12

RADC-TR-77-229
Final Technical Report
June 1977

A VERIFICATION SYSTEM FOR JOCIT/J3 PROGRAMS (RUGGED PROGRAMMING ENVIRONMENT - RPE/2)

Stanford Research Institute

Approved for public release; distribution unlimited.

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

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

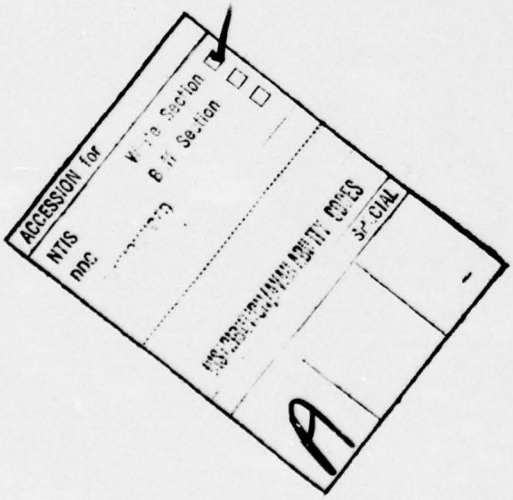
This report has been reviewed and approved for publication.

APPROVED:

JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED:

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division



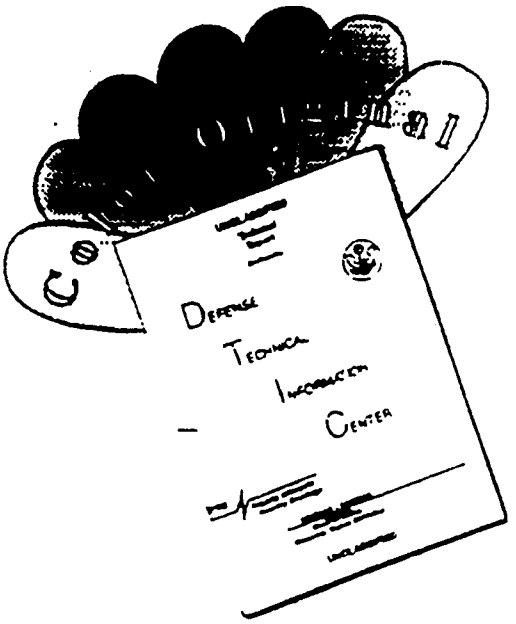
FOR THE COMMANDER:

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

Do not return this copy. Retain or destroy.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-229	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A VERIFICATION SYSTEM FOR JOCIT/J3 PROGRAMS (RUGGED PROGRAMMING ENVIRONMENT - RPE/2),	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 9 Apr 76 — 8 Apr 77
	6. PERFORMING ORG. REPORT NUMBER: SRI-5042-1
7. AUTHOR(s): B. Elspas, R. E. Shostak, J. M. Spitzen	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0204
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford Research Institute, 333 Ravenswood Avenue, Menlo Park CA 94025	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810273
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: July 1977
	13. NUMBER OF PAGES: 113
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:
Captain John M. Ives (ISIS)

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

Program verification	Higher order language reliability
Proof of correctness	Software tools
JOVIAL (JOCIT) language	Mechanical deduction
Rugged programming environment	Inductive assertions
Verification/Validation	Assertion language

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

This report describes work done during the second year of a research and development program aimed ultimately at a "Rugged Programming Environment" for JOVIAL. The RPE/1 verification system designed and built during the first year has been greatly extended and improved in several ways. The basic method of verification remains the same--that of inductive assertions.

The input processor has been modified to handle virtually all of JOCIT instead of the small subset covered by the RPE/1 system. The overall speed of

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

332500

verification has been increased by a factor of over 25. Ease of user interaction with the system has been greatly enhanced by adding facilities for carrying out and saving partial proofs of programs, for extending the assertion language, and for enabling top-down and botton-up proofs for well-structured programs. Moreover, the entire system has been translated into MACLISP, the system files have been transferred to the RADC-MULTICS Honeywell 6180 computer, and a sample verification (shown in the report) has been carried out entirely on the RADC computer.

CONTENTS

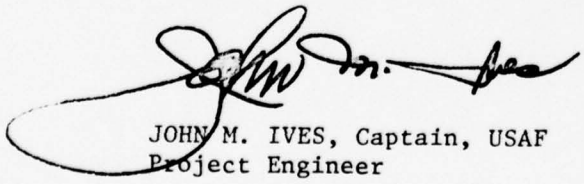
ILLUSTRATIONS

ACKNOWLEDGMENTS

R.S. Boyer has been a major contributor to this work. We thank O. Roubine who developed the SHOWGRAMMAR grammar analysis package and J S. Moore who developed the MACLISPIFY translation system and contributed to the development of the parser generator.

EVALUATION

This effort continues the successful development of a software proof of correctness system started under F30602-75-C-0042, the Rugged Programming Environment. Where this first effort proved the feasibility of applying the proof of correctness concept to a JOVIAL-like Higher Order Language, the current effort applied it to essentially the whole JOVIAL (JOCIT) language. And in doing so, SRI, the contractor, increased the working efficiency, flexibility, and usability by their software rewrite and inclusion of philosophical and user oriented features. These factors, together with the movement of the rewritten files to the RADC Multics System (which was done beyond the intended scope of the statement of work) now allows accessibility to two sources, the SRI TENEX system and the RADC Multics system, for the serious user to attempt verification of his JOVIAL programs. While the system is yet to be simplified so that the general programmer can conveniently access it, it is a major tool for inclusion in the Disciplined Programming Environment being developed under RADC's Software Cost Reduction Program.



JOHN M. IVES, Captain, USAF
Project Engineer

I INTRODUCTION AND CONCLUSIONS

This report describes our second year of research and development effort aimed at making formal proof of program correctness by means of inductive assertions a practical technique for JOVIAL programming. During the first year (September 1974 - October 1975), we developed a pilot system--the Rugged Programming Environment (RPE/1), written in INTERLISP, capable of formally verifying small programs written in a limited subset of JOVIAL. Having demonstrated the feasibility of proving correctness for a limited subset of JOVIAL programs, we set out in April 1976 to extend the power, utility, and general capabilities of our verifier in several directions.

The RPE/1 and RPE/2 verifiers have much the same overall structure. Both versions employ the method of correctness proof by inductive assertions. There are three major subsystems,

* Parser/transducer
* Verification condition generator
* Deductive system

The parser/transducer and verification condition generator together constitute the input processor of the verification system. The RPE/2 parser/transducer accepts a JOCIT program (annotated with input/output specifications and inductive assertions) from a text file, parses it, and saves the parsed version on a new file. The purpose of the verification condition generator (VCG) is to analyze this parsed form of the program, thereby creating a set of formulas in first-order predicate calculus whose valdity is a sufficient condition for consistency of the program with its specifications. The parser incorporates a complete syntactic characterization of JOCIT, while the VCG embodies knowledge of JOCIT semantics in order to generate the verification conditions (VCs).

The deductive system carries out the actual proof of validity of VCs under user guidance, making use of several subsystems for performing special kinds of deductive inference. The top-level of the deductive system, with which the user interacts, is a proof supervisor executive based on the method of analytic tableaux [16]. This Tableaux Executive also provides for the display of the dynamic proof state, maintains an audit trail for future examination of the details of the proof, and can save partial proofs for future completion. VCs that are propositionally valid are handled entirely by the tableaux executive acting in a completely automatic mode. Formulas involving logical quantifiers are either handled automatically by the tableaux system, or by user-supplied instantiation. A mechanism is also provided in the Tableaux Proof System for the invocation of axioms or previously proved formulas, with instantiations of free variables supplied by the system user. Formulas whose validity depends on the interpretation of equality/inequality relations, algebraic operations, or function symbols can be demonstrated by user invocation of either (a), a specialized decision mechanism for an extension of Presburger arithmetic, or (b), an expression simplifier for algebraic formulas which need not be limited to Presburger arithmetic. The Presburger mechanism also constructs concrete numerical counterexamples for invalid formulas. Both subsystems (a) and (b) are essentially automatic, once they have been invoked by the user through the Tableaux Executive.

Communication between the three major subsystems of the verifier takes place by the creation and reading of files, i.e., the original JOCIT program to be verified, the parsed form of the program, and the VCs for that program.

Our efforts were devoted to four major tasks, which may be summarized as follows:

* Task A - Modify the input processor to handle as much as possible of JOVIAL (JOCIT version).
* Task B - Increase the overall speed of verification by at least a factor of 2, and if possible by a factor of 5.
* Task C - Greatly enhance the ease of user interaction with the system by developing facilities for carrying out and saving partial proofs of programs, for extending the assertion language, and for enabling top-down/bottom-up proofs for well-structured programs.

* Task D - Begin transfer of our verification technology to the Air Force by implementing enough of the verification system on the RADC-MULTICS computer to permit verification of a simple program entirely on that machine.

The structure of this report reflects these tasks: the next four sections deal with the issues involved in meeting the respective requirements of the four tasks.

The main accomplishments of this project have been:

* The extension of program verification techniques for the first time to a large, fairly complex, real programming language in wide current use, and the demonstration of their feasibility in that domain.
* The development of an extremely flexible deductive system that, without compromising generality or processing speed, is able to handle user-guided hierarchical proofs of correctness, has facilities for saving and reentering partial proofs, and is easily integrated with special-purpose deduction modules.
* The development and implementation of a new, efficient algorithm for deciding validity for a large class of mathematical formulas. (The formulas are in an extension of universal Presburger arithmetic, described in Section III, Subsection C.) This algorithm also constructs numerical counterexamples for invalid formulas supplied to it.

Section II describes the modification of the parser/transducer to accommodate JOCIT syntax (except for certain implementation-dependent features), and the extension of the verification condition generator to handle the semantics of all features specifically required by the Work Statement as well as a good many others.

Section III deals with the modifications we made to increase the speed of verification, both in the machine time and the user time required. It also presents some measurements of the time required for each of the phases of verification. An overall speedup by a factor of approximately 27 compared to the RPE/1 system very amply fulfilled the goal of Task B.

Section IV describes the interactive features that have been added to the system. Most of these are concerned with user interaction in the deductive system. The RPE/1 system attempted to use a set of automatically invoked, fixed deductive strategies incorporated in a "goal-driven" deductive

system. That system turned out to be both extremely slow and cumbersome for all but the simplest deductions. It was also incapable of handling logical quantifiers or of instantiating axioms (except those built-in as procedural strategies). The RPE/2 deductive system is based on the method of analytic tableaux (q.v. Section IV, Subsection B) and eliminated many deficiencies of its predecessor. A simpler version of analytic tableaux had been implemented under RPE/1, but it had not been integrated with other deduction tools. Most of the increase in speed of user interaction with the system was due to this improved tableaux facility. Another aspect of user interaction discussed in Section IV is the user facilities associated with procedural abstraction and the carrying out of top-down (or bottom-up) proofs for suitably structured programs.

Finally, Section V describes the steps that led to the carrying out of a sample verification on the RADC-MULTICS Honeywell 6180 computer. This required rewriting the system in MACLISP, transferring system files to RADC-MULTICS, and actually demonstrating the system. This section of the report also includes a detailed discussion of some supporting software (developed in part under this contract) that greatly facilitated the actual translation process--so much so that it became feasible to translate the final RPE/2 system, instead of simply the much more primitive RPE/1 version.

Appendix A shows a detailed BNF grammar for JOCIT developed in connection with Task A.

Appendix B is a glossary of the deductive system including the commands that the user can issue and the parameters that the user can set.

Appendix C details a verification run at RADC-MULTICS.

The two years we have devoted to the development of practical verification tools for JOVIAL have led us to the following conclusions:

* Program verification continues to be a promising technique which, when used in conjunction with modern structured design and formal specification methodologies, will ultimately reduce the cost of developing and maintaining Air Force software systems.
* Program verification is applicable to complex real-world languages such as JOVIAL. The associated problems of applying

verification to such languages--the development of supporting parsers and verification condition generators--can be solved in straightforward ways with existing technology. It is also straightforward to use languages such as JOVIAL harmoniously with formal design disciplines such as the SRI Hierarchical Methodology [13].

* The bottleneck in developing an automatic verification technology is the development of more potent deductive mechanisms than are currently available. A related problem area is the difficulty of inventing the inductive assertions required for correctness proofs. The development of more powerful deduction tools will do much to overcome this problem. This area has great promise and potential but urgently needs further attention.

* Given the lack of a competent automatic deduction system for the mathematics of computer programs, the use of verification technology in practice requires the development of semiautomatic deductive facilities. The user interface of such facilities must be carefully engineered to permit flexible and informed user control over the myriad details of program theorem proving. The Tableaux Proof System is a major step toward such facilities.

Consequently, we hope to continue our efforts in three major areas. First, we will expand the deductive facilities of the system. In particular, we will increase the sophistication of our techniques for supervising interactive proofs, enhance the power of the automatic deductive mechanisms in areas such as quantification, nonlinear arithmetic (q.v. Section III, Subsection C), and canonical form rewriting systems.

Second, we will couple modern design techniques such as [13] and [9] with our verification system. This will involve developing a superset of the JOCIT language allowing the implementation of programs with hierarchical and modular structure. We intend to describe how advanced design tools under development at SRI, such as [14], can be applied to this superset language to provide a coherent environment for designing verifiable modular programs.

Third, we intend to tune and make robust all aspects of user interaction with the system to maximize ease of use. A major design criterion will be to develop a system that can be widely used outside of SRI.

II MODIFICATIONS TO THE INPUT PROCESSOR

A. Introduction

Task A, Item 4.1.1 of our Statement of Work called for the following effort:

> The contractor shall modify the current input processor from the ability to handle programs written in the JOVIAL J3/J73 subset defined under contract F30602-75-C-0042 to handle, in the greatest extent possible, the complete JOVIAL (J3). The following features, among others yet to be accommodated are the file- and table-declarations, the 'alternative', 'exchange', and 'return' statements, file I/O operations, and multiple entry points on procedures.

The input processor of our verification system comprises two successive stages of processing:

* Parsing and Transduction
* Verification Condition Generation (VCG)

The first is concerned only with the syntactic recognition of JOVIAL constructs, while the VCG stage makes use of the semantic aspects of JOVIAL. In Section II, Subsection B, we discuss the modification of the parser/transducer to fulfill the requirements of Item 4.1.1. Section II, Subsection C describes the changes made to the verification condition generator to handle the features listed as well as others. This subsection also points out ambiguities in the JOCIT language definition [3] that were made apparent by our efforts.

B. The Parser/Transducer

The first step was to define precisely the version of JOVIAL (J3) to be accommodated by the input processor. Our effort was based on the JOCIT version of JOVIAL, as documented in [3]. This document was, therefore, used for the definition of JOCIT syntax and semantics.

In the next three subsections we discuss, in turn, construction of a formal grammar for JOCIT, the building of a parser/transducer by means of a parser-generator, and how to use the parser-transducer on JOCIT programs.

1. Grammar Construction

Our previous parser/transducer used the Earley algorithm [5], and was based on a syntactic description of a subset of J3/J73 in the form of modified BNF syntax equations [7]. In the new version we proposed to make use of a much more efficient parsing algorithm, effective with SLR grammars (see [4]), to meet the requirements of Task B. A detailed discussion of parser techniques and their speed capabilities appears in Section III of this report. The next step was to express the syntactic constraints of JOCIT in the form of BNF equations that would form an SLR(1) grammar. SLR(1) was chosen for reasons of parsing efficiency.

We first rewrote the semiformal descriptions presented in [3] as strict BNF equations, subsuming nonterminals under common forms when necessary or convenient. The resulting formal grammar for JOCIT appears in Appendix A. The highly readable form in which the grammar is shown there was produced by a grammar display package due to O. Roubine of the SRI Computer Science Laboratory staff. This package, developed under another DoD contract, is available on file [SRI-KL]<ROUBINE>SHOWGRAMMAR.COM. We should note that this grammar covers all of JOCIT as defined in [3] with the following system-dependent exceptions:

* MONITOR
* Direct JOVIAL code (including direct:assign)
* COMPOOL
* MODE directive
* DEFINE directive

Several passes over the grammar were required to achieve this final form because it is usually not apparent whether a grammar is SLR(1). In fact, the best test for the SLR(1) property is to submit the grammar to an SLR(1) parser-generator. Three or four such design passes were required for the JOCIT grammar; minor modifications to the grammar were made whenever conflicts were discovered.

2. Using the Parser Generator

Our parser-generator has three phases: the first and third are implemented in INTERLISP and the second in ECL. The first phase prepares, from Lisp data structures, the input to the second phase, which is an SLR parser system developed at Harvard by Griffiths, Shostak and Townley and implemented in ECL [10]. The tables produced by ECL are then processed by the final INTERLISP phase, which converts them into INTERLISP or MACLISP code and combines the result with user specified transductions and lexical analysis routines to produce the final parser. Here is a diagram that describes this process:

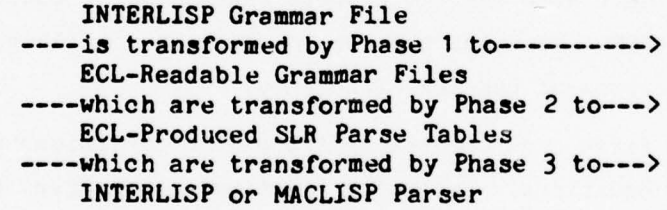


Figure 1. Parser Generation

Compiled code for this parser-generator is stored in [SRI-KL]<ROUBINE>INTERPG.COM. In addition to the grammar, the parser-generator takes as input a description of the lexical tokens of JOCIT, from which a finite state lexical analyzer is synthesized to build input for the parser. The purpose of the parser/transducer is not merely to parse the subject JOCIT program but also to output a transduced version of the program as a Lisp data structure. This parse is the input for the second phase (VCG) of the input processor. The data structure that contains the formal grammar also provides transduction augments for each nonterminal of the language. These augments are user-supplied and define the structures of transductions produced by the parser.

The construction of the parser begins with loading the parser-generator into INTERLISP by calling:

```
_LOAD(<ROUBINE>INTERPG.COM)
```

Several explanatory messages are typed out at the user's terminal. If the file (JOCIT$) containing the grammar is ready, the user proceeds according to the instructions; if, on the other hand, there have been changes in the grammar file since the last construction of a parser, one must invoke the ECL program that rebuilds the grammar file by calling (from within INTERPG.COM):

```
_ECLPR(JOCIT)
```

This action loads the file JOCIT$ (containing the grammar, transductions, and lexical analyzer description) into the environment. If the user then responds with "YES" to a query asking whether ECL should build new tables, ECLPR will perform that action.

Before we continue our description of the user actions needed to construct the lexical analyzer portion of the parser, we digress briefly to make some general remarks about its design and function. The finite-state machine that actually performs lexical analysis during parsing is itself synthesized by the parser-generator from a data structure specifying the lexical level of the grammar. This particular data structure, FSM, was hand-designed by analyzing the various ways we may validly combine individual characters to produce JOCIT lexemes, such as integers, fixed-point numbers, floating-point numbers, JOCIT names, octal constants, character constants, status constants, special one-character lexemes ($, I, A, S, U, P, V, R, etc.), and special two-character lexemes (**, ($, $), (*, *), and ==).

The resulting finite-state machine specification had 48 states, with 31 terminal states. It is shown diagrammatically in Figure 2, and resides in the grammar file JOCIT$ as the binding of the Lisp variable FSM.

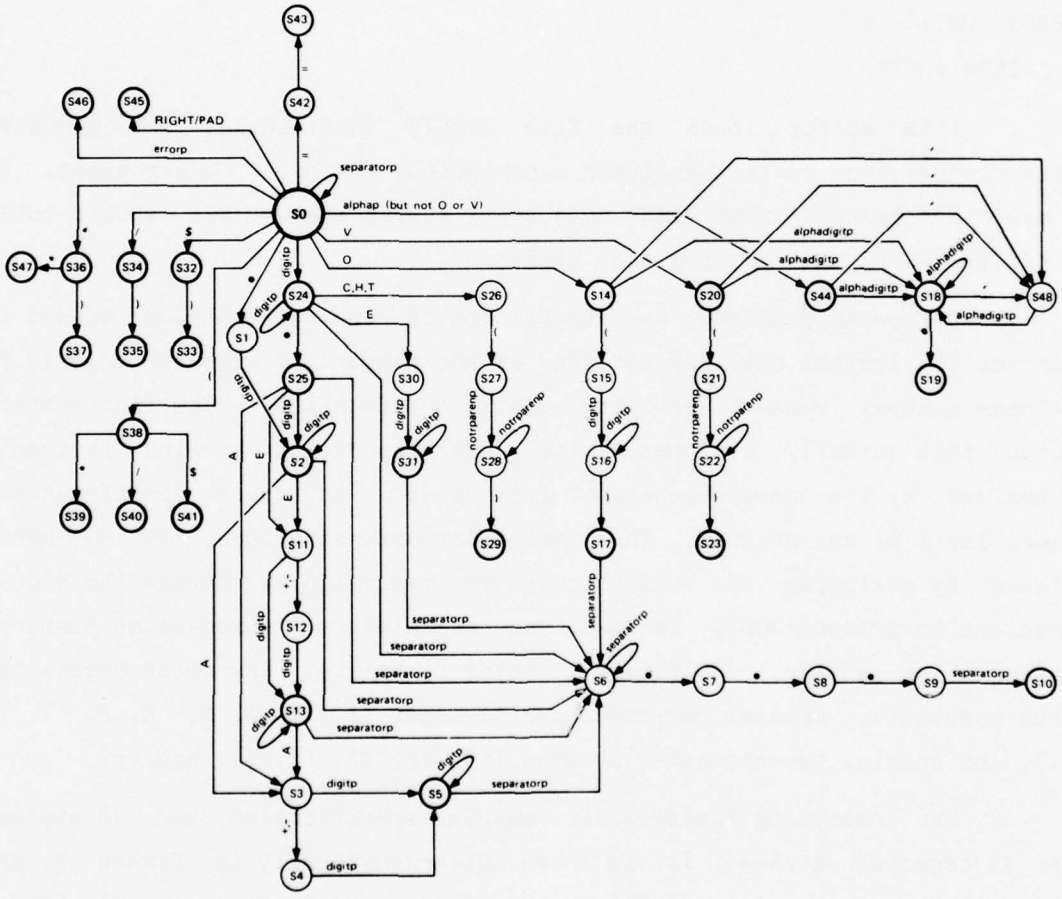


Figure 2. Finite State Machine Lexical Analyzer

KEY: 1. S0 is the initial state.

