2.H1 VC.2.H2 VC.2.H3
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
2079 conses
1.5 seconds
```

```
Proved
31.    VC.2.C.2
32.    (PR* VC.2.C.2 VC.2.H1 VC.2.H2 VC.2.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
4411 conses
3.45 seconds
Proved
33.    VC.2.C.3
34.    (PR* VC.2.C.3 VC.2.H1 VC.2.H2 VC.2.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------

collecting lists
10638, 10638 free cells
1542 conses
1.3 seconds
Proved
35.    VC.3.H1
36.    VC.3.H2
37.    VC.3.H3
38.    VC.3.C
39.    (PR* VC.3.C VC.3.H1 VC.3.H2 VC.3.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
1471 conses
1.1 seconds
Proved
40.    VC.4.H1
41.    VC.4.H2
42.    VC.4.H3
43.    VC.4.H4
44.    VC.4.C.1
45.    (PR* VC.4.C.1 VC.4.H1 VC.4.H2 VC.4.H3 VC.4.H4
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
Want instance for VC.4.H4? N
----------Proving----------
1374 conses
.9 seconds
Proved
46.    VC.4.C.2
47.    (PR* VC.4.C.2 VC.4.H1 VC.4.H2 VC.4.H3 VC.4.H4
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.4.H4?
----------Proving----------
1634 conses
1.15 seconds
Proved
```

```
48.     VC.5.H1
49.     VC.5.H2
50.     VC.5.H3
51.     VC.5.H4
52.     VC.5.H5
53.     VC.5.C.1
54.     (PR* VC.5.C.1 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         VC.5.H5 ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? Y
   J/ 0
Want instance for VC.5.H5? Y
   J/ 0
----------Proving----------

collecting lists
9885, 10397 free cells
4481 conses
3.65 seconds
Proved
55.     VC.5.C.2
56.     (PR* VC.5.C.2 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4 VC.5.H5
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? Y
   J/ J:D
Want instance for VC.5.H5? Y
   J/ J:D
----------Proving----------

collecting lists
10146, 10146 free cells

106724 conses
68.45 seconds
Proved
57.     VC.5.C.3
58.     (PR* VC.5.C.3 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         VC.5.H5 ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? N
Want instance for VC.5.H5? N
----------Proving----------
1715 conses
1.2 seconds
Proved
```

3. Verification of "pop"

```
43_LEF(POP.STP)


1.      (DEFINEQ (PR# (NLAMBDA ARGS
                        (COND
                          ((EQ (QUOTE Y)
                               (ASKUSER NIL NIL
                                        "Attempt proof now? "
                                        (QUOTE ((Y "es")
                                                (N "o")))
                                        T NIL
                                        (QUOTE (CONFIRMFLG T))))
                           (APPLY (FUNCTION PR)
                                  ARGS)
                           (OR (AND (LISTP EVENTS)
                                    (EQ (CAAR EVENTS)
                                        (QUOTE PR))
                                    (EQ (CADAR EVENTS)
                                        (CAR ARGS)))
                               (USEREXEC "!")))
                          (T (USEREXEC "!"))))))

        "Beginning of the actual proof"

1.      (DT STATE)
2.      (DT STATE)
Redundant event- ignored
2.      (DS STATE NEXT (STATE))
3.      (DSV INTEGER ARG.1)
4.      (DSV INTEGER ARG.2)
5.      (DSV INTEGER ARG)
6.      (DSV INTEGER J)
7.      (DS STATE ARRAY_MOD.BEGIN.STATE)
8.      (DSV STATE FREE.ARRAY_MOD.STATE)
9.      (DS INTEGER MAXARRAYSIZE)
10.     (DS INTEGER MAXINT)
11.     (DS INTEGER READ (INTEGER STATE))
12.     (DD INTEGER IPLUS (ARG.1 ARG.2)
            (PLUS ARG.1 ARG.2))
13.     (DD INTEGER IDIFFERENCE (ARG.1 ARG.2)
            (DIFFERENCE ARG.1 ARG.2))
14.     (DD INTEGER ITIMES (ARG.1 ARG.2)
            (TIMES ARG.1 ARG.2))
15.     (DD INTEGER PTR (FREE.ARRAY_MOD.STATE)
            (READ 0 FREE.ARRAY_MOD.STATE))
16.     (DD INTEGER STACK_VAL
            (ARG FREE.ARRAY_MOD.STATE)
            (IF (GREATERP ARG 0)
                (READ ARG FREE.ARRAY_MOD.STATE)
                0))
17.     (DD INTEGER MAX_STACK_SIZE
            NIL
```

