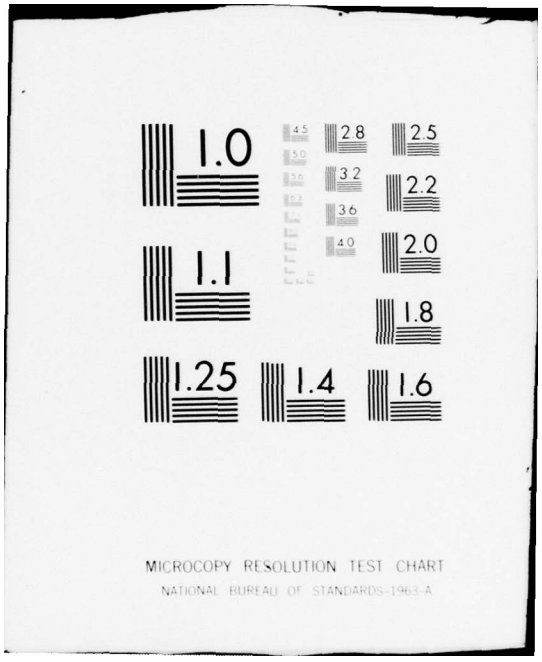
AD-A061 919 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 9/2
THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVIN--ETC(U)
AUG 78 B ELSPAS, R E SHOSTAK F44620-73-C-0068
UNCLASSIFIED AFOSR-TR-78-1491 NL

1 OF 2
AD A061919


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(18) (19) AFOSR-TR-78-1491

THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS.

(9) Final Report
Covering the Period 1 July 1977 through 30 June 1978

(15) SRI Project 2686
Contract F44620-73-C-0068

(11) August 1978

(12) 116 p.

(10) By: Bernard Elspas, Staff Scientist
Robert E. Shostak, Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332

(16) 2304
(11) A2

Attention:

Lt. Col. George W. McKemie, Contract Monitor
Directorate of Mathematical and Information Sciences

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
Cable: SRI INTL MNP
TWX: 910-373-1248

THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS

Final Report
Covering the Period 1 July 1977 through 30 June 1978

SRI Project 2686
Contract F44620-73-C-0068

August 1978

By: Bernard Elspas, Staff Scientist
Robert E. Shostak, Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332

Attention:

Lt. Col. George W. McKemie, Contract Monitor
Directorate of Mathematical and Information Sciences

Approved:

Jack Goldberg, Director
Computer Science Laboratory

David H. Brandin, Executive Director
Computer Science and Technology Division

SRI International

ABSTRACT

This final report describes progress over the period 1 July 1977 through 30 June 1978 on a five-year project aimed at the problem of synthesizing so-called inductive assertions to support the Floyd-Hoare method for proving correctness of computer programs.

During the first few years of the project, the emphasis was on more or less direct approaches toward alleviating this problem. Initially, we concentrated on building and using a mechanical solver for finite difference equations to synthesize inductive assertions. This approach had limited success. Unfortunately, neither this approach nor the allied approaches pursued simultaneously by Katz, Manna, Wegbreit, and German have proved to be adequate in any general sense. Therefore, during the period 1975-77 we explored alternatives that gave promise of at least alleviating the problem or of bypassing it entirely. These alternatives encompassed the transformation of programs into primitive recursive form prior to verification, the method of generator induction for proof of properties of complex data structures, the use of a hierarchical design methodology (HDM) to structure programs so as to minimize the need for loop-cutting assertions, and the methods allied to subgoal induction and computational induction. The general tenor of these alternative schemes is that to facilitate program verification considerable care must be taken in properly structuring both the programs to be proved and their specifications. An ideal situation would be one in which all the specifications are written in a formal language that can be processed by a powerful theorem prover. For the Boyer-Moore theorem prover recursive function theory is such a language.

In the fifth year of our research, reported here, we concentrated on using the Boyer-Moore system to prove several quite different kinds of programs. The first set of programs verified here form a system of LISP functions implementing a verification condition generator (VCG) for a simple block-structured language. The specifications for this VCG are given as standard Hoare axioms and rules. The second set of programs

are algorithms for achieving synchronization among several clocks. This application arose in connection with the design of an operating system for a fault-tolerant avionics computer (SIFT) with hardware and software redundancy features.

A separate problem addressed during the fifth year is the application of directed graph theory to the design of an efficient algorithm for deciding inequalities for sets of integer variables. This work represents a further extension of a series of efficient decision algorithms for Presburger arithmetic (under various restrictions). Most of these algorithms have been implemented (in LISP) as part of experimental program verifiers built at SRI during the past few years.

PROJECT PERSONNEL

The following people constituted the project team for this effort at various times during the period 1 July 1977 through 30 June 1978:

Robert S. Boyer

Bernard Elspas

Milton W. Green

Robert E. Shostak

Drs. Elspas and Shostak served as co-principal investigators.

The authors wish to acknowledge their considerable debt to Drs. Robert S. Boyer and J Strother Moore for many valuable discussions, in particular for assistance in using their recursive function theorem prover. We also wish to express our gratitude for helpful discussions on a variety of subjects to the following individuals: Woody Bledsoe, John Guttag, Paul Gloess, Shmuel Katz, Ralph London, and Mark Moriconi.

CONTENTS

APPENDICES

I INTRODUCTION AND SUMMARY

A. Introduction.

This is a final report covering progress during the last year of a five-year research effort on problems entailed in making the Floyd-Hoare approach to program verification [1,2]* a practical technique. An extensive summary of the work of the preceding four years appears in our last interim report [3].

The Floyd-Hoare technique is a method for applying mathematical-logical tools to proving "correctness" of computer programs. It is now among the most promising approaches to the achievement of reliable computer programs--currently a source of major concern to the Air Force. The Floyd-Hoare technique has already had considerable impact on the fields of language design [4,5], formal specification and design of software [6]. Various aspects of the method are currently being employed for the design and verification of specific properties of developmental software systems [7,8], especially such properties as security and fault tolerance, which are of critical importance in Air Force systems. A number of developmental program verifiers (at various stages of development) are in experimental use at such laboratory environments as USC-ISI, Stanford University, University of Texas, and System Development Corporation, as well as at SRI International. Closely allied prototype systems using symbolic execution also exist at IBM Watson Research Center [9,10] and General Research Corporation [11]. One of SRI's program verifiers (supported by RADC) is scheduled for completion in November 1980 as part of an integrated environment for the design, development, debugging, and verification of JOVIAL J73/I programs.

However, the method of proof of correctness has not yet come into widespread use as an everyday technique for attaining confidence in the correctness of software products. This is attributable in part to the usual time lapse between a laboratory demonstration of feasibility and practical use. For formal program verification the transition has been

*References are listed following Section IV.

hampered by several factors which are perhaps unique to this discipline. First, the Floyd-Hoare method (indeed, any formal method for proving correctness) demands unusual mathematical rigor, considerable skill in formal logic, and sophistication beyond the capabilities of most research programmers (let alone production programmers!). Second, the necessity for inventing inductive assertions has been found to be a serious stumbling block to the practical application of the Floyd-Hoare method, even when machine aids are provided. It has generally been recognized that there is an education gap in the training of programmers which must be overcome if such formal devices are to be brought into the practical arena. Fortunately, most university Computer Science departments are now making loop (inductive) assertions an integral part of the teaching of iterative and recursive programming.

The intrinsic difficulty of writing adequate inductive assertions was recognized as long ago as 1969 [12], and it was, of course, the initial motivation for this project. Despite our best efforts, and those of a number of other researchers [13,14,15] it cannot be said that this problem has been "solved" in general. The results of our work, and of the others just cited has made it possible to mechanize the invention of inductive assertions in special cases and for special domains. Thus, the difference equation technique, especially when combined with the "outside-in" heuristics of Katz-Manna [13], usually provides adequate inductive assertions for integer (and real number) programs. A great deal of insight has been gained from the alternative views of the problem posed by the subgoal induction method of Morris and Wegbreit [16], and by the analysis of loop schemes due to Basu and Misra [15]. In particular, many have noted that the invention of inductive assertions is analogous to the "generalization step" often required in carrying out inductive proofs. Our work on primitive recursive transformation [17] carried this notion one step further. There also emerged from all of this work the underlying conclusion that an important part of the problem of loop assertions really lies in the basic difficulty of adequately specifying software. For this reason we spent part of our effort in exploring the capabilities of two

specification methodologies--the method of algebraic specifications [18] and methods of hierarchical specification of abstract modules. The latter is best exemplified by the SRI Hierarchical Development Methodology (HDM) [6]. Both approaches serve to simplify the problems posed by inductive assertions by (1) minimizing the need for loop-cutting assertions, and (2) providing powerful assertion languages (e.g., the SRI-HDM language SPECIAL) in which to write specifications. In Appendix A of our last interim report [3] we applied HDM methodology to proving properties of a real program. Section III of the present report provides a similar example as to how the algebraic specification technique can be employed.

In general we have become impressed with the definitional power provided by writing specifications in the form of recursive function definitions, as required, e.g., in making use of the Boyer-Moore theorem prover. The structure of such definitions enforces on the programmer a strict discipline that automatically results in clean partitions between pieces of code, and the extensive induction capabilities of the Boyer-Moore system provide the generalization "power" that substitutes for the invention of inductive assertions in the strict Floyd-Hoare approach. Consequently, much of our recent work has used the Boyer-Moore system, as exemplified in the body of this report.

B. Relation to Other Computer Science Laboratory Projects

In our Computer Science Laboratory there have been, over the past few years, a number of related projects concerned with program verification, either in the use or development of these techniques. The existence of these related efforts has been of mutual benefit to all of the projects concerned. Thus, the present effort has benefited from the strong motivation provided by the need for enhanced deduction tools by the application-oriented projects. Likewise, our project work during the past year in particular has benefited by the availability of a preliminary version of the Boyer-Moore theorem prover for recursive functions. Conversely, some aspects of the efficient decision

procedures for Presburger arithmetic developed largely on this project have been used both in our work for Rome Air Development Center (on verifiers for several versions of the JOVIAL language) and also in a new version of the Boyer-Moore system. Still other application-oriented projects have needed (or will shortly require) sophisticated deduction tools for the verification of security and fault-tolerance properties of system software. We list some of these related efforts below.

Our work for Rome Air Development Center has been in progress almost continuously since 1975. Under contracts F30602-75-C-0042 and F30602-76-C-0204 ("Rugged Programming Environment", Phases RPE/1 and RPE/2) we developed early versions of program verifiers for (a subset of) JOVIAL/J3 [19] and for the JOCIT version of JOVIAL [20]. Our current contract with RADC (F30602-78-C-0031) calls for the development of a programming environment for JOVIAL-J73/I in which an Air Force programmer can design, implement, debug, and prove correctness for programs in this language. This contract runs until October 1980. In all three of these efforts we have made extensive use of deduction tools developed initially under the present AFOSR contract.

Mutually beneficial relationships have arisen also with several other government-supported projects in this laboratory. Among these are:

* "Equivalence-Preserving Transformations Between Programs," Principal Investigator: Robert S. Boyer; supported under ONR Contract N00014-75-C-0816.
* "Theorem Prover for Recursive Functions," Principal Investigator: Robert S. Boyer; supported under NSF Grant DCR72-03737A01.
* "Mechanizing the Mathematics of Computer Program Analysis," Principal Investigators: Robert S. Boyer and J Strother Moore; supported under NSF Grant MCS76-81425.
* "Methodology for Hierarchical Design, Development, and Verification of Computer Programs," Principal Investigator: Lawrence Robinson; supported under NSF Grant DCR74-18661.
* "Development and Evaluation of a Software Implemented Fault Tolerant (SIFT) Computer," Project Leader: J. Goldberg; supported under contracts from NASA-Langley.

* "Development of Software Fault Tolerance Techniques," Project Leader: P. Michael Melliar-Smith; supported under contract from NASA-Langley.

C. Report Overview

Section II describes research by R. Shostak on the use of cycle graphs for decision procedures with respect to linear inequalities over the integers. This work has provided an algorithm which is more efficient than several variants of the Presburger algorithm described in our earlier reports [21,3].

In Section III we provide in some detail an unconventional example of program verification dealing with the consistency proofs for an implementation (in pure LISP) of a verification condition generator (VCG) for a simple block-structured programming language. This exercise is unusual in two respects--first, it combines hand proofs with mechanical proofs (the latter executed on the Boyer-Moore theorem prover for recursive functions), second, it is only the second example of a VCG correctness proof of which we are aware (see [22]). The proof is structured into two overall parts. The first portion shows (by manual proofs) that a set of algebraic specifications for the VCG is consistent with standard Hoare axioms for the target language. "Consistency" is to be interpreted here in the (one way) sense that the Hoare axioms are satisfied by any function that satisfies the algebraic specifications, i.e., that the Hoare axioms are derivable from the specifications. It is known that the converse is not true, since a Hoare axiomatization does not determine a unique function for generating verification conditions (nor even unique verification conditions). These algebraic specifications are given in the style of Guttag [18]. The second (mechanical) portion of the proof was carried out on the Boyer-Moore system to show that the implementation satisfies the constraints of the algebraic specifications. We believe that this work represents a first step toward countering an objection frequently voiced about work in program proving: that the researchers in this field do not prove correctness for their own software.

Section IV of the report is concerned with the application of inductive methods to set properties. The section focuses on a detailed proof of one aspect of a clock synchronization algortithm recently devised by Pease, Shostak, and Lamport. This proof was likewise carried out on the Boyer-Moore theorem prover. The motivation for this algorithm arose in connection with the SIFT project under way in our Laboratory under NASA-Langley sponsorship. A paper describing the algorithm is given in Appendix B.

Three other appendices contain subsidiary material as follows:

Appendix A contains traces of the operation of the recursive function prover, and the formal definitions provided to that prover in connection with the work reported in Section III.

Appendix C contains details of the machine proofs of the clock synchronization algorithm described in Section IV.

Appendix D summarizes the other activities (papers published and conferences attended) that were carried out under this project during the final year.

II DECIDING LINEAR INEQUALITIES BY COMPUTING LOOP RESIDUES

A. Introduction

Procedures for deciding whether a given set of linear inequalities has solutions often play an important role in deductive systems for program verification [19,23,28,30,32,33, 40,45]. Array bounds checks and tests on index variables are but two of the many common programming constructs that give rise to formulas involving inequalities. A number of approaches have been used to decide the feasibility of sets of inequalities, ranging from goal-driven rewriting mechanisms [45,26,27,42] to the powerful simplex techniques [29,31,35] of linear programming. The simpler methods are well suited to the small, trivial problems that most often arise, but are insufficiently general. Simplex-based techniques, on the other hand, are general and fast for medium to large problems, but they do not take advantage of the trivial structure of the small problems encountered most frequently.

The algorithm presented here retains the generality needed in the exceptional case, without sacrifice of speed and simplicity in the more typical situation. It builds on V. Pratt's observation [38,36] that most of the inequalities that arise from verification conditions are of the form $x \leqslant y + c$, where x and y are variables and c is a constant. Pratt showed that a conjunction of such inequalities can be decided quickly by examining the loops of a graph constructed from the inequalities of the conjunction. We generalize this approach, first to inequalities with no more than two variables and with arbitrary coefficients, and then to arbitrary linear inequalities. Our generalization reduces to Pratt's test for inputs having the simple structure he describes.

The discussion is presented in six sections. Sections B and C are concerned with preliminary definitions and with a statement of the method for inequalities with two variables and arbitrary coefficients. Section D discusses issues of complexity and usefulness for integer problems, and relates the method to Pratt's. Sections E and F deal with the extension of the method to sets having strict inequalities and to sets with arbitrary linear inequalities. The last section presents a proof of the theorem that underlies the method.

B. Definitions

Let S be a set of linear inequalities each of whose members can be written in the form $ax + by \leqslant c$, where x, y are real variables and a, b, c are reals. Without loss of generality, we require that all variables appearing in S other than a special variable v_0, called the *zero variable*, have nonzero coefficients. We also assume that v_0 appears only with coefficient zero.

Construct an undirected multi-graph G from S as follows. Give G a vertex for each variable occurring in S and an edge for each inequality. Let the edge associated with an inequality $ax + by \leqslant c$ connect the vertex for x with the vertex for y. Label each vertex with its associated variable* and each edge with its associated inequality. G is said to be the *graph for S*.

Now let P be a path through G, given by a sequence $v_1, v_2, \ldots, v_{n+1}$ of vertices and a sequence $e_1, e_2, \ldots, e_n$ of edges, $n \geqslant 1$. The *triple sequence* for P is given by:

$$\langle a_1, b_1, c_1\rangle, \langle a_2, b_2, c_2\rangle, \ldots, \langle a_n, b_n, c_n\rangle \quad ,$$

where for each i, $1 \leqslant i \leqslant n$, $a_i v_i + b_i v_{i+1} \leqslant c_i$ is the inequality labeling e_i.** P is *admissible* if, for $1 \leqslant i \leqslant n - 1$, b_i and a_{i+1} have opposite signs; i.e., one is strictly positive and the other is negative.

Intuitively, admissible paths correspond to sequences of inequalities that form transitivity chains. For example, the sequence $x \leqslant y$, $y \leqslant z$, $z \leqslant 3$ gives rise to an admissible path, as does

$$2x \geqslant 3y - 4,\ 2y \geqslant 4 - z,\ -z \geqslant x \quad .$$

Note that the sequence:

$$x \leqslant y,\ y \leqslant z,\ -z \leqslant r$$

---

*In what follows, it is notationally convenient to write v for both the variable v and the vertex associated with that variable.

**In the case where v_i and v_{i+1} happen to be identical (i.e., e_i is a self-loop), an arbitrary choice is made as to the ordering of the first two components of the associated triple.

cannot label an admissible path, since the coefficients of z have the wrong relative signs.

A path is a *loop* if its first and last vertices are identical. A loop is *simple* if its intermediate vertices are distinct.

Note that the reverse of an admissible loop is always admissible, and that the cyclic permutations of a loop P are admissible if and only if a_1 and b_n are of opposite sign, where $\langle a_1, b_1, c_1\rangle \ldots \langle a_n, b_n, c_n\rangle$ is the triple sequence for P. In this case, we say P is *permutable*. Note also that, since v_0 appears in S only with coefficient 0, no admissible loop with initial vertex v_0 is permutable.

Now define, for a given admissible path P, the *residue inequality of P* as the inequality obtained from P by applying transitivity to the inequalities labeling its edges. For example, if the inequalities along P are

$$x \leqslant 2y + 1,\ y \leqslant 2 - 3z,\ -z \leqslant w \quad ,$$

we have:

$$x \leqslant 2y + 1 \leqslant 2(2 - 3z) + 1 \leqslant 2(2 + 3w) + 1 = 6w + 5$$

The residue inequality of P is thus $x - 6w \leqslant 5$.

More formally, define the *residue* r_p of P as the triple $\langle a_P, b_P, c_P\rangle$ given by:

$$\langle a_P, b_P, c_P\rangle = \langle a_1, b_1, c_1\rangle * \langle a_2, b_2, c_2\rangle * \ldots * \langle a_n, b_n, c_n\rangle \quad ,$$

where $\langle a_1, b_1, c_1\rangle \ldots \langle a_n, b_n, c_n\rangle$ is the triple sequence for P and where * is the binary operation on triples defined by:

$$\langle a, b, c\rangle * \langle a', b', c'\rangle = \langle kaa', -kbb', k(ca' - c'b)\rangle$$

$$\text{and } k = \frac{a'}{|a'|} \quad .$$

The *residue inequality* of P is then given by $a_P x + b_P y \leqslant c_P$, where x and y are the first and last vertices, respectively, of P.