2. Boldface indicates final states.

3. Lexemes associated with final states are:

S2, S13, S25	FLOATING\CONSTANT
S5	FIXED\CONSTANT
S10	RANGE\PRFX
S14, S20, S44	LETTER
S17	OCTAL\INTEGER
S18	NAME
S19	NAMEDOT
S23	STATUS\CONSTANT
S24, S31	DECIMAL\INTEGER
S29	CHARACTER\CONSTANT
S32	$
S33	$)
S34	/
S35	/)
S36	*
S37	*)
S38	(
S39	(*
S40	(/
S41	($
S42	=
S43	==
S45	RIGHT\PAD
S46	ILLEGAL:LEXEME
S47	**

4. The meanings of the transition predicates are:

separatorp	any separator character, i.e. space, linefeed, carriage return, formfeed, or eol
alphap	any alphabetic character
digitp	any digit
alphadigitp	any alphabetic character or digit
notrparenp	any character other than ')'
errorp	any character which is illegal as the initial character of a JOCIT lexeme, i.e. any character not an alphabetic, a digit, a RIGHT\PAD (which indicates end-of-file on parser input), a separator, or one of =, ., (, $, /, or *.

The actual construction of the lexical analyzer is carried out mechanically in JOCIT$ by calling the function MAKE.NEXTTYPE. This action creates a function definition for an INTERLISP function, NEXTTYPE, which is the lexical analyzer portion of the parser. Also created at the same time by MAKE.NEXTTYPE is a function (another part of the lexical analyzer) called BEGINCHECK. BEGINCHECK assembles alphabetical tokens and also, by means of buffering and look-ahead, distinguishes two uses of BEGIN in the JOCIT grammar that are not distinguished by our SLR(1) grammar. We had to provide this feature in order to make the grammar parseable with SLR(1) techniques. The construction of the lexical analyzer also uses a description of the final states of the finite state machine which is given in the variable FINALSTATES. FINALSTATES is a list of triples, where the first member of each triple is the state name (e.g., S17); the second member is the generic state name (e.g., octal\integer); and the third member is the particular Lisp function used to pack the characters together to make that kind of token. Examples of these "packing" functions are NCONCAT, BEGINCHECK, and STATE.

In operation, the lexical analyzer works as follows:

1. A function PEEKBUF in the analyzer reads a character from the file being parsed, and places it into a BUFFER.

2. Another lexical analyzer function NXTCHR reads the character into a second buffer, called INPUTS2.

3. Characters are eventually read into a third buffer, INPUTS, from which they are packed into accepted lexemes by one or another of the "packing" functions named above.

The extra buffers are needed because the lexical analyzer is always trying to recognize as a legal token the longest possible string of consecutive characters. If, for example, the string ABCDE is being read, where ABC, and ABCD both determine legal final states but ABCDE is illegal, then eventually ABCD finds its way into INPUTS but E remains in INPUTS2 to be placed back in the BUFFER before recognition of the next lexeme.

The parser-generator finishes by writing a file containing the synthesized parser. The user may choose to create either an INTERLISP or a MACLISP parser. We wanted to construct a MACLISP parser so that our whole system would be able to run in the MACLISP environment available on the RADC-

MULTICS computer. The file containing our parser/transducer is called JMAC.MAC, and it is directly loadable into any MACLISP environment.

The principal function of our parser is called JMAC. Its use is described in the next subsection.

3. Use of the Parser/Transducer

In order to use a parser/transducer synthesized by the above parser-generator, one must first have available a text file containing a JOCIT program to be parsed. Suppose that such a JOCIT program is on a file called "testprog.joc". One first loads the synthesized parser on top of a MACLISP environment (containing our function library) by typing '(bload jmac mac)'. At MULTICS the corresponding command is '(load "jmac.mac")'.

The user should follow the steps shown below. First, initialize the parser by calling:

```
(jmacinitialize)
```

Next, apply the parser/transducer to the selected file (taking care to save the result on a Lisp variable, say "parse"), by typing:

```
(setq parse (jmac '(testprog joc) 0))
```

When parsing is completed, the transduced program is typed out at the terminal. The user will usually wish to save the result of parsing on a Maclisp-loadable file. In our system this can be done by setting, say testvars, to '(parse), and calling (mfile test). This makes a file with the binding of parse saved on it.

When the parser aborts because of a syntax error in the JOCIT text file, an error message indicates the first illegal lexeme detected. The INTERLISP parser/transducer has interactive debugging aids that assist the user in correcting the error. (Unfortunately, the I/O and text editor features of MACLISP do not permit that flexibility. However, one may obtain the similar data by tracing the function BEGINCHECK and reexecuting the parse.)

C. Verification Condition Generator

1. Introduction

The semantic description of JOCIT used by our system is embodied in the second part of the input processor--its Verification Condition Generator (VCG). The VCG design is based on the same document [3] as the input transducer. The only difference is that while the syntactic description of JOCIT in [3] is in the form of BNF syntax equations, the semantic description is in natural language. As always with natural language specifications, the possibility of misinterpretation cannot be dismissed. Moreover, some aspects of JOCIT are inherently undefined, or they are dependent on the machine implementation. Examples of such incompletely specified aspects of JOCIT semantics are the following:

* The order of evaluation of subexpressions; see [3], p. 3-13,
* Transfer of control into a FOR statement; see [3], p. 4-21, and
* Evaluation of a switch that involves a function call which reinvokes the same switch; [3], p. 4-7.

We hope that the predicate transformer specifications given below are sufficiently unambiguous to be checked by JOCIT compiler experts.

The basic purpose of the VCG is to map a JOCIT program--together with formal specifications--into a set of logical formulas expressing the necessary and sufficient conditions for consistency between the program text and the specifications. The VCG operates with predicate transformers on the abstract version of the program to be verified. The basic predicate transformer is realized by a function WP (standing for "weakest precondition") that computes the weakest liberal precondition P for each pair (S,Q) where S is an executable JOCIT statement (or declaration) and Q is a logical assertion. P is said to be the "weakest liberal precondition for (S,Q)" if the truth of P before executing S implies that Q will hold after executing S, provided S terminates. The use of liberal preconditions rather than strong preconditions reflects the intent to prove partial correctness. Proofs of termination are carried out independently.

The verification condition generator has at its top level a user-invoked function, VCG, which takes as its single argument a Lisp variable bound to the (abstract) main program text for which VCs are to be computed. The parser/transducer will have produced the abstract program from an annotated JOCIT program. For example, the Lisp variable might be called MAIN. To compute VCs for MAIN, the user types in:

(VCG MAIN)

and the result of this invocation of VCG will be to bind a new Lisp variable, MAIN*, to a computed list of VCs for MAIN. In addition, any subprograms, procedures, functions, or closed subroutines contained within MAIN will be identified and analyzed, and their VCs computed, as explained below. In each case, the VCs are bound to the Lisp variable formed by appending the character * to the name of that particular subentity.

For each control path lying between successive assertion points in the main program, the function VCG computes a formula (VC) in predicate calculus expressing a condition of correctness for that path. Let the program text between successive assertions p and q be denoted by {S}. We represent the tagged path by p{S}q. The VC for path p{S}q is given by:

p implies WP[S; q]

VCG systematically considers the whole main program supplied to it and delivers as its result a list of such VCs--one for each elementary path between assertions. The function VCG does this by beginning with the main program's output assertion and applying the predicate transformer WP successively to program statements preceding that assertion until some point with an attached assertion p is reached. At that point, a single path VC will have been constructed, and a free variable called "VCS" (local to VCG) starts to accumulate such VCs. Construction of the next path VC is initiated, with the assertion p now playing the role of the output assertion. This process continues until all assertion points have been traversed and the main program input assertion attained. The last path VC to be constructed is, therefore, the VC for the initial path segment lying between the input assertion and the first assertion point. The list of VCs accumulated on the Lisp variable VCS is assigned to the external output variable MAIN*, an

appropriate message is printed out on the user's terminal, and VCG is completed.

The next subsection contains brief discussions of how WP acts on each of the types of statements that were present in the RPE/1 subset of JOVIAL. The subsection following that describes in detail the action of WP for features of JOCIT that were added during the present phase. Most readers will prefer, at least on a first reading of this report, to omit these two highly detailed subsections and proceed to Section III.

2. Basic Features of the Precondition Operator WP

The primitive statement types handled by WP in the RPE/1 system were:

* assignment (to simple variables)
* assignment (to arrays)
* *simple conditional statement*
* compound statement
* iterative statement (a do-while type of statement from J73)

We shall need to refer both to JOCIT statements and their transduced (abstract) counterparts. To make this correspondence obvious, we introduce the metalinguistic convention that form' shall stand for the transduction of the JOCIT expression (statement, declaration, or other expression) represented by form.

For each of the these five statement types, we exhibit below the following information:

* The JOCIT syntax for s (q.v. Appendix A)
* The abstract syntax s'
* The definition of WP[s;q]

(a) Let simple:asst be the JOCIT statement:

```
lhs = rhs $
```

[In abstract syntax: simple:asst' = (:= lhs' rhs')]

```
WP[simple:asst';q'] = the result of substituting rhs'
                      for each (free) occurrence of lhs'
                      in q'.
```

(b) Let array:asst be the JOCIT assignment to a component of an array (possibly multi-dimensional):

```
array:ref = rhs $
```

[In abstract syntax:

```
array:asst' = (:= (name' index-list) rhs')
```

where index-list = (SUBSCRIPTS . <a-list-of-indices>)]

Then WP[array:asst'; q'] =

```
WP[(:= name' (CHANGE name' index-list rhs')); q']
```

Thus WP for array assignments is handled by applying WP for simple-variable assignments to a virtual simple variable having as its name the array name. This feature appeared in much the same form in the RPE/1 system.

(c) Let cond:stat be the JOCIT statement:

```
IF bool $ stat
```

[In abstract syntax: cond:stat' = (IF bool' stat')]

WP[cond:stat'; q'] =

```
(AND (IMPLIES bool' (WP stat' q'))
     (IMPLIES (NOT bool') q'))
```

This is identical to the corresponding feature in RPE/1.

(d) Let compd:stat be the JOCIT statement:

```
BEGIN stat1 stat2 ... statn END
```

[Note: each stati contains its own $ terminator]

[In abstract syntax:

```
compd:stat' = (BEGIN stat1' stat2' ... statn')]
```

The semantics are given recursively by:

WP[compd:stat';q'] =

```
WP[stat1'; WP[(BEGIN stat2' stat3' ...) ; q']]
```

(e) Next, we describe the iteration statement. We have augmented the JOCIT language here by adding an assertion. This assertion is syntactically optional but required for verification. The JOCIT syntax is:

```
FOR i = a, b, c {ASSERT bool} $ stat
```

and the corresponding abstract form is:

(FOR i ((ASSERT bool') (a' b' c')) stat').

If the optional assertion is omitted the transducer supplies a vacuous assertion clause, (ASSERT T). Here i stands for any single-character control index, and a, b, c represent any numeric (integer-type) expressions. This syntax differs from that of the iterative statement type in RPE/1--a while-do/do-until form derived from J73. See [7] for details of VCG for the RPE/1 iterative statement. We discuss here only the features that had to be added to the RPE/2 VCG to handle the JOCIT iterative statement.

The semantics of the JOCIT complete-for-statement shown above is defined to VCG by post-transduction processing of the iterative statement into an explicit loop iterative:stat*.

WP[iterative:stat'; q] = WP[iterative:stat*; q]

where iterative:stat* is the compound statement:

```
(BEGIN (:= i* a*) (LABEL g*) (ASSERT bool*)
       stat*
       (:= i* (PLUS i* b*))
       (IF (LTQ i* c*) (GOTO g* NIL)))
```

The system generates the unique name i* for the local control variable i within the loop. Similarly, g* is a system-generated name for the loop return point. The terms bool*, stat*, a*, b*, and c* refer to the transduced loop invariant assertion, loop body, initial value, increment, and final value expressions, respectively, after they have had all occurrences of i replaced by i*. Observe that the order of the events: initialization, execution of the body, incrementing, and branching, follow the sequence shown in [3], p. 4-14.

The translation as actually implemented in VCG is somewhat more complex than indicated above. For one thing, "downward" incrementing (i.e., where b<0) implies a different exit test in the IF-clause of the translation, viz., (GTQ i* c*). The implementation tests b and if it is literally a number, chooses the appropriate inequality operator. If b is not a number (i.e., it is a numeric expression with a run-time value) the VCG implementation generates a conditional expression to cover both possibilities, b>=0 and b<0. Moreover, JOCIT also provides for degenerate

cases of the iterative statement (complete, 3-factor FOR-statement) shown above, viz, the incomplete 2-factor and 1-factor cases. These are also translated to compound statements though we do not show their forms in detail. The incomplete 2-factor FOR-statement (where no exit value, c, is specified) is transduced into a form equivalent to the complete (3-factor) case, with c taking the symbolic value INFINITY. The VCG implementation checks for c=INFINITY and if so, suppresses the exit inequality that would otherwise appear in the post-transduction form. The trivial case of a 1-factor FOR-statement uses an obvious, separate translation to straight-line code. In the 2-factor and 3-factor cases, VCG makes a separate VC for the loop path when it encounters the loop assertion, passing back the loop assertion bool* as the value to be returned by WP.
This concludes our discussion of WP for the statement types handled by the RPE/1 VCG subsystem.

3. Features Added to VCG in RPE/2.

The new features are:

* alternative-statement (ifeith/orif-statement)**
* goto's (to labels, item:switches and index:switches)
* return statement**
* optional entry point to a main program**
* exchange statement**
* assignments to functional modifier variables
* data declarations (simple items, arrays, tables and files)**
* file I/O operations**
* processing declarations (switch declarations, procedures, functions and closes)
* procedure call statement

Items in the above list marked with terminal ** were specifically called for in the Statement of Work.

We now define WP for each of the above types just as was done in the preceding subsection.

(a) Let alt:stat be the JOCIT form:

```
IFEITH bool $ stat ORIF bool1 $ stat1
       ORIF bool2 $ stat2
       ...
END
```

After transduction this assumes the abstract form, alt:stat':

```
(IFEITH (bool' stat')
        (bool1' stat1')
        (bool2' stat2') ... )
```

Then

```
WP[alt:stat'; q'] =
   (AND (IMPLIES bool' (WP stat' q))
        (IMPLIES (NOT bool')
                 (AND (IMPLIES bool1' (WP stat1' q))
                      (IMPLIES (NOT bool1')
                               (AND ... ... )) ...)
```

(b) Let goto:stat be either of the JOCIT forms:

```
          GOTO name $
  [abstract form: (GOTO name NIL)]

      GOTO switch:expr $
  [abstract form: (GOTO sw:name indices)]
```

where name is a statement:name and switch:expr is an indexed expression (e.g., XSW(1) for an index switch). These forms are handled by VCG at a higher level than WP (as are the related RETURN statement and the optional entry to a main program, both of which are discussed below). The following technique is used:

A preprocess phase of VCG scans the program for labels and switch statements. We insist that each label be followed by an assertion because in principle any such point may be targeted by a goto. The preprocessor replaces the goto:stat by the assertion appearing at the targeted point. If the goto target is a label (statement:name followed by a dot), the corresponding (ASSERT bool') statement replaces the goto, except that the form used is (ASSERT bool' STOP), where STOP is a flag to the VCG that this point is not to be used for initiating construction of a new path VC. It merely terminates the current path VC. The actual assertion (ASSERT bool')

following the targeted label will, of course, initiate a new path. For gotos addressing a switch (index:switch or item:switch) a similar device is employed, except that the preprocessor pursues the chain of gotos implied by a succession of switches to an ultimate target label. At each step, an appropriate conditional expression is constructed, involving the possible equality of either the item:switch variable to the values in the item:switch list, or the index:switch variable to 0,1,2,... . Note that it is thereby forbidden to have a switch statement readdress itself, either directly or entirely through a chain of other switches. Although this is probably a restriction on JOCIT practice, we do not believe the restriction to be serious.

(c) RETURN statement

The JOCIT form:

RETURN $

[abstract form: (RETURN)]

may be part of the body of a procedure, function, or close declaration. Execution of the RETURN implies that control will be passed to the exit functions for these entities. Therefore, the same effect is generated in VCG if the form (RETURN) is replaced at each occurrence by an assertion (ASSERT exit-assertion' STOP) much as with the gotos discussed above. The exit-assertion is, of course, derived from the specification attached to the procedure, function, or close in which the return is embedded.

(d) Optional entry to a main program

The syntax for main:program provides for an optional name appearing immediately preceding the final $ terminator. The semantics given in [3], p. 4-7 state that:

> Execution of a program begins with the first statement in the program unless a main program has specified a statement name following the TERM bracket. In this case, program execution begins with the statement bearing the specified label. Of course, this label must be accessible, i.e., it must not be declared within a procedure, function, or close declaration within the main program.

The same semantic effect can be provided to the VCG by simply inserting a goto statement targeting this statement label at the head of the executable code. The VCG preprocessor provides this translation by executing a function TRANS:RETURN on the body before it begins generating VCs.

This feature was implemented in VCG to fulfill the requirement in Task A referring to "multiple entry points on procedures," since, in fact, multiple entry procedures are not part of JOCIT. The optional entry point to main programs is, however, an analogous feature.

(e) Exchange statement

Let exch:stat be the JOCIT statement:

```
var1 == var2 $
```

The abstract form is (EXCHANGE var1' var2'), and the semantics are defined (see [3], p. 4-4) by the sequence of statements:

```
temp1 = var1 $
var1 = var2 $
var2 = temp1 $
```

for the case of exchange of simple variables (for subscripted variables, additional complications may arise from the evaluation order of indices). Accordingly,

```
WP(exch:stat';q') = WP[(BEGIN (:= temp1 var1')
                              (:= var1' var2')
                              (:= var2' temp1)); q']
```

In the implementation of this definition, the variable denoted by temp1 is actually a unique system-generated name (gensym) to prevent the occurrence of name conflicts with any other variables that may be present in the program or its assertions.

(f) Assignments to functional modifier variables

The JOCIT functional modifiers ODD, CHAR, MANT, and POS have been handled specially in VCG to permit WP to process assignment statements where forms such as ODD(XX) appear on the lhs of assignments.

ODD: ODD(XX) refers to the least significant bit of the binary word stored in XX. When ODD(XX) appears on the left side of an assignment, e.g., ODD(XX)=VV $, the effect is to set this least significant bit to (the least

significant bit of) VV. These semantics are captured by defining WP[(:= (ODD XX) VV); Q] as:

WP[(:= XX (SUBTRACT (PLUS XX (ODD VV)) (ODD XX)); Q']

where Q' is the result of substituting VV for each occurrence of (ODD XX) in Q. (The point is to define the semantics in terms of an assignment with a simple identifier on the left-hand-side.)

CHAR: CHAR(FF) refers to the characteristic of the (floating point) number FF. Thus, an assignment to CHAR(FF) (e.g., CHAR(FF)=CC $) changes the characteristic to CC, leaving the mantissa MANT(FF) unchanged. For any floating point number FF:

FF = MANT(FF)*2**CHAR(FF)

Hence, the above assignment makes FF = MANT(FF)*2**CC. These semantics are captured by defining WP[(:= (CHAR FF) CC); Q] as:

WP[(:= FF (TIMES (MANT FF) (EXPT 2 CC))); Q']

where Q' is the result of substituting CC for each occurrence of (CHAR FF) in Q.

MANT: MANT(FF) is analogous to CHAR(FF). Thus, the assignment MANT(FF)=MM makes FF = MM*2**CHAR(FF) and WP[(:= (MANT FF) MM); Q] is defined by:

[(:= FF (TIMES MM (EXPT 2 (CHAR FF)))); Q']

where Q' is the result of substituting MM for each occurrence of (MANT FF) in Q.

POS: POS(FI) refers to the file position pointer of a file object, FI. Thus, POS(FI) is UNDEFINED if the file is not open; it is the integer 0 if the file FI has just been opened; and is otherwise a positive integer. Assignments to POS(FI) move the file pointer. Our VCG defines an internal variable, FI:FILEPTR, for each declared file FI. This internal variable is an alias for (POS FI). That is, assignments to (POS FI), whether they result from explicit assignment statements, or from POS(FI) being an actual output parameter of a procedure or function, or from POS(FI) being a parameter in an input list of an INPUT statement, also result in the same virtual assignments to the variable FI:FILEPTR. (File I/O operations are discussed below.)

Thus, WP[(:= (POS FI) PP); Q] can be defined as:

WP[(:= FI:FILEPTR PP); Q']

where Q' = the result of substituting PP for each occurrence of (POS FI) in Q.

Assignments to the other types of JOCIT functional modifiers, viz, BIT, BYTE, NENT, ENTRY, and ENT are not presently accommodated by the VCG. Attempts to use them on the lhs of assignments will produce an error message from VCG.

(g) Data declarations

We consider separately these various types of data declarations provided in JOCIT:

* simple item declarations
* array declarations
* table declarations
* file declarations

(g.1) Simple item declarations

Simple items comprise numeric (integer, fixed- and floating-point numbers), Boolean, character-constant, and status-constant items. In each case, the JOCIT syntax is mapped into an abstract form beginning with the key word ITEM, and other portions of the transduced declaration will identify it as an item of type: I (integer), A (integer or fixed-point item, depending on the presence or absence of an integer declaring the number of fractional bits used in the representation), F (floating), B (Boolean), H/T/C (character-constant of type Hollerith, Transmission, or ASCII), or S (status-constant).

This information can be useful in verification: the deductive system may need to know the type of a variable to justify its use in particular contexts. In each case, the action of the subfunction of WP that handles item declarations is to assert a post-transduction logical version of the information contained in the declaration. This effect can be obtained by making WP[item:decl; q] return the expression:

(IMPLIES item:decl* q)

where item:decl* is the post-transduction forrrangitem:decl'. In practice we

have arranged to collect the post-transduction forms, item:decl*, in one large conjunction with the input assertion for the main program (or other such program unit, e.g., subprogram, procedure, etc.).

These post-transduction versions, item:decl*, vary from one type of item to another. We have attempted to anticipate what types of information will actually be useful in verification and to carry over such information into the post-transduction forms. For example, a declaration (for an unsigned integer II):

ITEM II I 20 U $

becomes, after transduction and post-transduction transformation:

item:decl* = (AND (INT II) (GTQ II 0))

In the present system, we have not attempted to use detailed number representation data; hence, the information "20 bits per item" is not carried over. In a later version of our system, where the semantics of machine representation may become significant, we will modify these forms. Presets to values are currently handled, however. Thus, the signed integer declaration:

ITEM JJ I 15 S P 1000 $

which provides for initialization of JJ to the value 1000, maps over into:

item:decl* = (AND (INT JJ) (EQ JJ 1000)).

Note that the clause (GTQ JJ 0) is absent because JJ was declared signed.