```
          (IDIFFERENCE (MAXARRAYSIZE)
                       1))
18.    (DD INTEGER READ_OP
          (ARG FREE.ARRAY_MOD.STATE)
          (READ ARG FREE.ARRAY_MOD.STATE))
19.    ARRAY_MOD.ASSERTION#1
20.    STACK.ASSERTION#1
21.    VC.1.H1
22.    VC.1.H2
23.    VC.1.C
24.    (PR* VC.1.C VC.1.H1 VC.1.H2 ARRAY_MOD.ASSERTION#1
           STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
1140 conses
.8 seconds
Proved
25.    VC.2.H1
26.    VC.2.H2
27.    VC.2.H3
28.    VC.2.C.1
29.    (PR* VC.2.C.1 VC.2.H1 VC.2.H2 VC.2.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
816 conses
.65 seconds
Proved
30.    VC.2.C.2
31.    (PR* VC.2.C.2 VC.2.H1 VC.2.H2 VC.2.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
2826 conses
1.8 seconds
Proved
32.    VC.2.C.3
33.    (PR* VC.2.C.3 VC.2.H1 VC.2.H2 VC.2.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
802 conses
.5 seconds
Proved
34.    VC.3.H1
35.    VC.3.H2
36.    VC.3.H3
37.    VC.3.C.1
38.    (PR* VC.3.C.1 VC.3.H1 VC.3.H2 VC.3.H3
        ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
1100 conses
.75 seconds
```

```
Proved
39.     VC.3.C.2
40.     (PR* VC.3.C.2 VC.3.H1 VC.3.H2 VC.3.H3
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------

collecting lists
10948, 10948 free cells
1291 conses
1.0 seconds
Proved
41.     VC.4.H1
42.     VC.4.H2
43.     VC.4.H3
44.     VC.4.C
45.     (PR* VC.4.C VC.4.H1 VC.4.H2 VC.4.H3
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
----------Proving----------
1152 conses
.9 seconds
Proved
46.     VC.5.H1
47.     VC.5.H2
48.     VC.5.H3
49.     VC.5.H4
50.     VC.5.C.1
51.     (PR* VC.5.C.1 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now? Yes
Want instance for VC.5.H4? N
----------Proving----------
1901 conses
1.5 seconds
Proved
52.     VC.5.C.2
53.     (PR* VC.5.C.2 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? Y
   J/ 0
----------Proving----------
4012 conses
2.5 seconds
Proved
54.     VC.5.C.3
55.     (PR* VC.5.C.3 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? Y
   J/ ARG:D
```