It is straightforward to show that * is associative, so that r_P is in fact uniquely defined. The idea that the residue inequality of a path is implied by the inequalities labeling the path is expressed in the following lemma:

Lemma 1. Any point (i.e., assignment of reals to variables) that satisfies the inequalities labeling an admissible path P also satisfies the residue inequality of P.

Pf. Straightforward by induction on the length of P.

C. Procedure for Inequalities with Two Variables

In the case where P is a loop with initial vertex, say, x, Lemma 1 asserts that any point satisfying the inequalities along P must also satisfy $a_P x + b_P x \leqslant c_P$. If it happens that $a_P + b_P = 0$ and $c_P < 0$, the residue inequality of P is false, and we say that P is an *infeasible loop.*

It follows that a set S of inequalities is unsatisfiable if the graph G for S has an infeasible loop. The converse, however, does not hold in general. Figure 1, for example, shows the graph for $S = \{x \leqslant y,\ 2x + y \leqslant 1,\ z \leqslant x,\ w \leqslant z,\ z \leqslant 1 + w,\ z \geqslant \tfrac{1}{2}\}$. Although S is unsatisfiable, the graph has no infeasible loops, simple or otherwise.

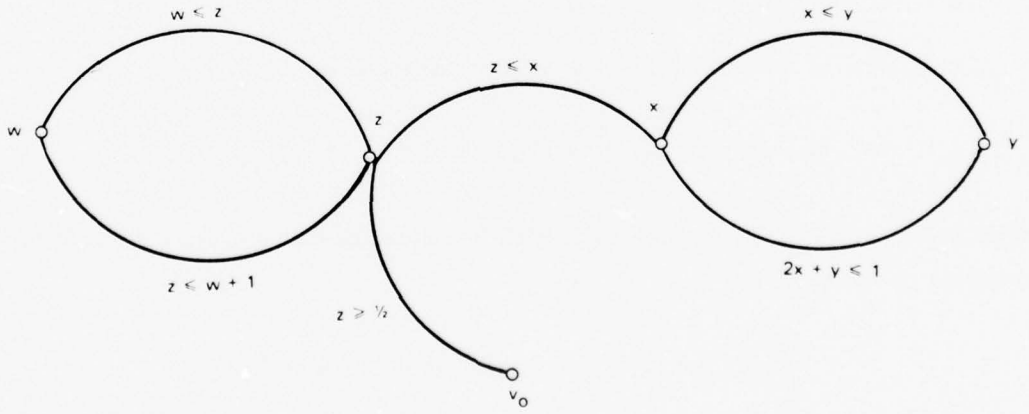


FIGURE 1 GRAPH G FOR $S = \{x \leqslant y,\ 2x + y \leqslant 1,\ z \leqslant x,\ w \leqslant z,\ z \leqslant w + 1,\ z \geqslant \tfrac{1}{2}\}$

The gist of our main theorem is that G can be modified to obtain a graph G' that has an infeasible simple loop if and only if S is unsatisfiable:

Definition: Let G be the graph for S. Obtain a *closure* G' of G by adding, for each simple admissible loop P (modulo permutation and reversal) of G a new edge labelled with the residue inequality of P.

Note that closures are not necessarily unique, since the initial vertex of each permutable loop can be chosen arbitrarily.

Theorem: S is unsatisfiable if and only if G' has an infeasible simple loop.

Figure 2 shows the unique closure of the graph of Figure 1. Note that the only loop of G contributing an edge to G' is the xyx loop. The v_0xzv_0 loop of G' is infeasible (having residue $\langle 0, 0, -1/3 \rangle$); hence the example S, according to the theorem, must be unsatisfiable.

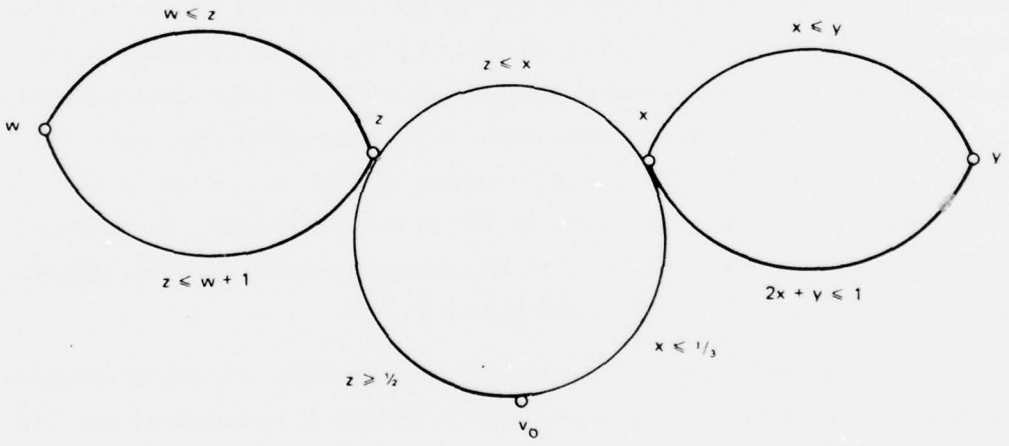


FIGURE 2 CLOSURE OF G

We show later that any cyclic permutation of an infeasible permutable loop is itself infeasible, and that the reverse of an infeasible loop is also infeasible. We thus have the following decision procedure for satisfiability of S:

(1) The simple admissible loops of G are enumerated modulo cyclic permutation and reversal, and their residues are computed. If any loops are found to be infeasible, S is unsatisfiable.

(2) Otherwise, the closure of G is formed by adding a new edge for each residue inequality. The residues of all newly formed simple admissible loops are now computed. If any are found to be infeasible, S is unsatisfiable. Otherwise S has solutions.

Note that this procedure, as stated, does not actually construct a solution if S is feasible. The proof of the main theorem, given in Section G, provides such a construction. Note also that the new admissible loops formed in (2) must have initial vertex v_0.

D. Efficiency and Other Issues

Any implementation of the procedure must, of course, incorporate some means of generating the simple loops of a graph. For this purpose, several algorithms exist (Johnson [34], Read and Tarjan [39], Szwarcfiter and Lauer [43]) that operate in time order $\ell(|V| + |E|)$, and space order $|V| + |E|$, where ℓ is the number of loops generated. These algorithms are easily modified to generate only admissible loops without adversely affecting efficiency. Since each loop has length on the order of $|V|$, these algorithms require little more time than that needed for output. A graph may, of course, have quite a few simple loops -- exponentially many (in $|E|$), in fact, in the worst case. One can show that the procedure we have described, like the simplex method, exhibits exponential worse-case asymptotic behavior. (See also [37,44].)

In practice, however, one does not encounter such behavior. The sets of inequalities that arise from verification conditions usually have the form of transitivity chains. The corresponding graphs are treelike, seldom having more than a few loops. Most of the loops that do occur are 2-loops, which are easily tested at the time the graph is constructed.

V. Pratt [38] has noted that these sets often fall within what he has termed *separation theory*. All the inequalities of such sets are of the form $x \leqslant y + c$. The residue of a loop whose labeling inequalities are of this form is given by one of $\langle 1, -1, m\rangle$, $\langle -1, 1, m\rangle$, where m is the sum of the constants c around the loop. The graph

for a set S in separation theory is thus its own closure, so the main theorem of the last section reduces, in this case, to Pratt's observation that such a set S is infeasible if and only if the sum of the constants around some simple loop is negative. Pratt notes that this condition can be tested in order $(|V| + |E|)^3$ time by taking a max/+ transitive closure of the incidence matrix of the graph. In practice, however, it may be more efficient to generate loops using one of the algorithms mentioned earlier.

Note that a set of inequalities in separation theory with integer constants is integer feasible if and only if it is real feasible. While the main theorem therefore decides integer feasibility in this case, it cannot decide integer feasibility in general. It has been observed [41], however, that the transformations Bledsoe [24,25] describes for reducing formulas in integer arithmetic to sets of inequalities tends to produce sets that are integer feasible if and only if they are real feasible. The main theorem thus provides a useful, but not complete, test for integer feasibility.

E. Strict Inequalities

The procedure is trivially generalized to handle strict inequalities (i.e., inequalities of the form $ax + by < c$). Let an admissible loop be *strict* if one or more of its edges is labeled with a strict inequality. A strict loop P with residue $\langle a_P, b_P, c_P \rangle$ is *infeasible* if $a_P + b_P = 0$ and $c_P \leqslant 0$. If the definition of closure is now modified in such a way that new edges arising from strict loops are labeled with strict inequalities, the main theorem still holds.

F. Extension to Arbitrary Sets of Inequalities

The method can be further generalized to sets of inequalities with arbitrary coefficients and arbitrary numbers of variables.

The basic idea is illustrated by the following example. Consider the set

$$S = \{ x \leqslant y,\ y \leqslant z,\ z \leqslant y - x + 1,\ x \geqslant 2 \} \quad .$$

Note that the inequality $z \leqslant y - x + 1$ has three variables. As shown in Figure 3, we choose two of the three (say z and y) as the endpoints of the edge corresponding to this

inequality in the graph G for S. The term $(-x + 1)$ becomes the "constant" of this inequality. The residue of the only simple loop (y z y) is given by

$$\langle 1, -1, 0\rangle * \langle 1, -1, -x + 1\rangle$$

and is computed "symbolically" to obtain $\langle 1, -1, -x + 1\rangle$. Note that this loop is infeasible unless $-x + 1 \geqslant 0$. If the residue inequality $-x + 1 \geqslant 0$ is now added to the graph, an infeasible simple loop $(v_0 x v_0)$ results, thus making S unsatisfiable.

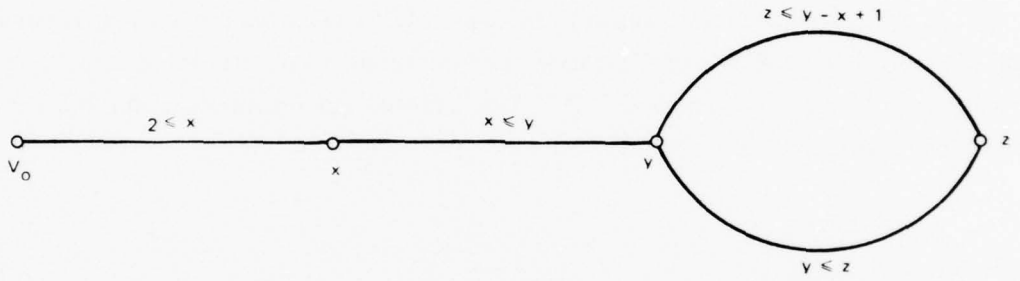


FIGURE 3 GRAPH G FOR $\{x \leqslant y, y \leqslant z, z \leqslant y - x + 1, x \geqslant 2\}$

We now describe the procedure for an arbitrary set S. We assume that the variables of S other than v_0 are ordered in some way. Each variable that is the lowest or second lowest ranked variable in every inequality in which it appears is said to be a *primary variable*. We adopt the convention that the edge corresponding to a given inequality is always attached to the two nodes corresponding to its primary variables. If it has only one primary variable, one end is attached to v_0, and if it has no primary variables, both ends are attached to v_0. The procedure is as follows:

(1) Compute a closure G′ of the graph G for S as usual, evaluating residues "symbolically" as in the example. If G′ has an infeasible loop, terminate returning "unsatisfiable." Otherwise, if all the variables of S are primary, terminate returning "satisfiable."

(2) Otherwise, repeat the procedure using the set of residue inequalities of G′ in place of S.

Note that the procedure must terminate since the number of non-primary variables must decrease each iteration. One can prove as an extension of the main theorem that the general procedure is complete as well as sound.

R. Tarjan* has observed that any set of inequalities can be polynomially transformed to one with no more than three variables per inequality through the addition of new variables. The inequality $w + x + y + z \leqslant 1$, for example, is replaced by $w + x \leqslant v$, $w + x \geqslant v$, $v + y + z \leqslant 1$. For sets with inequalities having no more than three variables, only two iterations of the procedure are ever required. There does not seem to be any fast way to transform a set of inequalities to one having inequalities with no more than two variables.

G. Proof of the Main Theorem

It follows from Lemma 1 and from the definition of closure that a set S of inequalities (each having no more than two variables) is satisfiable if and only if S′ is unsatisfiable, where S′ labels the edges of a closure of the graph for S. If we define a *closed* graph as one that is a closure of itself, the main theorem can thus be restated as follows:

Theorem: If G is a closed graph for S, then S is satisfiable if and only if G has no infeasible simple loop.

The proof of the theorem requires a number of technical lemmas. Proofs are omitted for the more trivial of these.

Notation: Where P and Q are paths, let PQ denote the concatenation of P with Q.

Lemma 2. If P and Q are admissible paths, then PQ is admissible if and only if b_Q and a_P are of opposite sign.

Notation: Let $T = \langle a, b, c \rangle$ be a triple of reals. Then $T^{\sim}$ denotes the triple $\langle b, a, c \rangle$.

Lemma 3. If T_1, T_2 are triples, $T_1 * T_2 = (\tilde{T}_2 * \tilde{T}_1)^{\sim}$.

Corollary 4. If Q is the reverse of an admissible path P, then $r_P = r_Q^{\sim}$.

Corollary 5. The reverse of an infeasible loop is itself infeasible.

*Private Communication.

Lemma 6. Any permutation of an infeasible permutable loop is infeasible.

Pf. Say P is infeasible and P′ is a permutation of P. Then there are paths Q and R such that P = QR and P′ = RQ. Thus,

$$r_P = \langle k a_Q a_R , -k b_Q b_R , k(c_Q a_R - c_R b_Q) \rangle$$

and

$$r_{P'} = \langle k' a_R a_Q , -k' b_R b_Q , k'(c_R a_Q - c_Q b_R) \rangle \quad ,$$

where $k = \frac{a_R}{|a_R|}$ and $k' = \frac{a_Q}{|a_Q|}$. Note that by admissibility of P and P′, both a_R and a_Q are nonzero. By infeasibility of P, $a_R a_Q - b_Q b_R = 0$ and $k(c_Q a_R - c_R b_Q) < 0$.

$$\therefore \; a_R \left(\frac{c_Q b_R b_Q}{a_Q} - c_R b_Q \right) < 0$$

$$\therefore \; a_R b_Q \left(\frac{c_Q b_R}{a_Q} - c_R \right) < 0$$

$$\therefore \; c_R - \frac{c_Q b_Q}{a_Q} < 0 \quad \text{(since } a_R \text{ and } b_Q \text{ have opposite signs)}$$

$$\therefore \; k'(c_R a_Q - c_Q b_R) < 0 \quad .$$

Recalling that $a_R a_Q - b_Q b_R = 0$, we thus have that P′ is infeasible

Q.E.D.

Notation: Where u, v, w are reals, let $u \overset{w}{<} v$ mean that $u < v$ if $w \geqslant 0$ and $u > v$ if $w < 0$.

Definition: Where P is an admissible path, the *discriminant* d_P of P is given by $\frac{c_P}{a_P + b_P}$.

Note that an infeasible loop is one with discriminant $-\infty$.

Lemma 7. If PQ is an admissible loop from v_0 to v_0, then PQ is infeasible iff $d_P \overset{a_Q}{\geq} d_Q$ iff $d_P \overset{a_P}{<} d_Q$.

Notation: In the following, let $\langle a_1, b_1, c_1 \rangle$, $\langle a_2, b_2, c_2 \rangle$, $\langle a_3, b_3, c_3 \rangle$, and $\langle a_P, b_P, c_P \rangle$, respectively, denote the residues of P_1, P_2, P_3, and P.

Lemma 8. If G is closed and has an infeasible loop from v_0 to v_0, G has an infeasible simple loop.

Pf. Let P be a shortest infeasible loop from v_0 to v_0 in G. If P is simple, we are done. Otherwise, since, by admissibility, the intermediate vertices of P are distinct from v_0, P can be expressed $P_1 P_2 P_3$, where P_2 is simple. We claim that P_2 is also infeasible.

Suppose not. Then either $a_2 + b_2 = 0$ and $c_2 \geq 0$, or d_{P_2} is finite. In the former case, a_2 and b_2 have opposite signs. It follows from Lemma 2 that b_1 and a_3 must as well, hence $P_1 P_3$ is admissible. Now since

$$r_{P_1 P_2} = \langle 0, b_1, c_1 \rangle * \langle a_2, b_2, c_2 \rangle = \frac{a_2}{|a_2|} \langle 0, -b_1 b_2, c_1 a_2 - c_2 b_1 \rangle \quad ,$$

we have:

$$d_{P_1 P_2} = \frac{c_1 a_2 - c_2 b_1}{-b_1 b_2} = \frac{c_2}{b_2} - \left(\frac{a_2}{b_2}\right) d_{P_1} = \frac{c_2}{b_2} + d_{P_1} \quad .$$

Since P is infeasible, we have from Lemma 7 that

$$\frac{c_2}{b_2} + d_{P_1} \overset{a_3}{\geq} d_{P_3} \quad .$$

Thus,

$$c_2 + b_2 d_{P_1} \overset{a_3 b_2}{\geq} b_2 d_{P_3}$$

$\therefore c_2 + b_2 d_{P_1} < b_2 d_{P_3}$ (since a_3 and b_2 have opposite signs)

$\therefore b_2 d_{P_1} < b_2 d_{P_3}$ (since $c_2 \geqslant 0$)

$\therefore d_{P_1} \overset{b_2}{<} d_{P_3}$

$\therefore d_{P_1} \overset{a_3}{>} d_{P_3}$ (since b_2 and a_3 are of opposite sign).

But then $P_1 P_3$ is infeasible by Lemma 2, contradicting our assumption that P is the shortest such loop.

Now if d_{P_2} is finite, the closedness of G provides that some vertex x on P_2 must be connected to v_0 via an edge E labeled $ax \leqslant c$, where c/a is the discriminant of some cyclic permutation P_2' (possibly = P_2) of P_2. We now have three cases:

Case I. P_2 is not permutable.

Then $P_2' = P_2$, $a = a_2 + b_2$, $c = c_2$, and by Lemma 2, a_2 and b_2 are of the same sign. Also, a must be of this sign; hence both $P_1 E$ and EP_2 are admissible. An argument similar to the one above gives that one or the other of $P_1 E$, EP_2 must be infeasible, contradicting the shortness of P.

Case II. P_2 is permutable and $P_2' = P_2$.

In this case, we have from Lemma 2 that a_2 and b_2 have opposite signs; hence b_1 and a_3 do as well. An argument similar to that given earlier shows that one of $P_1 P_3$, $P_1 E$, and EP_2 must be infeasible, again contradicting the shortness of P_2.

Case III. P_2 is permutable and $P_2' \neq P_2$.

Let P_4 be the initial subpath of P_2 which terminates at x, and let P_5 be the final subpath of P_2 which originates at x (so that $P_2 = P_4 P_5$). In this case, it can be shown that $P_1 P_3$ is admissible, that one of $P_1 P_4 E$, $EP_5 P_3$ is admissible, and that one of these three paths must be infeasible. The shortness of P is thus once again contradicted.

Q.E.D.

Theorem. Let G be a closed graph for S. Then S is satisfiable if and only if G has no simple infeasible loop.

Pf. It follows from Lemma 1 that, if G has a simple, infeasible loop, S must be unsatisfiable. Conversely, suppose G has no such loop. We will show that S is satisfiable by constructing a solution.

Let $v_1, \ldots, v_r$ be the variables of S other than v_0. We construct a sequence $\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_r$ of reals and a sequence $G_0, G_1, \ldots, G_r$ of graphs inductively as follows:

(1) Let $\hat{v}_0 = 0$ and $G_0 = G$.