Floating item and fixed item declarations are mapped into logical forms like (REAL FF). We are not attempting to distinguish between fixed and floating-point items, both types are treated as real numbers in the deductive system. Presets are handled just as they are for integer items.

Boolean items are mapped into forms such as:

(AND (BOOLEAN BB) (EQ BB 0))

or simply,

(BOOLEAN BB)

if the preset is absent. The deductive system can be informed (by means of an axiom) that (BOOLEAN BB) is equivalent to (OR (EQ BB 0) (EQ BB 1)) in conformity with JOCIT semantics.

The treatment of other types of items is similar, with Hollerith, Transmission, and ASCII items all mapped without distinction into (CHARACTER-CONST item:name).

(g.2) Array declarations

Since these are more complex than simple item declarations we have attempted to carry more information over into the post-transduction form. For example, the JOCIT declaration:

ARRAY AA 2 3 I 15 S $ (See [3], p. 5-7)

becomes in post-transduction form:

```
(AND (IS-ARRAY AA) (EQ (DIMENSION AA) 2)
     (EQ (UPPERBOUND 1 AA) 2)
     (EQ (UPPERBOUND 2 AA) 3))
```

since AA is here a two-dimensional array with two elements along the first ("column") dimension, and three elements along the second (or "row") dimension The verifier does not currently perform array bound checks, but it will be relatively simple to make use of this bound information in future extensions.

Array declarations need to invoke another VCG mechanism, however, because of name scoping (see [3], p. 8-3). We have arranged that, upon entering the name scope (e.g., a procedure or function declaration) within which an array (or table, file, or switch) is declared, the array information given above is also stored on the property list of the array name. This information is available locally, i.e., while VCG is acting within that name scope, and the information is destroyed upon exiting that scope. A list (ARRAYNAMES) is also maintained (and updated) which collects the names of currently declared arrays. The global variables, SWITCHNAMES, TABLENAMES, FILENAMES, and PROCNAMES play similar roles. Thus, the action of WP on an array (table, file, or switch) declarations also has a side effect within VCG with respect to this property list storage mechanism, in addition to the value that WP passes back.

(g.3) Table declarations

In most respects table declarations are handled in a manner similar to array declarations. However, the post-transduction logical form,

table:decl*, has been kept to the minimum information (IS-TABLE table:name). The side-effect of WP[table:decl'; q] is, as with array declarations, the creation of property list information relevant to the declared table. This includes the following:

Indicator	Property
TABLESIZE	table:size'
TABLESTRUCTURE	P/S
TABLEPACKING	N/M/D

where table:size' is (V number) or (R number), for "variable" or "rigid" sizes, respectively. TABLESTRUCTURE refers to P (parallel) or S (serial). TABLEPACKING distinguishes the three types: N (no packing, i.e., word units), M (medium packing, i.e., byte units), and D (dense packing in bit units). While the verifier does not currently make use of such machine represention information, we expect to do so in future improved versions. Provision for preset information was not included in this version.

(g.4) File declarations

File declarations are analogous to array and table declarations in the mechanisms used in VCG. The JOCIT file declaration:

```
FILE FI [C/H/B] n1 [V/R] n2 file:states device:name $
```

(where [C/H/B] = file:type, and [V/R] = length:type) becomes in transduction:

```
(FILE FI file:type' n1 length:type' n2
     file:states' device:name)
```

where n1 = file:size = the maximum number of records in the file, and n2 = record:size = either the maximum record size (for V-specificiation) or the fixed record size (for R-specification). Record size is given in bytes for H and C-type files, and in words for B (binary) files. The file:type' is defined as:

- ASCII, if file:type = C;
- HOLLERITH, if file:type = H;
- BINARY, if file:type = B.

The item file:states' = a list of the items in the supplied list of status constants, file:states, e.g., ((V SHUT) (V OPEN'INPUT) (V OPEN'OUTPUT)), if file:states happens to be V(SHUT) V(OPEN'INPUT) V(OPEN'OUTPUT).

The information stored (temporarily, i.e., within the name scope of the file declaration) on the property list of the filename, FI, is:

Indicator	Property
FILETYPE	ASCII/HOLLERITH/BINARY
FILESIZE	n1 (an integer)
LENGTHTYPE	V/R
RECORDSIZE	n2 (an integer)
FILESTATES	file:states'
DEVICENAME	device:name (e.g., DSK)

(h) File I/O statements

The file I/O statements comprise the INPUT, OUTPUT, OPEN, and SHUT statements. The transduced syntax for these forms is:

```
inp:stat' = (INPUT file:name v1 v2 . . .)
out:stat' = (OUTPUT file:name e1 e2 . . .)
ostat'    = (OPEN INPUT/OUTPUT file:name . optional-iolist)
shut:stat'= (SHUT INPUT/OUTPUT file:name . optional-iolist)
```

The elementary operations here are opening and shutting a file (with no iolist provided), and the INPUT and OUTPUT statements, inp:stat' and out:stat'. The semantics of the OPEN and SHUT statements with an iolist provided are defined by concatenation of a simple OPEN/SHUT statement with the corresponding INPUT/OUTPUT statement. In the case of OPEN, the file is opened and then the INPUT/OUTPUT is executed; in the case of SHUT, the concatenation is in reverse order--the INPUT/OUTPUT statement is followed by the simple SHUT.

Thus, we need only define WP for the simple OPEN/SHUT forms and the INPUT/OUTPUT forms. WP[(OPEN/SHUT INPUT/OUTPUT file:name); q] is defined by:

```
WP[(BEGIN (:= (MAKE:FILESTATE file:name)
          (OPEN/SHUT INPUT/OUTPUT))
          (:= (POS file:name) 0)
          (:= (MAKE:FILEPTR file:name) 0))
   ; q]
```

The definition of WP[(INPUT file:name item); q], i.e., the form for input of a single item, is:

```
WP[(BEGIN (:= item (READ-RECORD file:name
                                (POS file:name)))
          (:= (POS file:name) (PLUS 1 (POS file:name))))
   ; q]
```

Analogously, WP[(OUTPUT file:name item); q] is given by:

```
WP[(BEGIN (:= file:name
               (WRITE-RECORD file:name
                             (POS file:name) item))
          (:= (POS file:name) (PLUS 1 (POS file:name))))
   ; q]
```

WP for INPUT/OUTPUT statements with an iolist consisting of several items is defined in the obvious way by recursion on the length of this list. Thus, updating of the file position pointer, e.g., FF:FILEPTR = (POS FF), is handled by an explicit assignment to (POS FF) each time a record is read from, or written onto, the file FF.

The functions MAKE:FILEPTR and MAKE:FILESTATE used above are such that (MAKE:FILEPTR FF) returns FF:FILEPTR, and (MAKE:FILESTATE FF) returns FF:FILESTATE, where FF can be any JOCIT name. As mentioned earlier, FF:FILEPTR is equivalent to (POS FF); the VCs contain the information that they are the same quantity (integer, or UNDEFINED).

The forms (READ-RECORD file:name ptr) and (WRITE-RECORD file:name ptr item) are analogous to SELECT and CHANGE as used in the semantics of arrays described above.

(i) Processing declarations

Processing declarations comprise switch declarations, procedure declarations, function declarations, and close declarations. The last three are similar and will be discussed together. There are two somewhat different types of switch declarations, index switches and item switches, and they are discussed separately.

(i.1) Index switch declarations

Index switches are declared by:

```
SWITCH sw:name = ( form1, form2, ... , formn) $
```

where the forms are either empty or goto:formulas of the type:

```
goto:formula ::= name
             ::= item:switch:name
             ::= switch:name($ index $)
```

The transduced syntax for an index switch is:

```
(INDEXSWITCH sw:name form1' form2' ... formn')
```

where an empty form transduces to NIL, a name transduces to (name NIL), and an indexed formula, e.g., ISW1($1,2$), transduces to the abstract form (ISW1 (1 2)). For example, the index switch declaration:

```
SWITCH XSW1 = (L0, L1,, CLS, XSW2($2$), ISW1($1$)) $
```

is mapped into:

```
(INDEXSWITCH XSW1 (L0 NIL) (L1 NIL) NIL (CLS NIL)
                  (XSW2 (2)) (ISW1 (1)))
```

The semantics are like those of a computed goto; when the statement GOTO XSW1(0) $ is encountered (within the scope of the above declaration) the effect is that of GOTO L0 $, since the zeroth element of XSW1 is the target label L0 (presumably a statement name). Similarly, GOTO XSW1(1) $ means GOTO L1 $; since the second element of XSW1 is empty, GOTO XSW1(2) $ has no effect. (Note that NIL is the no-op statement in abstract syntax).

As far as the index switch declaration itself is concerned, WP merely stores the index switch list, i.e., the Lisp alist:

```
((0 . (L0 NIL)) (1 . (L1 NIL)) (2 . NIL) (3 . (CLS NIL))
                (4 . (XSW2 (2)) (5 . (ISW1 (1))))
```

on the property list of the switch:name (in this case, XSW1). The property indicator used is INDEXSWLIST. This information associates each integer position 0,...,5 with its corresponding target element. The name XSW1 is also placed on a global list, SWITCHNAMES. Both pieces of information are used in the VCG processing of GOTOs.

(i.2) Item switch declarations

Declarations of item switches are transduced into forms like:

```
(ITEMSWITCH ISW1 SWITEM (= 1 (L1 NIL)) (= -5 (CLS NIL)))
```

where ISW1 is the switch:name, SWITEM is the variable (switch:variable) evaluated when a GOTO ISW1 $ is encountered, and the list of lists (each beginning with =) indicates how the actual target of the goto is interpreted for each of a finite number of values of SWITEM. Thus, if SWITEM = 1 when GOTO ISW1 $ is encountered, the interpretation is GOTO L1 $; if SWITEM = -5, the meaning is GOTO CLS $. This interpretation is conducted while processing the GOTO statement, as described earlier. All that is needed in processing the item switch declaration is to store the name of the switch:variable

(e.g., SWITEM) and the switch comparison list under the properties ITEMSWVARIABLE and ITEMSWLIST, respectively, on the property list of the switch name (ISW1). This property list information is local to the name scope of the item switch and is deleted when VCG exits from that scope.

(i.3) Procedure, function, and close declarations

The actions of VCG on these three types of declarations are closely parallel (except for minor differences produced by the absence of an output parameter list for functions and closes). We confine our description to the case of procedure declarations. As in the case of switch declarations, the important action is the placement of relevant information extracted from the transduced declaration on the property list of the procedure:name.

A transduced procedure declaration has the form:

```
(PROC proc:name (input:list (OUT:PARS output:list))
     declarations:list
     (BEGIN stat1 stat2 ...))
```

where proc:name is the name of the procedure; the car of the second element is a list of the formal input parameters, e.g., a list like (XX YY); the cadr of the second element is a list like (OUT:PARS ZZ WW), which uses OUT:PARS as a (reserved) keyword; declarations:list contains the formal entry- and exit-assertions for the procedure; and the last element of (PROC ...) is the procedure body (a compound:statement).

The information stored (on the property list of proc:name) is as follows:

Indicator	Property
FORMAL-INPUT-PARAMETERS	input:list
FORMAL-OUTPUT-PARAMETERS	output:list
FORMAL-ENTRY-ASSERTION	input:assertion
FORMAL-EXIT-ASSERTION	output:assertion

These actions are performed by a function called PROCESS:PROCDECL1 (which is called from PROCESS:PROCDECLS when VCG is looking at the whole program or at a procedure, close, or subprogram within which the current procedure is declared). (Note: There is a minor difference with respect to the FORMAL-EXIT-ASSERTION property for function declarations in that the implicit output variable for a function is the function name; hence, this

internal name is replaced by the function name cons'ed onto the input:list, e.g., FOO [a function name] is replaced by (FOO XX YY) everywhere in the output:assertion) to get the FORMAL-EXIT-ASSERTION for functions.)

The information stored in this property list persists only while VCG is acting within the declaration scope, and it is used by the WP function for procedure:calls, WP:CALL, discussed below.

Invocation of WP on a procedure (function, subprogram, or close) declaration also invokes the following hierarchical series of events:

1. The user is informed that a subprocedure has been encountered, and he is asked whether he desires the VCs for this subprocedure computed now.

2. If the user assents, the whole procedure (function, subprogram, or close) is passed for analysis to the appropriate version of VCG. (It is called VCGL:PROC for procedures and functions; observe that the function VCG is for user invocation, and on main programs only.) The body of the procedure is then subjected to the same kind of analysis described for the main program, and only when its VCs have been calculated, does the system return to generating VCs for the main program. The VCs for the subprocedure will be bound as value to a system-generated name, (e.g., PROC1*, if the procedure is called PROC1), and PROC1 will also receive a value as a Lisp variable equal to the (abstract) declaration.

3. If the user does not assent, analysis of the body of PROC1 will be deferred, as will analysis of any subprocedures that PROC1 might contain. However, the user can pick up this process at a later time, since PROC1 (as a Lisp variable) will receive the appropriate binding to the declaration. In the interim, the VCs for the top-level program will exist in a form that assumes the consistency of PROC1 with its formal entry/exit specifications, and any calls made to PROC1 will make use of that information.

The above sequence of events is closely bound up with our mechanism for carrying out hierarchical proofs, as described in more detail in Section IV.

(j) Procedure call statement

The JOCIT procedure call statement is distinguished in the abstract syntax by the key word CALL:

```
(CALL proc:name
      (actual:input:list (OUT:PARS actual:output:list)))
```

Function calls differ only in that (1) the output:list is empty and (2)

function calls do not appear at the statement level, but only within expressions. They are, therefore, more properly referred to as "function references," although the JOCIT manual [3] uses both terms.

The action of VCG (through WP:CALL) on procedure calls is fairly complex. It is best described by a dummy example. Suppose we are in a program where a procedure, named PROC1, has been declared. Suppose, also, that PROC1 has formal input parameters XX and YY, one output parameter ZZ, an entry assertion IN:PROC1(XX, YY), and an exit assertion OUT:PROC1(XX, YY, ZZ). The analysis of the body of PROC1 will have generated some set of VCs, which, when they have been proved valid, substantiate the following quantified formula:

```
(FORALL XX (FORALL YY (FORSOME ZZ
   (IMPLIES (IN:PROC1 XX YY) (OUT:PROC1 XX YY ZZ))).
```

Suppose that a call to PROC1 is encountered by WP, e.g., the call:

```
proc:call' = (CALL PROC1 ((AA BB) (OUT:PARS CC)))
```

where AA, BB and CC are, respectively, the actual values corresponding to the formals XX, YY and ZZ. The above quantified formula will be instantiated by the actual parameter values, with a system-generated unique name, e.g., CC:0013, for the value of the actual output parameter after symbolic execution of the procedure. The current assertion q being passed back through the procedure call statement may contain references to the exit value of the actual output parameter. (In fact, it almost certainly will contain such references since the effects of calling PROC1 are on this parameter.) The WP action will, first, substitute a gensym, e.g., CC:0013, for all occurrences of CC in q, resulting in a formula q'. This q' is then used to construct the formula:

```
q'' = (AND (IN:PROC1 AA BB)
           (IMPLIES (OUT:PROC1 AA BB CC:0013) q'))
```

The formula q'' is returned as the value of WP[proc:call'; q]. Since any instances of CC appearing in q have been replaced by instances of the gensym CC:0013, any occurrences of CC in this formula can only have resulted from appearances of CC in the input assertion. The gensym-ing of CC to CC:0013 permits us to refer to both entry and exit values of the actual output parameter.

Let us analyze what q'' means in terms of the overall VC under generation. The current VC will embody q'' in the form:

(IMPLIES p WP[code; q''])

where "code" refers to the code intervening between some earlier control point (where the assertion p is attached) and the procedure call statement. In proving the above VC, it is incumbent on the prover (be he human or machine) to show that q'' will be true whenever control reaches the procedure call from the point p. In particular, proving this implication necessitates showing that (IN:PROC1 AA BB) will hold, i.e., that the procedure's input assertion is satisfied by the actual input values. It also requires proving that q' will follow from the conjunction of p and the assertion (OUT:PROC1 AA BB CC:0013). The latter is equivalent to showing that the desired relation q' holds among these variables, assuming that the procedure call establishes the specified relation (OUTPROC1) among AA, BB, and CC on exit.

This process effectively permits us to decouple the process of proving correctness for a formal procedure from the program (or subprogram, other procedure, function, or close) where it is invoked. The formal relation between entry- and exit-assertions for the procedure is instantiated in the VC for the host program (subprogram, etc.) by the actual parameters, and these instantiated specifications can be used to prove correctness for the host program.

III VERIFICATION TIMING

A. Introduction

Task B, Item 4.1.2, of our Statement of Work, requires the reduction of processing time for J3 program verification by a factor of at least 2 compared to the RPE/1 system, despite the greater scope of the RPE/2 verifier. The most obvious way of measuring a change, using the RPE/2 system to verify a program that took a known amount of time with RPE/1 and comparing the times required, is not feasible because the programs with which we dealt in RPE/1 were not valid JOCIT and consequently not valid as input to RPE/2. This prevents direct time comparisons for both parsing and verification condition generation but permits them for deduction because the classes of syntactically well-formed formulas accepted by the two deductive systems are substantially the same. For parsing, a reasonable comparison can be made between the two systems by measuring parsing in units of time per input lexical token. Similarly, we can compare the times to generate verification conditions for comparably sized programs.

We have made a set of such comparisons and conclude that the improvement in RPE/2 is substantially better than the requirement in the Statement of Work. A binary search program on which we reported in the RPE/1 Final Report required 210 seconds for parsing, 5 seconds for verification condition generation, and 600 seconds for deduction. In the RPE/2 system, comparable speeds are .7 seconds for parsing, 1 second for verification condition generation, and 27.5 seconds for deduction. The totals are 815 seconds in RPE/1 and 29.2 seconds in RPE/2 or a speedup by a factor of about 27. (However, it must be noted that the RPE/2 times are on a DEC KL-20 computer that is 3 to 5 times faster than the DEC KA-10 on which the RPE/1 measurements were made. Also, the present system is written in MACLISP which provides compiled object code that, in our application, appears to be approximately 1.5 times faster than RPE/1's INTERLISP. Both these conversion

factors are affected by variables whose measurement is outside our scope, such as the different ways the two lisp systems implement file input primitives; hence comparison is necessarily quite imprecise.)

Taking into account the change in implementation language and machine between RPE/1 and RPE/2, we conclude that the speed of verification condition generation is essentially unchanged although the language being verified is about 10 times larger in RPE/2. Especially since VCG is a negligible part of the time for verification, we are quite pleased with this result.

Although the comparisons presented above take only cpu time into account, we believe the time required for interaction has also been reduced substantially. This involves only the deductive facilities of the system. We compared the proofs of a moderately complex verification condition for a binary search program such as that done in RPE/1. The RPE/1 proof required about three hours of user time. In RPE/2, as a result of both the more powerful deductive mechanisms that have been implemented and the more flexible interactive facilities of tableaux, this proof requires only about fifteen minutes of user time.

The remaining subsections of this section describe the changes in four parts of the system: the parser, the Presburger deductive mechanism, the tableaux proof system, and the hashing utilities.

B. Parser Timing

Of the verifier's various components, the parsing mechanism showed the most dramatic speed improvement--about 30 to 1. As we noted above, part of this improvement is due to faster computing facilities and to the use of the MACLISP compiler. The most important reason for the speedup, however, is the incorporation of a parsing algorithm fundamentally different from the one used in the RPE/1 system.

The parsing mechanism used in our earlier effort is based on an algorithm [5] developed by J. Earley. An advantage of this algorithm is that it assumes only that the grammar in question is context-free. This made Earley's algorithm particularly attractive to us at the beginning of the

RPE/1 effort, when we had had relatively little experience with the JOVIAL language.

Earley's algorithm is easy to use but computationally inefficient. Our implementation in RPE/1 was able to parse at a rate of only about one token per second. While this level of performance was adequate for the early stages of system development, it is far below that needed for practical use of the verifier.

The parser implemented in the RPE/2 effort is based on the SLR(k) parsing algorithm first proposed by De Remer [4]. Unlike the Earley algorithm, the new technique requires the context-free grammar to possess certain structural properties. The inconvenience of modifying the grammar into acceptable form is balanced by a dramatic increase in parsing speed. In the case of the JOCIT parser, this speed turned out to be approximately 30 milliseconds for each lexical token.

The improvement given by the new parsing technique might be compared to that gained by using a compiler rather than an interpreter. We feel that the new parser is sufficiently fast for use in practice.

C. Presburger Deductive Mechanism

Much of the increased efficiency of the deductive component is due to improvements made in the Presburger decision mechanism.

This mechanism is able to prove valid formulas, and find counterexamples for invalid formulas in an extension of universal Presburger arithmetic. Roughly speaking, universal Presburger formulas are those that can be built up from integers, integer variables, addition, multiplication by constants, the usual arithmetical and propositional relations, and universal closure. The formula (FORALL x)(FORALL y)[3x+y = 2(x+y)+(x-y)], for example, is in the class. Our extension of universal Presburger introduces arbitrary uninterpreted function symbols. The formula

```
(FORALL x)(FORALL y)
    [[x < f(y)+1 AND f(y)< x+1] IMPLIES y+g(x) = g(f(y))+y]
```

is a member of this extended class.

The improvements to the Presburger code made in the RPE/2 effort can best be explained in relation to the decision method used in the earlier version. This method is carried out in two stages.

First, the closed formula F to be proved or disproved is transformed, by a kind of disjunctive normal form expansion, to a set of integer linear programming problems (ILPs). The ILPs have the property that F is valid if and only if no ILP is solvable.

Next, the ILPs are tested (one by one) for solvability by using an improved version of the SUP-INF method first proposed by Bledsoe [1]. If one of the ILPs is found to have an integer solution, that solution provides a model, and therefore a counterexample, for F. If none is found solvable, F is reported valid.

The improved Presburger mechanism differs from the earlier version in two important respects: both the technique used to test ILPs for solvability and the means for handling function symbols have been changed.

As we noted above, a modification of the SUP-INF method (described in detail in [15]) was used to test ILPs for solvability in the RPE/1 version. The SUP-INF approach to integer linear programming may be viewed as that of transforming an integer problem into the real domain, solving it in that domain, and then interpreting the result in the integer domain. The advantage of this approach over other methods of integer linear programming is speedy solution of small problems. The efficiency of this method derives partly from ease of use (it requires no matrix initialization as does other methods) and partly from the fact that the real problem is easier to solve than the integer problem. The chief disadvantage of the SUP-INF method is incompleteness--it cannot determine feasibility for a certain class of problems whose solvability depends on Diophantine behavior. Concerned about maintaining completeness in the Presburger mechanism, we decided to implement a more traditional method of solving ILPs--the Gomory [8] algorithm--in the RPE/2 effort. The Gomory algorithm entails a substantial amount of matrix initialization overhead, compared to the SUP-INF method, and so it is not quite as efficient for small problems. On the other hand, the array manipulation operations used by the Gomory algorithm are handled quite

efficiently by MACLISP, so the overhead is not substantial in our implementation. We have found, in fact, that for medium to large problems the Gomory implementation is as fast or faster than the Interlisp SUP-INF implementation. Using the Gomory algorithm has significantly extended the domain of completeness of the Presburger mechanism as a whole.