```
----------Proving----------

collecting lists
10466, 10466 free cells

34393 conses
22.4 seconds
Proved
56.     VC.5.C.4
57.     (PR* VC.5.C.4 VC.5.H1 VC.5.H2 VC.5.H3 VC.5.H4
         ARRAY_MOD.ASSERTION#1 STACK.ASSERTION#1)
Attempt proof now?
Yes
Want instance for VC.5.H4? N
----------Proving----------
1325 conses
.9 seconds
Proved
```

REFERENCES

[1]
Preliminary Ada Reference Manual
ACM, SIGPLAN Notices, Vol. 14, No. 6, 1979.

[2]
R.S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, 1979.

[3]
R.S. Boyer and J S. Moore.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
Technical Report CSL-108, SRI International, Menlo Park, California 94025, December, 1979.

[4]
R.S. Boyer and J S. Moore.
A Theorem-Prover for Recursive Functions: A User's Manual.
Technical Report CSL-91, SRI International, Menlo Park, California 94025, June, 1979.

[5]
R.S. Boyer and J S. Moore.
A Verification Condition Generator for FORTRAN.
Technical Report, SRI International, Menlo Park, California 94025. June, 1980.

[6]
JOCIT Compiler User's Manual, Revision 3
Computer Sciences Corp., El Segundo, CA, 1975.

[7]
JOVIAL J73/I Computer Programming Manual
Computer Sciences Corporation, El Segundo, California, 1977.

[8]
O.J. Dahl.
The SIMULA 67 Common Base Language.
Technical Report, Norwegian Computing Center, Oslo, 1968.

[9]
L.P. Deutsch.
An Interactive Program Verifier.
PhD thesis, Univ. of Calif, Berkeley, 1973.

[10]
E. W. Dijkstra.
Notes on Structured Programming.
Academic Press, New York, 1972, .
C. A. R. Hoare, ed.

[11]
E.W. Dijkstra.
A Discipline of Programming.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[12]
B. Elspas, et. al.
A Verification System for JOVIAL/J3 Programs (Rugged Programming Environment)--RPE/1.
Technical Report 3756-1, SRI International, Menlo Park, California 94025, January, 1976.
Also issued as RADC-TR-76-58, March 1976, by Rome Air Development Center, Griffiss AFB, NY 13441.

[13]
B. Elspas, et. al.
A Verification System for JOCIT/J3 Programs (Rugged Programming Environment) -- RPE/2.
Technical Report 5042-1, SRI International, Menlo Park, California 94025, April, 1977.
Also issued as RADC-TR-77-229, June 1977, by Rome Air Development Center, Griffiss AFB, NY 13441.

[14]
B. Elspas.
The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness.
Technical Report, SRI International, Menlo Park, California 94025, August, 1978.

[15]
Rugged JOVIAL Environment--User Manual
SRI International, Menlo Park, California 94025, 1981.
This user manual covers the RJE verification system developed under Contract F30602-78-C-0031 with Rome Air Development Center. The theorem prover portion of the system is described in greater detail in a separate user manual by Boyer and Moore dated June 1979.

[16]
R.J. Feiertag, P.M. Melliar-Smith, and J.M. Spitzen.
The YELLOW Programming Language: Preliminary Design Phase Report and Language Specification.
Technical Report, SRI International, Menlo Park, California 94025, February, 1978.
Sponsored by the Defense Advanced Research Projects Agency under Contract MDA903-77-C-0032.

[17]
R.W. Floyd.
Assigning Meanings to Programs.
In J.T. Schwartz, editor, Mathematical Aspects of Computer Science, Vol. 19, pages 19-32. Am. Math. Soc., Providence, RI, 1967.

[18]
S.L. Gerhart, et. al.
An Overview of AFFIRM.
In Proceedings IFIPS-80, pages 343-348. IFIPS, October, 1980.

[19]
J.A. Goguen and J. Meseguer.
OBJ-1: A Study in Executable Algebraic Formal Specification.
Technical Report, SRI International, Menlo Park, California 94025, July, 1981.
Supported by the Office of Naval Research under Contract No. N00014-80-0296.

[20]
J. V. Guttag.
The Specification and Application to Programming of Abstract Data Types.
PhD thesis, Department of Computer Science, University of Toronto, 1975.
Computer Science Research Group Tech. Report CSRG-59.

[21]
M. Hamilton and S. Zeldin.
Higher Order Software -- a Methodology for Defining Software.
IEEE Trans. Softw. Eng. SE-2(1):9-32, March, 1976.

[22]
S.L. Hantler and J.C. King.
An Introduction to Proving the Correctness of Programs.
Comp. Surveys of the ACM 8(3):331-353, September, 1976.

[23]
C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
CACM 12(10):576-583, October, 1969.

[24]
J. D. Ichbiah, J. P. Rissen, J. C. Heliard.
The Two-Level Approach to Data Independent Programming in the LIS System Implementation Language.
North-Holland Pub. Co., Amsterdam, 1974, .

[25]
W. Teitelman.
Interlisp Reference Manual.
Technical Report, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California, 94304, October, 1978.

[26]
S. Katz and Z. Manna.
Logical Analysis of Programs.
CACM 19(4):188-206, April, 1976.

[27]
J.C. King.
A Program Verifier.
PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, September, 1969.

[28]
D. E. Knuth.
Structured Programming with Go To Statements.
Comp. Surveys 6(4):261-301, December, 1974.

[29]
K. N. Levitt, L. Robinson, B. A. Silverberg.
The HDM Handbook, Vol III: A Detailed Example in the Use of HDM.
Technical Report, SRI International, June, 1979.

[30]
B. H. Liskov, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder.
Abstraction Mechanisms in CLU.