(2) Suppose $\hat{v}_i$ and G_i have been determined for $0 \leq i < j \leq r$. Let $\sup_j = \min\{d_P | P \text{ is an admissible path from } v_j \text{ to } v_0 \text{ in } G_{j-1} \text{ and } a_P > 0\}$. $\inf_j = \max\{d_P | P \text{ is an admissible path from } v_0 \text{ to } v_j \text{ in } G_{j-1} \text{ and } b_P < 0\}$. (where it is understood that $\min\emptyset = \infty$ and $\max\emptyset = -\infty$). Then let $\hat{v}_j$ be any value in the interval $[\inf_j, \sup_j]$. (We show momentarily that $\inf_j \leq \sup_j$.) Let G_j be obtained from G_{j-1} by adding two new edges from v_j to v_0, labeled $v_j \leq \hat{v}_j$ and $v_j \geq \hat{v}_j$, respectively.

To ensure that the $\hat{v}_j$'s and G_j's are well defined, we must show that, for $1 \leq j \leq r$, $\inf_j \leq \sup_j$. It will then remain to show that the $\hat{v}_j$'s do indeed give a solution for S.

We need the following claim:

Claim. (i) For $1 \leq j \leq r$, $\inf_j \leq \sup_j$

(ii) For $0 \leq j \leq r$, G_j has no infeasible simple loops.

Pf. By induction on j.

Basis. $j = 0$.

In this case, (i) holds vacuously, and (ii) holds since $G_0 = G$.

Induction Step. $0 < j \leq r$.

For (i), suppose, to the contrary, that $\inf_j > \sup_j$. Then in G_{j-1} admissible paths P_1, P_2 exist from v_0 to v_j and v_j to v_0, respectively, with $b_{P_1} < 0$,

$a_{P_2} > 0$, and $d_{P_1} > d_{P_2}$. By Lemma 2, $P_1 P_2$ is an admissible loop, and by Lemma 7, $P_1 P_2$ is infeasible. By Lemma 8, then, G_{j-1} has a simple infeasible loop, contradicting (ii) of the induction hypothesis.

For (ii), suppose G_j has an infeasible simple loop P. Since G_{j-1} has no such loop, and since the loop formed by the two new edges added to G_{j-1} to obtain G_j is not infeasible, P (or its reverse) must be of the form P'E, where E is one of the two new edges (say the one labeled $v_j \leqslant \hat{v}_j$; the other case is handled similarly), and P' is a path from v_0 to v_j in G_{j-1}. But then, by Lemma 7, $d_{P'} > d_E = \hat{v}_j$, contradicting $\hat{v}_j \geqslant \inf_j \geqslant d_{P'}$. (Note that $b_{P'} < 0$ from the admissibility of P'E.)

Q.E.D.

It now remains to show that the $\hat{v}_j$'s satisfy S. So let $ax + by \leqslant c$ be an inequality of S. We claim that $a\hat{x} + b\hat{y} \leqslant c$. We treat the case in which $a > 0$ and $b < 0$; the other cases are argued similarly. Let E be the edge labeled $ax + by \leqslant c$ in G_r. Then, where E_1 is the edge labeled $\hat{x} \leqslant x$ in G_r, and E_2 is the one labeled $y \leqslant \hat{y}$, $E_1 E E_2$ forms an admissible loop. The residue of this loop is

$$\langle 0, -1, -\hat{x} \rangle * \langle a, b, c \rangle * \langle 1, 0, \hat{y} \rangle = \langle 0, 0, -a\hat{x} - b\hat{y} + c \rangle \quad .$$

Since, by the claim proved above, and by Lemma 8, G_r has no infeasible loops from v_0 to v_0, we have $-a\hat{x} - b\hat{y} + c \geqslant 0$. Thus $a\hat{x} + b\hat{y} \leqslant c$ as required.

Q.E.D.

Acknowledgments

The author gratefully acknowledges the insights provided by R. Tarjan, R. Boyer, J. Moore, and M. W. Green.

III CONSISTENCY PROOFS FOR A SIMPLE VERIFICATION CONDITION GENERATOR

A. Introduction

This section of the report describes an application of both mechanical and human theorem proving to the proof of correctness of a simple verification condition generator (VCG). It will be recalled that the VCG is an important component of most program verifiers. Its purpose is to transform a program module, already annotated with assertions, into a list of theorems from which the control semantics of the program have been eliminated. That is, the input to a VCG is an annotated program module and the output is a list of 'staticized' theorems that must be proved to verify that the program and assertions are consistent. Thus, the VCG must incorporate knowledge about the semantics of the programming language, in particular, its control semantics.

It is, therefore, a matter of considerable importance that these semantics are properly reflected in the VCG. If they are not, the verification conditions (VCs) generated by the VCG may be inappropriate to the program under verification. They may be either too weak or too strong. In the first case the verifier may be able to report an incorrect program as "verified" (unsoundness of the verifier). In the second case the faulty VCs may be impossible to validate even when the program is actually correct (incompleteness). Since theorem provers are (necessarily) incomplete over most domains of reasoning, the latter problem is less serious, but it is still a problem. However, to be able to guarantee that a VCG correctly incorporates the semantics of a programming language, it is necessary (as with any formal proof of consistency) to have a formal description of these semantics. This description will be provided in Subsection C.

In Subsection B we define the concrete and abstract syntax for a simple programming language SL that will serve as the vehicle for this VCG verification.

Subsections D and E contain the proofs of consistency for the VCG. These proofs are carried out in two stages:

* First, in Subsection D, we shall demonstrate by means of hand proofs (i.e., conventional, but quite rigorous mathematical arguments) that the formal semantic definition for our language is satisfied by a set of algebraic specifications (in the style of Guttag) for the verification condition generator.

* Second, we use the Recursive Function Theorem Prover of Boyer and Moore to prove (entirely automatically) that an implementation of VCG satisfies the algebraic axioms. The detailed proof traces produced by this system appear in Appendix A. However, the general discussion of what was proved and how the Prover was set up to handle the proofs is given in Subsection E below.

The first step in the consistency proof entails making a correspondence for each of the statement types of the language between a Hoare axiom for that construct and one or more of the algebraic specifications. We have not seen arguments of this sort carried out elsewhere before, at least not at this level of formalism. The proofs are quite straightforward, but they were not entirely free from surprises. In particular we have realized, as a result of carrying them out, that several quite distinct notions of "assertion" are current in verification methodology, that each has distinct advantages and disadvantages, and that they lead to different sorts of verification conditions. In addition, each type is amenable to seemingly different, but actually equivalent, axiomatizations in terms of Hoare logic. These equivalent axiomatizations for the assertion constructs shed light on the general problem of semantic definition.

The second step--that of mechanically proving consistency between a LISP implementation of the function VCG and its algebraic specifications--may at first glance appear to be trivial, because of the close correspondence between the LISP code of the implementation and the LISP-like language of the specifications. Moreover, unlike some VCGs, this implementation was written in pure applicative LISP. Nevertheless, it turned out to be much harder actually to get the recursive function theorem prover to prove all the required theorems than to carry out the

hand proofs of the first step. In part this was due to technicalities associated with the mechanical proving system, particularly conventions regarding "quoted" atomic names and the need for specifying fixed numbers of arguments for functions.

Some of the proofs were nontrivial because the implementation of VCG to be verified makes use of an intermediate function VCR. This function is like the specified function VCG, except that it operates on a reversed statement list for reasons of efficiency. Thus, VCR is recursive through CDRs (in the normal way), whereas VCG is specified by defining it to recurse by removing the last element of the statement list forming the first argument to VCG. The structure of VCR and its relation to VCG was, of course, reflected exactly in the definitions provided to the Boyer-Moore system. Thus, the machine proof helps to certify that no errors have crept into the implementation as a result of this inversion of lists of statements in the recursive calls. In fact, one such error was detected in the process of verification.

B. Syntax for a Simple Language

Below we give definitions of the syntax and semantics for the simple block-structured language SL which serves as a vehicle for our study of VCG correctness proofs. Two versions of the syntax are given for SL--a set of BNF productions for the concrete source language, and (somewhat less formally) similar definitions of the syntax of abstract forms of the nonterminals of the language. The abstract forms are intended for use as internal representations to be input to a verification condition generator.

1. Concrete Syntax for SL

The modified BNF syntax equations for SL are given with nonterminals in lower case characters, and terminals shown in upper case (or quoted where there is no corresponding upper case character). The

metalinguistic symbol '|' is used to separate alternative righthand sides.

```
stats        ::= stat | stats stat

stat         ::= empty\stat | assert\stat | assume\stat
                 | prove\stat | asst\stat | block\stat
                 | ifelse\stat | while\stat | goto\stat
                 | label\stat | abort\stat

empty\stat   ::= ';' | SKIP ';'

assert\stat  ::= ASSERT pred ';'

assume\stat  ::= ASSUME pred ';'

prove\stat   ::= PROVE pred ';'

asst\stat    ::= var ':=' expr ';'

block\stat   ::= BEGIN stats END

ifelse\stat  ::= IF boolexpr THEN stat ELSE stat ENDIF

while\stat   ::= WHILE '(' ASSERTING pred ')' boolexpr
                 DO stat ENDWHILE

goto\stat    ::= GOTO label '(' ASSERTING pred ')' ';'

label\stat   ::= label ':'

abort\stat   ::= ABORT ';'
```

The nonterminals pred, boolexpr, expr, var, and label are not defined here. Any standard syntax for predicates, Boolean expressions, expressions, (simple) variables, and label names, respectively will serve.

2. The Abstract Syntax

The abstract forms of the above constructs are LISP S-expressions defined as follows:

```
assert\statA  ::=  '(' ASSERT predA ')'
```

```
assume\statA  ::=  '(' ASSUME predA ')'

prove\statA   ::=  '(' PROVE predA ')'

asst\statA    ::=  '(' ':=' varA exprA ')'

block\statA   ::=  BEGIN . statsA

ifelse\statA  ::=  '(' IF boolexprA statA statA ')'

while\statA   ::=  '(' WHILE predA boolexprA statA ')'

goto\statA    ::=  '(' GOTO label predA ')'

label\statA   ::=  '(' LABEL label ')'

empty\statA   ::=  '(' SKIP ')'

abort\statA   ::=  '(' ABORT ')'

statsA        ::=   '(' statA ')' |  statsA @ '(' statA ')'

statA         ::=  assert\statA | assume\statA | prove\statA
                 | asst\statA | block\statA | ifelse\statA
                 | while\statA | goto\statA | label\statA
                 | empty\statA | abort\statA
```

Notes: 1. '.' denotes Lisp cons
2. @ denotes LISP append, as an infix operator.
3. Each nonterminal ending in the letter A denotes the abstract form corresponding to the concrete nonterminal with that A deleted.

Just as with the concrete syntax, we do not specify the abstract syntax for expressions, Boolean expressions, and predicates. What we have in mind are the sorts of S-expression forms currently used in several program verifiers [e.g., (PLUS x (TIMES y z)) for the expression x+y*z, and (AND (EQUAL A B) (LESSP C D)) for the Boolean expression (A=B)&(C<D)]. However, the reader is free to imagine his own expression language. Simple variables and labels are, of course, atomic words. Thus they are not affected by the transition from concrete to abstract syntax.

C. Formal Semantics for the Language SL

In general, three kinds of formal semantics have been used for language specification--axiomatic, operational, and denotational semantics. We shall be concerned here entirely with an axiomatic definition since this type is most directly matched to the issues in question--the relations between preconditions and postconditions across an execution of program segments. Moreover, both of the other kinds of semantic definition entail questions, more intimately concerned with execution models, which are largely irrelevant to the matter of VC generation. In brief, the other modes of semantic definition lie at too detailed a level of abstraction--they say too much--whereas an axiomatic definition tells us exactly what needs to be known about a language in order to generate VCs for it.

The axiomatic method of semantic definition (due to Hoare [2]) requires the provision, for each statement construct of the language, say the statement "stat", either an axiom (usually called a "Hoare axiom") of the form:

P{stat)Q

or an inference rule (a "Hoare rule") of the form:

$$\frac{D1,\ldots,Dn}{P\{stat\}Q}$$

where P and Q are predicate expressions in some base logic, and the Di are either similar predicate expressions, or other (Hoare) formulas of the form p{...}q. The Di are called subsidiary deductions, and they must be validated from axioms of the system, or by application of one or more inference rules to these axioms. The axioms themselves are just that--they are "facts" to be assumed as basic to the inference system. In general, they define the semantics of the primitive statement types of the language, such as assignment, jumps, abort, and also the assertion constructs (assert and assume statements), which are needed in order that Floyd-Hoare verification may be applied. The meaning of a

Hoare formula such as p{stats}q is that if execution of the program segment stats is initiated at a control point where the predicate p (over the program variables) is true, and if execution of stats comes to a proper termination, the predicate q must be satisfied (by the current values of the program variables) at that termination point. The predicates p and q are referred to as precondition and postcondition, respectively. The Hoare inference rules, in general, axiomatize the compound statement types of the language (such as its block, conditional or iterative statements), i.e., those which contain statements as syntactic elements. The verification of such compound statements clearly entails the establishment of subsidiary deductions about the execution of their constituent substatements, hence the need for the subsidiary formulas Di in these inference rules.

In addition to the Hoare inference rules relating to particular language constructs, certain language independent rules are also needed which are basic to the whole formalism. These basic rules are listed below without discussion.

1. The Consequence Rule

$$\frac{P \rightarrow P1, P1\{S\}Q1, Q1 \rightarrow Q}{P\{S\}Q}$$

2. The Conjunction Rule

$$\frac{P\{S\}Q, P\{S\}R}{P\{S\}Q\&R}$$

3. The Disjunction Rule

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

4. The Concatenation Rule

$$\frac{P\{S1\}Q, Q\{S2\}R}{P\{S1;S2\}R}$$

The above basic inference rules may be used, together with the other (language-specific) rules and axioms in carrying out inferences about program execution in the Hoare calculus. Note carefully that all of the rules provide sufficient conditions for such inferences. One must be careful to resist the temptation to apply them in the other direction. The Floyd-Hoare approach, which we follow, proves (partial) correctness by allowing a human prover (possibly aided by a machine) to invent appropriate inductive assertions, Qi, such that all execution paths from input to output are "covered" by proven Hoare formulas, Qi{seg}Qj, where seg is the program segment lying between the assertions Qi and Qj. In view of this relaxation, one can be content also with axiomatizing the constructs of the programming language by Hoare "sufficiency" rules, as we have done. This becomes significant particularly for those constructs that implicitly call for the invention of inductive assertions, e.g., the iterative (while...do) statement.

Thus, our viewpoint regarding proof of partial correctness is that the program to be verified has already been annotated with inductive assertions at all necessary points. These include loop invariants for the while...do statement (already mentioned) and mandatory assertions at every labelled statement. The latter may not always be necessary (since some labels may never be targeted), but the job of the VCG is greatly simplified if we make this assumption, and if we, moreover, assume that a preprocessing of the program has incorporated the target label assertion lexically into each goto statement addressing this label. Thus, we shall write gotos as:

```
GOTO label (ASSERTING pred);
```

rather than simply as:

```
GOTO label;
```

As we shall see, this convention permits us to express the Hoare rule for GOTOs as:

$$\frac{P \rightarrow pred}{P\{\text{GOTO label (ASSERTING pred)}\}Q,}$$

or, still more simply as an axiom:

GOTO Axiom: pred{GOTO label (ASSERTING pred)}Q.

The more usual alternative is to posit the axiom, P{GOTO label}false. Then, since false->Q is tautologous, one has (by the Consequence Rule)

P{GOTO label}pred

regardless of the predicate pred! We find the first approach more intuitive and satisfying.

Similarly, the Hoare rule for WHILE statements is usually given as:

$$\frac{I\&B\{S\}I}{I\{\text{WHILE B DO S ENDWHILE}\}I\&\sim B}$$

This latter entails the invention (by the human or machine verifier) of the auxiliary assertion I, which does not occur in the WHILE statement itself. It seems more natural to incorporate the inductive assertion I directly in the iteration statement, in the form:

WHILE (ASSERTING I) B DO S ENDWHILE.

This suggestion has already been made from several quarters (see, e.g., Wegbreit [46]), often with the use of other key words, such as "maintaining", in place of "asserting".

One further notion is worth mentioning before we proceed on to more specific matters. We wish to single out those Hoare axioms of the form:

pre{S}post

where post = Q is a (free) predicate variable, rather than an arbitrary expression, but where pre may still be any predicate expression. Examples of this form are:

Q&A{ASSERT A}Q

Q{SKIP}Q

A->Q{ASSUME A)Q

We shall refer to such special Hoare axioms as being in backwards canonical form. The reason for this name is that such rules correspond directly to definitions of Dijkstra's [47] predicate transformer wlp (the "weakest liberal precondition" operator). More precisely, from the rule A->Q{ASSUME A}Q on can infer that (A->Q)->wlp(ASSUME A, Q). In this simple case it also makes sense to take A->Q as the definition of wlp(ASSUME A, Q), not just as an upper bound for it. Thus,

wlp(ASSUME A, Q) = [A->Q]

may be used as a definition for the semantics of the ASSUME statement in Dijkstra's terms. It is easily verified that all of Dijkstra's 'axioms' for wlp are satisfied here. In other cases the identity between wlp(S,Q) and the precondition of a Hoare axiom in backwards canonical form does not follow; one can only assert the weaker implication, pre->wlp(S,Q).

Observe also that an axiom that is not in this canonical form, e.g., the axiom:

Q{ASSUME A}Q&A

can sometimes be transformed into an equivalent one in canonical form. In the example cited, since Q is a free predicate variable, we may replace it by A->Q', yielding the axiom:

A->Q'{ASSUME A}(A->Q')&A

which is equivalent to A->Q'{ASSUME A}Q'&A since (A->Q')&A = Q'&A is a tautology. This version also implies

(A->Q'){ASSUME A}Q'

by the Consequence Rule (since Q'&A->Q'). But, this axiom differs from the original (canonical) ASSUME axiom only in the name of the free variable Q. Conversely, the canonical form implies the noncanonical version by replacing Q' by Q&A, noting that Q->(A->Q&A), and using the Consequence Rule.

We now present, without further discussion, the Hoare-type axioms and rules defining the semantics of the language SL.

3.1 ASSERT Axiom

A{ASSERT A}A

Alternative equivalent form of Axiom 3.1:

3.1a: $\dfrac{P \rightarrow A \& A \rightarrow Q}{P\{\text{ASSERT } A\}Q}$

3.2 ASSUME Axiom

A->Q{ASSUME A}Q

Alternative equivalent form of Axiom 3.2:

3.2a: Q{ASSUME A}Q&A

Note: Axiom 3.2 can also be used as an axiom for assertions that are subject to run-time checking (i.e., so called "checked assertions"; see e.g., the axiomatization for EUCLID in London, et al., [4]).

3.3 PROVE Axiom

Q&A{PROVE A}Q&A

Alternative equivalent forms of Axiom 3.3:

3.3a: Q&A{PROVE A}Q

3.3b: $\dfrac{Q \rightarrow A}{Q\{\text{PROVE } A\}Q}$

Notes: 1. Axiom 3.3 is given by London, et al. [4] as an axiom for unchecked assertions in EUCLID.

2. The difference between ASSERT and PROVE is seen to be that ASSERT serves as a complete break between VCs, whereas PROVE does not create a new VC, but merely forces verification of its predicate while conjoining this predicate to any other preconditions for use in proving the next ASSERT/PROVE to be

encountered. ASSUME plays a role similar to this last aspect of PROVE, but, of course, does not demand proof for its predicate. Some authors have used ASSERT in the sense which we use PROVE here.

3.4 Assignment AXIOM

Q(e/x){x:=e}Q

Note 1. The notation 'Q(e/x)' stands for the result of replacing each (free) occurrence of x in Q by a (free) instance of the expression e. If this substitution results in the capture of free variables in e by quantifiers occurring in Q, the quantified variables must be systematically renamed (by fresh variables) before carrying out the indicated substitution.

Note 2. This axiom assumes that evaluation of the expression e produces no side effects.