A second important improvement in the mechanism concerns the method used to deal with the semantics of function symbols. These semantics require that counterexamples for invalid formulas satisfy the substitutivity axiom of equality; for example, if the variables x and y are given the same numerical value in a counterexample, the terms f(x) and f(y) must also be given the same value. The mechanism used to enforce this axiom in the earlier implementation involved intricate, inefficient, and often unreliable code interlaced with the code for the SUP-INF algorithm. In the newer version, we use a more efficient and conceptually cleaner method. The new method is independent of how integer feasibility is tested and is therefore more reliable. It permits the proof of some theorems that could not be proved by the earlier mechanism in reasonable amounts of time.

D. Tableaux System Timing

The deductive system has been substantially improved over that of RPE/1 by the fact that we now use the Tableaux Mechanism to manage all proofs. This facility acts as the sole user interface and is responsible for transmitting user directives to various specialized deductive mechanisms. By comparison, our RPE/1 deductive system had an ad hoc mechanism for interaction and lacked the present system's capacity to structure proofs and manage proof strategy. This new facility is described in Section IV.

E. Hash Timing

In areas of the system requiring the use of hash tables, we have adopted Balbine's double hashing algorithm [11] using twin prime table sizes so that division by the table size and the table size minus two can be used to compute hash probes. This greatly reduces the secondary clustering that limited the speed of our original single hashing scheme.

IV INTERACTION

A. Introduction

Task C, Item 4.1.3 of our Statement of Work, requires the addition to the RPE/1 verification system of features to increase the ease of interaction. We have implemented each of the specific features required--the ability to save partial proofs, the capacity to extend the assertion language with user-defined constructs, and the capability of carrying out structured proofs of programs--as well as a large number of other facilities intended to reduce the quantity and improve the quality of user interaction during verification.

The two remaining subsections of this section describe the interactive features of the tableaux proof system and the user facilities for carrying out hierarchical proofs.

B. Tableaux Deductive System

Most of these facilities have been implemented within the framework of our Tableaux Deductive System, and so we begin by describing the ideas underlying this system. It is based on the "analytic tableaux" of Smullyan [16]. In this method, we prove a formula F by constructing a tree T such that:

* the nodes of T are formulas,
* the root formula of T is F,
* each of the nodes of T is derived from some ancestor node according to one of the rules (enumerated below) for extending a tableau, and
* each of the branches of T is "closed" in the sense that it contains a pair of formulas P and (NOT P).

To understand why this technique works, consider the formula f(T) which is constructed from a tree T by conjoining, over each branch B, the disjunction

of the formulas on B. Suppose we apply a rule R to T to obtain the extended tableau R[T]. The critical fact is that for each rule R, f(T) is equivalent to f(R[T]). It follows by induction on the number of rules applied in going from the initial tableau T0 with only the root node containing F to the final closed tableau TC that f(T0) is equivalent to f(TC). But f(T0) is just (AND (OR F)) which is equivalent to F. And f(TC) is a conjunction of disjunctions each of which is tautologically true because it contains a formula P and its denial (NOT P). Thus F is equivalent to a tautologically true formula and hence must be a theorem.

Before enumerating the rules for extending tableaux, let us explain why we have chosen analytic tableaux as the basis of an interactive proof system. The previous paragraph is the sketch of a proof (given in detail in Chap. 2 of Smullyan) that the method is consistent, i.e., it can never construct a closed tableau for a non-theorem. It can also be proved (see Smullyan, pp. 57-60) that if F is a valid first-order formula, then there is a finite closed tableau rooted with F. Thus the method of analytic tableaux is as good in theoretical terms as any proof procedure for first-order logic.

We believe that, besides its theoretical merits, this method is practical for present-day verification of interesting programs. This is because it lends itself to a harmonious and flexible mixture of automatic and interactive proof. The user of our Tableaux System has available a variety of interactive facilities for dividing a large proof into smaller parts, performing primitive logical operations, invoking lemmas, and investigating the state of a partial proof. The user also has available two powerful automatic facilities for simplifying algebraic expressions and proving theorems in a quantifier-free theory of Presburger arithmetic (augmented with uninterpreted functions). This combination of tools leads to a proof style in which the user does parts of a proof in small, manual steps until reaching formulas within the domain of the automatic facilities that may then be used to complete the proof.

We like this arrangement for several reasons. First, as we have remarked, the interactive facilities amplify the power of the completely mechanical (but semantically restricted) provers now available to permit the

proof of interesting programs. Second, the structure of the system is quite flexible: as new automatic techniques are developed, we can incorporate them into our uniform user interface, both simplifying the burden of the user and extending the potency of the complete system. Finally, we have structured the system so that the dependence of a proof on particular lemmas, or the results of automatic theorem-proving components is made explicit. We believe that the presence of these easily understandable "audit trails" in the presentation of a proof greatly increases the credibility of the system compared to that of a system whose final result was a bare TRUE or FALSE.

The rules for augmenting a tableau are as follows:

* ALPHA: If a node contains the formula (OR d1 d2 ...), then the chain of nodes containing d1, d2, ... may be added at each leaf below the formula. The intuitive content of this operation is that to prove a disjunction, it suffices to prove any one of the disjuncts. This rule also applies to the formula (IMPLIES F1 F2) because of its equivalence to (OR (NOT F1) F2), to the formula (NOT (NOT F)), equivalent to (OR F), and to the formula (NOT (AND c1 c2 ...)), equivalent to (OR (NOT c1) (NOT c2) ...).

* BETA: If a node contains the formula (AND c1 c2 ...), then the open leaves below it may each be augmented by adding, as sons, each of the conjuncts. Note that this differs from the ALPHA rule where a chain is added with d2 the son of d1, d3 the son of d2, etc. Here the conjuncts are added as brothers. This rule permits us to prove a conjunction by proving each of the conjuncts separately. It also applies to the formula (NOT (OR d1 d2 ...)), equivalent to (AND (NOT d1) (NOT d2) ...) and to the formula (NOT (IMPLIES F1 F2)), equivalent to (AND F1 (NOT F2)).

* GAMMA: If a node contains the formula (FORSOME x (e x)) (where (e x) is any formula with free variable x), then the unclosed leaves below the formula may be augmented by nodes containing the formula (e a) for any term a. That is, to prove that (e x) holds for some x it is sufficient to exhibit any such x. Equivalently, below the formula (NOT (FORALL x (e x))), there may be appended any instance (NOT (e a)).

* DELTA: This rule provides for formulas of the form (FORALL x (e x)). The unclosed leaves below such formulas may be augmented by nodes containing the formula (e (sfi)), where (sfi) is a newly introduced Skolem function, a function with no special properties. The basis of this rule is that, if we can prove (e (sfi)) with no knowledge about sfi, then this is tantamount to proving (FORALL x (e x)). Similarly, the open branches containing a formula (NOT (FORSOME x (e x))) may may be augmented by the formula (NOT (e (sfi))) where, in the same way, sfi is newly introduced.

* INSTANCE: This rule combines the GAMMA and DELTA rules into one, allowing the user to strip any subset of the quantifiers in a formula by substituting arbitrary terms for the existential variables and Skolem functions for the universal variables. For example, given the formula (FORSOME x (FORALL y (FORSOME z (p x y z)))), the user might choose to eliminate quantification over y and z but retain quantification over x. This yields the formula (FORSOME x (p x (sfi x) t)), where t is any user supplied term and sfi is a system- supplied Skolem function. Note that to retain soundness, all enclosing existential indicial variables must be included as parameters of the Skolem function.

* INVOKE: This rule provides that if a formula p is valid, then the negation of its universal closure may be appended to any of the branches of a tableau. Observe that if p is valid, then the negation of its universal closure is unsatisfiable. Thus, adding this negation to a branch preserves the validity of the disjunction of the branch's formulas.

* IDENTITY: Suppose two nodes c1 and c2 contain a formula e and the negation of a universally quantified identity whose matrix is (EQ eq1 eqr) or (IFF eq1 eqr). Then any proper substitution instance of e with respect to this identity may be appended to leaves that are descendants of both c1 and c2. (A substitution is proper if none of the variables of the substituted terms are captured by existential quantifiers of e.) This is the usual rule of substitutivity of identities.

* ALGEBRA: Suppose a node contains a formula F and the algebraic simplifier reduces F to F'. Then F' may be appended to open leaves below F.

* ARITH: Suppose c is a node some of whose ancestors contain the formulas F1, F2, ..., FN. Suppose the Presburger decision mechanism can prove the formula P: (IMPLIES (AND (NOT F1) ... (NOT FN)) F). Then the formula (NOT F) may be appended to the open leaves below c. This rule may be understood as an application of the INVOKE rule to the lemma P and the BETA rule to the resulting node containing (NOT P).

We will describe the facilities now included in the system (which is still being actively developed) by guiding the reader through the proofs of two sample theorems. These theorems are two of the verification conditions that arise in proving the correctness of a string searching algorithm developed by Boyer and Moore [2]. (In Appendix B, we give a complete specification of the user command set for carrying out tableaux proofs.)

The first condition to be proved is:

```
(IMPLIES (AND (AND (EQ PATLEN (LENGTH PAT))
                   (EQ STRLEN (LENGTH STR)))
```

```
        (AND (INT STRLEN) (GTQ STRLEN 0))
        (AND (INT PATLEN) (GTQ PATLEN 0))
        (AND (BOOLEAN FLG) (EQ FLG 0))
        (AND (IS-ARRAY STR)
             (EQ (ARRAYTYPE STR) CHARACTER-CONST)
             (EQ (DIMENSION STR) 1)
             (EQ (UPPERBOUND 1 STR) 1000))
        (AND (IS-ARRAY PAT)
             (EQ (ARRAYTYPE PAT) CHARACTER-CONST)
             (EQ (DIMENSION PAT) 1)
             (EQ (UPPERBOUND 1 PAT) 100)))
   (AND (AND (AND (EQ PATLEN (LENGTH PAT))
                  (EQ STRLEN (LENGTH STR)))
             (EQ 0 0))
        (OR (EQ (STRPOS PAT STR) 0)
            (GTQ (PLUS (STRPOS PAT STR) PATLEN)
                 (PLUS PATLEN 1))))))
```

Assuming that this condition is the value of the variable vc:1, we begin a proof of the theorem by the command

```
*(newproof vc:1)
```

In the annotated interactions that follow, we will present the user commands in lower case and preceded by "*". System responses will be in upper case. The system responds to the newproof command by asking for the declaration of a number of the unknown identifiers that occur in the theorem.

```
HOW SHOULD PATLEN BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD PAT BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD LENGTH BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD STRLEN BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD STR BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD INT BE DECLARED? TYPE TERM OR FORMULA
formula
HOW SHOULD FLG BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD BOOLEAN BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD IS-ARRAY BE DECLARED? TYPE TERM OR FORMULA
formula
HOW SHOULD ARRAYTYPE BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD CHARACTER-CONST BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD DIMENSION BE DECLARED? TYPE TERM OR FORMULA
term
```

```
HOW SHOULD UPPERBOUND BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD STRPOS BE DECLARED? TYPE TERM OR FORMULA
term

(IMPLIES (AND ...) (AND ...))

TABLEAU SETUP COMPLETED
```

The system is now ready to proceed with the proof. We begin by telling it to proceed automatically as far as possible.

```
*(proofgo)

NEW NODES: (27 26)

(NODE 30 HAS BEEN CLOSED WRT NODE 4)

NEW NODES: (31 30)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 26 31 BETA)
THE UNCLOSEDLEAVES ARE: ((31) 29)
(NONLOGICAL IS ((31 29 28 25 24 23 22 21 20 19 18 17 16 15
                 14 13 12 11 10)))
```

At this point a tableau has been grown with 31 nodes and many of the branches have been closed automatically. Two branches remain open: those that end in the leaves 31 and 29. We ask to see the formula at node 31.

```
*(pf 31)

(EQ 0 0)
```

Although this formula is obviously true and can be proved by our Presburger arithmetic theorem prover, the system cannot presently invoke this prover automatically. So we turn its attention to the formula by means of the command

```
*(nextnode 31)
NONLOGICAL
```

which responds with the logical type of the formula--NONLOGICAL. Next we use the arithmetic theorem prover. One form of the ARITH rule is invoked by the function arithc; it tries to prove the current formula by using the formulas at the argument nodes as hypotheses. In this case, we require no hypotheses, so the command is just

```
*(arithc)
```

and the system responds by proving the formula and closing the branch:

```
(EQ 0 0) HAS BEEN VERIFIED

(NODE 32 HAS BEEN CLOSED WRT NODE 31)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 31 32 ARITH)
THE UNCLOSEDLEAVES ARE: (NIL 29)
(NONLOGICAL IS ((31 29 28 25 24 23 22 21 20 19 18 17 16
                 15 14 13 12 11 10)))
```

The only open branch is now that ending at node 29. We position the system at this node

```
*(nextnode 29)
NONLOGICAL
```

and issue a command to display the nodes of this branch. (Note that the current node is denoted by the "@" symbol.)

```
*(visible)

29@(GTQ (PLUS (STRPOS PAT STR) PATLEN) (PLUS PATLEN 1))
28 (EQ (STRPOS PAT STR) 0)
25 (NOT (EQ STRLEN (LENGTH STR)))
24 (NOT (EQ PATLEN (LENGTH PAT)))
23 (NOT (GTQ STRLEN 0))
22 (NOT (INT STRLEN))
21 (NOT (GTQ PATLEN 0))
20 (NOT (INT PATLEN))
19 (NOT (EQ FLG 0))
18 (NOT (BOOLEAN FLG))
17 (NOT (EQ (UPPERBOUND 1 STR) 1000))
16 (NOT (EQ (DIMENSION STR) 1))
15 (NOT (EQ (ARRAYTYPE STR) CHARACTER-CONST))
14 (NOT (IS-ARRAY STR))
13 (NOT (EQ (UPPERBOUND 1 PAT) 100))
12 (NOT (EQ (DIMENSION PAT) 1))
11 (NOT (EQ (ARRAYTYPE PAT) CHARACTER-CONST))
10 (NOT (IS-ARRAY PAT))
```

We now observe that one of the properties of the subroutine STRPOS is needed to complete the proof. The required property, called STRPOS:AX1 is

```
(FORALL S1 (FORALL S2  (OR (EQ (STRPOS S1 S2) 0)
                           (GTQ (STRPOS S1 S2) 1)))))
```

It is invoked by the command

```
*(invoke (makethm strpos:ax1))
HOW SHOULD S2 BE DECLARED? TYPE TERM OR FORMULA
term
```

and the user then selects the appropriate instance of the lemma (whose

negation signifies in the tableaux method that it is assumed) by means of the dialogue

```
(NOT (FORALL ::1
             (FORALL ::2
                     (OR (EQ (STRPOS :1 :2) 0)
                         (GTQ (STRPOS :1 :2) 1)))))

(SUBSTITUTE AT ::1 ?) y OK
(WHAT SUBSTITUTION ?) pat

(SUBSTITUTE AT ::2 ?) y OK
(WHAT SUBSTITUTION ?) str

(THE RESULT OF SUBSTITUTION IS:)

(NOT (OR (EQ (STRPOS PAT STR) 0) (GTQ (STRPOS PAT STR) 1)))
```

This instantiation of the lemma is of type BETA; appended to the tableau, it will split into two cases, one with the node (NOT (EQ (STRPOS PAT STR) 0)) and the other with the node (NOT (GTQ (STRPOS PAT STR) 1)). Noting that the first of these contradicts node 28 and will close immediately, we proceed to add the instantiated lemma to the tableau.

```
*(proofgo)

(NODE 34 HAS BEEN CLOSED WRT NODE 28)

NEW NODES: (35 34)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 33 35 BETA)
THE UNCLOSEDLEAVES ARE: ((35))
(NONLOGICAL IS ((35 31 29 28 25 24 23 22 21 20 19 18 17
                 16 15 14 13 12 11 10)))
```

This leaves a single open branch terminating in the second case that arises from the lemma. And this branch can be closed by a simple arithmetic deduction using the formula at node 29. Thus the proof is concluded by

```
*(nextnode 35)
NONLOGICAL
*(arithc 29)

(NOT (GTQ (STRPOS PAT STR) 1)) HAS BEEN VERIFIED

(NODE 36 HAS BEEN CLOSED WRT NODE 35)
```

```
    (THE PROOF IS COMPLETE)
```

The second condition is:

```
    (IMPLIES
      (AND (AND (AND (EQ PATLEN (LENGTH PAT))
                     (EQ STRLEN (LENGTH STR)))
                (EQ FLG 0.)
           (OR (EQ (STRPOS PAT STR) 0)
               (GTQ (PLUS (STRPOS PAT STR) PATLEN)
                    (PLUS II 1))))
      (AND (IMPLIES (GT II STRLEN)
                    (AND (EQ FLG 0) (EQ (STRPOS PAT STR) 0)))
           (IMPLIES (NOT (GT II STRLEN))
                    (AND (AND (AND (AND (EQ PATLEN
                                            (LENGTH PAT))
                                       (EQ STRLEN
                                           (LENGTH STR)))
                                  (EQ FLG 0))
                             (EQ (SUBSTRING PAT (PLUS 1 PATLEN)
                                            PATLEN
                                 (SUBSTRING STR (PLUS 1 II)
                                            (SUBTRACT (PLUS II PATLEN)
                                                      PATLEN))))
           (OR (EQ (STRPOS PAT STR) 0)
               (GTQ (PLUS (STRPOS PAT STR) PATLEN)
                    (SUBTRACT (PLUS (PLUS II PATLEN) 1)
                              PATLEN)))))))
```

The proof of this condition, bound to the variable vc:2, is begun by the command

```
    *(newproof vc:2)
```

to which the system responds

```
    (IMPLIES (AND ...) (AND ...))

    TABLEAU SETUP COMPLETED
```

Noting the occurrences of SUBSTRING in the body of this formula, we decide that we will need to substitute instances of the SUBSTR:NIL lemma which says that

```
    (FORALL S (FORALL K (FORALL L
        (IMPLIES (GT K L) (EQ (SUBSTRING S K L) NIL))))))
```

The lemma invocation function, like the functions for instantiation, identity substitution, and algebraic simplification (unless its argument is proved) shows the user the result that may be appended to the tableau but requires a proofgo to do the appending. Since we do not, for the moment, want this proofgo to apply the ALPHA and BETA rules to vc:2, we will force the system to come back to the user after each rule is applied.

```
*(setq automatic nil)
NIL
*(invoke (makethm substr:nil))
HOW SHOULD K BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD L BE DECLARED? TYPE TERM OR FORMULA
term
HOW SHOULD NIL BE DECLARED? TYPE TERM OR FORMULA
term

(NOT (FORALL ::1
       (FORALL ::2
         (FORALL ::3
           (IMPLIES (GT :2 :3)
                    (EQ (SUBSTRING :1 :2 :3) NIL))))))

(SUBSTITUTE AT ::1 ?) y OK
(WHAT SUBSTITUTION ?) pat

(SUBSTITUTE AT ::2 ?) y OK
(WHAT SUBSTITUTION ?) (plus 1 patlen)

(SUBSTITUTE AT ::3 ?) y OK
(WHAT SUBSTITUTION ?) patlen

(THE RESULT OF SUBSTITUTION IS:)

(NOT
 (IMPLIES (GT (PLUS 1 PATLEN) PATLEN)
          (EQ (SUBSTRING PAT (PLUS 1 PATLEN) PATLEN) NIL)))
```

Now, we add this instance of the lemma to the tableau.

```
*(proofgo)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 1 2 ALPHA)
THE UNCLOSEDLEAVES ARE: ((2))
(ALPHA IS ((1)))
(BETA IS ((2)))
```

Next, we create and add to the tableau the other required instance of the lemma.

```
*(invoke (makethm substr:nil))

(NOT
  (FORALL ::1
    (FORALL ::2
      (FORALL ::3
        (IMPLIES (GT :2 :3)
                 (EQ (SUBSTRING :1 :2 :3) NIL))))))
```

```
(SUBSTITUTE AT ::1 ?) y OK
(WHAT SUBSTITUTION ?) str

(SUBSTITUTE AT ::2 ?) y OK
(WHAT SUBSTITUTION ?) (plus 1 ii)

(SUBSTITUTE AT ::3 ?) y OK
(WHAT SUBSTITUTION ?) ii

(THE RESULT OF SUBSTITUTION IS:)

(NOT (IMPLIES (GT (PLUS 1 II) II)
              (EQ (SUBSTRING STR (PLUS 1 II) II) NIL)))
*(proofgo)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 1 3 ALPHA)
THE UNCLOSEDLEAVES ARE: ((3))
(ALPHA IS ((1)))
(BETA IS ((3 2)))
```

Although the ALPHA and BETA rules are applicable to the instances just created, we wish to avoid using them in this way. This is done by

```
*(defer 2)
*(defer 3)
```

Next, we want to proceed automatically from the current node 1.

```
*(setq automatic t)
T
*(proofgo)

NEW NODES: (13 12)

NEW NODES: (17 16 15 14)

(NODE 28 HAS BEEN CLOSED WRT NODE 9)

(NODE 29 HAS BEEN CLOSED WRT NODE 12)

NEW NODES: (29 28)

NEW NODES: (31 30)

(NODE 32 HAS BEEN CLOSED WRT NODE 12)

(NODE 33 HAS BEEN CLOSED WRT NODE 6)

NEW NODES: (34 33)
```

```
(NODE 35 HAS BEEN CLOSED WRT NODE 9)

NEW NODES: (36 35)

NEW NODES: (38 37)

(NODE 41 HAS BEEN CLOSED WRT NODE 6)

NEW NODES: (42 41)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 37 42 BETA)
THE UNCLOSEDLEAVES ARE: ((42) 34 36 40)
(BETA IS (NIL 2 3))
(NONLOGICAL IS ((42 40 39 36 34 26 25 21 20 13
                 12 11 10 9)))
```