Technical Report, MIT Laboratory for Computer Science Computation Structures Group, July, 1978.
Memo 161.

[31]
B. H. Liskov and V. Berzins.
An Appraisal of Program Specifications.
MIT Press, 1979, .

[32]
B. H. Liskov and S. Zilles.
Specification Techniques for Data Abstraction.
IEEE Trans. Softw. Eng. SE-1(1):7-19, March, 1975.

[33]
R.L. London.
Perspectives on Program Verification.
In R. T. Yeh, editor, Current Trends in Programming Methodology, Vol. II - Program Validation, pages 151-172. Prentice-Hall, 1977.

[34]
R.L. London.
Proof Rules for the Programming Language EUCLID.
Acta Informatica 10, 1978.

[35]
J. McCarthy.
Towards a Mathematical Science of Computation.
In Congress 1962, Amsterdam, The Netherlands: North-Holland, pages 21-28. IFIP, 1961.

[36]
P.M. Melliar-Smith and R.L. Schwartz.
Hierarchical Specification of the SIFT Fault Tolerant Flight Control System.
In Conference Proceedings No. 303--Tactical Airborne Distributed Computing and Networks. North Atlantic Treaty Organization, June, 1981.

[37]
P.M. Melliar-Smith and R.L. Schwartz.
Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System.
Technical Report CSL-133, SRI International, Menlo Park, California 94025, January, 1982.
Supported by NASA Langley Research Center under Contract NAS1-15428.

[38]
H. D. Mills.
How to Write Correct Programs and Know It.
In Intl. Conf. on Reliable Software, 13-15 April 1975, Los Angeles, CA, in SIGPLAN Notices, pages 363-370. , June, 1975.

[39]
M.S. Moriconi and R.L. Schwartz.
Automatic Construction of Verification Condition Generators from Hoare Logics.
In Proceedings of the 8th International Colloquium on Automata, Languages, and Programming. ICALP, July, 1981.
Held in Haifa, Israel.

[40]
M.S. Moriconi.
A Designer/Verifier's Assistant.
Technical Report CSL-80, SRI International, Menlo Park, California 94025, 1978.

[41]
P.G. Neumann.
Computer Security Evaluation.
In National Computer Conference 1978, Vol. 47, pages 1087-1095. AFIPS, 1978.

[42]
D. L. Parnas.
Information Distribution Aspects of Design Methodology.
In Information Processing 71, pages 339-344. IFIP, 1972.

[43]
D. L. Parnas.
A Technique for Software Module Specification with Examples.
Comm. ACM 15(5):330-336, May, 1972.

[44]
D. L. Parnas.
On the Criteria to Be Used in Decomposing Systems into Modules.
Comm. ACM 15(12):1053-1058, December, 1972.

[45]
D. L. Parnas.
On a 'Buzzword': Hierarchical Structure.
In Information Processing 74, pages 336-339. IFIP, 1974.

[46]
B.K. Reid and J.H. Walker.
SCRIBE--Introductory User's Manual, Second Edition
Carnegie-Mellon University, 1979.

[47]
L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena.
A Formal Methodology for the Design of Operating System Software.
Prentice-Hall, New York, 1977, .

[48]
L. Robinson and K. N. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Comm. ACM 20(4):271-283, April, 1977.

[49]
L. Robinson.
The HDM Handbook, Vol I: The Foundations of HDM.
Technical Report, SRIInternational", June, 1979.

[50]
O. M. Roubine and L. Robinson.
The SPECIAL Reference Manual.
Technical Report, SRI International, CSL-45, January, 1977.

[51]
R.E. Shostak, R.L. Schwartz, and P.M. Melliar-Smith.
STP: A Mechanized Logic for Specification and Verification.
Technical Report CSL-130, SRI International, Menlo Park, California 94025, 1981.
To appear in the Proceedings of the 6th Conference on Automated Deduction.

[52]
R.E. Shostak, P.M. Melliar-Smith, and R.L. Schwartz.
Techniques for Automatic Deduction.
Technical Report, SRI International, Menlo Park, CA 95025, October, 1981.
Prepared for Air Force Office of Scientific Research, Captain William Price, Technical Monitor. Contract Number F49620-79-C-0099.

[53]
R.E. Shostak.
Deciding Combinations of Theories.
Technical Report CSL-132, SRI International, Menlo Park, California 94025, February, 1982.

[54]
B.A. Silverberg.
A User's Guide to the INTERPG Parser Generator System.
Technical Report CSL-78, SRI International, Menlo Park, California 94025, September, 1978.

[55]
B. A. Silverberg, L. Robinson, K. N. Levitt.
The HDM Handbook, Vol II: The Languages and Tools of HDM.
Technical Report, SRI International, June, 1979.

[56]
B.A. Silverberg.
An Overview of the SRI Hierarchical Development Methodology.
Technical Report CSL-116, SRI International, Menlo Park, California 94025, July, 1980.

[57]
Military Standard JOVIAL J73 (Draft)
SofTech, Inc., 460 Totten Pond Road, Waltham, MA 02154, 1979.
A draft version of MIL-STD-1589A.

[58]
D.L. Shirley.
Standard Computer Programming Language: JOVIAL J73 Subsetting
Rome Air Development Center, Griffiss AFB, NY 13441, 1973.

[59]
Military Standard JOVIAL (J73/I), MIL-STD-1589 (USAF)
Department of the Air Force, 1977.

[60]
Military Standard, JOVIAL (J73), MIL-STD-1589A, (USAF)
U.S. Air Force, Department of Defense, 1979.

[61]
Military Standard, JOVIAL (J73), MIL-STD-1589B (USAF)
U.S. Air Force, Department of Defense, 1980.

[62]
W.A. Wulf, R.L. London, and M. Shaw.
An Introduction to the Construction and Verification of ALPHARD Programs.
IEEE Trans. Soft. Eng., Vol. 2 :253-265, December, 1976.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. 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.

END

FILMED

2-83

DTIC