3.5 BEGIN...END Block Rule

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

3.6 Conditional Rule

$$\frac{P\&B\{\text{stat1}\}Q,\ P\&\tilde{}B\{\text{stat2}\}Q}{P\{\text{IF B THEN stat1 ELSE stat2 ENDIF}\}Q}$$

3.7 WHILE Rule

$$\frac{I\&B\{\text{stat}\}I}{I\{\text{WHILE (ASSERTING I) B DO stat ENDWHILE}\}I\&\tilde{}B}$$

Alternative equivalent form for Rule 3.7

3.7a: $$\frac{P\text{->}I,\ I\&B\{\text{stat}\}I,\ I\&\tilde{}B\text{->}Q}{P\{\text{WHILE (ASSERTING I) B DO stat ENDWHILE}\}Q}$$

3.8 GOTO Axiom

A{GOTO Lab (ASSERTING A)}Q

Alternative equivalent form for Axiom 3.8

3.8a: $$\frac{P\text{->}A}{P\{\text{GOTO Lab (ASSERTING A)}\}Q}$$

Note: Our form of GOTO, assumes that the assertion A occurring at the targeted label appears explicitly within the "asserting" clause of the goto statement. This assumption requires that every label in the program be followed by an ASSERT statement, and that (if necessary) a preprocessor place these assertions redundantly within the corresponding goto statements.

3.9 LABEL Axiom

Since the label's role is taken over by the above GOTO convention, labels become no-ops to the verification condition generator. Hence, the semantics of the label\stat (to the VCG) are given by:

Q{LABEL Lab}Q

3.10 Empty Statement Axioms

Q{;}Q
Q{SKIP}Q

3.11 ABORT Axiom

P{ABORT}false

D. Specifications of a VCG for SL--Consistency with the Axioms.

Algebraic Specifications

In this subsection we present, again without much discussion, a set of algebraic specifications for a verification condition generator (VCG) that are rigorously based on the Hoare-type axioms listed above. The specifications we present here are based on a set due to D. Musser of the USC Information Sciences Institute (private communication). We have added the WHILE specification (S7) and the ABORT specification (S11) to Musser's set. In several other cases (notably for the IF statement specification S6), we have also experimented with alternative forms, but we decided to stick with Musser's versions, even though they entail some duplication of VCs.

These specifications form a set of 12 rewrite rules specifying a function VCG (which is supposed to compute a list of verification conditions consistent with the above Hoare rules) whenever VCG is

supplied with two arguments--(1) a list of abstract statement forms representing a segment of SL source code, and (2) an arbitrary postcondition predicate, Post. Thus, VCG has the functionality:

VCG: StatList x Pred -> PredList

where StatList is the set of all legal program segments, Pred is the set of all (abstract form) predicate expressions, and PredList is the set of all finite (length n=1,2,...) lists of predicates from Pred. Post is an arbitrary member of Pred, and StL is an arbitrary member of StatList. StatList includes the empty program, denoted by NIL. We use angle brackets to represent lists, and the infix operator @ to denote "append" on lists.

We specify:

S0: VCG(NIL, Post) = <Post>

S1: VCG(StL; ASSERT A, Post) = VCG(StL, A) @ <A->Post>

S2: VCG(StL; ASSUME A, Post) = VCG(StL, A->Post)

S3: VCG(StL; PROVE A, Post) = VCG(StL, A) @ VCG(StL, A->Post)

S4: VCG(StL; x:=e, Post) = <Post(e/x)>

S5: VCG(StL1; BEGIN StL2 END, Post) = VCG(StL1 @ StL2, Post)

S6: VCG(StL; IF B THEN stat1 ELSE stat2 ENDIF, Post) =

VCG(StL; ASSUME B; stat1)

@ VCG(StL; ASSUME (NOT B); stat2, Post)

S7: VCG(StL; WHILE (ASSERTING I) B DO stat ENDWHILE, Post) =

VCG(StL, I)

@ VCG(ASSUME I; ASSUME B; stat, I)

@ <I&~B->Post>

S8: VCG(StL; GOTO Lab (ASSERTING A), POST) = VCG(StL, A)

S9: VCG(StL ; , Post) = VCG(StL, Post)

S10: VCG(StL; SKIP, Post) = VCG(STL, Post)

S11: VCG(StL; ABORT, Post) = VCG(StL, true)

In the next part of this Subsection we present detailed proofs that the specifications S0-S11 for VCG are consistent with the Hoare axioms and rules appearing in Subsection C.

Manual Consistency Proofs

We show here, by manual proofs, that the algebraic specifications S0-S11 given above for the function VCG are consistent with (i.e., imply the validity of) the Hoare-type axiomatization (also given above) of the simple programming language. Each proof consists of applying a particular VCG specification (say, the one for "stat") to the statement list ASSUME P; stat and an arbitrary postcondition Q. We then expand this application according to the specification, applying various reductions, and we interpret the final result, VCG(ASSUME P; stat, Q) = L, by means of the following:

Correspondence Rule:

If VCG(ASSUME P; stat, Q) = <P1,...,Pn> then the Hoare-type rule:

$$\frac{P1, P2, \ldots, Pn}{P\{stat\}Q}$$

is satisfied.

The Correspondence Rule serves to establish the connection between the Hoare formalism and the VCG function, in that the elements of the list L obtained by expanding the application VCG(ASSUME P; stat, Q) are to be interpreted as sufficient conditions P1, P2,..., Pn to infer that P{stat}Q. In some cases these subsidiary deductions may themselves be relations of the form, VCG(statements, post) = L, and consequently they also need to be interpreted as Hoare statements, pre{statements}post. Usually, however, the Pi will simply be statements in the base logic.

It should be noted that the proofs also make use of the four fundamental rules of the Hoare formalism given earlier. Their use in

proving the language-dependent rules is legitimate since they are basic to the Hoare formalism and are independent of any particular language.

1. ASSUME Statement

VCG(ASSUME P; ASSUME A, Q) = VCG(ASSUME P, A->Q)

= <P -> (A->Q)>

Hence, by the Correspondence Rule,

$$\frac{P->(A->Q)}{P\{\text{ASSUME A}\}Q}$$

Specialization of P to A->Q yields:

A->Q{ASSUME A}Q

2. ASSERT Statement

VCG(ASSUME P; ASSERT A, Q) = VCG(ASSUME P, A) @ <A->Q>

= <P->A> @ <A->Q>

= <P->A, A->Q>

Hence, by the Correspondence Rule,

$$\frac{P->A, A->Q}{P\{\text{ASSERT A}\}Q}$$

Specialization of both P and Q to A yields:

A{ASSERT A}A

3. PROVE Statement

VCG(ASSUME P; PROVE A, Q) = VCG(ASSUME P, A)

@ VCG(ASSUME P, A->Q)

= <P->A> @ <P -> (A->Q)>

= <P->A, P&A->Q>

By the Correspondence Rule we obtain:

$$\frac{P \rightarrow A,\ P\&A \rightarrow Q}{P\{\text{PROVE } A\}Q}$$

Letting P = Q&A, we find:

Q&A{PROVE A}Q

A familiar alternative form (equivalent to the Hoare-type axiom just derived) is obtained by replacing Q by Q&A:

Q&A{PROVE A}Q&A

This alternative form also clearly implies the first form, by the Consequence Rule; hence, the two forms are equivalent.

4. Assignment Axiom

VCG(ASSUME P; x := e, Q) = VCG(ASSUME P, Q(e/x))

= <P -> Q(e/x)>

By the Correspondence Rule,

$$\frac{P \rightarrow Q(e/x)}{P\{x:=e\}Q}$$

Letting P=Q(e/x) yields:

Q(e/x){x:=e}Q

5. Block Rule

VCG(ASSUME P; StL1; BEGIN StL2 END, Q)

= VCG(ASSUME P; StL1; StL2, Q)

from which the Correspondence Rule directly yields:

$$\frac{P\{\text{StL1; StL2}\}Q}{P\{\text{StL1; BEGIN StL2 END}\}Q}$$

6. Conditional Rule

VCG(ASSUME P; IF B THEN S1 ELSE S2 ENDIF, Q)

= VCG(ASSUME P; ASSUME B; S1, Q)

@ VCG(ASSUME P; ASSUME ~B; S2, Q)

from which direct applications of the Correspondence Rule yield:

$$\frac{P\{\text{ASSUME B; S1}\}Q, P\{\text{ASSUME ~B; S2}\}Q}{P\{\text{IF B THEN S1 ELSE S2 ENDIF}\}Q}$$

However, the relation P{ASSUME B; S1}Q is implied by P&B{S1}Q, and similarly P{ASSUME ~B; S2}Q is implied by P&~B{S2}Q. [These implications follow from the Rule of Concatenation and the ASSUME Axiom]. Hence, from the above rule we also obtain:

$$\frac{P\&B\{S1\}Q, P\&\sim B\{S2\}Q}{P\{\text{IF B THEN S1 ELSE S2 ENDIF}\}Q}$$

which is the desired Hoare Rule for conditional statements.

7. WHILE Statement

VCG(ASSUME P; WHILE (ASSERTING I) B DO S ENDWHILE, Q)

= VCG(ASSUME P, I) @ VCG(ASSUME I&B; S, I) @ <I&~B->Q>

= <P->I> @ VCG(ASSUME I&B; S, I) @ (I&~B->Q>

Using the Correspondence Rule we find that:

$$\frac{P\to I, I\&B\{S\}I, I\&\sim B\to Q}{P\{\text{WHILE (ASSERTING I) B DO S ENDWHILE}\}Q}$$

By letting Q be I&~B we obtain:

$$\frac{P\to I, I\&B\{S\}I}{P\{\text{WHILE (ASSERTING I) B DO S ENDWHILE}\}I\&\sim B}$$

Additional simplification is obtained by letting P=I, so that:

$$\frac{I\&B\{S\}I}{I\{\text{WHILE (ASSERTING I) B DO S ENDWHILE}\}I\&\sim B}$$

which is the usual Hoare rule for while...do statements (and also our WHILE Rule 3.7).

8. GOTO Axiom

VCG(ASSUME P; GOTO L (ASSERTING A), Q)

= VCG(ASSUME P, A) = <P->A>

Hence, by the Correspondence Rule:

$$\frac{P \rightarrow A}{P\{\text{GOTO L (ASSERTING A)}\}Q}$$

This can be simplified to the (equivalent) form:

$$A\{\text{GOTO L (ASSERTING A)}\}Q$$

9. No-Op Axiom

If we let SKIP stand for any of the no-op statements of the language,

VCG(ASSUME P; SKIP, Q) = VCG(ASSUME P, Q) = <P->Q>

Hence, by the Correspondence Rule:

$$\frac{P \rightarrow Q}{P\{\text{SKIP}\}Q}$$

The simplest form is obtained by letting P=Q, so that we get the axiom:

$$Q\{\text{SKIP}\}Q$$

10. ABORT Axiom

VCG(ASSUME P; ABORT, Q) = VCG(ASSUME P, true)

= <P->true> = <true>

Hence, by the Correspondence Rule:

$$P\{\text{ABORT}\}Q$$

holds for any predicates P and Q. In particular, we have the standard form of the ABORT axiom:

$$P\{\text{ABORT}\}\text{false}$$

where Q=false, and from which P{ABORT}Q follows from the Consequence Rule, since false->Q.

This concludes the proofs of consistency between the axiomatization and the algebraic specifications for VCG.

E. Verification of the Implementation in Terms of the Specifications

We present here the second portion of the consistency proof for VCG. This portion of the proof verifies that the LISP implementation (or, more precisely, a paraphrase of that implementation in the theorem prover's syntax) is consistent with the algebraic specifications discussed above. The proof is carried out entirely on the Boyer-Moore theorem prover [48] for recursive functions. The theorems to be proved are (paraphrases of) the algebraic specification equations. There is a separate proof for each algebraic specification equation. Once having been initiated (by calling the theorem prover function PROVE on a theorem), the proof of each theorem proceeds automatically. However, there is a substantial amount of information that must be supplied to the theorem prover prior to that point. In this subsection we show and discuss this initial "setup" process. The detailed proof traces of the individual automatic proofs are collected in Appendix A, along with the verified LISP code and the corresponding definitions supplied to the prover.

1. Definitions Supplied to the Theorem Prover

To be able to prove a theorem containing references to function symbols (or predicate symbols) the theorem prover must previously have been supplied with (recursive function) definitions for those symbols. The theorem prover comes already supplied with definitions for most list-processing primitives (such as CONS, CAR, CDR, APPEND, REVERSE, and EQUAL) as well as with the arithmetic primitives (PLUS, TIMES, DIFFERENCE, QUOTIENT, and EXPT). The latter, however, are not needed for the proofs in question.

Some of the definitions we needed to supply are simply (nonrecursive) abbreviations, e.g., it is convenient to use the LISP primitive LIST (which happens not to be incorporated into the prover initially) as LIST(x) <- (CONS x "NIL"). [The reason why LIST was not built in is, apparently, because LISP "LIST" is usually defined for any

number of arguments, and the theorem prover insists on a definite fixed number. Since one argument suffices for our purposes, this is what we have done.] Note also that the LISP atom NIL appears as the string constant "NIL" in the prover's syntax; it is identical to the value returned by the zero-argument function (NIHIL). Many of the definitions for the syntax of the language SL are also nonrecursive. For example, an assignment statement is defined by the predicate ASSTP as defined for the prover by the command:

```
DEFN(ASSTP (S) (AND (EQUAL (CAR S) ":=") (PLISTP (CDR S))))
```

The prover predicate PLISTP is true for proper lists, i.e., those ending in "NIL", and for "NIL" itself. Thus, (ASSTP S) will be true precisely for proper lists whose car is the assignment key word ":=" of SL.

Similar definitions suffice for the other types of legal statements of SL. For example, it is useful to lump together all the no-op statements of SL into a single predicate NULLP defined by:

```
DEFN(NULLP (S) (IF (LISTP S)
                   (OR (EQUAL (CAR S) "SKIP")
                       (EQUAL (CAR S) "LABEL"))
                   (EQUAL S "NIL")))
```

These definitions and the other syntax predicates have been combined into a single predicate LEGALSTATP such that (LEGALSTATP S) is true if and only if S is a legal statement of the language SL. Because the compound statements, e.g., the IF statement, contain statement components, LEGALSTATP is a recursive function. A separate predicate LEGALP is used to define the notion of a legal program (i.e., "NIL" or a list of legal statements). Thus, LEGALP is also recursive and calls LEGALSTATP.

Recursive definitions pose a special problem for the present version of the Boyer-Moore theorem prover in that one must be able to convince the prover that such definitions are well-founded, i.e., that

the recursive function in question is total. The prover knows about such functions as the LISP function COUNT, and is generally able to deduce well-foundedness where the recursive calls take place on CDR'd arguments. When the system is unable to deduce well foundedness it emits a FAILED message after an attempted DEFN definition, but tentatively accepts the definition anyway, with a caveat to the user that the deductions may be unsound as a result. Incidentally, the DEFN mechanism absolutely precludes giving any mutually recursive definitions. Thus, it would have been impossible to define LEGALSTATP in terms of LEGALP, while also defining LEGALP in terms of LEGALSTATP. Since the latter definition is essential, we had to forego defining LEGALSTATP (for BEGIN blocks) by a recursive call to (LEGALP (CDR block)).

The main definitions supplied to the prover concern the VCG implementation itself (the syntax predicates discussed above are rather subsidiary, but they are convenient to use in the VCG functions). The VCG is implemented entirely in pure LISP (no SETQ's or PROGs are used). The top-level function VCS accepts two arguments (the first must be a legal statement list and the second a logical formula) and returns as its value a list of logical formulas purporting to be the verification conditions (VCs) for that statement list and postcondition. VCG is essentially a backward-acting verification condition generator, i.e., it "pushes back" the second argument past the last statement in the statement list to determine the weakest precondition holding at the point just ahead of this last statement. VCG then proceeds by calling itself recursively on the rest of the statement list with this modified postcondition. Since list-processing recursion is more efficiently handled by recursing down a list by CARs and CDRs, the actual implementation defines VCS(L, Q) in terms of a function VCR which acts on the reversed list RL = (REVERSE L). Thus,

```
VCS(L,Q) <- (VCR (REVERSE L) Q)
```

where VCR(RL, Q) is defined by the usual CAR/CDR recursion, with a

separate kind of action depending on the syntactic type of the statement (CAR RL). Thus, for example, if (CAR RL) [remember: this is the last statement in L] is a no-op statement such as "NIL", (SKIP), or (COMMENT text...), then VCR(RL, Q) simply returns (VCR (CDR RL) Q). That is, the no-op statement is ignored by the VCG. The reader is referred to Appendix A for the details of this and other definitions.

It would be best, however, to comment here on the syntactic differences between the actual LISP code implementing the VCG and the corresponding definitions made to the prover. The key words of the language SL, such as IF, BEGIN, :=, and WHILE appear in the implementation as quoted atoms. Their counterparts in the prover syntax, however, are required to be quoted character strings (e.g., "IF", "BEGIN", and so forth). The actual LISP function LIST (of an arbitrary number of arguments) cannot be used in the prover (as already mentioned above), so that it is necessary to expand out, e.g., (LIST a b c) to (CONS a (CONS b (CONS c "NIL"))). Because LIST has been defined for the single-argument case, this can (but need not) be shortened to (CONS a (CONS b (LIST c))).

Occurrences of the prover function PLISTP have been introduced in various places in the prover definitions (where nothing corresponding to this appeared in the LISP code). These introductions proved necessary to allow the prover to attempt reasonable inductive proofs (by inducting on the list structure of those arguments forced to satisfy PLISTP).

The statements (theorems) to be proved by the Boyer-Moore system also require transcription before input to the system. In fact, the syntactic differences are greater for these theorems than they are between LISP code and prover DEFN forms. Consider, for example, the specification for VCS when applied to a (legal) statement list ending in an ASSERT statement. When written in conventional (concrete) form this specification appears as:

VCS(STL; ASSERT A, Q) = VCS(STL, A) @ <A -> Q>

This form is certainly highly readable. We could have insisted (with some loss in readability) on writing this specification in a LISP-like prefix form such as:

```
(EQUAL (VCS (APPEND STL '((ASSERT A))) Q)
       (APPEND (VCS STL A) '((IMPLIES A Q)))
```

where the prefix function symbols EQUAL, APPEND, and IMPLIES replace the respective infix forms =, @, and ->, and the metalanguage angle brackets < >, denoting an explicit list, are replaced by LISP parentheses. However, even this mild paraphrase does some violence to the real issues, for writing '((ASSERT A)) makes A a quoted atom (instead of a free variable denoting a logical formula). Even worse, Q is treated as a quoted atom in one place and as a variable (to be evaluated) in another. In order for the Boyer-Moore system to handle this theorem properly we need to take into account the fact that A and Q are not quoted atoms (indeed, the prover provides for no such data type; it would have to be a string-quoted expression, viz., "A"). A little thought shows that what is needed is:

```
(EQUAL (VCS (APPEND STL
                    (CONS (CONS "ASSERT" (CONS A "NIL")) "NIL"))
            Q)
       (APPEND (VCS STL A)
               (CONS (CONS "IMPLIES" (CONS A (CONS Q "NIL")))
                     "NIL")))
```

This could have been abbreviated somewhat by making use of the single-argument function LIST:

```
(EQUAL (VCS (APPEND STL (LIST (CONS "ASSERT" (LIST A))))
            Q)
```

```
(APPEND (VCS STL Q)
        (LIST (CONS "IMPLIES" (CONS A (LIST Q))))))
```

Either of the two preceding versions could have been supplied to the theorem prover as theorems to be proved, and the proof would have succeeded. In fact, we took the precaution of adding as an additional hypothesis the fact (LEGALP STL) which is implicit in the original formulation. After all, if STL is not a legal list of SL statements, we do not care what VCS would compute. In practice, however, the definitions supplied to the theorem prover must provide a default in the event of illegal inputs. We choose this to be the (string) constant "UNDEFINED", as can be seen from the definition of VCR. In a practical form of the implementation we would probably guard against illegal inputs of this sort by providing syntactic tests with error invocation upon failure. [In fact, exactly this device was used in another version of this VCG. It has been seen (along with other similar tests) to be a useful debugging feature)]. In a finished system, i.e, one comprising a front end parser, such syntactic checks are carried out by the parser, and can therefore be eliminated from the VCG.