This grows 42 nodes and closes several branches. To see what to do next, we obtain a summary of all the open branches.

```
*(av)

(FROM NODE 34)

34 (EQ (SUBSTRING PAT (PLUS 1 PATLEN) PATLEN)
       (SUBSTRING STR (PLUS 1 II)
                      (SUBTRACT (PLUS II PATLEN) PATLEN)))
25 (GT II STRLEN)
12 (NOT (EQ (STRPOS PAT STR) 0))
11 (NOT (EQ STRLEN (LENGTH STR)))
10 (NOT (EQ PATLEN (LENGTH PAT)))
9 (NOT (EQ FLG 0))

(FROM NODE 36)

36 (EQ (STRPOS PAT STR) 0)
21 (NOT (GT II STRLEN))
13 (NOT (GTQ (PLUS (STRPOS PAT STR) PATLEN) (PLUS II 1)))
11 (NOT (EQ STRLEN (LENGTH STR)))
10 (NOT (EQ PATLEN (LENGTH PAT)))
9 (NOT (EQ FLG 0))

(FROM NODE 40)

40 (GTQ (PLUS (STRPOS PAT STR) PATLEN)
        (SUBTRACT (PLUS (PLUS II PATLEN) 1) PATLEN))
39 (EQ (STRPOS PAT STR) 0)
20 (GT II STRLEN)
13 (NOT (GTQ (PLUS (STRPOS PAT STR) PATLEN) (PLUS II 1)))
11 (NOT (EQ STRLEN (LENGTH STR)))
10 (NOT (EQ PATLEN (LENGTH PAT)))
9 (NOT (EQ FLG 0))
```

```
(FROM NODE 42)

42 (EQ (SUBSTRING PAT (PLUS 1 PATLEN) PATLEN)
       (SUBSTRING STR (PLUS 1 II)
                      (SUBTRACT (PLUS II PATLEN) PATLEN)))
20 (GT II STRLEN)
13 (NOT (GTQ (PLUS (STRPOS PAT STR) PATLEN) (PLUS II 1)))
11 (NOT (EQ STRLEN (LENGTH STR)))
10 (NOT (EQ PATLEN (LENGTH PAT)))
9 (NOT (EQ FLG 0))

DONE
```

First, we observe that node 40 follows from node 13 with a little arithmetic.

```
*(nextnode 40)
NONLOGICAL
*(arithc 13)

(GTQ (PLUS (STRPOS PAT STR) PATLEN)
     (SUBTRACT (PLUS (PLUS II PATLEN) 1) PATLEN))

HAS BEEN VERIFIED

(NODE 43 HAS BEEN CLOSED WRT NODE 40)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 40 43 ARITH)
THE UNCLOSEDLEAVES ARE: (NIL 34 36 42)
(BETA IS (NIL 2 3))
(NONLOGICAL IS ((42 40 39 36 34 26 25 21 20 13
                 12 11 10 9)))
```

Next, we observe that nodes 42 and 34 can be closed by using arithmetic and the lemma instances created earlier.

```
*(nextnode 42)
NONLOGICAL
*(arithc 2 3)

(EQ (SUBSTRING PAT (PLUS 1 PATLEN) PATLEN)
    (SUBSTRING STR (PLUS 1 II) (SUBTRACT (PLUS II PATLEN)
                                         PATLEN)))

HAS BEEN VERIFIED

(NODE 44 HAS BEEN CLOSED WRT NODE 42)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 42 44 ARITH)
THE UNCLOSEDLEAVES ARE: (NIL 34 36)
(BETA IS (NIL 2 3))
```

```
(NONLOGICAL IS ((42 40 39 36 34 26 25 21 20 13 12 11 10 9)))

T
*(nextnode 34)
NONLOGICAL
*(arithc 2 3)

(EQ (SUBSTRING PAT (PLUS 1 PATLEN) PATLEN)
    (SUBSTRING STR (PLUS 1 II) (SUBTRACT (PLUS II PATLEN)
                                         PATLEN)))

HAS BEEN VERIFIED

(NODE 45 HAS BEEN CLOSED WRT NODE 34)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 34 45 ARITH)
THE UNCLOSEDLEAVES ARE: (NIL 36)
(BETA IS (NIL 2 3))
(NONLOGICAL IS ((42 40 39 36 34 26 25 21 20 13 12 11 10 9)))

T
```

A single open branch remains, and to close it we must invoke another of the properties of STRPOS.

```
*(nextnode 36)
NONLOGICAL
*(invoke (makethm strpos:ax2))
HOW SHOULD S2 BE DECLARED? TYPE TERM OR FORMULA
term

(NOT (FORALL ::1
          (FORALL ::2
                  (OR (EQ (STRPOS :1 :2) 0)
                      (LTQ (PLUS (STRPOS :1 :2) (LENGTH :1))
                           (PLUS 1 (LENGTH :2)))))))

(SUBSTITUTE AT ::1 ?) y OK
(WHAT SUBSTITUTION ?) pat

(SUBSTITUTE AT ::2 ?) y OK
(WHAT SUBSTITUTION ?) str

(THE RESULT OF SUBSTITUTION IS:)

(NOT (OR (EQ (STRPOS PAT STR) 0)
         (LTQ (PLUS (STRPOS PAT STR) (LENGTH PAT))
              (PLUS 1 (LENGTH STR)))))
```

```
*(proofgo)

(NODE 47 HAS BEEN CLOSED WRT NODE 36)

NEW NODES: (48 47)

(TABLEAU IS NOT CLOSED)

(CURRENT:LASTNODE:CURRENTTYPE ARE RESPECTIVELY 46 48 BETA)
THE UNCLOSEDLEAVES ARE: ((48))
(BETA IS (NIL 2 3))
(NONLOGICAL IS ((48 42 40 39 36 34 26 25 21
                 20 13 12 11 10 9)))

*(visible 48)

48 (NOT (LTQ (PLUS (STRPOS PAT STR) (LENGTH PAT))
             (PLUS 1 (LENGTH STR))))
36 (EQ (STRPOS PAT STR) 0)
21 (NOT (GT II STRLEN))
13 (NOT (GTQ (PLUS (STRPOS PAT STR) PATLEN) (PLUS II 1)))
11 (NOT (EQ STRLEN (LENGTH STR)))
10 (NOT (EQ PATLEN (LENGTH PAT)))
9 (NOT (EQ FLG 0))
```

A proofgo has closed off the easy case of the lemma, and a summary of the open branch that is left shows that some appropriate arithmetic deduction will finish up the proof.

```
*(nextnode 48)
NONLOGICAL
*(arithc 21 13 11 10)

(NOT (LTQ (PLUS (STRPOS PAT STR) (LENGTH PAT))
          (PLUS 1 (LENGTH STR))))

HAS BEEN VERIFIED

(NODE 49 HAS BEEN CLOSED WRT NODE 48)

(THE PROOF IS COMPLETE)
```

C. Other Interactive Facilities

1. Facilities Supporting Hierarchical Verification

In our earlier report, [7], Sec. VII, we described a plan for a set of interactive facilities to support hierarchical verification. These facilities have been implemented during the RPE/2 phase of our work, in fulfillment of Task C of our Statement of Work which calls for "means for carrying out top-down/bottom-up proofs of program correctness for well-structured programs". In this subsection we describe the user-oriented facilities that our system provides for conducting such proofs.

The program structuring feature of JOCIT that best lends itself to hierarchical verification is procedure abstraction. We include therein the closely allied notion of functional abstraction--a function declaration differs from procedure declarations only in that no formal output parameter list is provided for functions. We have, therefore, based our approach to hierarchical verification (either top-down or bottom-up) on the use of JOCIT formal procedures. Another important aspect of hierarchical verification methodologies [13] is the notion of data abstraction. Unfortunately, JOCIT does not provide primitive constructs supporting data abstraction, and thus it is not within the scope of this effort.

a. Top-Down Verification

The verification system features discussed here are embedded in the verification condition generator (VCG) subsystem; they have already been described briefly in Section C. We describe here how these features enable one to carry out top/down program verification with our system.

A JOCIT main program will be processed by VCG (after transduction) to produce a set of VCs for the main program. These VCs will be stored in a Lisp variable named, for example, MPG*, if the main program is MPG. (Main programs do not have names in JOCIT, hence a suitable name must be provided by the user.) If MPG contains no embedded procedures (i.e., is nonhierarchical), all of these VCs pertain directly to MPG at a single abstraction level. Proof of their validity by our tableaux deductive system concludes the single-level (nonhierarchical) verification process.

Suppose, on the other hand, that MPG contains one or more procedure declarations. In the process of computing the VCs (MPG*) for MPG, the VCG subsystem will report to the user the names of procedures that it has discovered, and it will ask the user if he wishes their VCs to be computed at that time. If the user is employing the system to conduct verification concurrently with a top-down design, he will want to defer that computation. For example, a declaration for some procedure PROC1 contained in MPG may consist essentially of a formal parameter list and declarations, and it may contain only a dummy body, such as BEGIN END or a comment. The declarations within this unelaborated procedure should, however, contain at least an entry assertion and an exit assertion. These assertions specify what the procedure is supposed to do after it has been implemented. They must, of course, refer to the formal input and output parameters of the procedure. In this incomplete state, it would be meaningless to have VCG compute the VCs for PROC1, and the user will, therefore, defer that part of the VCG process. This approach corresponds to a top-down programming style, i.e., programming by successive refinements. Even though MPG might contain one or more calls to the still-unelaborated procedure PROC1, the VCG will produce an appropriate set of VCs for MPG, making use of the formal entry/exit assertions for PROC1. If these assertions are consistent with the way in which PROC1 is used in MPG, the main program VCs, MPG*, will be valid and presumably provable by our deductive system. That proof will show that the abstract implementation of MPG is consistent with the specifications for PROC1 (and any other unelaborated procedures directly contained within MPG.) Note, in particular, that, when this proof is carried out, the deductive system will not have to be separately informed (e.g., by means of axioms) about the procedural construct PROC1. The relevant information (extracted from the formal parameter lists and the entry/exit assertions) will be instantiated by the actual values for each call, and it will have been inserted into MPG*. (See the discussion about WP:CALL in Section C). This top-down verification style is suitable when procedures contained in a main program are to be explicated later. The important point here is that it will be possible to verify the consistency of MPG with the specifications for PROC1 (and any other procedures contained directly within MPG) before writing the actual code for these procedures.

However, at some later time the human programmer/verifier will wish to implement PROC1. At that time he will supply the missing body to PROC1, which might in turn contain references to other, lower-level procedures. In carrying out the next level of top-down programming and proof, the programmer has several choices. He may wish to go back to the original JOCIT main program and fill in the missing body for PROC1. After that, the next step would be to redo the parse and transduction of the modified main program. This is, of course, the most natural and systematic approach. However, a more direct approach can also be taken if one assumes that the user is sufficiently conversant with the abstract syntax to write the missing body for PROC1 directly in transduced form. He can do this within the VCG environment by invoking the MACLISP editor facility on the Lisp data structure PROC1, which is extracted from the main program MPG at the time the top-level VCG pass is made. The editor is invoked on proc1 by the commands:

```
(edit)
yp proc1 value <esc> <esc>
```

whereupon the transduced PROC1 declaration will appear in "windowed" form and the user can replace the dummy body by an actual one. For example, the dummy body might appear as "(BEGIN)," corresponding to the JOCIT syntax, BEGIN END. The user could use the Lisp editor's i(nsert) command to place the desired body statements after the word BEGIN. (Note: this description is based on the TECO-like string editor provided in TENEX MACLISPs; the edit function in MULTICS MACLISP is somewhat different in style.) The user would, probably, however, wish to confirm at some point that the abstract syntax he has thus hand-coded really corresponds to actual JOCIT. Thus, when he ultimately rewrites the JOCIT main program, he should redo the parsing and check that the parsed PROC1 is identical to the one he wrote directly. The virtue of working directly in the abstract syntax is that one may, thereby, eliminate several iterations through the parser. This would be the case if the code for PROC1 requires several successive changes before it is verifiable.

It should be noted that this method of elaborating a procedure declaration does not require the user to edit the abstract main program within which the procedure is declared. The corresponding editing of the

main program will occur automatically because Lisp uses structure sharing, and PROC1 simply points to the procedure declaration substructure under MPG. Thus, when PROC1 has been proved correct at the next lower level of verification, the user can rewrite the JOCIT program in accordance with the edited MPG, reparse the program, and compare the resulting parse, say newparse, with the edited MPG (by means of the MACLISP function EQUAL), thus:

(equal MPG newparse)

The next level of verification consists in generating the VCs for PROC1. This is the step that was deferred in generating only the top-level VCs for MPG. The user can do this either by again calling VCG on MPG or simply by invoking VCG:ALL (which handles VCG for main programs, subprograms, functions, procedures, and closes) on PROC1 (and any other procedures directly contained in MPG). If he elects the first option, the user will recompute the same top-level VCs (i.e., MPG*) and will again receive a message regarding the computation of VCs for PROC1. This time, he should answer "yes" to the system's query. The body of PROC1 will then be analyzed by the verification condition generator, and VCs for it will be computed and stored under the variable PROC1*. If the body of PROC1 contains (sub)procedure declarations, the process already described for the MPG level will be repeated, with the same user options, at the level of the procedure PROC1.

When all procedures, subprocedures, sub-subprocedures, etc., have been explicated in this way down to the level of JOCIT primitives, the user will have in hand a set of pointers to the main program and all of its subentities. Their names appear on the global variable, PROGRAMSLIST. He will also have a set of pointers to all of the VCs that have been generated. The names of these pointers appear on another global list, VCSLIST. These lists record for the user (and for the file-making mechanism) what portions of the main program have been analyzed by VCG. Upon each completion of VCG, the user is asked whether he wishes to save the computed VCs on a file (together with the main program). This option need not be exercised until the entire top-down VCG process is complete, unless it is desired to save the partial results. Whenever the answer is "yes," a MACLISP-loadable file containing MPG and all VCs determined up to that time will be stored in the

user's directory under the name MPG.VCS (where MPG is the name of the main program).

The user need not defer verification of the VCs until all VC levels have been computed. In fact, there is likely to be some advantage in using the deductive system to check the validity of VCs at each hierarchy level before proceeding to the next. In this way the user can detect programming errors before refinement has gone too far. In this case, the user should follow the following protocol:

> Answer "yes" to the file-making query at each stage of refinement of his program.
>
> Exit to the Tableaux system environment.
>
> Load the file *.VCS that has just been written by VCG, and invoke NEWPROOF successively on each clause of each entity (e.g., MPG*, or PROC1*, etc.) in the VCSLIST.
>
> Reenter the VCG environment, and proceed to the next stage of refinement.

b. Bottom-Up Verification

We have just described how program verification concurrent with design can be carried out under our system according to a top-down, successive refinement paradigm. The corresponding bottom-up process is supported equally well by our system; however, the purely bottom-up approach appears intrinsically less attractive than the top-down method for reasons independent of our implementation. These reasons will become clear in the discussion that follows. In summary, it will be established that some top-down thinking is a necessary part of bottom-up design/verification.

To carry out a bottom-up design with concurrent verification, the user must begin with some notion, at least, of the overall procedural hierarchy. He must, for example, have in mind some of the lower-level procedures that will ultimately be needed in his overall design. The user begins with a tentative design for one of the JOCIT procedures at the lowest level of the planned hierarchy. This procedure is first parsed and transduced by the methods described in Section II, Subsection B. The specifications attached to this procedure should, however, be adequate to

prove VCs for the next higher level of the procedural hierarchy. The user must be aware in advance of how he plans to use each procedure at the next higher level. A safe strategy is to specify low-level procedures with the strongest possible assertions--that is, to include in the exit assertion all properties of the exit parameters that could possibly be proved, and to make the entry assertion as general (i.e., as weak) as possible, subject to the correct functioning of the procedure. An example of the latter would be to refrain from restricting the domains of input parameters unless this is absolutely required by the functions and operations to be contained in the procedure body. VC generation is carried out for this low-level procedure (and others at the same level) exactly as described for the top-down process in the preceding subsection. These VCs are then proved by means of the deductive system, just as under the top-down process. This completes the design, implementation, and proof for the lowest level.

The user then proceeds to the next higher level by writing JOCIT procedures that call the lowest-level procedures. These next-level procedures are subjected successively to parsing/transduction and VCG. However, the analysis of the low-level procedures by VCG can be inhibited (by answering "no" to the system-generated query) because VCG has already been carried out on these procedures. The user also allows the VCG system to make a file containing the verification conditions for that level (accumulating those at the lower levels as well, if he wishes--this is determined entirely by the names appearing on PROGRAMSLIST and VCSLIST). Finally, the tableaux system is applied to the new VCs. In this way the user will build up his JOCIT program from the bottom, verifying at each stage that the visible program is consistent with the assertions at that level and with the (already proved) properties of the procedures it calls. When the top level (the final program) is reached, verification of the top-level VCs (e.g., MPG#) by Tableaux shows that the main program is consistent with its assertions, and this completes the proof. It would be wise at this point to regenerate all the VCs from the top level on down, as a double check that the bookkeeping operations in making successive VC files have not gone astray. These regenerated VCs can be checked mechanically against the saved VCs that were proved by tableaux during the bottom-up process. The means for doing the

checking were described in the preceding subsection in relation to comparing two parses.

In summary, it should be clear from the foregoing discussion that this process of building up a program by writing successive layers of procedures starting from the lowest level and working up demands a good deal of planning and forethought by the programmer. The general plan for the overall program must initially be, at least, in the back of the programmer's mind, if not well thought out. Moreover, entry and exit assertions for procedures at each level must reflect their anticipated uses at the next level. Finally, the data structures involved in low-level procedures will have to reflect the data structures at the higher (as yet undesigned) levels if the technique is to succeed. This last observation is relevant to considerations of data abstraction which we have necessarily excluded from consideration in this project. The reader is referred to [13] for a detailed discussion.

To conclude this subsection we should note that when our system is used purely for ex post facto verification of an existing program, then it is essentially irrelevant whether proof is carried out by the top-down or the bottom-up process. The only difference is the order in which procedures of the hierarchy are verified in terms of the specifications for lower-level procedures.

2. The Use of Dummy Assertions

Next, we discuss some additional features that were not covered in the preceding discussions of the top-down and bottom-up protocols. It is often inconvenient to include detailed assertions (especially for the loop assertions) in the early stages of top-down verification. One reason for this is that they may have to be modified when the program is debugged. However, the parser is relatively insensitive to the syntactic forms of assertions. In consequence, one may use one or the other of two informal kinds of "assertion abstraction" to simplify or curtail iterations through the parser.

In the first method to be described, the user supplies only mnemonic JOCIT names for one or more of the assertions. These names will later be replaced by fully explicated Boolean forms. The second method is a refinement of the first wherein dummy predicate forms are used for the assertions, but where the dependence of these predicates on selected program variables is made explicit.

We begin with a description of the first technique. The user writes a top-level program in JOCIT with all loop assertions (and perhaps also the input and output assertions) in the form of single mnemonic words, e.g., INASRT, OUTASRT, LOOPASRT1, LOOPASRT2, etc. Of course, these words must be acceptable to JOCIT syntax, so only legal JOCIT names are allowed. The following forms are permitted, respectively, for loop assertions, input assertions, and output assertions:

```
ASSERT LOOPASRT1 $
ASSERTIN INASRT $
ASSERTOUT OUTASRT $
```

The parser/transducer will convert these assertions, respectively, into the transduced forms:

```
(ASSERT LOOPASRT1)
(ASSERTIN INASRT)
(ASSERTOUT OUTASRT)
```

One advantage in deferring the choice of explicit assertions is that the user can thereby concentrate initially on achieving syntactic correctness for the JOCIT program itself, apart from the assertions. Another advantage is that he can execute VCG on this dummy-asserted program to check the flow of control between assertion points. This process is equivalent to a limited symbolic execution of the program in which mainly the control semantics are checked. The checking is not, however, automatic; the user must examine the generated VCs by eye and decide whether they conform to his expectations. He can check that the correct number of VCs is being generated. (This number will not change when the assertions are later explicated). He can also check the branch points of the program to see that at each branch, control is being transferred to the expected assertion point. Branch points (stemming from IF and IFEITH statements) correspond in the VCs to forms such as:

```
(AND (IMPLIES test-condition DUMMY-ASRT1)
     (IMPLIES (NOT test-condition) DUMMY-ASRT2))
```

These dummy-name assertions are not meaningful for the final VCG process because the dependence of the assertions on the program variables must be made explicit before the VCs will reflect partial correctness.

At this point, the user can apply either of the following two strategies or, as we shall see, some convenient combination of the two:

* Introduce explicit assertions in place of the dummy names used thus far, or
* Introduce dummy functional forms that parameterize the assertions without committing the user to specific functional forms.

The second of these strategies is the method of dummy predicate forms which we mentioned at the beginning of this subsection. The choice between these modes will depend on the user's confidence in getting the right assertions quickly. If he is confident in his ability to do so, he should write assertions (in abstract syntax) using the program variables and bind these assertions to the corresponding Lisp atoms used as dummy assertions. For example, the user might choose to define LOOPASRT1 completely by typing in:

```
(SETQ LOOPASRT1 '(AND (GTQ XX 0) (LTQ XX YY) (LT YY NN)))
```

He then needs to modify the transduced main program (MPG), where LOOPASRT1 appears, by expanding LOOPASRT1 in accordance with this definition. This is most easily done by typing:

```
(DSUBST LOOPASRT1 'LOOPASRT1 MPG).
```

Each dummy assertion used in MPG must be dealt with similarly. As a result, MPG will be a version of the transduced MPG containing fully explicated assertions. This version can then be subjected to the generation of VCs, viz, by calling (VCG MPG). Observe, too, that if these dummy assertions were buried within subprocedures to MPG, any existing bindings to these subprocedures will be left intact because DSUBST uses the same structure sharing as the editor, i.e., it does not make copies. The same desirable result holds when the DSUBST is carried out at the level of a subprocedure, i.e., the main program (or any procedures above the DSUBST'd procedure) will be altered correspondingly.

On the other hand, the second mode for instantiation of dummy-name assertions permits the user to approach the construction of loop assertions (and entry/exit assertions) in a more gradual manner. The technique is similar to that of the first mode except that the user need only supply the appropriate program variable names for each dummy assertion, leaving the functional forms of the assertions to be determined later. For example, the loop assertion, LOOPASRT1, might be defined by typing:

```
(SETQ LOOPASRT1 '(LA1 XX YY NN))
```

where the user anticipates that only the variables XX, YY, NN will be needed in the ultimate assertion, and he leaves the form of the dummy predicate LA1 to be determined later. After all the dummy-name assertions have been defined either in this partially expanded form or by explicit forms, and the appropriate DSUBST operations have been carried out, VCG may then be invoked on the program. The user needs to examine the resulting VCs closely to determine the functional forms for the dummy predicates LA1, etc., that satisfy these VCs. In general this is a difficult process, but the human program verifier should be able to bring to bear his understanding of the program and its intent to facilitate the task. In fact, he may combine the two modes of instantiation by writing partial explications for some assertions in such forms as:

```
(SETQ LOOPASRT1 '(AND (GTQ XX 0) (LTQ XX YY) (LA11 YY NN))).
```

Here, the user presumably knew (or guessed) the relations among 0, XX, and YY, but was unsure of the proper relation between YY and NN. By leaving that portion of the assertion in parameterized dummy form, he may be able to determine quickly from the VCs that the appropriate relational operator is LT (and not, e.g., LTQ).