As already mentioned, the detailed proof traces resulting from the action of the Boyer-Moore system on the (transcribed) algebraic specifications are shown in Appendix A. The reader is encouraged to examine these traces carefully, noting, in particular, that the fairly voluminous explanatory output shown there is automatically generated by the system to help the reader follow the line of reasoning established by the prover. We have not added any parenthetic remarks or comments between any invocation of PROVE and the final PROVED which terminates a successful call to the prover.

Unfortunately, it has not been possible to get the prover to verify consistency between the implementation and the IF axiom. We believe that this is due to a quirk in the prover rather than to any basic flaw in either the way the theorem has been set up or in the DEFNs provided to the prover. Part of the problem is the propensity of the prover to

exhaustively consider all possible statement types for the statements S1, S2, appearing in an IF statement, "IF B THEN S1 ELSE S2". Since S1 and S2 can themselves be IF statements, a rather sophisticated induction is called for--one which the prover seems to be unable to provide automatically. Another problem noted in attempting this proof is that the prover attempts to induct on the list structure of the Boolean test B of the IF statement--an induction that is doomed to failure. In our opinion the proof should be capable of success without recourse to either sort of induction. We are still working on this problem and hope to overcome it by proving some prior lemmas by means of the prover function PROVE.LEMMA. This function is like PROVE but also stores away the resulting theorem as a rewrite rule or as an induction lemma for future use. In this way the user of the system can exercise some control over the deduction strategy taken by the prover.

F. Some Observations

Admittedly, the language SL for which we have designed and verified a verification condition generator is an extremely simple one. Even though it includes structured conditional and iteration statements, as well as gotos and an abort statement, many aspects of modern high-level languages were not covered, such as name scoping, procedure and function calls (with or without side effects), modules, case statements, jumps out of blocks, exception handling, or parallelism. It remains to be seen whether similar means will suffice to provide correspondingly convincing proofs of correctness for a more realistic VCG. However, the statement types of SL are virtually certain to be part of any reasonable block-structured language. One suspects that proofs of the VCG features for these statements would not be radically altered by virtue of interactions with other constructs (except for side effects). Still, just writing a VCG for one of the more realistic languages is an ambitious undertaking, let alone carrying out a formal proof of its correctness. Nevertheless, we hope that we, or others, will be inspired to undertake such an exercise in the near future.

One case in point is a VCG that was designed and implemented by us for a subset of JOVIAL-J3 (JOCIT version) under contract with Rome Air Development Center [20]. We were reasonably confident at the time of its completion that this implementation was substantially correct for the subset it was supposed to handle. Subsequent analysis by the same techniques used above (hand methods only--no machine proofs) revealed that there were, in fact, several bugs. These were not revealed in the course of routine testing, simply because none of the test programs contained features that would exercise the faulty code. Although the bugs were not serious ones, and were easily repaired, it is still disturbing to us that they could occur. This confirms our feeling that there is no substitute for some level of formality both in the description of program semantics and in the carrying out of correctness arguments for a VCG, if that VCG is to be considered reliable.

Under our current contract with Rome Air Development Center we are building a much more ambitious program verifier, this one for JOVIAL-J73/I. The statement types of SL were chosen partly because they are a core subset of J73/I that would need handling in any case. As our effort progresses with J73/I, we propose to subject the design of its verification condition generator to the scrutiny used above for SL, even though the level of formality may be necessarily somewhat less severe.

IV INDUCTIVE PROOF OF SET PROPERTIES

This section is concerned with the application of inductive techniques to the verification of programs involving set constructs. The usefulness of set-theoretic structures in the specification of algorithms is clear; the power and richness of expression that set-theory provides is affirmed by its standing as the formal basis of almost all of mathematics. Unfortunately, set-theoretic constructs do not directly lend themselves to recursive formulation, and hence, to inductive proof. (Indeed, the principle of induction is not even stated as an axiom of set-theory, though a version of it can be derived.) The essential difficulty lies in the dependence of inductive methods on the existence of a well-founded partial ordering over some aspect of the structure to which they are to be applied. Arbitrary sets, of course, do not impose an order upon their constituent elements; more to the point, the order in which elements are added in the construction of a set is not reflected in the end product.

Nevertheless, it is possible to formulate properties of sets in a recursive fashion, at least in the finite case. The basic idea is to map each finite set to some permutation (represented as a List) of its members. For each set operator or predicate, one finds a corresponding list operator or predicate that homomorphically preserves the value or truth of its correspondent, modulo representation. The list operators and predicates are defined in a recursive manner.

Suppose, for example, one wishes to prove that $A \cup B = B \cup A$ for arbitrary finite sets A and B. We define the recursive function UNION for lists by:

```
UNION(X Y) =
   (IF (NLISTP X)
       Y
       (IF (MEMBER (CAR X) Y)
           (UNION (CDR X) Y)
           (CONS (CAR X)(UNION (CDR X) Y))))
```

where IF is the conventional 3-placed conditional.

Similarly, the predicate SETEQUAL is defined by:

SETEQUAL(X Y) = (AND (SUBSETP X Y)(SUBSETP Y X)),

where SUBSETP(X Y) =

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

The theorem to be proved thus becomes:

```
   (SETEQUAL (UNION A B)(UNION B A))
```

By virtue of its recursive formulation, this last formula is easy to prove by induction. Its validity, moreover, necessarily implies that of the original theorem. To see that this is true, let us suppose the existence of two finite sets X and Y that violate the original theorem, i.e., such that $X \cup Y \neq Y \cup X$. Letting $\hat{X}$ and $\hat{Y}$ denote, respectively, two arbitrary list representations of X and Y, we have, by the homomorphic property of UNION (which we have posited, but not proved) that $\hat{X}$ UNION $\hat{Y}$ must be a list representation of $X \cup Y$, and that $\hat{Y}$ UNION $\hat{X}$ must be a list representation of $Y \cup X$. Then using the homomorphic property of

SETEQUAL, it follows from $X \cup Y \neq Y \cup X$ that (SETEQUAL (UNION $\hat{X}$ $\hat{Y}$)(UNION $\hat{Y}$ $\hat{X}$)) is false, giving a contradiction.

This illustration, is of course, much too simple to give a valid indication of the usefulness of the technique. The example on which we focused our study is far more difficult. It is taken from a rather elaborate algorithm (due to Pease, Shostak, and Lamport) for obtaining synchronization among a group of mutually-suspicious processors, some of which may be faulty, in a distributed computing system. The paper in which the algorithm is described is attached as Appendix B. We will assume for the remainder of this chapter that the reader has at least scanned that material.

Our effort was largely concerned with using the Boyer-Moore theorem-prover to obtain an automatic proof of one aspect of the correctness of the algorithm. While there was not sufficient time in the course of the project to complete the proof, the main lemma required for the verification was successfully demonstrated. A listing of the definitions and the chain of lemmas leading to the main lemma is supplied in Appendix C.

The most difficult aspect of carrying out the example was not the proof itself, but rather the recursive formulation of the algorithm and the statement of its correctness.

The most natural formulation of the algorithm (described on p. B-6 of Appendix B) requires two mutually-recursive functions; one to compute a single-element of an interactive consistency vector, and the other to compute an entire vector. Because the Boyer-Moore system does not allow introduction of mutually-recursive functions, it was necessary to combine the two into a singly-recursive function, IC.VECTOR (defined on p. C-8 of Appendix C .) Note that IC.VECTOR takes 7 arguments: P, PROCS, PROCSCDRS, N, M, SUFFIX, LIARS. The function returns the interactive consistency vector (represented as an ASSOC-list) that processor P would compute given that the subset LIARS

of PROCS consists of faulty-processors. The quantities N and M are as in the statement of the algorithm in the paper. The arguments PROCSCDRS and SUFFIX are artificial, and come into play only on internal recursive calls. In the initial call, PROCSCDRS is bound to PROCS, and SUFFIX to NIL.

The proved property is that the elements of the interactive consistency vector corresponding to non-faulty processors give the private values (as defined by the uninterpreted function PV) of those processors. The crux of this property is the lemma STRONG.TEST.SETS.N, given on p. C-13 of Appendix C. It should be noted that this lemma is the culmination of a long string of lemmas. The proof required about 100 hours of human interaction and about several hours of CPU time.

REFERENCES

1. R. W. Floyd, "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, Vol. 19, J.T.Schwartz (ed.), pp. 19-32, American Mathematics Society, Providence, Rhode Island (1967).

2. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," CACM, Vol. 12, No. 10, pp. 576-583 (October 1969).

3. B. Elspas, "The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness," Interim Report, SRI Project 2686, SRI International, Menlo Park, California (November 1977).

4. R. London, et al., "Proof Rules for the Programming Language EUCLID," Acta Informatica (to appear).

5. R. J. Feiertag, P. M. Melliar-Smith, and J. M. Spitzen, "The Yellow Programming Language--Preliminary Design Phase Report and Language Specification," SRI International, Menlo Park, California (February 1978).

6. L. Robinson, "HDM--Command and Staff Overview," Technical Report CSL-49, SRI International, Menlo Park, California (February 1978).

7. P. G. Neumann, et al., "A Provably Secure Operating System: The System, Its Applications, and Proofs," Final Report, SRI Project 4332, SRI International, Menlo Park, California (February 11, 1978).

8. J. H. Wensley, et al., "Design Study of Software-Implemented Fault-Tolerance (SIFT) Computer," Interim Technical Report 1, SRI Project 4026, SRI International, Menlo Park, California (June 1978).

9. S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," Computing Surveys of the ACM, Vol. 8, No. 3, pp. 331-353 (September 1976).

10. W. C. Carter, W .H. Joyner, and D. Brand, "Microprogram Verification Considered Necessary," Research Report RC 7053, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (December 13, 1977).

11. S. Saib, et al., "Advanced Software Quality Assurance," Final Report CR-3-770, General Research Corporation, Santa Barbara, California (May 1978).

12. J. C. King, "A Program Verifier," Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania (September 1969).

13. S. Katz and Z. Manna, "Logical Analysis of Programs," CACM, Vol. 19, No. 4, pp. 188-206 (April 1976).

14. S. M. German and B. Wegbreit, "A Synthesizer of Inductive Assertions," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, pp. 68-75 (March 1975).

15. S. K. Basu and J. Misra, "Proving Loop Programs," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, pp. 76-86 (March 1975).

16. J. H. Morris and B. Wegbreit, "Subgoal Induction," CACM, Vol. 20, No. 4, pp. 209-222 (April 1977).

17. R. S. Boyer, J S. Moore, and R. E. Shostak, "Primitive Recursive Program Transformation," Proc. 3rd ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, pp. 171-174 (January 1976).

18. J. Guttag, "Abstract Data Types and the Development of Data Structures," CACM, Vol. 20, No. 6, pp. 396-404 (June 1977).

19. B. Elspas, et al., "A Verification System for JOVIAL/J3 Programs (Rugged Programming Environment--RPE/1)," Technical Report 3756-1, Stanford Research Institute (January 1976).

20. B. Elspas, et al., "A Verification System for JOCIT/J3 Programs (Rugged Programming Environment--RPE/2)," Final Report, SRI Project 5042, SRI International, Menlo Park, California (April 1977).

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

22. L. C. Ragland, "A Verified Program Verifier," Ph.D. thesis, University of Texas at Austin (1973).

23. W. W. Bledsoe, "Program Correctness," Mathematics Department Memo ATP-14, The University of Texas at Austin (January 1974).

24. W. W. Bledsoe, "The Sup-Inf Method in Presburger Arithmetic," Mathematics Department Memo ATP-18, The University of Texas at Austin (December 1974).

25. W. W. Bledsoe, "A New Method for Proving Certain Presburger Formulas," Advance Papers, 4th Int. Joint Conf. on Artificial Intelligence, pp. 15-21, Tbilisi, Georgia U.S.S.R. (September 1975).

26. W. W. Bledsoe, R. S. Boyer, and W. H. Henneman, "Computer Proofs of LImit Theorems," Artificial Intelligence, Vol. 3, pp. 27-60 (1972).

27. W. W. Bledsoe and P. Bruell, "A Man-Machine Theorem-Proving System," Artificial Intelligence, Vol. 5, pp. 51-72 (1974).

28. D. C. Cooper, "Programs for Mechanical Program Verification," in Machine Intelligence, Vol. 6, pp. 43-59, American Elsevier, New York (1971).

29. G. B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, New Jersey (1962).

30. L. P. Deutsch, "An Interactive Program Verifier," Ph.D. thesis, University of California, Berkeley, California (1973).

31. R. E. Gomory, "An Algorithm for Integer Solutions to Linear Programs," Princeton IBM Math. Res. Report (November 1958); also in R. L. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming, pp. 269-302, McGraw-Hill, New York (1963).

32. D. I. Good, R. L. London, and W. W. Bledsoe, "An Interactive Verification System," Proc. Int. Conf. on Reliable Software, Los Angeles, California (April 1975).

33. S. Igarashi, R. L. London, and D. C. Luckham, "Automatic Program Verification 1: A Logical Basis and Its Implementation," Stanford AI Memo 200 (May 1973) and USC Information Sciences Institute Report ISI/RR-73-11 (May 1973).

34. D. B. Johnson, "Finding All the Elementary Circuits of a Directed Graph," SIAM J. Computing, Vol. 4, pp. 77-84 (1975).

35. R. D. Lee, "An Application of Mathematical Logic to the Integer Linear Programming Problem," Notre Dame J. Formal Logic, Vol. XIII, No. 2 (April 1972).

36. S. D. Litvintchouk and V. R. Pratt, "A Proof Checker for Dynamic Logic," 5th Int. Joint Conf. on Artif. Intell., pp 552-558, Cambridge, Massachusetts (August 1977).

37. M. Prabhaker and N. Deo, "On Algorithms for Enumerating All Circuits of a Graph," SIAM J. Computing, Vol. 5, No. 1 (March 1976).

38. V. R. Pratt, "Two Easy Theories Whose Combination is Hard," M.I.T. Technical Report, Cambridge, Massachusetts (September 1977).

39. R. C. Read and R. E. Tarjan, "Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees," ERL Memo M-433, Electronics Research Laboratory, University of California, Berkeley (1973).

40. R. Shostak, "An Efficient Decision Procedure for Arithmetic with Function Symbols," Presented at 5th Int. Joint Conf. on Artif. Intell., Cambridge, Massachusetts (August 1977).

41. R. Shostak, "On the Sup-Inf Method for Proving Presburger Formulas," J. ACM, Vol. 24, No. 4, pp. 529-543 (October 1977).

42. N. Suzuki, "Verifying Programs by Algebraic and Logical Reduction," Proc. Int. Conf. on Reliable Software (Sigplan Notices), Vol. 10, No. 6, (June 1975).

43. J. L. Szwarcfiter and P. E. Lauer, "Finding the Elementary Cycles of a Directed Graph in O(n+m) Per Cycle," No. 60, University of Newcastle Upn Tyne, Newcastle Upon Tyne, England (May 1974).

44. R. Tarjan, "Enumeration of the Elementary Circuits of a Directed Graph," SIAM J. Computing, Vol. 2, (1973).

45. R. J. Waldinger and K. N. Levitt, "Reasoning About Programs," J. Artif. Intell., Vol. 5, pp. 235-316 (1974).

46. B. Wegbreit, "Constructive Methods in Program Verification," IEEE Trans. on Software Engineering, Vol. SE-3, No. 3, pp. 193-209 (May 1977).

47. E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976).

48. R. S. Boyer and J S. Moore, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory," Proc. Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts (August 1977).

Appendix A

MACHINE PROOFS OF CONSISTENCY BETWEEN ALGEBRAIC SPECIFICATIONS AND A VCG IMPLEMENTATION

Appendix A

MACHINE PROOFS OF CONSISTENCY BETWEEN ALGEBRAIC SPECIFICATIONS AND A VCG IMPLEMENTATION

1. Proof of VCS Specification for the Empty Statement List.

Here we show the machine proof of consistency for the function VCS over an empty list of statements and an arbitrary predicate, viz., that

VCS("NIL",Q) = <Q>

This proof is, of course, accomplished trivially by definitional expansion, as made clear by the mechanically-generated trace output.

In this proof and also the succeeeding ones, lines typed by the user are prefaced by the prompt character _. All other lines are machine generated. The user of the prover first asks for an indented print-out (PP) of each theorem to be proved. Following each proof the system prints out some timing statistics for that call to the function PROVE.

```
_PP(NIL.THM1)

  (EQUAL (VCS "NIL" Q)
         (CONS Q "NIL"))
(NIL.THM1)

_(PROVE NIL.THM1)
This formula simplifies, unfolding the definition of VCS, to:

      (TRUE).

Q.E.D.

Load average during proof:  1.117388
Elapsed time:  .201 seconds
CPU time (devoted to theorem proving):  .091 seconds
IO time:  .093 seconds
CONSes consumed:  84


PROVED
```

2. Proof of VCS over statement list ending in a no-op.

If STAT is a no-op statement, it satisfies the predicate (NULLP STAT). In that case VCS(STL; STAT, Q) is supposed to be the same as VCS(STL, Q) where STL is any legal list of statements. This theorem is shown first, then the machine proof, which must consider four cases corresponding to the four types of no-op statements. Observe that in this version the definition for no-op statements included the statement type COMMENT (which was later eliminated in the formal language definitions shown in Section II).

```
_PP NULL.THM

  (IMPLIES (AND (LEGALP STL)
                (NULLP STAT))
           (EQUAL (VCS (APPEND STL (CONS STAT "NIL"))
                       Q)
                  (VCS STL Q)))
NULL.THM
_(PROVE NULL.THM)
```

This simplifies, using the lemmas APPEND.REVERSE, CAR.CONS, and CDR.CONS, and expanding the definitions of NULLP, AND, VCS, IMPLIES, APPEND, REVERSE, VCR, and LEGALP, to the following four new goals:

```
Case 1.    (IMPLIES (AND (LISTP STAT)
                         (EQUAL (CAR STAT) "LABEL"))
                    (EQUAL (VCR "NIL" Q)
                           (CONS Q "NIL"))).
```

However this again simplifies, unfolding the definition of VCR, to:

```
   (TRUE).
```

```
Case 2.    (IMPLIES (AND (LISTP STAT)
                         (EQUAL (CAR STAT) "COMMENT"))
                    (EQUAL (VCR "NIL" Q)
                           (CONS Q "NIL"))),
```

which again simplifies, expanding the function VCR, to:

```
   (TRUE).
```

```
Case 3.    (IMPLIES (AND (LISTP STAT)
                         (EQUAL (CAR STAT) "SKIP"))
                    (EQUAL (VCR "NIL" Q)
                           (CONS Q "NIL"))),
```

which we again simplify, opening up the definition of VCR, to:

```
   (TRUE).

Case 4.    (EQUAL (VCR "NIL" Q)
                  (CONS Q "NIL")),
```
which again simplifies, unfolding VCR, to:

```
   (TRUE).

Q.E.D.

Load average during proof:  .9096334
Elapsed time:  63.322 seconds
CPU time (devoted to theorem proving):  12.399 seconds
IO time:  .703 seconds
CONSes consumed:  16453


PROVED
```

3. Proof of VCS over statement list ending in an ASSUME.

Here we prove that the specification:

VCS(STL; ASSUME P, Q) = VCS(STL, P->Q)

is satisfied, where STL is any legal list of statements, and P, Q are arbitrary predicates.

```
_PP(ASSUME.THM)

  (IMPLIES (LEGALP STL)
           (EQUAL (VCS (APPEND STL
                               (CONS (CONS "ASSUME"
                                           (CONS P "NIL"))
                                     "NIL"))
                       Q)
                  (VCS STL
                       (CONS "IMPLIES"
                             (CONS P (CONS Q "NIL")))))))
(ASSUME.THM)

_(PROVE ASSUME.THM)
```
This formula simplifies, applying APPEND.REVERSE, CAR.CONS, and CDR.CONS, and opening up the definitions of VCS, IMPLIES, APPEND, REVERSE, LEGALP, and VCR, to two new goals:

```
Case 1. (IMPLIES (AND (LEGALP STL) (LISTP STL))
                 (EQUAL (VCR (CONS (CONS "ASSUME"
                                         (CONS P "NIL"))
                                   (REVERSE STL))
                             Q)
                        (VCR (REVERSE STL)
```

```
                                (CONS "IMPLIES"
                                      (CONS P (CONS Q "NIL"))))))),
```

which we again simplify, applying the lemmas CDR.CONS and CAR.CONS, and opening up the function VCR, to:

```
        (TRUE).

Case 2. (EQUAL (VCR "NIL"
                    (CONS "IMPLIES"
                          (CONS P (CONS Q "NIL"))))
               (CONS (CONS "IMPLIES"
                           (CONS P (CONS Q "NIL")))
                     "NIL")).
```

This again simplifies, expanding the function VCR, to:

```
        (TRUE).
```

Q.E.D.

```
Load average during proof:  1.338743
Elapsed time:  33.306 seconds
CPU time (devoted to theorem proving):  3.452 seconds
IO time:  .536 seconds
CONSes consumed:  6343


PROVED
```

4. Proof of VCS over a statement list ending in an ASSERT.

Here we show that the specification:

VCS(STL; ASSERT A, Q) = VCS(STL, A) @ <A->Q)>

is satisfied, where STL is any legal statement list, and A, Q are arbitrary predicates.

```
_PP ASSERT.THM

  (IMPLIES (LEGALP STL)
           (EQUAL (VCS (APPEND STL
                               (CONS (CONS "ASSERT" (CONS P
                                                          "NIL"))
                                     "NIL"))
                       Q)
                  (APPEND (VCS STL P)
                          (CONS (CONS "IMPLIES"
                                      (CONS P (CONS Q "NIL")))
                                "NIL")))))

ASSERT.THM
```

```
_(PROVE ASSERT.THM)
```
This simplifies, using the lemmas APPEND.REVERSE, CAR.CONS, and CDR.CONS, and expanding VCS, IMPLIES, APPEND, REVERSE, VCR, and LEGALP, to:

```
(EQUAL (APPEND (VCR "NIL" P)
               (CONS (CONS "IMPLIES"
                           (CONS P (CONS Q "NIL")))
                     "NIL"))
       (CONS P
             (CONS (CONS "IMPLIES"
                         (CONS P (CONS Q "NIL")))
                   "NIL"))),
```

which we again simplify, using the lemmas CDR.CONS and CAR.CONS, and expanding the functions VCR and APPEND, to:

```
      (TRUE).
```

Q.E.D.

```
Load average during proof:  2.392729
Elapsed time:  32.954 seconds
CPU time (devoted to theorem proving):  2.898 seconds
IO time:  .741 seconds
CONSes consumed:  5412


PROVED
```

5. Proof of VCS over statement list ending in assignment.

Here we prove that the specification:

VCS(STL; X:=A, Q) = VCS(STL, SUBST(A X Q))

is satisfied, where STL is any legal statement list, Q is an arbitrary predicate, X is a variable name, and A is any expression. The function SUBST (known to the theorem prover) is like the LISP function, i.e., SUBST(X,Y,Z) is the result of substituting an occurrence of X for each occurrence of Y in Z.

```
_PP ASST.THM

  (IMPLIES (LEGALP STL)
           (EQUAL (VCS (APPEND STL
                               (CONS (CONS ":="
                                           (CONS X
                                                 (CONS A
                                                       "NIL")))
```

```
                                        "NIL"))
                    Q)
                (VCS STL (SUBST A X Q))))
ASST.THM

_(PROVE ASST.THM)
```

This simplifies, using the lemmas APPEND.REVERSE, CAR.CONS, and CDR.CONS, and opening up the functions VCS, IMPLIES, APPEND, REVERSE, VCR, and LEGALP, to:

```
      (EQUAL (VCR "NIL" (SUBST A X Q))
             (CONS (SUBST A X Q) "NIL")).
```

This again simplifies, expanding the definition of VCR, to:

```
      (TRUE).
```

Q.E.D.

```
Load average during proof:  1.596421
Elapsed time:  15.719 seconds
CPU time (devoted to theorem proving):  2.491 seconds
IO time:  .329 seconds
CONSes consumed:  4727


PROVED
```

6. Proof of VCS over statement list ending in GOTO.

Here we prove the specification:

VCS(STL; GOTO LABEL (ASSERTING PRED), Q) = VCS(STL, PRED)

where STL is any legal statement list, LABEL is any statement label, and PRED is the assertion attached to the labelled statement.

```
_PP GOTO.THM

  (IMPLIES (LEGALP STL)
           (EQUAL
              (VCS (APPEND STL
                           (CONS (CONS "GOTO"
                                       (CONS LABEL
                                             (CONS PRED "NIL")))
                                 "NIL"))
                   Q)
              (VCS STL PRED)))
GOTO.THM
```

```
_(PROVE GOTO.THM)
```
This formula simplifies, applying the lemmas APPEND.REVERSE, CAR.CONS, and CDR.CONS, and opening up the definitions of VCS, IMPLIES, APPEND, REVERSE, VCR, and LEGALP, to:

```
      (EQUAL (VCR "NIL" PRED)
             (CONS PRED "NIL")),
```

which again simplifies, unfolding the function VCR, to:

```
      (TRUE).
```

Q.E.D.

```
Load average during proof:  1.777559
Elapsed time:  16.588 seconds
CPU time (devoted to theorem proving):  2.566 seconds
IO time:  .32 seconds
CONSes consumed:  4291


PROVED
```

7. Proof of VCS over statement list ending in a statement block.

Here we prove that the specification:

VCS(STL1; BEGIN STL2 END, Q) = VCS(STL1 @ STL2, Q)

is satisfied where STL1 and STL2 are any legal statement lists.

```
_PP(BEGIN.THM)

  (IMPLIES (AND (LEGALP STL1)
                (LEGALP STL2))
           (EQUAL (VCS (APPEND STL1 (CONS (CONS "BEGIN" STL2)
                                          "NIL"))
                       Q)
                  (VCS (APPEND STL1 STL2)
                       Q)))
(BEGIN.THM)

_(PROVE BEGIN.THM)
```
This conjecture simplifies, applying APPEND.REVERSE, CAR.CONS, and CDR.CONS, and expanding the functions AND, VCS, IMPLIES, APPEND, REVERSE, and VCR, to two new conjectures:

```
Case 1. (IMPLIES (AND (LEGALP STL1)
                      (LEGALP STL2)
                      (NOT (EQUAL (APPEND STL1 STL2) "NIL"))
                      (NOT (LISTP (APPEND STL1 STL2))))
                 (EQUAL (VCR (APPEND (REVERSE STL2)
```

```
                                        (REVERSE STL1))
                                    Q)
                                "UNDEFINED")).
```

Name the above subgoal *1.

```
Case 2. (IMPLIES (AND (LEGALP STL1)
                      (LEGALP STL2)
                      (EQUAL (APPEND STL1 STL2) "NIL"))
                 (EQUAL (VCR (APPEND (REVERSE STL2)
                                     (REVERSE STL1))
                             Q)
                        (CONS Q "NIL"))),
```

which we would normally push and work on later by induction. But since we have already pushed one goal split off of the original input we will disregard all that we have previously done, give the name *1 to the original input, and work on it.

So now let's consider:

```
(IMPLIES (AND (LEGALP STL1) (LEGALP STL2))
         (EQUAL (VCS (APPEND STL1
                             (CONS (CONS "BEGIN" STL2) "NIL"))
                     Q)
                (VCS (APPEND STL1 STL2) Q))).
```

We gave this the name *1 above. Let us appeal to the induction principle. Four inductions are suggested by terms in the conjecture. They merge into three likely candidate inductions, none of which is unflawed. However, one is more likely than the others. We will induct according to the following scheme:

```
(AND (IMPLIES (NOT (LISTP STL1))
              (p STL1 STL2 Q))
     (IMPLIES (AND (LISTP STL1)
                   (p (CDR STL1) STL2 Q))
              (p STL1 STL2 Q))).
```

The inequality CDR.LESSP establishes that the measure (COUNT STL1) decreases according to the well-founded function LESSP in the induction step of the scheme. The above induction scheme leads to two new conjectures:

```
Case 1.    (IMPLIES
                (NOT (LISTP STL1))
                (IMPLIES (AND (LEGALP STL1) (LEGALP STL2))
                         (EQUAL (VCS (APPEND STL1
                                             (CONS (CONS "BEGIN"
                                                         STL2)
                                                   "NIL"))
                                     Q)
```

```
                                    (VCS (APPEND STL1 STL2) Q))))).
```

This simplifies, applying the lemmas CAR.CONS, CDR.CONS, PLISTP.REVERSE, and APPEND.RIGHT.ID, and expanding LEGALP, AND, APPEND, REVERSE, VCR, VCS, and IMPLIES, to:

```
(TRUE).
```

Case 2.

```
(IMPLIES
  (AND (LISTP STL1)
       (IMPLIES (AND (LEGALP (CDR STL1))
                     (LEGALP STL2))
         (EQUAL (VCS (APPEND (CDR STL1)
                             (CONS (CONS "BEGIN" STL2)
                                   "NIL"))
                     Q)
                (VCS (APPEND (CDR STL1) STL2) Q))))
  (IMPLIES (AND (LEGALP STL1) (LEGALP STL2))
           (EQUAL (VCS (APPEND STL1
                               (CONS (CONS "BEGIN" STL2)
                                     "NIL"))
                       Q)
                  (VCS (APPEND STL1 STL2) Q)))))
```

This simplifies, applying the lemmas APPEND.REVERSE, CAR.CONS, CDR.CONS, and ASSOCIATIVITY.OF.APPEND, and expanding the functions AND, APPEND, REVERSE, VCR, VCS, and IMPLIES, to:

```
(TRUE).
```

That finishes the proof of *1. Q.E.D.

```
Load average during proof:  1.191621
Elapsed time:  183.979 seconds
CPU time (devoted to theorem proving):  40.169 seconds
IO time:  2.05 seconds
CONSes consumed:  63919


PROVED
```

8. Proof of VCS over statement list ending in PROVE.

Here we prove that the specification:

VCS(STL; PROVE P, Q) = VCS(STL, P) @ VCS(STL, P->Q)

is satisfied, where STL is any legal statement list and P,Q are predicates.

```
_PP PROVE.THM

  (IMPLIES (LEGALP STL)
           (EQUAL (VCS (APPEND STL
                               (CONS (CONS "PROVE"
                                           (CONS P "NIL"))
                                     "NIL"))
                       Q)
                  (APPEND (VCS STL P)
                          (VCS STL
                               (CONS "IMPLIES"
                                     (CONS P (CONS Q "NIL")))))))
PROVE.THM

_(PROVE PROVE.THM)
This conjecture simplifies, applying the lemmas APPEND.REVERSE,
CAR.CONS, and CDR.CONS, and unfolding the definitions of VCS,
IMPLIES, APPEND, REVERSE, VCR, and LEGALP, to the formula:

 (EQUAL (APPEND (VCR "NIL" P)
                (VCR "NIL"
                     (CONS "IMPLIES"
                           (CONS P (CONS Q "NIL")))))
        (CONS P
              (CONS (CONS "IMPLIES"
                          (CONS P (CONS Q "NIL")))
                    "NIL"))).

However this again simplifies, rewriting with CDR.CONS and
CAR.CONS, and expanding VCR and APPEND, to:

      (TRUE).

Q.E.D.

Load average during proof:  4.81208
Elapsed time:  31.836 seconds
CPU time (devoted to theorem proving):  4.201 seconds
IO time:  .8 seconds
CONSes consumed:  6574


PROVED
```

9. Proof of VCS over statement list ending in an ABORT.

Here we show that the specification:

VCS(STL; ABORT, Q) = VCS(STL, T)

is satisfied, where STL is any legal statement list and Q is any predicate.

```
_PP ABORT.THM

  (IMPLIES (LEGALP STL)
           (EQUAL (VCS (APPEND STL (CONS (CONS "ABORT" "NIL")
                                         "NIL"))
                       Q)
                  (VCS STL T)))
ABORT.THM

_(PROVE ABORT.THM)
This simplifies, applying the lemmas APPEND.REVERSE, CDR.CONS,
and CAR.CONS, and unfolding the definitions of VCS, IMPLIES,
REVERSE, APPEND, VCR, and LEGALP, to:

      (TRUE).

Q.E.D.

Load average during proof:  1.684852
Elapsed time:  4.256 seconds
CPU time (devoted to theorem proving):  2.521 seconds
IO time:  .179 seconds
CONSes consumed:  4355


PROVED
```

This completes the proof traces generated by the recursive function theorem prover in demonstrating consistency between the LISP implementation shown below and the algebraic specifications S0-S11 given in Sec. II-D.

10. LISP Code for the Verified Implementation of VCG

Below we exhibit one verified implementation of VCG (the verification condition generator for the language SL). This particular version was written in MacLISP. We have also verified an InterLISP version that differs only in minor ways from the one shown. Another InterLISP version that has been written (but which is as yet unverified)

partitions the main function VCR into eleven subfunctions, such as VCR:IF, VCR:ASST, and VCR:ASSUME, each corresponding to one of the "cond" clauses in VCR shown below.

```
(DEFUN VCG (SL Q) (VCR (REVERSE SL) Q))

(DEFUN VCR (RL POST)
       (COND ((NULL RL) (LIST POST))
             ((NULLP (CAR RL)) (VCR (CDR RL) POST))
             ((EQ (CAAR RL) 'ASSUME)
              (VCR (CDR RL)
                   (LIST 'IMPLIES (CADR (CAR RL)) POST)))
             ((EQ (CAAR RL) 'ASSERT)
              (APPEND (VCR (CDR RL) (CADR (CAR RL)))
                      (LIST (LIST 'IMPLIES
                                  (CADR (CAR RL))
                                  POST))))
             ((EQ (CAAR RL) 'GOTO)
              (VCR (CDR RL) (CADDR (CAR RL))))
             ((EQ (CAAR RL) 'PROVE)
              (APPEND (VCR (CDR RL) (CADR (CAR RL)))
                      (VCR (CDR RL)
                           (LIST 'IMPLIES
                                 (CADR (CAR RL))
                                 POST))))
             ((EQ (CAAR RL) 'BEGIN)
              (VCR (APPEND (REVERSE (CDR (CAR RL))) (CDR RL))
                   POST))
             ((EQ (CAAR RL) 'IF)
              (APPEND (VCR (APPEND (LIST (CADDR (CAR RL)))
                                   (LIST (LIST 'ASSUME
                                               (CADR (CAR RL))))
                                   (CDR RL))
                           POST)
                      (VCR (APPEND (LIST (CADDDR (CAR RL)))
                                   (LIST (LIST 'ASSUME
                                               (LIST 'NOT
                                                     (CADAR RL))))
                                   (CDR RL))
                           POST)))
             ((EQ (CAAR RL) ':=)
              (VCR (CDR RL)
                   (SUBST (CADDR (CAR RL)) (CADR (CAR RL)) POST)))
             ((EQ (CAAR RL) 'WHILE)
              (APPEND (VCR (CDR RL)
                           (CADR (CAR RL)))
                      (VCR (APPEND (CDDDR (CAR RL))
                                   (LIST (LIST 'ASSUME
                                               (LIST 'AND
                                                     (CADR (CAR RL))
                                                     (CADDAR RL))))))
```

```
                    (CADR (CAR RL)))
                (LIST (LIST (QUOTE IMPLIES)
                            (LIST (QUOTE AND)
                                  (CADR (CAR RL))
                                  (LIST (QUOTE NOT)
                                        (CADDR (CAR RL))))
                            POST)))))
          ((EQ (CAAR RL) 'ABORT) (VCR (CDR RL) T))
          (T (PRINT (LIST 'ERROR:
                          (CAAR RL)
                          'IS
                          'UNDEFINED))
             (IOC G))))

(DEFUN NULLP (S)
       (OR (NULL S)
           (EQUAL S '(SKIP))
           (EQUAL (CAR S) 'COMMENT)
           (EQ (CAR S) 'LABEL)))
```

Observe that the function VCR is defined so that input of an unrecognized statement list (as argument RL) produces an error break (IOC G). In the actual definition of VCR supplied to the theorem prover this has been replaced by the result "UNDEFINED", since the theorem prover insists upon total functions. These definitions are shown in the next section of this appendix.

11. Definitions Provided to the Recursive Function Theorem Prover

We show here the formal definitions ("DEFNS") that were provided to the Boyer-Moore Theorem Prover for Recursive Functions in order to allow the Prover to demonstrate the consistency theorems.

Some of these definitions define the (abstract) syntax of SL programs. The functions NULLP (defining the syntax of no-ops in SL), ASRTNP (defining the syntax of predicates in SL), and ASSTP (defining the syntax for assignment statements in SL) are of this type. In addition, the function LEGALSTATP gives the syntax for "statement" in SL, while LEGALP defines the syntax for "statement-list".