The reader is referred to [6] for a more detailed discussion of techniques for generating adequate inductive assertions. In particular, the method of finite difference equations (described there) is applicable to the invention of many types of algebraic invariants.

3. User-Defined Constructs in the Assertion Language

In the immediately preceding discussion, we have described some user-oriented features that enable a human programmer/verifier to work with dummy assertions during the process of writing and verifying a program. By using those features he is, in effect, creating user-defined predicates of a very local and temporary sort. These assertion forms disappear before the final VCG process and, therefore, play no part in the actual deductive proof. In contrast, we next describe a mechanism whereby the user may introduce formal definitions for constructs that he may wish to use over a considerable period of time for verifying different programs. The algebraic simplifier contains built-in features (described below) that provide the system user with means for creating his own assertion language of functional (and predicate) abstractions and integrating them into the deductive mechanisms. The assertion language accepted by the deductive system is thereby made to be user-extendible. See also, [7], pp. 71-74, for a description of the RPE/1 version of this feature.

Assertions written in terms of JOCIT primitives tend to be rather lengthy, even for small, "toy" programs. In part, this is because the relational primitives (the equality, nonequality, and inequality relations) are at too low a level. For many programs one needs to have relational abstractions concerned with higher-level data structures, e.g., arrays, files, tables, records, etc. One set of facilities for introducing such abstractions exists in the Tableaux system's mechanism for invoking lemmas (see Section B).

However, invoking axioms for each occurrence of some user-defined construct can prove to be excessively tedious in running the deductive system. For one thing, the user must supply instantiations for each axiom invocation. We could, for example, handle deductions about the JOCIT intrinsic function for absolute value (ABS) entirely by introducing the axiom:

```
(FORALL X (AND (IMPLIES (GTQ X 0) (EQ X (ABS X))
               (IMPLIES (LT X 0) (EQ (MINUS X) (ABS X))))).
```

This axiom tells the deductive system all it needs to know about the function ABS. In practice, this would make proofs even of such elementary facts as

(EQ (ABS (ABS (ABS (ABS X)))) (ABS X))

rather tedious. The tableaux proof of this fact would require numerous separate instantiations of the above ABS axiom, each time with different user-supplied bindings.

An alternative we have investigated is to incorporate rewrite rules for these functions into the simplifier. The rule for ABS could contain the knowledge that, for numeric x,

(ABS x) = if x>0 then x else -x,

and also the knowledge that (ABS (ABS x)) = (ABS x), even though this fact is deducible from the definition, and is, therefore, redundant. Any invocation of ALGEBRA from the deductive system then automatically attempts to simplify forms of the type (ABS *) by rewriting them. In this approach instantiation is automatic, unlike the application of axioms by INVOKE in the Tableaux subsystem. Similar rules have been built into the simplifier for functions such as MAX, MIN, and, of course, for the algebraic functions PLUS, TIMES, SUBTRACT, MINUS, DIVIDE, and EXPT (which are primitives in *JOCIT*, under the usual infix operator names, +, *, -, /, and **) as well as for the relational operators, EQ, NEQ, LT, GT, LTQ, and GTQ (corresponding to the JOCIT infix primitives, EQ, NQ, LS, GR, LQ, and GQ).

It is highly desirable to allow the user to create his own rewrite rules for the simplifier to facilitate deductions for new, user-defined abstractions without the need for invoking axioms.

In the fast string search program that we have verified (see also Section B), extensive use was made of user-defined abstractions as parts of the program's loop assertions. For example, (LENGTH string), (STRPOS str1 str2), and (SUBSTRING string k 1) all figure in one or more assertions. STRPOS is central because the program is, in effect, supposed to compute (STRPOS pat str) = the first position, j, in str where pat matches the substring of str from j to (LENGTH pat)-1. The proof shown in Section B depends on the invocation of several plausible axioms relating these three abstractions. An alternative is to incorporate directly into the expression simplifier enough information about these string function abstractions to permit the simplifier to handle some or all of the proof. For example, the axiom, SUBSTR:NIL:

(IMPLIES (GT K L) (EQ (SUBSTRING str k 1) NIL))
could readily have been set up as a conditional rewrite rule for SUBSTRING.

In order to provide the system user with a convenient facility for constructing such rewrite rules, our simplifier contains a function, SPECIFY, which automates much, but not all, of the effort entailed in this construction. See [7], pp. 71-73 for a discussion of SPECIFY and how it is used.

To construct a rewrite rule for SUBSTRING the user calls SPECIFY with arguments supplying the name of the construct (i.e., SUBSTRING), a formal parameter list (in this case, str, k, and 1), and a list of special cases under which specified simplified forms are to be returned by the simplifier. There is also an optional argument that permits the user to specify (when appropriate) that the function has certain special properties, such as commutativity. SPECIFY then constructs a rewrite rule SUBSTRING$ that combines these simplifications in a single Lisp function. This rule is invoked whenever the simplifier sees an expression of the form (SUBSTRING * * *).

Our work with this part of our system has pointed up some problems associated with the use of systems of rewrite rules, and with SPECIFY in particular. First, the function, SPECIFY, is far from being an automatic rewrite rule generator. It lacks the ability to perform pattern-match compilation, which would be a very desirable feature. With pattern matching in the construction of rewrite rules, we could simply input a set of algebraic specifications to this new system, and have it construct for us the detailed Boolean tests needed to match these specifications. Given a suitable parser (e.g., one constructed by the parser-generator technique) one might even input algebraic specifications in a non-Lisp syntax. A second, more general, problem associated with this kind of deduction is that simplifier rules need to interact in peculiar ways, e.g., since the functions ABS, SQRT, and (TIMES x x) are all inherently nonnegative, one would like the forms, (ABS (ABS X)), (ABS (SQRT X)), and (ABS (TIMES X X)) simply to return the arguments to the outermost ABS. However, in order to make this fact known to the rule for ABS, one needs to put into ABS$ a great deal of

information about other functional constructs. Moreover, when the system is extended by adding new user-defined constructs, the possibilities for such interactions may multiply rapidly. It is likely that these considerations set some sort of practical limit to what one can conveniently do in deduction simply through the use of rewrite rules. Fortunately, our system is also able to handle arbitrary information in the form of predicate calculus axioms when the capabilities of algebraic simplification are strained by the above considerations. The two approaches complement one another in nice ways. It is likely that both types will continue to play a role in both automatic and semiautomatic deductive systems.

V RPE/2 ON RADC-MULTICS

A. Introduction

Task D (Item 4.1.4) of our Work Statement requires us to "investigate the transferability of the verification files by installing as much as necessary of the SRI PDP-10 files on the Honeywell 6180 MULTICS system to run a simple JOVIAL (J3) program for verification." Although we planned initially to transfer the RPE/1 system to the RADC-MULTICS computer to satisfy this requirement, it proved possible to transfer the entire RPE/2 verification system and use it for the demonstration run.

The files and sample programs required to carry out this verification are now part of user Spitzen's file directory on RADC-MULTICS. In this chapter, we first describe the translation of portions of the original INTERLISP verifier into the MACLISP language available both on SRI's KL-20 computer and RADC's Honeywell 6180 MULTICS system. We next describe the transfer of the resulting MACLISP programs to RADC-MULTICS. Finally, we describe the JOVIAL (J3) program that we verified at RADC-MULTICS. Appendix C gives detailed instructions for reproducing this verification run.

B. Translation to MACLISP

As indicated above, we initially set out to translate the existing RPE/1 verifier from its implementation in INTERLISP into a version of MACLISP available on the SRI KA-10 computer. As the translation effort progressed, it became clear that with relatively little extra effort we could carry on most of the development work for the RPE/2 system directly in MACLISP. From about June 1976 on, that strategy was followed. As a result, by the end of 1976 we had a usable MACLISP implementation for RPE/2. Some changes were carried out beyond that time, but the system framework was essentially complete.

The MACLISP system originally available on the KA-10 was Version 1132, running under TENEX. This has been successively replaced by Version 1251 and most recently by a version called 1MID or 1258. Observable differences between these Tenex MACLISPs have been negligible in their effect on our work. Apparently, the implementers of this language have been concerned with various improvements in efficiency of speed, storage, and garbage collection, but they have maintained upward compatibility between versions. In fact, code compiled under Version 1251 appears to run properly under the newest (1MID-1258) executive and even on our new machine (SRI-KL). Fortunately, we have not been plagued by compatibility and conversion problems as a result of these language and machine changes.

A second, relatively minor conversion was involved when we mounted the MACLISP system on the RADC-MULTICS (Honeywell 6180) computer. The MACLISP system on that machine was, naturally, a MULTICS version. Initially, we had some problems in getting accustomed to MULTICS conventions, but these problems largely solved themselves in time. Apart from minor differences in the I/O features and certain functions in MULTICS MACLISP which had to be synthesized at SRI, we encountered no problems in this second phase.

The starting point for the MACLISP RPE/2 system was taken from the RPE/1 INTERLISP files for the tableaux deductive system and the algebraic expression simplifier, the latter a subsidiary package used by the tableaux system. An improved Presburger mechanism was rewritten for RPE/2 in INTERLISP and converted to MACLISP in November-December 1976 by special means described below. The VCG subsystem was begun in INTERLISP, carried about half-way to completion (providing only the basic core constructs), then translated into MACLISP. This core of the final MACLISP VCG was used in the RADC-MULTICS sample demonstration. VCG was completed early in 1977 entirely in MACLISP by making modifications and additions to the core VCG subsystem. The tableaux deductive system was similarly developed in MACLISP from the initial MACLISP version.

The parser/transducer represents a different situation, because it is a machine-synthesized file package; its mode of construction is described in general terms at the end of this subsection. For details of the

parser/transducer the reader is referred to Section II-B. Documentation for MACLISP is comparatively poor, and somewhat out of date; the most recent definitive description is [12]. Fortunately, we were also able to obtain from MIT a draft document containing detailed comparisons of MACLISP with INTERLISP. These notes supplied, for each of the major functions of INTERLISP, a description of the corresponding feature in MACLISP, or--if none existed--with equivalent MACLISP code. We carefully tested each such equivalence before adopting it.

We now describe briefly the principal differences between INTERLISP and MACLISP that affected our translation. In addition to these, there are many other differences that affect the form of a user's commands to the system. For example, MACLISP lacks the evalquote mode, which lets the INTERLISP user type FOO(X) with the same effect as the more cumbersome (FOO 'X). Making files and saving core images differ not only between INTERLISP and MACLISP but also among different MACLISPs and different machines. Conventions for user interrupts (e.g., the use of control characters) are different between INTERLISP and MACLISP, but fortunately they are fairly standard between MACLISP implementations.

The following very useful functions of INTERLISP are entirely lacking in MACLISP (or are so different that they must be hand coded):

> ADDPROP, ALPHORDER, ARGLIST, ATTACH, CHANGENAME, CHANGEPROP, COPY, COUNT, DEFLIST, DREMOVE, DSUBST, EQP, EQLENGTH, EVERY, FASSOC, FILEPOS, FNTYP, GETD, INTERSECTION, LCONC, LDIFF, LISTP, LITATOM, LOADFNS, LOADVARS, MAKEFILE, MOVD, NCHARS, NCONC1, NEQ, NLISTP, NTH, PACK, PACKC, PROG1, REMOVE, SELECTQ, SOME, SPACES, SUBPAIR, SUBSET, TCONC, UNION, UNPACK.

To avoid confusion, in the remainder of this section we will refer to INTERLISP functions in uppercase characters and to MACLISP functions in lowercase. Note that some names occur in both languages, but with different meanings.

Most of the functions listed were created in terms of MACLISP primitives and saved in a basic Library Functions file (LIB.MAC). This file also contains a few other functions so widely used throughout the verifier that

they belong in a common library (even though they are not primitive to INTERLISP). LIB.MAC also contains some versions of MACLISP primitives so modified as to behave like their INTERLISP analogues. For example, (FIX -1.5) = -2, while (fix -1.5) = -1; hence fixx was defined to act like FIX. Similarly, since (zerop 'a) causes an error interrupt while (ZEROP 'A) = NIL, a function, zerops, was defined to match ZEROP. The library function file package LIB.MAC was later compiled into a binary file LIB.FAS.

Many of the differences between MACLISP and INTERLISP stem from fundamental differences in function definition and concept. For uncompiled functions, MACLISP has only the types "expr" and "fexpr" (corresponding to INTERLISP types, EXPR and FEXPR*). MACLISP's exprs also cover what are called EXPR*'s in INTERLISP. There is no MACLISP analog to INTERLISP "FEXPR". Whereas INTERLISP stores function definitions in a special "function cell", MACLISP stores them as expr or fexpr properties of the atomic function symbol. In MACLISP, calls to exprs must be supplied with exactly the right number of arguments, unlike the sometimes handy (but often dangerous) INTERLISP default: binding missing arguments to NIL.

Fortunately, J S. Moore developed a superb translation tool, called MACLISPIFY. A portion of the small development cost of this support software was borne by another project. MACLISPIFY is best described as an interactive translator-builder; it is coded in INTERLISP. Source code and compiled versions are on directory <MOORE>MACLISPIFY on SRI-KL. In its pristine state, MACLISPIFY is acquainted with the conventions of INTERLISP, including the names, numbers of arguments, and types of all INTERLISP functions. MACLISPIFY allows the user, as he proceeds, to build up a set of translation rules (called "translation augments") between INTERLISP and any other Lisp dialect. Moore also supplied us with a set of primitive translation augments for many of the most common MACLISP functions (including the special forms discussed below). The user of MACLISPIFY can simply turn it loose on a set of function definitions in INTERLISP by typing:

```
(APPLY 'MACLISPIFY fnslist T)
```

where fnslist is bound to the list of function names. MACLISPIFY is then invoked successively on the definitions of each function on fnslist. The

user is asked whether he wishes MACLISPIFY to recursively pursue the Maclispification of subroutines. He can reply either A(sk me each time for permission to Maclispify them), M(aclispify them without asking), or D(on't Maclispify them). When MACLISPIFY encounters an expression whose CAR lacks a translation rule, it asks the user if he wishes to supply one. The user's options at this point are to respond with one of the following:

<linefeed>

= <function-name>

N(o) => Always bring it to my attention.

M(aybe later) => The form will be brought to your attention when the parent form has been Maclispified. No augment is created for this function symbol now, and you will be asked this question again the next time the function symbol is seen.

Y(es) => causes the Editor to be entered with a standard augment template.

The first option converts the INTERLISP name to its lowercase (MACLISP) equivalent, and also proceeds to "Maclispify" each argument expression. This user response is appropriate much of the time, i.e., whenever MACLISP uses a function with the same name and semantics as INTERLISP. Examples of this are: CONS->cons, LIST->list, CAR->car, etc. All such obvious instances have already been provided with transduction augments in the initial set; thus, the user would not be asked this question for functions like CONS, LIST, CAR, and CDR--only for functions lacking a translation augment.

The second option is used when there is a semantically equivalent function in the MACLISP environment with a different name than the INTERLISP function to be translated. The pair MEMB, memq is an example.

Neither of the first two options is appropriate when INTERLISP and MACLISP have functions with the same names but different semantics. The mapping functions, MAPCAR, MAPC, MAP, MAPCON, and MAPLIST are good examples for several reasons. First, the argument order is reversed in MACLISP, i.e. (MAPCAR LST 'ATOM) is correctly rendered as (mapcar 'atom lst), and similarly for the other mapping functions. Also, mapcar, mapc, etc., can take any number of list arguments; that number must be equal to the number of arguments required by the functional argument. On the other hand, INTERLISP

mapping functions can take an optional third (functional) argument to be used (instead of the default CDR) for iterating down the list. (A minor difference: the sixth mapping function, MAPCONC, corresponds to mapcan, instead of "mapconc".) Our version of MACLISPIFY uses one fairly complicated augment, {STANDARD-MAP-AUGMENT}, to cover all these contingencies for the six mapping functions simultaneously. For example, when MACLISPIFY sees an INTERLISP form like (MAP LST 'ATOM 'CDDR), the warning message is printed out on the user's terminal: "MACLISP mapping functions do not accept the third INTERLISP argument."

A great virtue of MACLISPIFY is that the user interactively builds up a set of translation rules as he proceeds to translate his INTERLISP code. Once a translation rule has been supplied to it, MACLISPIFY will in the future automatically use that rule to perform translations for that function symbol. The MACLISPIFY system is extremely easy to use by virtue of the human engineering that has gone into its design. At places where the user might be in doubt as to how to proceed, typing "?" will tell him what options are open to him at that point.

MACLISPIFY stores translation augments as no-argument function definitions for INTERLISP names surrounded by curly braces, {}. For example, the simplest possible augment (created whenever the user uses the <linefeed> lowercasing option) is (LAMBDA NIL form). This is the augment for CONS, LIST, CAR, CDR, etc. Augments of the second type (created by = <function-name>) have the form, e.g., {MEMB} = (LAMBDA NIL (CONS 'memq arglst). More complicated augments than these must be written by the user, working within the INTERLISP editor.

For example, the translation augment:

{PUT} = (LAMBDA NIL (LIST 'putprop arg1 arg3 arg2)

would be written by the user to indicate that the translation for (PUT A I V) should be (putprop a v i). A useful convention with MACLISPIFY is that ARG1, ARG2,... denote the original (INTERLISP) arguments to a form, while arg1, arg2,... denote their Maclispified versions. A similar convention applies to the pairs, FNNAME, fnname; ARGLST, arglst; and FORM, form. All of these names are used freely by the translation functions, with FORM being bound to

the form currently under translation and FORM = (FNNAME ARG1 ARG2 ...) = (FNNNAME . ARGLST).

The so-called special forms of INTERLISP, such as SELECTQ, COND, and PROG, which do not first evaluate (all) their arguments (or which evaluate them in a nonstandard order), need special treatment for Maclispification. MACLISPIFY provides built-in handling for such forms. Note, too, that INTERLISP (but not MACLISP) allows initialization of PROG variables in the declaration clause. Hence, the INTERLISP code:

```
(PROG ((X 1)(Y LST)) ...)
```

is automatically rewritten as:

```
(prog (x y) (setq x 1) (setq y lst) ...)
```

by MACLISPIFY.

Finally, let us mention the translation of CLISP contructs into MACLISP. CLISP forms (see, e.g., [17]) provide a kind of "unofficial" INTERLISP in which a user can type in rather free form expressions (e.g., a great variety of iterative statements, pattern-matching expressions, infix forms, etc.) to INTERLISP, even though these forms are not directly comprehensible by the INTERLISP interpreter or compiler. They are intercepted by the DWIM ("Do What I Mean") mechanism (part of the INTERLISP Programmer's Assistant) and translated into standard INTERLISP before interpretation. DWIM contains canned rules for expanding CLISP iterative forms, e.g.,

```
(for X from 1 to 100 while (NOT (FOO X)) do ...),
```

into explicit PROG loops. MACLISPIFY uses this DWIM compiler first to translate CLISP forms into INTERLISP, but it also leaves the original CLISP in place as a comment in the expanded code if the user wishes. Then the main body of MACLISPIFY translates the INTERLISP code, leaving the comment still in the MACLISP translation for clarity.

The MACLISPIFY system also asks the user to specify, for each MACLISP function name, whether it refers to a S(ystem) function, a L(ibrary) function, a U(ser) function, or simply to F(orget it!). In this way, MACLISPIFY is able to allocate it to a suitable list for filing. "System" functions are supposed to be the primitives of MACLISP (or whatever Lisp dialect one is translating into); "library" functions have already been

described; "user" functions are supposed to be part of the package being translated. When MACLISPIFY exits, it creates both an INTERLISP file with all of the information that has been processed and a MACLISP-loadable file with definitions of functions for the translated code in the form of putprop statements. In the INTERLISP file, the definitions of MACLISP functions are saved as expr or fexpr properties of the (lowercase) function names. All of the translation augments (the initial set plus augments created by the user during the session) are also saved on the INTERLISP file for future use.

We now discuss the parser/transducer "translation" process. The details of our construction of the parser/transducer are given in Section II-B. Since the parser/transducer is a machine-synthesized subsystem of the verifier, it would have been inefficient to produce first a parser/transducer in INTERLISP and then translate it into MACLISP. Instead, the parser-generator (an INTERLISP software tool that produces the parser/transducer from a syntactic description of the target language) can produce either an INTERLISP or a MACLISP parser for the target language.

C. Transfer to RADC-MULTICS

Having obtained MACLISP source code for the verification system, we used the ARPANET File Transfer Program (FTP) running at MULTICS to move the source files from the SRI KL-20 to the RADC-MULTICS system. (Because of problems with FTP on the KL-20, we were not able to use it to "send" source files to RADC; all file transfer was achieved by "getting" files from SRI to RADC.)

We then attempted to run the verification system composed of these transferred files in MULTICS MACLISP. Minor problems were encountered:

* A number of primitive functions in MULTICS MACLISP, for example, charpos, must be synthesized in KL-20 MACLISP.
* The input and output primitives of MULTICS MACLISP and KL-20 MACLISP use slightly different conventions. Also, the file systems of the two machines use different methods for naming files.

In all cases, however, minor reprogramming sufficed to obtain source files that run correctly with MULTICS MACLISP.

D. Sample Verification at RADC-MULTICS

The program that we verified is given in Appendix C, Part 1. It searches for a floating point number VV in an array AA of length 1000. If the number is found within the initial segment of AA of length NN, the program halts with the Boolean FLG set to 1 and the integer LOCN set to the first index of VV in AA. If the number is not found in this segment, the program halts with FLG set to 0 and the temporary variable INDEX set to NN+1. These conditions are described in formal terms by an output assertion at the program label OUT; the verification also requires three inductive assertions at labels L1, L2, and L3. The entire program is 37 lines long and includes 23 lines of assertions and 14 of declarations and executable code.

The first step in verifying this program is to parse and transduce it into the abstract syntax processed by the verification condition generator. This is done by invoking the lisp system at MULTICS and issuing the command

```
(load "loadjmac.mac")
```

which causes the parser to be loaded into the MACLISP environment. When loading is complete, the loading program types the message

"Ready to parse."

The user may now invoke the parser by calling the function parse whose single argument is the name of a file containing a JOVIAL program to be verified. In the example, the required command is

```
(parse "search.joc")
```

Following the parse and transduction of the source file, the variable parse is bound to the resulting abstract form. Because we were running with a limited number of available segments, it was not possible to configure a single MACLISP environment containing the entire verification system. At this point, we must, therefore, send the abstract form to a file, which we do by issuing the commands

```
(setq svars '(abstractsearch))
(setq abstractsearch parse)
(mfile s)
```

This causes the file s.mac to be created containing the abstract syntax and thus completes this phase of verification. The abstract syntax for the sample program is given in Appendix C, Part 2.

Next, it is necessary to generate verification conditions from the abstract syntax. We must first escape from lisp to the MULTICS command level and invoke a new lisp environment. This environment is configured for generating verification conditions by issuing the command

(load "loadvcg.mac")

When the system is ready, we reload the abstract program by saying

(load "s.mac")

and are then ready to generate verification conditions by commanding

(vcg abstractsearch)

which results in the binding of the variable abstractsearch* to a list of the verification conditions for the program - five in this example. We then complete the verification condition generation phase by adding the elements abstractsearch*, a1, a2, a3, a4, and a5 to the list svars, binding a1, a2, a3, a4, and a5, respectively, to the five elements of abstractsearch*, and commanding

(mfile s)

to write a new s.mac including the verification conditions. These five conditions are given in Appendix C, Part 3.

We are now ready to prove these verification conditions. We escape to MULTICS command level, obtain a new lisp, and say

(load "loaddeduct.mac")

to configure an environment with the deductive system. We are then ready to prove, in turn, each of the five verification conditions. It turns out that a1 and a5 can be proved with just a few user commands, a3 and a4 can be proved automatically, and a2 requires a somewhat lengthy dialog. The details are given in Appendix D, Part 4.

Appendix A
JOCIT GRAMMAR

Appendix A

JOCIT GRAMMAR

1. Grammar Rules

```
0   1   **ROOT**          ::= <prog> RIGHT\PAD

1   2   <prog>            ::= <main\prog>
    3                     ::= <sub\prog>

2   4   <a\p\l>          ::= **empty**
    5                     ::= ( <fstrdot> <aopl> )

3   6   <alt\stat>       ::= IFEITH <f> $ <stat>
                                <orif\listpls> END

4   7   <aopl>           ::= **empty**
    8                     ::= = <onamepls>

5   9   <assign\stat>    ::= <variable> = <f> $

6   10  <bch>            ::= B
    11                    ::= CH

7   12  <body>           ::= **empty**
    13                    ::= <body> <decl>
    14                    ::= <body> <labelstr> <stat>

8   15  <cht>            ::= CH
    16                    ::= T

9   17  <const\list>     ::= BEGIN\ <dclpls> END

10  18  <const\listq>    ::= **empty**
    19                    ::= <const\list>

11  20  <constant>       ::= CHARACTER\CONSTANT
    21                    ::= STATUS\CONSTANT
    22                    ::= <numeric>

12  23  <controls>       ::= <controls2>
    24                    ::= <controls2> ASSERT\ASSUME <f>

13  25  <controls2>      ::= ALL ( NAME )
    26                    ::= <f>
    27                    ::= <f> , <f>
```

```
    28                  ::= <f> , <f> , <f>

14  29  <cstat>         ::= BEGIN <body> END

15  30  <dcl>           ::= <const\list>
    31                  ::= <os\const>

16  32  <dclpls>        ::= <dcl>
    33                  ::= <dclpls> <dcl>

17  34  <decl>          ::= ARRAY NAME <intpls> <i\desc> $
                            <const\listq>
    35                  ::= ASSERTINOUT <f> $
    36                  ::= CLOSE NAME $ <cstat>
    37                  ::= FILE NAME <bch> <file\size> <vr>
                            <record\size> FILE\STATES
                            NAME $
    38                  ::= ITEM NAME <i\desc> <pq> $
                            <const\listq>
    39                  ::= ITEM NAME <os\const> $
    40                  ::= PROC NAME <parameters> $
                            <pddpstr> <cstat>
    41                  ::= PROGRAM NAME $
    42                  ::= SWITCH NAME = ( <insdlpls> ) $
    43                  ::= SWITCH NAME ( NAME ) =
                             ( <itsdlpls> ) $
    44                  ::= <overlay\decl>
    45                  ::= <str\i\decl>
    46                  ::= <table\prfx> <packingq> $ BEGIN
                            <oei\decl> <oeid\sodpls> END
    47                  ::= <table\prfx> <packingq> L $
    48                  ::= <table\prfx> DECIMAL\INTEGER $
                            BEGIN <dtid\sidpls> END

18  49  <dti\decl>      ::= ITEM NAME <dti2> <packingq> $
                            <const\listq>

19  50  <dti2>          ::= <i\desc2> <integer> <integer>
    51                  ::= AI <integer> <su> <dti3>

20  52  <dti3>          ::= <plusmin> <integer> <dti4>
    53                  ::= <integer> <dti4>
    54                  ::= <dti4>

21  55  <dti4>          ::= R <dti5>
    56                  ::= <dti5>

22  57  <dti5>          ::= RANGE\PRFX <numeric> <dti6>
    58                  ::= <dti6>
```

```
23  59  <dti6>          ::= <integer> <integer>

24  60  <dtid\sid>      ::= <dti\decl>
    61                  ::= <str\i\decl>

25  62  <dtid\sidpls>   ::= <dtid\sid>
    63                  ::= <dtid\sidpls> <dtid\sid>

26  64  <exch\stat>     ::= <variable> == <variable> $

27  65  <f>             ::= <f> OR <f2>
    66                  ::= <f2>

28  67  <f2>            ::= <f2> AND <f3>
    68                  ::= <f3>

29  69  <f3>            ::= NOT <f3>
    70                  ::= <f4>

30  71  <f4>            ::= <f5> <rel\sufxstr>

31  72  <f5>            ::= <f5> <plusmin> <f6>
    73                  ::= <f6>

32  74  <f6>            ::= <f6> <multdiv> <f7>
    75                  ::= <f7>

33  76  <f7>            ::= <f7> ** <f8>
    77                  ::= <f7> LPAR\STAR <f> STAR\RPAR
    78                  ::= LPAR\SLSH <f> SLSH\RPAR
    79                  ::= <f8>

34  80  <f8>            ::= ( <f> )
    81                  ::= <constant>
    82                  ::= <func\ref>
    83                  ::= <variable>
    84                  ::= <plusmin> <f8>

35  85  <file\size>     ::= <constant>

36  86  <fpls>          ::= <f>
    87                  ::= <fpls> , <f>

37  88  <fplsdot>       ::= NAMEDOT
    89                  ::= <f>
    90                  ::= <fplsdot> , <f>
    91                  ::= <fplsdot> , NAMEDOT

38  92  <fstrdot>       ::= **empty**
    93                  ::= <fplsdot>
```

```
39  94  <func\ref>      ::= NAME ( <fstrdot> )

40  95  <goto\formula> ::= NAME <xq>

41  96  <goto\stat>    ::= GOTO <goto\formula> $

42  97  <i\desc>       ::= AI <integer> <su> <os\intq> <rq>
                           <rangeq>
    98                 ::= <i\desc2>

43  99  <i\desc2>      ::= B
    100                ::= F <rq>
    101                ::= S <integerq> <statuspls>
    102                ::= <cht> <integer>

44  103 <inout\stat>   ::= INOUT NAME <iolist> $
    104                ::= OPSH INOUT NAME <iolistq> $

45  105 <inputs>       ::= <inputs> , <oname>
    106                ::= <oname>

46  107 <inputsq>      ::= **empty**
    108                ::= <inputs>

47  109 <insdl>        ::= **empty**
    110                ::= <goto\formula>

48  111 <insdlpls>     ::= <insdl>
    112                ::= <insdlpls> , <insdl>

49  113 <integer>      ::= DECIMAL\INTEGER
    114                ::= OCTAL\INTEGER

50  115 <integerq>     ::= **empty**
    116                ::= <integer>

51  117 <intpls>       ::= DECIMAL\INTEGER
    118                ::= <intpls> DECIMAL\INTEGER

52  119 <iolist>       ::= <f>
    120                ::= <iolist> , <f>

53  121 <iolistq>      ::= **empty**
    122                ::= <iolist>

54  123 <itsdl>        ::= <os\const> = <goto\formula>

55  124 <itsdlpls>     ::= <itsdl>
    125                ::= <itsdlpls> , <itsdl>
```

```
56  126 <labelstr>      ::= **empty**
    127                 ::= <labelstr> NAMEDOT

57  128 <letter>        ::= AI
    129                 ::= B
    130                 ::= CH
    131                 ::= DMN
    132                 ::= EGJKOQWXYZ
    133                 ::= F
    134                 ::= L
    135                 ::= P
    136                 ::= R
    137                 ::= S
    138                 ::= T
    139                 ::= U
    140                 ::= V

58  141 <main\prog>     ::= START $ <body> TERM <nameq> $

59  142 <multdiv>       ::= *
    143                 ::= /

60  144 <namepls>       ::= NAME
    145                 ::= <namepls> , NAME

61  146 <nameplspls>    ::= = <namepls>
    147                 ::= <nameplspls> = <namepls>

62  148 <nameplsstr>    ::= **empty**
    149                 ::= <nameplspls>

63  150 <nameq>         ::= **empty**
    151                 ::= NAME

64  152 <numeric>       ::= FIXED\CONSTANT
    153                 ::= FLOATING\CONSTANT
    154                 ::= <integer>

65  155 <oei\decl>      ::= ITEM NAME <i\desc> $
                                <const\listq>

66  156 <oeid\sod>      ::= <oei\decl>
    157                 ::= <overlay\decl>

67  158 <oeid\sodpls>   ::= <oeid\sod>
    159                 ::= <oeid\sodpls> <oeid\sod>

68  160 <oname>         ::= NAME
    161                 ::= NAMEDOT
```

```
69  162 <onamepls>      ::= <oname>
    163                 ::= <onamepls> , <oname>

70  164 <orif\list>     ::= <labelstr> ORIF <f> $ <stat>

71  165 <orif\listpls>  ::= <orif\list>
    166                 ::= <orif\listpls> <orif\list>

72  167 <originq>       ::= **empty**
    168                 ::= <integer> =

73  169 <os\const>      ::= + <constant>
    170                 ::= - <constant>
    171                 ::= <constant>

74  172 <os\intq>       ::= <integerq>
    173                 ::= <plusmin> <integer>

75  174 <outputsq>      ::= **empty**
    175                 ::= = <inputs>

76  176 <overlay\decl>  ::= OVERLAY <originq> <namepls>
                                <nameplsstr> $

77  177 <packingq>      ::= **empty**
    178                 ::= DMN

78  179 <parameters>    ::= **empty**
    180                 ::= ( <inputsq> <outputsq> )

79  181 <pcall\stat>    ::= NAME <a\p\l> $

80  182 <pddpstr>       ::= **empty**
    183                 ::= <pddpstr> <decl>

81  184 <plusmin>       ::= +
    185                 ::= -

82  186 <pq>            ::= **empty**
    187                 ::= P <os\const>

83  188 <rangeq>        ::= **empty**
    189                 ::= RANGE\PRFX <numeric>

84  190 <record\size>   ::= <numeric>

85  191 <rel\sufxstr>   ::= **empty**
    192                 ::= <rel\sufxstr> REL\OP <f5>

86  193 <return\stat>   ::= RETURN $
```

```
87  194 <rq>            ::= **empty**
    195                 ::= R

88  196 <stat>          ::= ASSERT\ASSUME <f> $
    197                 ::= DIRECT JOVIAL
    198                 ::= FOR <letter> = <controls> $
    199                 ::= IF <f> $
    200                 ::= <alt\stat>
    201                 ::= <assign\stat>
    202                 ::= <cstat>
    203                 ::= <exch\stat>
    204                 ::= <goto\stat>
    205                 ::= <inout\stat>
    206                 ::= <pcall\stat>
    207                 ::= <return\stat>
    208                 ::= <stop\stat>
    209                 ::= <test\stat>

89  210 <statuspls>     ::= STATUS\CONSTANT
    211                 ::= <statuspls> STATUS\CONSTANT

90  212 <stop\stat>     ::= STOP $
    213                 ::= STOP NAME $

91  214 <str\i\decl>    ::= STRING NAME <i\desc2> <integer>
                            <integer> <packingq> <integer>
                            <integer> $ <const\listq>
    215                 ::= STRING NAME AI <integer> <su>
                            <str\cases> $ <const\listq>

92  216 <str\cases>     ::= <plusmin> <integer> <str\cases1>
    217                 ::= <integer> <str\cases1>
    218                 ::= <str\cases1>

93  219 <str\cases1>    ::= R <str\cases2>
    220                 ::= <str\cases2>

94  221 <str\cases2>    ::= RANGE\PRFX <numeric> <integer>
                            <integer> <str\cases3>
    222                 ::= <integer> <integer> <str\cases3>

95  223 <str\cases3>    ::= DMN <integer> <integer>
    224                 ::= <integer> <integer>

96  225 <su>            ::= S
    226                 ::= U

97  227 <sub\prog>      ::= <subprog\prfx> <body> TERM $

98  228 <subprog\prfx>  ::= CLOSE NAME $ START $
    229                 ::= START PROC NAME <parameters> $
```

```
99  230 <table\prfx>    ::= TABLE <nameq> <table\sizeq>
                            <tssq>

100 231 <table\size>    ::= <vr> <numeric>

101 232 <table\sizeq>   ::= **empty**
    233                 ::= <table\size>

102 234 <test\stat>     ::= TEST $
    235                 ::= TEST <letter> $

103 236 <tssq>          ::= **empty**
    237                 ::= P
    238                 ::= S

104 239 <variable>      ::= ENTRY ( NAME <x> )
    240                 ::= FUNC\MOD <xq> ( <variable> )
    241                 ::= NAME
    242                 ::= NAME <x>
    243                 ::= NAME LPAR\DOLL <f> ... <f>
                            DOLL\RPAR
    244                 ::= NAME LPAR\DOLL RANGE\PRFX <f>
                            DOLL\RPAR
    245                 ::= <letter>

105 246 <vr>            ::= R
    247                 ::= V

106 248 <x>             ::= LPAR\DOLL <fpls> DOLL\RPAR

107 249 <xq>            ::= **empty**
    250                 ::= <x>
```

2. Diagnostics

ERASING RULES	4 7 12 18 92 107 109 115 121 126 148 150 167 174 177 179 182 186 188 191 194 232 236 249
LEFT-RECURSIVE RULES	13 14 33 63 65 67 72 74 76 77 87 90 91 105 112 118 120 125 127 145 147 159 163 166 183 192 211
RIGHT-RECURSIVE RULES	69 84
SELF-EMBEDDING RULES	240

3. Cross References

a. Nonterminals

THE SYMBOL	OCCURS IN LHS OF	AND IN RHS OF
<a\p\l>	4 5	181
<alt\stat>	6	200
<aopl>	7 8	5
<assign\stat>	9	201
<bch>	10 11	37
<body>	12 13 14	13 14 29 141 227
<cht>	15 16	102
<const\list>	17	19 30
<const\listq>	18 19	34 38 49 155 214 215
<constant>	20 21 22	81 85 169 170 171
<controls>	23 24	198
<controls2>	25 26 27 28	23 24
<cstat>	29	36 40 202
<dcl>	30 31	32 33

b. Terminals

THE SYMBOL	OCCURS IN THE RHS OF
$	6 9 34 35 36 37 38 39 40 41 42 43 46 47 48 49 64 96 103 104 141 155 164 176 181 193 196 198 199 212 213 214 215 227 228 229 234 235
(	5 25 42 43 80 94 180 239 240
)	5 25 42 43 80 94 180 239 240
*	142
**	76
+	169 184
,	27 28 87 90 91 105 112 120 125 145 163
-	170 185
...	243
/	143
=	8 9 42 43 123 146 147 168 175 198
==	64
A1	51 97 128 215 ***SPECIAL SYMBOL***
ALL	25
AND	67
ARRAY	34
ASSERTINOUT	35 ***SPECIAL SYMBOL***
ASSERT\ASSUME	24 196 ***SPECIAL SYMBOL***
B	10 99 129
BEGIN	29 46 48

```
BEGIN\                  17

CH                      11 15 130  ***SPECIAL SYMBOL***

CHARACTER\CONSTANT      20

CLOSE                   36 228

DECIMAL\INTEGER         48 113 117 118

DIRECT                  197

DMN                     131 178 223
                                   ***SPECIAL SYMBOL***

DOLL\RPAR               243 244 248

EGJKOQWXYZ              132     ***SPECIAL SYMBOL***

END                     6 17 29 46 48

ENTRY                   239     ***SPECIAL SYMBOL***

F                       100 133

FILE                    37

FILE\STATES             37

FIXED\CONSTANT          152

FLOATING\CONSTANT       153

FOR                     198

FUNC\MOD                240     ***SPECIAL SYMBOL***

GOTO                    96

IF                      199

IFEITH                  6

INOUT                   103 104    ***SPECIAL SYMBOL***

ITEM                    38 39 49 155

JOVIAL                  197

L                       47 134
```

c. Special Symbols

```
AI              : A I
ASSERTINOUT     : ASSERTIN ASSERTOUT
ASSERT\ASSUME   : ASSUME ASSERT
CH              : C H
DMN             : D M N
EGJKOQWXYZ      : E G J K O Q W X Y Z
ENTRY           : ENTRY ENT
FUNC\MOD        : ABS POS NENT NWDSEN BIT BYTE ODD LOC
                  CHAR MANT
INOUT           : INPUT OUTPUT
OPSH            : OPEN SHUT
REL\OP          : EQ GQ GR LQ LS NQ
```

Appendix B
TABLEAUX GLOSSARY

Appendix B

TABLEAUX GLOSSARY

(Asterisks denote optional arguments.)

ALPHA The CAR of this variable is the list of active ALPHA nodes. The CDR is the list of deferred ALPHA nodes. See also: FORMULATYPE(F).

ALGEBRA(C) The algebraic simplifier is applied to the formula at node C, which must be an ancestor of CURRENT. If the result is T, it is appended to the tree below CURRENT (see PROOFGO for details) and closes the open leaves accessible from CURRENT. If the result is NIL, the user is so informed; this means that the node is useless in completing the proof. Otherwise, the user is asked whether the simplified form should be appended to the tree and may respond with N (for No) or Y or O (for Yes or OK).

ANCESTOR(C) Is C an ancestor of CURRENT?

ALIST(C1 C2) Returns the instantiation (if any) that may be used as a second argument to INSTANCE to instantiate the formula at node C1 so that the result contradicts the formula at node C2.

ARITH(F NODES) Apply the Presburger procedure to the assertion that the conjunction of the negations of the formulae at the nodes specified by the list, NODES, (each node of which should be an ancestor of CURRENT) implies F. If this assertion is proved, the negation of F is appended to the tree below CURRENT. If the assertion is not proved, a possible counterexample is given. F is a formula, either typed in by the user, or supplied by, e.g., (GETFORMULA c). The first option is useful for deriving intermediate results not appearing in the tableau.

ARITHC(C1 C2 ...) This is equivalent to
(ARITH (GETFORMULA CURRENT) '(C1 C2 ...)).

AUTOMATIC If AUTOMATIC is NIL, then each PROOFGO will perform a single tableaux operation; otherwise, PROOFGO will repeat as long as there is an unused ALPHA, DELTA, or BETA node.

AV() For each unclosed leaf C, type the message (FROM NODE C) and then do a VISIBLE(C).

BETA The CAR of this variable is the list of active BETA nodes. The CDr is the list of deferred BETA nodes. See also: FORMULATYPE(F).

CURRENT This is the node in the tableau that will be used in the next tree augmenting operation. For details, see PROOFGO, ALGEBRA, ARITH, ARITHC, INSTANCE, and INVOKE.

CURRENTTYPE This is the type of the next operation to be performed. It is equal to FORMULATYPE(CURRENT) unless a call has just been made on ALGEBRA, ARITH, ARITHC, INSTANCE, or INVOKE.

DEFER(C) Give node C "deferred" status; automatic PROOFGO will not consider it, and VISIBLE will hide it. If C is CURRENT, then CURRENTTYPE is set to NIL.

DELTA The CAR of this variable is the list of active DELTA nodes. The CDR is the list of deferred DELTA nodes. See also: FORMULATYPE(F).

DEFAULTNODES This integer determines the maximum number of nodes that will be permitted in the tableau begun by a NEWPROOF.

FATHER(C) Returns -1 if C is the ROOT.

FILTERS If K is an element of this list, then SHOW will suppress pairs (K . V) in printing nodes. See SHOW for a list of the possibile settings.

FORMULATYPE(F) The types are ALPHA (OR, IMPLIES, NOT-AND, NOT-NOT), BETA (AND, NOT-IMPLIES, NOT-OR), GAMMA (FORSOME, NOT-FORALL), DELTA (FORALL, NOT-FORSOME), NONLOGICAL (everything else).

GAMMA The CAR of this variable is the list of active GAMMA nodes. The CDR is the list of deferred GAMMA nodes. A GAMMA node is automatically deferred by a PROOFGO from that node. See also: FORMULATYPE(F).

GETFACT(K) Returns the dotted pair (K . V), if any, such that (K . V) is recorded at an ancestor of CURRENT, and no pair (K . V') is recorded at any more recent ancestor of CURRENT.

GETJUSTIFICATION(C) Returns the JUSTIFICATION, if any, of node C. This is equivalent to (CADR (LOCALGETFACT 'JUSTIFICATION) C).

GETFORMULA(C) Returns the formula at node C.

HCSIZE This is the size of the HCONS hash array. A REHASH(N) sets HCSIZE to TABLESIZE(N), a twin prime greater than or equal to N. Note that TABLESIZE(TABLESIZE(N))=TABLESIZE(N).

IDENTITY(C) Invoke the identity substitution rule. C must be either an ancestor or a descendant of CURRENT, and the formula at C must be a negated universally quantified identity. Both EQ and IFF are acceptable identity relations. Refer to the formula at CURRENT as F. If C is a descendant of CURRENT, a NEXTNODE(C) is automatic. If there is exactly one instance of the right- or left-hand side of the identity in F, the only possible substitution is made; the result becomes NEWFORM, and the JUSTIFICATION becomes IDENTITY. The NEWFORM is pretty-printed, and the user may do a PROOFGO to append it below (the new) CURRENT. If more than one substitution is possible, the user is asked to designate a subset of the possibilities and the designated substitutions determine the NEWFORM.