The main recursive function definitions are those for VCS, the top-level function of the VCG, and its subfunction VCR, which does the actual work (on a reversed copy of the first argument to VCS, a statement-list). Both of these are intended to be accurate Boyer-Moore DEFN versions of the actual implementation (which happens to be in MacLisp). This translation step was done by hand, but, in principle, it could have been done by a mechanical translation since there is little more to do than translate such things as LISP "cond's" into Boyer-Moore three-argument IFs, and the like.

```
(DEFN NULLP (S)
            (IF (LISTP S)
                (IF (OR (EQUAL (CAR S)
                               "SKIP")
                        (EQUAL (CAR S)
                               "LABEL")
                        (EQUAL (CAR S)
                               "COMMENT"))
                    T F)
                (EQUAL S "NIL"))
            NIL)

(DEFN ASSTP (S)
      (IF (LISTP S)
          (EQUAL (CAR S)
                 ":=")
          F)
      NIL)

(DEFN ASRTNP (A)
      (IF (NLISTP A)
          T
          (PLISTP A))
      NIL)

(DEFN
  LEGALSTATP
  (S)
  (IF
    (NLISTP S)
    (EQUAL S "NIL")
    (IF
      (NULLP S)
      T
      (IF
        (ASSTP S)
        T
        (IF
```

```
            (AND (EQUAL (CAR S)
                        "ASSERT")
                 (ASRTNP (CADR S)))
            T
            (IF
              (AND (EQUAL (CAR S)
                          "ASSUME")
                   (ASRTNP (CADR S)))
              T
              (IF
                (AND (EQUAL (CAR S)
                            "PROVE")
                     (ASRTNP (CADR S)))
                T
                (IF
                  (AND (EQUAL (CAR S)
                              "BEGIN")
                       (PLISTP (CDR S)))
                  T
                  (IF
                    (EQUAL (CAR S)
                           "GOTO")
                    T
                    (IF (EQUAL (CAR S)
                               "ABORT")
                        T
                        (IF (AND (EQUAL (CAR S)
                                        "WHILE")
                                 (ASRTNP (CADR S))
                                 (ASRTNP (CADDR S))
                                 (LEGALSTATP (CADDDR S)))
                            T
                            (AND (EQUAL (CAR S)
                                        "IF")
                                 (ASRTNP (CADR S))
                                 (LEGALSTATP (CADDR S))
                                 (LEGALSTATP (CADDDR S))...)
  NIL)

(DEFN LEGALP (L)
      (IF (NLISTP L)
          (EQUAL L "NIL")
          (AND (LEGALSTATP (CAR (REVERSE L)))
               (LEGALP (REVERSE (CDR (REVERSE L))))))
      NIL)

(DEFN
  VCR
  (RL Q)
  (IF
    (EQUAL RL "NIL")
    (CONS Q "NIL")
```

```
(IF
  (NULLP (CAR RL))
  (VCR (CDR RL)
       Q)
  (IF
    (ASSTP (CAR RL))
    (VCR (CDR RL)
         (SUBST (CADDR (CAR RL))
                (CADR (CAR RL))
                Q))
    (IF
      (EQUAL (CAAR RL)
             "ASSUME")
      (VCR (CDR RL)
           (CONS "IMPLIES" (CONS (CADR (CAR RL))
                                 (CONS Q "NIL"))))
      (IF
        (EQUAL (CAAR RL)
               "ASSERT")
        (APPEND (VCR (CDR RL)
                     (CADR (CAR RL)))
                (CONS (CONS "IMPLIES"
                            (CONS (CADR (CAR RL))
                                  (CONS Q "NIL")))
                      "NIL"))
        (IF
          (EQUAL (CAAR RL)
                 "GOTO")
          (VCR (CDR RL)
               (CADDR (CAR RL)))
          (IF
            (EQUAL (CAAR RL)
                   "BEGIN")
            (VCR (APPEND (REVERSE (CDR (CAR RL)))
                         (CDR RL))
                 Q)
            (IF
              (EQUAL (CAAR RL)
                     "IF")
              (APPEND
                (VCR
                  (CONS
                    (CADDR (CAR RL))
                    (CONS (CONS "ASSUME"
                                (CONS (CADR (CAR RL))
                                      "NIL"))
                          (CDR RL)))
                  Q)
                (VCR
                  (CONS
                    (CADDDR (CAR RL))
                    (CONS
```

```
            (CONS "ASSUME"
                  (CONS "NOT"
                        (CONS (CADR (CAR RL))
                              "NIL")))
            (CDR RL)))
      Q))
(IF
  (EQUAL (CAAR RL)
         "PROVE")
  (APPEND
    (VCR (CDR RL)
         (CADR (CAR RL)))
    (VCR (CDR RL)
         (CONS "IMPLIES"
               (CONS (CADR (CAR RL))
                     (CONS Q "NIL")))))
  (IF
    (EQUAL (CAAR RL)
           "ABORT")
    (VCR (CDR RL)
         T)
    (IF
      (EQUAL (CAAR RL)
             "WHILE")
      (APPEND
        (VCR (CDR RL)
             (CADR (CAR RL)))
        (APPEND
          (VCR
            (CONS
              (CADDDR (CAR RL))
              (CONS
                (CONS
                  "ASSUME"
                  (CONS
                    (CONS
                      "AND"
                      (CONS
                        (CADR (CAR RL))
                        (CONS
                          (CADDR (CAR RL))
                          "NIL")))
                    "NIL"))
                "NIL"))
            (CADR (CAR RL)))
          (CONS
            (CONS
              "IMPLIES"
              (CONS
                (CONS
                  "AND"
                  (CONS
```

```
                                                    (CADR (CAR RL))
                                                    (CONS
                                                      (CONS
                                                        "NOT"
                                                        (CONS
                                                          (CADDR (CAR RL))
                                                          "NIL"))
                                                      "NIL")))
                                                (CONS Q "NIL")))
                                            "NIL")))
                                        "UNDEFINED")))))))))))
        NIL)

      (DEFN VCS (STL Q)
            (IF (LISTP STL)
                (VCR (REVERSE STL)
                     Q)
             "UNDEFINED"))
            NIL)))
```

The LISP functions, TLIST and TLIST1, whose definitions appear below, are auxiliary (utility) functions designed to tranlate LISP list expressions that represent SL programs such as:

```
'((ASSUME B) (:= X A) (IF P S1 (BEGIN S2 S3)))
```

into the required Boyer-Moore syntax, in this case into:

```
'(CONS (CONS "ASSUME" (CONS B "NIL"))
       (CONS (CONS ":=" (CONS X (CONS A "NIL"))
             (CONS (CONS "IF"
                         (CONS P (CONS S1
                                       (CONS (CONS "BEGIN"
                                                   (CONS S2
                                                         (CONS S3 "NIL")...)
```

```
(TLIST
  [LAMBDA (L)
    (COND
      ((STRINGP L))
      ((ATOM L)
        (MKSTRING L))
      (T (LIST (QUOTE CONS)
               (TLIST (CAR L))
               (TLIST1 (CDR L]))

(TLIST1
  [LAMBDA (L)
    (COND
      ((STRINGP L))
```

```
((NULL L)
  "NIL")
((ATOM L)
  L)
(T (LIST (QUOTE CONS)
         [COND
           ((LISTP (CAR L))
             (TLIST (CAR L)))
           (T (TLIST1 (CAR L]
         (TLIST1 (CDR L])
```

TLIST was used to help translate the algebraic specifications to be proved into the Theorem Prover's syntax.

Appendix B

REACHING AGREEMENT IN THE PRESENCE OF FAULTS

by

M. C. Pease
R. E. Shostak
L. Lamport

Appendix B

REACHING AGREEMENT IN THE PRESENCE OF FAULTS

1. Introduction

Fault-tolerant systems often require a means by which independent processors or processes can arrive at an exact mutual agreement of some kind. It may be necessary, for example, for the processors of a redundant system to synchronize their internal clocks periodically. Or they may have to settle upon a value of a time-varying input sensor that gives each of them a slightly different reading. In the absence of faults, reaching a satisfactory mutual agreement is usually an easy matter. In most cases, it suffices simply to exchange values (times, in the case of clock synchronization) and compute some kind of average. In the presence of faulty processors, however, simple exchanges cannot be relied upon; a bad processor might report one value to a given processor, and another value to some other processors, causing each to calculate a different "average."

One might imagine that the effects of faulty processors could be dealt with through the use of voting schemes involving more than one round of information exchange; such schemes might force faulty processors to reveal themselves as faulty, or at least to behave consistently enough with respect to the nonfaulty processors to allow the latter to reach an exact agreement. As we will show, it is not always possible to devise schemes of this kind, even if it is known that the faulty processors are in a minority. Algorithms that allow exact agreement to be reached by the nonfaulty processors do exist, however, if they sufficiently outnumber the faulty ones.

Our results are formulated using the notion of interactive consistency, which we define as follows. Consider a set of n isolated processors, of which it is known that no more than m are faulty. It is not known, however, which processors are faulty. Suppose that the processors

can communicate only by means of two-party messages. The communication medium is presumed to be fail-safe and of negligible delay. The sender of a message, moreover, is always identifiable by the receiver. Suppose also that each processor p has some private value of information V_p (such as its clock value or its reading of some sensor). The question is, for given m, $n \geq 0$, whether it is possible to devise an algorithm based on an exchange of messages that will allow each nonfaulty processor p to compute a vector of values with an element for each of the n processors, such that:

(1) The nonfaulty processors compute exactly the same vector.

(2) The element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

Note that the algorithm need not reveal which processors are faulty, and that the elements of the computed vector corresponding to faulty processors may be arbitrary--it matters only that the nonfaulty processors compute exactly the same value for any given faulty processor.

We will say that such an algorithm achieves interactive consistency, since it allows the nonfaulty processors to come to a consistent view of the values held by all the processors, including the faulty ones. The computed vector is called an interactive consistency (i.c.) vector. Once interactive consistency has been achieved, each nonfaulty processor can apply an averaging or filtering function to the i.c. vector, according to the needs of the application. Since each nonfaulty processor applies this function to the same vector of values, an exact agreement is necessarily reached.

We will show in the following sections that algorithms can be devised to guarantee interactive consistency for and only for n,m such that $n \geq 3m + 1$. It will follow, in particular, that a minimum of four processors is required in the single-fault case. We will also show, however, that interactive consistency can be assured for arbitrary $n \geq m \geq 0$ if it is assumed that faulty processors can refuse to pass on information obtained from other processors, but cannot falsely report this information. This assumption can be approximated in practice using authenticators, which we discuss in Section 5.

We begin in Section 2 with a description of the single-fault case. Section 3 is concerned with the generalization to $n \geq 3m + 1$ and Section 4 with an impossibility argument for $n \leq 3m$. Section 5 gives an algorithm for arbitrary $n \geq m \geq 0$ that works under the restricted assumption stated above. Conclusions and issues for future study are given in Section 6.

Problems similar to the one considered here have been studied by Davies and Wakerly [1].

2. The Single-Fault Case

In order to give the reader a feeling for the problem, we begin with a procedure for obtaining interactive consistency in the simple case of $m = 1$, $n = 4$.

The procedure consists of an exchange of messages, followed by the computation of the interactive consistency vector on the basis of the results of the exchange.

Two rounds of information exchange are required. In the first round, the processors exchange their private values. In the second round, they exchange the results obtained in the first round. The faulty processor (if there is one) may "lie," of course, or refuse to send messages. If a nonfaulty processor p fails to receive a message it expects from some other processor, p simply chooses a value at random and acts as if that value had been sent.

The exchange having been completed, each nonfaulty processor p records its private value V_p for the element of the interactive consistency corresponding to p itself. The element corresponding to every other processor q is obtained by examining the three received reports of q's value (one of these was obtained directly from q in the first round, and the others from the remaining two processors in the second round). If at least two of the three reports agree, the majority value is used. Otherwise, a default value such as "NIL" is used.

To see that this procedure assures interactive consistency, first note that if q is nonfaulty, p will receive V_q both from q and from the

other nonfaulty processor(s). p will thus record V_q for q as desired. Now suppose q is faulty. We must show only that p and the other two nonfaulty processors record the same value for q. If every nonfaulty processor records NIL, we are done. Otherwise, some nonfaulty processor, say p, records a non-NIL value v, having received v from at least two other processors. Now if p received v from both of the other nonfaulty processors, each other nonfaulty processor must receive v from every processor other than p (and possibly from p as well); every nonfaulty processor will thus record v. Otherwise, p must have received v from all processors other than some other nonfaulty processor p'. In this case p' received v from all processors other than q (so p' records v) and all other nonfaulty processors received v from all processors other than p'. All nonfaulty processors therefore record v as required.

3. A Procedure for $n \geq 3m + 1$

Recall that the procedure given in the last section requires two rounds of information exchange, the first consisting of communications of the form "my private value is" and the second consisting of communications of the form "processor x told me his private value is...". In the general case of m faults, m + 1 rounds are required. In order to describe the algorithm, it will be convenient to characterize this exchange of messages in a more formal way.

Let P be the set of processors, and V a set of values. For $k \geq 1$, we define a k-level scenario as a mapping from the set of nonempty strings over P of length $\leq k + 1$, to V. For a given k-level scenario σ and string $w = p_1 p_2 \ldots p_r$, $2 \leq r \leq k + 1$, $\sigma(w)$ is interpreted as the value p_2 tells p_1 that p_3 told p_2 that p_4 told p_3 ... that p_r told p_{r-1} is p_r's private value. For a single-element string p, $\sigma(p)$ simply designates p's private value V_p. A k-level scenario thus summarizes the outcome of a k-round exchange of information. Note that, for a given subset of nonfaulty processors, only certain mappings are possible scenarios; in

particular, since nonfaulty processors are always truthful in relaying information, a scenario must satisfy:

$$\sigma(pqw) = \sigma(qw)$$

for each nonfaulty processor q, arbitrary processor p, and string w.

The messages a processor p receives in a scenario σ are given by the restriction σ_p of σ to strings beginning with p. The procedure we present now for arbitrary $m \geq 0$, $n \geq 3m + 1$, is described in terms of p's computation, for a given σ_p, of the element of the interactive-consistency vector corresponding to each processor q. The computation is as follows:

(1) If for some subset Q of P of size $> (n+m)/2$ and some value v, $\sigma_p(pwq) = v$ for each string w over Q of length $\leq m$, p records v.

(2) Otherwise, the algorithm for m-1, n-1 is recursively applied with P replaced by P - $\{q\}$, and σ_p by the mapping $\hat{\sigma}_p$ defined by:

$$\hat{\sigma}_p(pw) = \sigma_p(pwq)$$

for each string w of length $\leq m$ over p - $\{q\}$. If at least $\lfloor (n+m)/2 \rfloor$ of the n-1 elements in the vector obtained in the recursive call agree, p records the common value, otherwise p records NIL.

Note that $\hat{\sigma}_p$ corresponds to the m-level subscenario of σ in which q is excluded and in which each processor's private value is the value it obtains directly from q in σ. Note also that the algorithm essentially reduces to the one given in the last section in the case m = 1, n = 4.

The proof that the algorithm given above does indeed assure interactive consistency proceeds by induction on m:

Basis m = 0.

In this case, no processor is faulty, and the algorithm always terminates in step (1) with p recording V_q for q.

Induction Step $m > 0$.

First note that if q is nonfaulty, $\sigma_p(pwq) = V_q$ for each string w of length $\leq m$ over the set of nonfaulty processors. This set has n-m members (which, since $n \geq 3m$, is $> (n+m)/2$) and so satisfies the requirements for Q in step (1) of the algorithm. Any other set satisfying these requirements, moreover, must contain a nonfaulty processor (since it must be of size $> (n+m)/2$, and $n \geq 3m + 1$), and must therefore also yield V_q as the common value. The algorithm thus terminates at step (1), and p records V_q for q as required.

Now suppose that q is faulty. We must show that the value p records for q agrees with the value each other nonfaulty processor p' records for q.

First consider the case in which both p and p' exit the procedure at step (1), each having found an appropriate set Q. Since each such set has more than $(n+m)/2$ members, and since P has only n members in all, the two sets must have more than $2((n+m)/2) - n = m$ common members. Since at least one of these must be nonfaulty, the two sets must give rise to the same value v, as required.

Next suppose that p' exits at step (1), having found an appropriate set Q and common value v, and that p executes step (2). We claim that in the vector of n-1 elements that p computes in the recursive call, the elements corresponding to members of $\hat{Q} = Q - \{q\}$ are equal to v. Since $\hat{Q}$ has at least $\lfloor (n+m)/2 \rfloor$ members, it will then follow that p records v in accordance with step (2). To see that the elements corresponding to members of $\hat{Q}$ are indeed equal to v, recall that the mapping $\hat{\sigma}_p$ that p uses to compute the vector in the recursive call is the restriction, to strings beginning with p, of the m-level scenario $\hat{\sigma}$ defined by:

$$\hat{\sigma}(w) = \sigma(wq)$$

for each string w of length $\leq m$ over $P - \{q\}$. By induction hypothesis, this vector is identical to the one p' would have computed using the restriction $\hat{\sigma}_{p'}$ of $\hat{\sigma}$ had p' made the recursive call. Moreover, the value p' would have computed for the element of this vector corresponding to a

given q' in $\hat{Q}$ must be v, since $\hat{Q}$ and v satisfy step (1) of the algorithm. (Note that $\hat{Q}$ is of size $\geq \lfloor (n+m)/2 \rfloor > [(n-1) + (m-1)]/2$, and that $\sigma_{p'}(pwq') = \sigma_{p'}(p'wq'q) = v$ for each string w of length $\leq m-1$ over $\hat{Q}$).

The case in which p exits at step (1) and p' exits at step (2) is handled similarly.

In the one remaining case, both p and p' exit at step (2). In this case both recurse, and must, by induction hypothesis, compute exactly the same vector, and hence the same value for q. Q.E.D.

4. Proof of Impossibility for $n < 3m + 1$

The procedure of the last section guarantees interactive consistency only if $n \geq 3m + 1$. In this section it is shown that the $3m + 1$ bound is tight. We will prove not only that it is impossible to assure interactive consistency for $n < 3m + 1$ with $m + 1$ rounds of information exchange, but also that it is impossible, even allowing an infinite number of rounds of exchange (i.e., using scenarios mapping from all nonempty strings over P to V).

Just to gain some intuitive feeling as to why 3m processors are not sufficient, consider the case of three processors A, B, C, of which one, say C, is faulty. By prevaricating in just the right way, C can thwart A's and B's efforts to obtain consistency. In particular, C's messages to A can be such as to suggest to A that C's private value is, say, 1, and that B is faulty. Similarly, C's messages to B can be such as to suggest to B that C's private value is 2, and that A is faulty. If C plays its cards just right, A will not be able to tell which of B and C is faulty, and B will not be able to tell which of A and C is at fault. A will thus have no choice but to record 1 for C's value while B must record 2, defeating interactive consistency.

In order to give a precise statement of the impossibility result and its proof, a few formal definitions are needed.

First, define a scenario as a mapping from the set P^+ of all nonempty strings over P, to V. For a given $p \in P$ define a p-scenario as a mapping from the subset of P^+ consisting of strings beginning with p, to V.

Next, for a given choice $N \subseteq P$ of nonfaulty processors, and a given scenario σ, say that σ is consistent with N if for each $p \in N$, $q \in P$ and $w \in P^*$ (set of all strings over P), $\sigma(pqw) = \sigma(qw)$. (In other words, σ is consistent with N if each processor in N always reports what it knows or hears truthfully.)

Now define the notion of interactive consistency as follows. For each $p \in P$, let F_p be a mapping that takes a p-scenario σ_p and a processor q as arguments, and returns a value in V. (Intuitively, F_p gives the value that p computes for the element of the interactive consistency vector corresponding to q on the basis of σ_p.) We say that $\{F_p \mid p \in P\}$ assure interactive consistency for m faults if for each choice of $N \subseteq P$, $|N| \geq n - m$, and each scenario σ consistent with N,

(i) For all $p,q \in N, F_p(\sigma_p,q) = \sigma(q)$

(ii) For all $p,q \in N$, $r \in p$, $F_p(\sigma_p,r) = F_q(\sigma_q,r)$

where σ_p and σ_q denote the restrictions of σ to strings beginning with p and q, respectively.

Intuitively, clause (i) requires that each nonfaulty processor p correctly compute the private value of each nonfaulty processor q, and clause (ii) requires that each two nonfaulty processors compute exactly the same vector.

Theorem. If $|V| \geq 2$ and $n \leq 3m$, there exists no $\{F_p \mid p \in P\}$ that assures interactive consistency for m faults.

Proof. Suppose, to the contrary, that $\{F_p \mid p \in P\}$ assure interactive consistency for m faults. Since $n \leq 3m$, P can be partitioned into three nonempty sets A, B, and C, each of which has no more than m members. Let v and v' be two distinct values in V. Our

general plan is to construct three scenarios α, β and σ such that α is consistent with $N = A \cup C$, β with $N = B \cup C$ and α with $N = A \cup B$. The members of C will all be given private value v in α and v' in β. Moreover, α, β, and σ will be constructed in such a way that no processor $a \in A$ can distinguish α from σ (i.e., $\alpha_a = \sigma_a$) and no processor $b \in B$ can distinguish β from σ (i.e., $\beta_b = \sigma_b$). It will then follow that for the scenario σ processors in A and B will compute different values for the members of C.