INSTANCE(NODE ALIST) Construct an instance of the formula at NODE, which must be an ancestor of CURRENT. The user specifies quantifiers to be skolemized or instantiated in either of two ways. If ALIST is not NIL, it is assumed to specify the names of the indicial variables to be instantiated and the instantiations desired. If ALIST is NIL, the system asks the user which indicials are to be instantiated and what instantiations are desired. Following instantiation, the resulting

formula is pretty-printed on the user's terminal. At this point NEWFORM is the instantiation, and JUSTIFICATION is (CONS 'INSTANCE ALIST). The user may call PROOFGO to append NEWFORM to the tableau below CURRENT (or the the user can forget the instantiation by issuing a NEXTNODE, calling INSTANCE again, etc.).

INVOKE(LEMMA) The universal closure of LEMMA is instantiated in the second way described under INSTANCE (the null ALIST case). NEWFORM becomes the negation of the instantiation, and JUSTIFICATION becomes (CONS 'LEMMA ALIST). The user proceeds by calling PROOFGO (or discarding the instantiation) as described under INSTANCE.

JUSTIFICATION This is the justification (see SHOW for a list of the possibilities) for the operation to be done on CURRENT.

LASTNODE This is the number of the node most recently added to the tableau; it is set to the literal atom UNDONE if that node has been excised by UNDO.

LEAFP(C) Is C a leaf?

LEFTBROTHER(C) Returns -1 if C is leftmost of the brethren.

LEVEL(C) Returns the depth of C in the tableau.

LMB(C) Returns the leftmost brother of node C.

LOADPROOF(FILE) The argument FILE is not evaluated. The file FILE.PRF must exist on the user's directory. It is assumed that this file was created by a call to SAVEPROOF, and the proof state at the time of the SAVEPROOF is reinstated.

LOCALGETFACT(K C) Returns the dotted pair (K . V), if any, recorded at node C.

MAKETHM(X) The result is the formula X formatted as a lemma and suitable as an argument to INVOKE. MAKETHM is normally called automatically when a proof is completed successfully. It may also be used directly by the user, but it must be noted that the use of unverified lemmas can invalidate an otherwise sound proof.

NEWPROOF(THM NAME*) Initialize a tableau for a proof of THM. If any of the symbols in THM are not known to the system, it will ask that they be declared as TERMs or FORMULAs. If the proof succeeds in closing all branches, then NAME is set to MAKETHM(THM). Several parameters may be changed from their normal values prior to this call: the parameter HCSIZE controls the size of the initial hcons hash array and should be set to 2000 for proving large formulas; the parameter DEFAULTNODES controls the maximum number of nodes that can be generated during the course of a proof and should be set to 300 or 400 for large formulas.

NEXTNODE(C) Consider the node C, which becomes the CURRENT node. CURRENTTYPE is set to FORMULATYPE(C). If CURRENTTYPE is ALPHA, BETA, GAMMA, or DELTA, a PROOFGO() will augment the tableau according to the appropriate tableaux rule. Alternatively, the user may proceed by invoking some other deductive procedure such as ARITH, ALGEBRA, etc.

NONLOGICAL The CAR of this variable is the list of active nonlogical nodes, i.e., nodes containing formulas not of one of the forms (AND ...), (OR ...), (IMPLIES p q), (FORALL x p), (FORSOME x p), the negation of one of these forms, or (NOT (NOT p)).

OFFSPRING(C) Returns the list of nodes whose SOURCE is C.

ONLYLEAVES If this variable is bound and is not NIL, it controls the augmentation of the tableau. See PROOFGO for details.

PPRINSKEL(C) A list structure showing the tree structure of the subtableau rooted at C is pretty-printed on the user's terminal.

PF(C) Causes the formula at node C to be pretty-printed on the user's terminal.

PROOFGO(TERM*) Augment the tree according to the CURRENTTYPE. If ONLYLEAVES is bound to a list of nodes, then new nodes are added below leaves in the intersection of REACHABLEUNCLOSEDLEAVES() and ONLYLEAVES; if ONLYLEAVES is NIL or unbound, new nodes are added below all reachable unclosed leaves. If CURRENTTYPE is GAMMA, then GETFORMULA(CURRENT) is instantiated to the user-supplied parameter TERM; otherwise, TERM may be omitted. PROOFGO is automatic after a successful call of ARITH, ARITHC, or ALGEBRA. If CURRENTTYPE is ALPHA, BETA, or DELTA, PROOFGO continues automatically except when the parameter AUTOMATIC is NIL.

REACHABLEUNCLOSEDLEAVES(C*) Returns a list of the unclosed leaves below C, or, if C is omitted, below CURRENT.

REHASH(N) To be called when HCONSARRAY overflows. HCSIZE is set to TABLESIZE(N), a twin prime not less than N. After it returns, the last top-level tableau command should be repeated--it will not have been correctly completed.

RIGHTBROTHER(C) Returns -1 for a rightmost brother; otherwise, the rightmost brother of C.

RIGHTSON(C) Returns -1 if C is a leaf; otherwise, the rightmost son of C.

SAVEPROOF(FILE) The argument FILE is not evaluated. The state of the proof is saved on the file FILE.PRF. It may be reinstated by calling LOADPROOF. Note that printing parameters, such as FILTERS, are not saved.

SHOW(C NUM*) The subtableau rooted at node C is output. If NUM is T, then only node numbers are typed. Otherwise, the formula at each node is output along with the <Key:Value> pairs stored at the node for each KEY not an element of the controlling parameter FILTERS. The default value of NIL suppresses typing of the keys INSTANCE, CLOSED, JUSTIFICATION, SOURCE, and OFFSPRING. The meanings of these keys are as follows:

INSTANCE: the term that was substituted for the indicial variable in the formula at the SOURCE to get this formula.

CLOSED: the number of a node which is an ancestor of this node and contains a formula contradicting this formula.

JUSTIFICATION: the name of the deductive mechanism and node numbers of any hypotheses used in obtaining this formula from its SOURCE, e.g., ARITH, ALGEBRA, INSTANCE, IDENTITY, LEMMA, or TABLEAUX.

SOURCE: the node from which this node was derived.

OFFSPRING: the numbers of any nodes that have been derived from this node.

SONS(C) Returns the lists of sons of node C.

SOURCE(C) Returns the node from which node C was derived.

UNCLOSEDLEAVES This is a list of the leaves of open branches of the tableau. Each time new nodes are added, it is modified appropriately. See also: WHAT().

UNDEFER(TYPES) Undefer all nodes in TYPES, a subset of (ALPHA BETA GAMMA DELTA NONLOGICAL). If TYPES is NIL, undefer everything.

UNDO(C) C must be a node (other than the root) that the user wants to excise from the tree. To do this soundly, all descendants of FATHER(C) must be undone as well. The user is given this list and must confirm that these nodes are all to be deleted. If this is confirmed, the nodes are deleted and all nodes from which they were derived are restored to their type lists (i.e., ALPHA, BETA, etc.).

VISIBLE(C DF* NUM*) Print nodes between C and ROOT, starting with C. Normally, only visible (i.e. useful and undeferred) nodes are printed. Setting DF to T causes deferred nodes to be printed, too. Setting NUM to T causes only node numbers to be printed. If the CURRENT node is printed, it is marked with the symbol "@".

WHAT() Type a status report giving CURRENT, LASTNODE, CURRENTTYPE, ALPHA, BETA, GAMMA, DELTA, NONLOGICAL, and UNCLOSEDLEAVES. Any of ALPHA, BETA, GAMMA, DELTA, or NONLOGICAL that are empty are omitted. If nonempty, they are typed in the format ((N1 N2 ...) D1 D2 ...) where the Ni are active nodes of that type and the Di are deferred nodes. The output format of UNCLOSEDLEAVES is ((R1 R2 ...) U1 U2 ...) where all the Ri and Ui are unclosed leaves but only the Ri are reachable from CURRENT.

WFF(F) Is the formula F syntactically well-formed?

Appendix C

MULTICS Verification Run

Appendix C

MULTICS Verification Run

1. The Search Program

```
START $
    ITEM VV F $
    ITEM NN I 10 U $
    ITEM INDEX I 10 U 0 ... 1000 P 0 $
    ITEM LOCN I 10 U 0 ... 1000 $
    ITEM FLG B P 0 $
    ARRAY AA 1000 F $
    ASSERTIN T $
    ASSERTOUT (FLG EQ 0)
          AND (INDEX EQ NN+1)
          AND NOT(FORSOME(J, 0 LQ J AND J LQ NN
                                  AND AA($J$) EQ VV))
           OR (FLG EQ 1)
          AND (LOCN GQ 0)
          AND (LOCN LQ NN)
          AND AA($LOCN$) EQ VV $
L1. ASSERT FLG EQ 0
               AND
               (INDEX GQ 0) AND (INDEX LQ NN+1) AND
               FORALL(J, NOT( 0 LQ J AND J LS INDEX
                                AND AA($J$) EQ VV)) $
    IF INDEX EQ NN+1 $   GOTO L3 $
    IF AA($INDEX$) EQ VV $
            BEGIN LOCN=INDEX $ FLG=1 $ GOTO L2 $ END
    INDEX = INDEX + 1 $
    GOTO L1 $
L2. ASSERT (FLG EQ 1) AND (INDEX EQ LOCN)
            AND (0 LQ LOCN) AND (LOCN LQ NN)
            AND AA($LOCN$) EQ VV $
    GOTO OUT $
L3. ASSERT FLG EQ 0 AND
           (INDEX EQ NN+1) AND
            NOT(FORSOME(J, 0 LQ J AND J LQ NN AND
                           AA($J$) EQ VV)) $
OUT. ASSERT (FLG EQ 0)
        AND (INDEX EQ NN+1)
        AND NOT(FORSOME(J, 0 LQ J AND J LQ NN
                                 AND AA($J$) EQ VV))
         OR (FLG EQ 1) AND (0 LQ LOCN) AND (LOCN LQ NN)
                       AND AA($LOCN$) EQ VV $
TERM $
```

2. The Abstract Form of the Program

```
(MAINPROGRAM NIL
          (ITEM VV (F NIL) NIL NIL)
          (ITEM NN (I 10. U NIL NIL NIL) NIL NIL)
          (ITEM INDEX (I 10. U NIL NIL (0. 1000.))
                0. NIL)
          (ITEM LOCN (I 10. U NIL NIL (0. 1000.)) NIL
                NIL)
          (ITEM FLG (B) 0. NIL)
          (ARRAY AA (1000.) (F NIL) NIL)
          (ASSERTIN T)
          (ASSERTOUT
            (OR (AND (AND (EQ FLG 0.)
                          (EQ INDEX (PLUS NN 1.)))
                          (NOT (FORSOME J
                                  (AND (AND (LTQ 0. J)
                                            (LTQ J NN))
                                       (EQ (AA (J))
                                           VV)))))
                          (AND (AND (AND (EQ FLG 1.)
                                         (GTQ LOCN 0.))
                                    (LTQ LOCN NN))
                               (EQ (AA (LOCN)) VV))))
          (LABEL L1)
          (ASSERT (AND (AND (AND (EQ FLG 0.)
                                 (GTQ INDEX 0.))
                            (LTQ INDEX (PLUS NN 1.)))
                       (FORALL J
                         (NOT (AND (AND (LTQ 0. J)
                                        (LT J INDEX))
                                   (EQ (AA (J))
                                       VV))))))
          (IF (EQ INDEX (PLUS NN 1.)) (GOTO L3 NIL))
          (IF (EQ (AA (INDEX)) VV)
              (BEGIN (:= LOCN INDEX)
                     (:= FLG 1.)
                     (GOTO L2 NIL)))
          (:= INDEX (PLUS INDEX 1.))
          (GOTO L1 NIL)
          (LABEL L2)
          (ASSERT (AND (AND (AND (AND (EQ FLG 1.)
                                       (EQ INDEX LOCN))
                                 (LTQ 0. LOCN))
                            (LTQ LOCN NN))
                       (EQ (AA (LOCN)) VV)))
          (GOTO OUT NIL)
          (LABEL L3)
          (ASSERT (AND (AND (EQ FLG 0.)
                            (EQ INDEX (PLUS NN 1.)))
                       (NOT (FORSOME J
                              (AND (AND (LTQ 0. J)
                                        (LTQ J NN))
```

```
                                        (EQ (AA (J))
                                            VV))))))
            (LABEL OUT)
            (ASSERT
              (OR (AND (AND (EQ FLG 0.)
                            (EQ INDEX (PLUS NN 1.)))
                       (NOT (FORSOME J
                              (AND (AND (LTQ 0. J)
                                        (LTQ J NN))
                                   (EQ (AA (J)) VV)))))
                       (AND (AND (AND (EQ FLG 1.)
                                      (LTQ 0. LOCN))
                                 (LTQ LOCN NN))
                            (EQ (AA (LOCN)) VV))))))
```

3. The Verification Conditions

* A1 is

```
(IMPLIES (AND T
               (REAL VV)
               (AND (INT NN) (GTQ NN 0.))
               (AND (INT INDEX) (GTQ INDEX 0.)
                    (EQ INDEX 0.))
               (AND (INT LOCN) (GTQ LOCN 0.))
               (AND (BOOLEAN FLG) (EQ FLG 0.))
               (AND (ARRAY AA)
                    (EQ (ARRAYTYPE AA) REAL)
                    (EQ (DIMENSION AA) 1.)
                    (EQ (UPPERBOUND 1. AA) 1000.)))
          (AND (AND (AND (EQ FLG 0.) (GTQ INDEX 0.))
                    (LTQ INDEX (PLUS NN 1.)))
               (FORALL J
                  (NOT (AND (AND (LTQ 0. J)
                                 (LT J INDEX))
                            (EQ (AA J) VV)))))))
```

* A2 is

```
(IMPLIES
 (AND (AND (AND (EQ FLG 0.) (GTQ INDEX 0.))
           (LTQ INDEX (PLUS NN 1.)))
      (FORALL J
              (NOT (AND (AND (LTQ 0. J) (LT J INDEX))
                        (EQ (AA J) VV)))))
 (AND
  (IMPLIES (EQ INDEX (PLUS NN 1.))
           (AND (AND (EQ FLG 0.) (EQ INDEX (PLUS NN 1.)))
                (NOT (FORSOME J
                              (AND (AND (LTQ 0. J)
                                        (LTQ J NN))
```

```
                                        (EQ (AA J) VV))))))
    (IMPLIES (NOT (EQ INDEX (PLUS NN 1.)))
             (AND (IMPLIES (EQ (AA INDEX) VV)
                           (AND (AND (AND (AND (EQ 1. 1.)
                                               (EQ INDEX
                                                   INDEX))
                                          (LTQ 0. INDEX))
                                     (LTQ INDEX NN))
                                (EQ (AA INDEX) VV)))
                  (IMPLIES (NOT (EQ (AA INDEX) VV))
                           (AND (AND (AND (EQ FLG 0.)
                                          (GTQ (PLUS INDEX
                                                     1.)
                                               0.))
                                     (LTQ (PLUS INDEX 1.)
                                          (PLUS NN 1.)))
                                (FORALL J
                                  (NOT (AND (AND (LTQ 0. J)
                                                 (LT
                                                   J
                                                   (PLUS
                                                     INDEX
                                                     1.)))
                                            (EQ
                                              (AA J)
                                              VV)))))))))
```

* A3 is

```
  (IMPLIES (AND (AND (AND (AND (EQ FLG 1.)
                               (EQ INDEX LOCN))
                          (LTQ 0. LOCN))
                     (LTQ LOCN NN))
                (EQ (AA LOCN) VV))
           (OR (AND (AND (EQ FLG 0.)
                         (EQ INDEX (PLUS NN 1.)))
                    (NOT (FORSOME J
                                  (AND (AND (LTQ 0. J)
                                            (LTQ J NN))
                                       (EQ (AA J)
                                           VV)))))
               (AND (AND (AND (EQ FLG 1.)
                              (LTQ 0. NN))
                         (LTQ LOCN NN))
                    (EQ (AA LOCN) VV)))))
```

```
* A4 is

  (IMPLIES (AND (AND (EQ FLG 0.)
                     (EQ INDEX (PLUS NN 1.)))
                (NOT (FORSOME J
                       (AND (AND (LTQ 0. J) (LTQ J NN))
                            (EQ (AA J) VV)))))
           (OR (AND (AND (EQ FLG 0.)
                         (EQ INDEX (PLUS NN 1.)))
                    (NOT (FORSOME J
                           (AND (AND (LTQ 0. J)
                                     (LTQ J NN))
                                (EQ (AA J) VV)))))
               (AND (AND (AND (EQ FLG 1.)
                              (LTQ 0. LOCN))
                         (LTQ LOCN NN))
                    (EQ (AA LOCN) VV)))))


* A5 is

  (IMPLIES (OR (AND (AND (EQ FLG 0.)
                         (EQ INDEX (PLUS NN 1.)))
                    (NOT (FORSOME J
                           (AND (AND (LTQ 0. J)
                                     (LTQ J NN))
                                (EQ (AA J) VV)))))
               (AND (AND (AND (EQ FLG 1.)
                              (LTQ 0. LOCN))
                         (LTQ LOCN NN))
                    (EQ (AA LOCN) VV)))
           (OR (AND (AND (EQ FLG 0.)
                         (EQ INDEX (PLUS NN 1.)))
                    (NOT (FORSOME J
                           (AND (AND (LTQ 0. J)
                                     (LTQ J NN))
                                (EQ (AA J) VV)))))
               (AND (AND (AND (EQ FLG 1.)
                              (GTQ LOCN 0.))
                         (LTQ LOCN NN))
                    (EQ (AA LOCN) VV)))))
```

4. Proving the Verification Conditions

The proof is begun by setting up a tableau to prove a1, using the command

```
(newproof a1)
```

The system next requests the declaration, as term or formula, of the symbols appearing in a1. In this example, the user must declare "t", "real", "int", "boolean", "array", "arraytype", and "and" as formulae and "vv", "nn", "index", "locn", "flg", "aa", "dimension", "upperbound", and "j" as terms. When declaration of symbols is complete, the system types

```
tableau setup completed
```

and the user may proceed with the proof. For a1, a sufficient set of tableaux commands to complete the proof is

```
(proofgo)
(nextnode 30)
(arithc 16 18 21 23 28 29)
(nextnode 32)
(arithc 21 23)
```

Of course, to construct these commands, a user must issue various commands to inspect the state of the partial tableau and decide which facility to use at each step. However, the point of the present chapter is just to demonstrate that a verification can be run at RADC-MULTICS; a tutorial on the use of tableaux in practice is given in Appendix B.

The proof of a2 is the most complex of the five proofs; it is achieved by the sequence of commands

```
(defer 1)
(algebra 1)
(nextnode 3)
(proofgo)
(nextnode 12)
(proofgo)
(nextnode 14)
(arithc 7)
(nextnode 15)
(arithc 13)
(nextnode 10)
(proofgo)
(nextnode 26)
(arithc 5 6 7 11 25)
(nextnode 27)
(proofgo)
(nextnode 30)
```

```
(arithc 7)
(nextnode 32)
(arithc 5 6 11)
(nextnode 33)
(arithc 5 6 11)
(nextnode 37)
(instance 4 nil)
(j:2)
(proofgo)
(nextnode 41)
(proofgo)
(nextnode 42)
(arithc 37)
(nextnode 44)
(arithc 35)
(nextnode 20)
(instance 4 nil)
(j:1)
(proofgo)
(nextnode 47)
(proofgo)
(nextnode 48)
(arithc 20)
(nextnode 49)
(arithc 5 6 13 18 19)
(nextnode 50)
(arithc 18)
(nextnode 43)
(arithc 5 6 11 28 35 36 37)))
```

Continuing with the proof of the verification conditions, we see that a3 and a4 require essentially no user interaction and are proved by the commands

```
(newproof a3)
(proofgo)
(newproof a4)
(proofgo)
```

Finally, the proof of a5 is quite simple. It is completed by

```
(newproof a5)
(proofgo)
(nextnode 11)
(arith '(not (ltq 0 locn)) '(11))
```

REFERENCES

1. Bledsoe, W.W., "The SUP-INF Method in Presburger Arithmetic," Automatic Theorem Proving Project Technical Report ATP-18, University of Texas, Austin, Texas (December 1974).

2. Boyer, R.S. and Moore, J S., "A Fast String Searching Algorithm," CACM (to appear).

3. Computer Sciences Corporation, "JOCIT Compiler Users Manual," Revision 3, El Segundo, California (September 1975).

4. De Remer, F.L., "Simple LR(k) Grammars," CACM, Vol. 14, No. 7, pp. 453-460 (July 1971).

5. Earley, J., "An Efficient Context-Free Parsing Algorithm," CACM, Vol. 13, No. 2, pp. 94-102 (February 1970).

6. Elspas, B., "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness," Interim Report, SRI Project 2686, Stanford Research Institute, Menlo Park, California (July 1974).

7. Elspas, B. et al, "A Verification System for JOVIAL/J3 Programs (Rugged Programming Environment - RPE/1)," Final Report, SRI Project 3756, Stanford Research Institute, Menlo Park, California (January 1976). Also available from Rome Air Development Center, Griffiss AFB, Rome, New York 13441 as ISIS Report RADC-TR-76-58, (A024395).

8. Gomory, R.E., "An Algorithm for Integer Solutions to Linear Problems," in R.L. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming, (McGraw-Hill, New York, 1963), pp. 269-302.

9. Guttag, J.V., Horowitz, E., and Musser, D., "Abstract Data Types and Software Validation," CACM (to appear). Also available as USC Information Sciences Institute Research Report ISI/RR-76-48, 4676 Admiralty Way, Marina del Rey, California 90291 (August 1976).

10. Holloway, G. et al, "ECL Programmer's Manual," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts (December 1974).

11. Knuth, D.E., "Sorting and Searching," Vol. 3: The Art of Computer Programming, (Addison-Wesley Publishing Company, Reading, Massachusetts, 1973), pp. 521-524.

12. Moon, D.A., "MACLISP Reference Manual," Revision 0, Project MAC, M.I.T., Cambridge, Massachusetts (April 1974).

13. Robinson, L. and Levitt, K.N., "Proof Techniques for Hierarchically Structured Programs," CACM, Vol. 20, No.4, pp. 271-283 (April 1977).

14. Robinson, L. and Roubine, O., "SPECIAL--A Specification and Assertion Language," Computer Science Laboratory Technical Report CSL-46, Stanford Research Institute, Menlo Park, California (January 1977).

15. Shostak, R.E., "On the SUP-INF Method for Proving Presburger Formulas," JACM (to appear).

16. Smullyan, R.M., First-Order Logic, (Springer-Verlag Publishing Company, New York, 1971).

17. Teitelman, W., "INTERLISP REFERENCE MANUAL," Xerox Palo Alto Research Center, Palo Alto, California (December 1975).