We define the scenarios α, β, and σ recursively as follows:

(i) For each $w \in P^+$ not ending in a member of C, let

$$\alpha(w) = \beta(w) = \sigma(w) = v.$$

(ii) For each $a \in A$, $b \in B$, $c \in C$ let

$$\alpha(c) = \alpha(ac) = \alpha(bc) = \alpha(cc) = v$$
$$\beta(c) = \beta(ac) = \beta(bc) = \beta(cc) = v'$$
$$\sigma(c) = \sigma(ac) = \sigma(bc) = \sigma(cc) = v$$

(iii) For each $a \in A$, $b \in B$, $c \in C$, $p \in P$, $w \in P^*c$ (i.e., w is any string over P ending in c), let

$$\alpha(paw) = \alpha(aw)$$
$$\alpha(pbw) = \beta(bw)$$
$$\alpha(pcw) = \alpha(cw)$$

$$\beta(paw) = \alpha(aw)$$
$$\beta(pbw) = \beta(bw)$$
$$\beta(pcw) = \beta(cw)$$

$$\sigma(paw) = \sigma(aw)$$
$$\sigma(pbw) = \sigma(bw)$$
$$\sigma(acw) = \alpha(cw)$$
$$\sigma(bcw) = \beta(cw)$$

It is easy to verify by inspection that α, β, and σ are in fact consistent with $N = A \cup C$, $B \cup C$, $A \cup B$, respectively. Moreover, one can show by a simple induction proof on the length of w that:

$$\alpha(aw) = \sigma(aw), \ \beta(bw) = \sigma(bw)$$

for all $a \in A$, $b \in B$, and $w \in P^*$.

It then follows from the definition of interactive consistency that for any $a \in A$, $b \in B$, $c \in C$,

$v = \alpha(c) = F_a(\alpha_a, c) = F_a(\sigma_a, c) = F_b(\sigma_b, c) = F_b(\beta_b, c) = v'$ giving a contradiction. Q.E.D.

5. An Algorithm Using Authentications

The negative result of the last section depends strongly on the assumption that a faulty processor may refuse to pass on values it has received from other processors or may pass on fabricated values. This section addresses the situation in which the latter possibility is precluded. We will assume, in other words, that a faulty processor may "lie" about its own value, and may refuse to relay values it has received, but may not relay altered values without betraying itself as faulty.

In practice, this assumption can be satisfied to an arbitrarily high degree of probability using authenticators. An authenticator is a redundant augment to a data item that can be created, ideally, only by the originator of the data. A processor p constructs an authenticator for a data item d by calculating $A_p[d]$, where A_p is some mapping known only to p. It must be highly improbable that a processor q other than p can generate the authenticator $A_p[d]$ for a given d. At the same time, it must be easy for q to check, for a given p, v, and d, that $v = A_p[d]$. The problem of devising mappings with these properties is a cryptographic one. Methods for their constructions are discussed in [2] and [3]. For many applications in which faults are due to random errors rather than to malicious intelligence, any mappings that "suitably randomize" the data suffice.

A scenario σ is carried out in the following way. As before, let $v = \sigma(p)$ designate p's private value. p communicates this value to r by sending r the message consisting of the triple $\langle p, a, v \rangle$, where $a = A_p[v]$. When r receives the message, it checks that $a = A_p[v]$. If so, r takes v as the value of $\sigma(rp)$. Otherwise r lets $\sigma(rp) = NIL$. More generally, if r receives exactly one message of the form $(p_1, a_1(p_2, a_2 \ldots (p_k, a_k, v)\ldots))$, where $a_k = A_k[v]$ and for $1 \leq i \leq k-1$, $a_i = A_i[(p_{i+1}, a_{i+1} \ldots (p_k, a_k, v)]$, then $\sigma(rp_1 \ldots p_k) = v$. Otherwise, $\sigma(rp_1 \ldots p_k) = NIL$.

A scenario σ constructed in this way is consistent with a given choice N of faulty processors, if for all processors $p \in N$, $q \in P$ and strings w, w' over P,

(i) $\sigma(qpw) = \sigma(pw)$

(ii) $\sigma(w'pw)$ is either $\sigma(pw)$ or NIL

Condition (i) insures that nonfaulty processors are always truthful. Condition (ii) guarantees that a processor cannot relay an altered value of information received from a nonfaulty processor.

We now present a procedure, using m+1-level authenticated scenarios, that guarantees interactive consistency for any $n \geq m$. As before, the procedure is described in terms of the value a nonfaulty processor p records for a given processor q on the basis of σ_p:

> Let S_{pq} be the set of all non-NIL values $\sigma_p(pwq)$, where w ranges over strings of distinct elements with length $\leq m$ over $P - \{p,q\}$. If S_{pq} has exactly one element v, p records v for q; otherwise, p records NIL.

To see that interactive consistency is assured consider first the case in which q is nonfaulty. In this case $\sigma_p(pwq)$ is either $\sigma(q)$ or NIL for each appropriate w by condition (ii). Since, in particular, $\sigma_p(pq) = \sigma(q)$ by (i), $S_{pq} = \{\sigma(q)\}$. p thus records $\sigma(q)$ for q as required.

If q is faulty, it suffices to show only that for each two nonfaulty processors p and p', $S_{pq} = S_{p'q}$. So suppose $v \in S_{pq}$, i.e., $v = \sigma_p(pwq)$ for some string w having no repetitions, with length $\leq m$ over $P - \{p,q\}$. If p' occurs in w, (say $w = w_1 p' w_2$) then $\sigma(pwq) = \sigma(p' w_2 q)$ by (ii), hence $v = \sigma(pwq) \in S_{p'q}$. If p' does not occur in w and w is of length $< m$, then pw is of length $\leq m$, so $v = \sigma(pwq) = \sigma(p'pwq) \in S_{p'q}$. Finally, if p' does not occur in w and w is of length m, w must be of the form $w_1 r w_2$ where r is nonfaulty, giving that $v = \sigma(pwq) = \sigma(rw_2q)$ (by (ii)) = $\sigma(p'rw_2q)$ (by (i)) $\in S_{p'q}$. In each case $v \in S_{p'q}$. A symmetrical argument shows that if $v \in S_{p'q}$, $v \in S_{pq}$. Hence $S_{p'q} = S_{pq}$ as required. Q.E.D.

6. Conclusions

The problem of obtaining interactive consistency appears to be quite fundamental to the design of fault-tolerant systems in which executive control is distributed. In the SIFT [4] fault-tolerant computer under development at SRI, the need for an interactive consistency algorithm arises in at least three aspects of the design--synchronization of clocks, stabilization of input from sensors, and agreement upon results of diagnostic tests. In the preliminary stages of the design of this system, it was naively assumed that simple majority voting schemes could be devised to treat these situations. The gradual realization that simple majorities are insufficient led to the results reported here.

These results by no means answer all the questions one might pose about interactive consistency. The algorithms presented here are intended to demonstrate existence. The construction of efficient algorithms and algorithms that work under the assumption of restricted communications is a topic for future research. Other questions that will be considered include those of reaching approximate agreement and reaching agreement under various probabilistic assumptions.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the substantial contribution of ideas to this paper by K. N. Levitt, P. M. Melliar-Smith, and J. H. Wensley.

REFERENCES

1. Davies, D., and Wakerly, J., "Synchronization and Matching in Redundant Systems," IEEE Transactions on Computers, C-27, 6, pp. 531-539 (June 1978).

2. Diffie, W., and Hellman, M., "New Directions in Cryptography," IEEE Trans. Information Theory, IT-22, 6, pp. 644-654 (Nov. 1976).

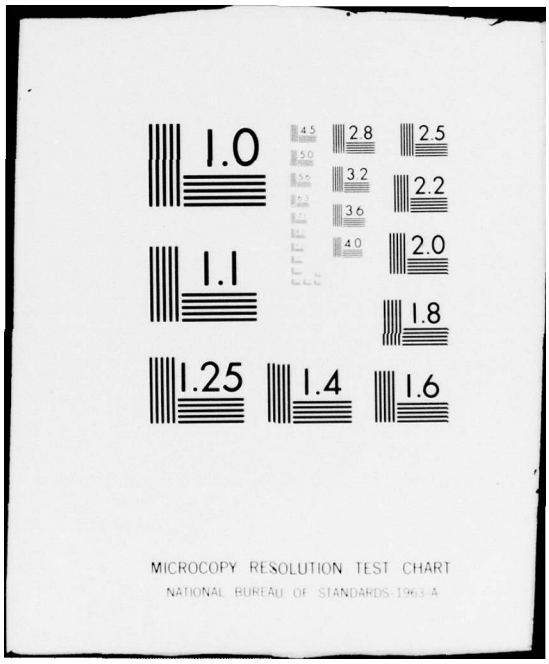
3. Rivest, R. L., Shamir, A., and Adleman, L. A., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Comm. ACM, 21, pp. 120-126 (Feb. 1978).

4. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proceedings of the IEEE, to appear.

AD-A061 919 SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/G 9/2
THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVIN--ETC(U)
AUG 78 B ELSPAS, R E SHOSTAK F44620-73-C-0068
UNCLASSIFIED AFOSR-TR-78-1491 NL

2 OF 2
AD A061919

END
DATE FILMED
2 79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Appendix C

MACHINE PROOFS OF THE SYNCHRONIZATION ALGORITHM

Appendix C

MACHINE PROOFS OF THE SYNCHRONIZATION ALGORITHM

This appendix contains a history file showing the definitions used with the Boyer-Moore theorem prover and the lemmas proved in connection with the verification of some aspects of the synchronization algorithm discussed in Section IV of the report.

```
(FILECREATED " 6-Sep-78 20:21:51" <SHOSTAK>NEWHIST..2 16587


     changes to:  NEWHIST)


(PRETTYCOMPRINT NEWHISTCOMS)

(RPAQQ NEWHISTCOMS (NEWHIST))

(RPAQQ NEWHIST ((DCL PV (P)
                    NIL)
        (DCL LIE (STRING)
             NIL)
        (DEFN SETP (X)
              (IF (PLISTP X)
                  (IF (NLISTP X)
                      T
                      (AND (NOT (MEMBER (CAR X)
                                        (CDR X)))
                           (SETP (CDR X))))
                  F)
              NIL)
        (DEFN CEILING.QUOTIENT (X Y)
              (IF (EQUAL X (TIMES Y (QUOTIENT X Y)))
                  (QUOTIENT X Y)
                  (ADD1 (QUOTIENT X Y)))
              NIL)
        (DEFN DISTRIB1 (Y Z)
              (IF (NLISTP Y)
                  "NIL"
                  (CONS (CONS Z (CAR Y))
                        (DISTRIB1 (CDR Y)
                                  Z)))
              NIL)
        (DEFN DISTRIB (X Y)
              (IF (NLISTP X)
                  "NIL"
                  (APPEND (DISTRIB1 Y (CAR X))
                          (DISTRIB (CDR X)
                                   Y)))
              NIL)
```

```
(DEFN K.PERMS.OVER.S (K S)
      (IF (NUMBERP K)
          (IF (EQUAL K 0)
              (CONS "NIL" "NIL")
              (DISTRIB S (K.PERMS.OVER.S (SUB1 K)
                                         S)))
          "NIL")
      NIL)
(DEFN SCEN1 (STRING LIARS)
      (IF (NLISTP STRING)
          "NIL"
          (IF (NLISTP (CDR STRING))
              (PV (CAR STRING))
              (IF (MEMBER (CADR STRING)
                          LIARS)
                  (LIE STRING)
                  (SCEN1 (CDR STRING)
                         LIARS))))
      NIL)
(DEFN SCEN (STRING SUFFIX LIARS)
      (SCEN1 (APPEND STRING SUFFIX)
             LIARS)
      NIL)
(DEFN TEST.STRING (STRING P Q VALUE SUFFIX LIARS)
      (EQUAL VALUE (SCEN (CONS P (APPEND STRING (CONS Q "NIL")))
                         SUFFIX LIARS))
      NIL)
(DEFN TEST.STRINGS (STRINGS P Q VALUE SUFFIX LIARS)
      (IF (NLISTP STRINGS)
          T
          (AND (TEST.STRING (CAR STRINGS)
                            P Q VALUE SUFFIX LIARS)
               (TEST.STRINGS (CDR STRINGS)
                             P Q VALUE SUFFIX LIARS)))
      NIL)
(DEFN TEST.SET (SET P Q VALUE LENGTH SUFFIX LIARS)
      (TEST.STRINGS (K.PERMS.OVER.S LENGTH SET)
                    P Q VALUE SUFFIX LIARS)
      NIL)
(DEFN HAS.K.INSTANCES (K VALUE BAG)
      (IF (NUMBERP K)
          (IF (EQUAL K 0)
              T
              (IF (NLISTP BAG)
                  F
                  (IF (EQUAL (CAR BAG)
                             VALUE)
                      (HAS.K.INSTANCES (SUB1 K)
                                       VALUE
                                       (CDR BAG))
                      (HAS.K.INSTANCES K VALUE (CDR BAG)))))
          F)
      NIL)
```

```
(DEFN VOTE (NUMBER BALLOTLIST)
      (IF (NLISTP BALLOTLIST)
          "NIL"
          (IF (HAS.K.INSTANCES (SUB1 NUMBER)
                               (CAR BALLOTLIST)
                               (CDR BALLOTLIST))
              (CAR BALLOTLIST)
              (VOTE NUMBER (CDR BALLOTLIST))))
      NIL)
(DEFN
  K.COMBS.OVER.S
  (K S)
  (IF (NUMBERP K)
      (IF (EQUAL K 0)
          (CONS "NIL" "NIL")
          (IF (NLISTP S)
              "NIL"
              (APPEND (DISTRIB1 (K.COMBS.OVER.S (SUB1 K)
                                                (CDR S))
                                (CAR S))
                      (K.COMBS.OVER.S K (CDR S)))))
      "NIL")
  NIL)
(DEFN STRIP (ALIST)
      (IF (NLISTP ALIST)
          "NIL"
          (CONS (CDR (CAR ALIST))
                (STRIP (CDR ALIST))))
      NIL)
(DEFN DELETE (X Y)
      (IF (NLISTP Y)
          Y
          (IF (EQUAL X (CAR Y))
              (DELETE X (CDR Y))
              (CONS (CAR Y)
                    (DELETE X (CDR Y)))))
      NIL)
(DEFN LAST.ELT (STRING)
      (IF (NLISTP STRING)
          "NIL"
          (IF (NLISTP (CDR STRING))
              (CAR STRING)
              (LAST.ELT (CDR STRING))))
      NIL)
(DEFN HAS.NO.LIARS (STRING LIARS)
      (IF (NLISTP STRING)
          T
          (IF (MEMBER (CAR STRING)
                      LIARS)
              F
              (HAS.NO.LIARS (CDR STRING)
                            LIARS)))
      NIL)
```

```
(PROVE.LEMMA NO.LIARS (REWRITE)
             (IMPLIES (AND (LISTP STRING)
                           (HAS.NO.LIARS STRING LIARS))
                      (EQUAL (SCEN1 STRING LIARS)
                             (PV (LAST.ELT STRING))))
             NIL NIL)
(DEFN NON.FAULTIES (PROCS LIARS)
      (IF (NLISTP PROCS)
          "NIL"
          (IF (MEMBER (CAR PROCS)
                      LIARS)
              (NON.FAULTIES (CDR PROCS)
                            LIARS)
              (CONS (CAR PROCS)
                    (NON.FAULTIES (CDR PROCS)
                                  LIARS))))
      NIL)
(PROVE.LEMMA LASTELT.APPEND (REWRITE)
             (IMPLIES (AND (PLISTP X)
                           (PLISTP Y)
                           (LISTP X))
                      (EQUAL (LAST.ELT (APPEND Y X))
                             (LAST.ELT X)))
             NIL NIL)
(PROVE.LEMMA NO.LIARS.CONS (REWRITE)
             (IMPLIES (AND (LISTP STRING)
                           (HAS.NO.LIARS STRING LIARS))
                      (EQUAL (SCEN1 (CONS X STRING)
                                    LIARS)
                             (PV (LAST.ELT STRING))))
             NIL NIL)
(PROVE.LEMMA MEMBER.CAR (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (MEMBER (CAR X)
                                   LIARS))
                      (NOT (HAS.NO.LIARS X LIARS)))
             NIL NIL)
(PROVE.LEMMA APPEND.CAR (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (MEMBER (CAR (APPEND X
                                                (CONS Q "NIL")))
                                   LIARS))
                      (NOT (HAS.NO.LIARS X LIARS)))
             NIL NIL)
(PROVE.LEMMA MEM.APPEND (REWRITE)
             (EQUAL (CAR (APPEND X Y))
                    (IF (LISTP X)
                        (CAR X)
                        (CAR Y)))
             NIL NIL)
```

```
(PROVE.LEMMA TEST.STRING.NO.LIARS (REWRITE)
             (IMPLIES (AND (PLISTP STRING)
                           (LISTP STRING)
                           (HAS.NO.LIARS STRING LIARS)
                           (NOT (MEMBER Q LIARS)))
                      (EQUAL (SCEN1 (CONS P
                                          (APPEND STRING
                                                  (CONS Q "NIL")
                                                  ))
                                    LIARS)
                             (PV Q)))
             NIL NIL)
(DEFN MEMBER.STRINGS.HAVE.NO.LIARS (STRINGS LIARS)
      (IF (NLISTP STRINGS)
          T
          (IF (HAS.NO.LIARS (CAR STRINGS)
                            LIARS)
              (MEMBER.STRINGS.HAVE.NO.LIARS (CDR STRINGS)
                                            LIARS)
              F))
      NIL)
(PROVE.LEMMA DISTRIB.NO.LIARS (REWRITE)
             (IMPLIES (AND (MEMBER.STRINGS.HAVE.NO.LIARS Y
                                                         LIARS)
                           (HAS.NO.LIARS S LIARS))
                      (MEMBER.STRINGS.HAVE.NO.LIARS
                        (DISTRIB S Y)
                        LIARS))
             NIL NIL)
(PROVE.LEMMA K.PERMS.NO.LIARS (REWRITE)
             (IMPLIES (HAS.NO.LIARS S LIARS)
                      (MEMBER.STRINGS.HAVE.NO.LIARS
                        (K.PERMS.OVER.S K S)
                        LIARS))
             NIL NIL)
(PROVE.LEMMA STRONGER.TEST.STRING.NO.LIARS (REWRITE)
             (IMPLIES (AND (PLISTP STRING)
                           (HAS.NO.LIARS STRING LIARS)
                           (NOT (MEMBER Q LIARS)))
                      (EQUAL (SCEN1 (CONS P
                                          (APPEND STRING
                                                  (CONS Q "NIL")
                                                  ))
                                    LIARS)
                             (PV Q)))
             NIL NIL)
(DEFN LIST.OF.PLISTS (STRINGS)
      (IF (PLISTP STRINGS)
          (IF (NLISTP STRINGS)
              T
              (IF (PLISTP (CAR STRINGS))
                  (LIST.OF.PLISTS (CDR STRINGS))
                  F))
          F)
      NIL)
```

```
(PROVE.LEMMA K.PERMS.IS.LIST.OF.PLISTS (REWRITE)
             (IMPLIES (PLISTP S)
                      (LIST.OF.PLISTS (K.PERMS.OVER.S K S)))
             NIL NIL)
(PROVE.LEMMA TEST.STRINGS.NO.LIARS (REWRITE)
             (IMPLIES (AND (MEMBER.STRINGS.HAVE.NO.LIARS
                                                    STRINGS
                                                      LIARS)
                           (NOT (MEMBER Q LIARS))
                           (LIST.OF.PLISTS STRINGS))
                      (TEST.STRINGS STRINGS P Q (PV Q)
                                    "NIL" LIARS))
             NIL NIL)
(PROVE.LEMMA TEST.SET.NO.LIARS (REWRITE)
             (IMPLIES (AND (PLISTP S)
                           (HAS.NO.LIARS S LIARS)
                           (NOT (MEMBER Q LIARS)))
                      (TEST.STRINGS (K.PERMS.OVER.S K S)
                                    P Q (PV Q)
                                    "NIL" LIARS))
             NIL NIL)
(PROVE.LEMMA K.COMBS.IS.LIST.OF.PLISTS (REWRITE)
             (IMPLIES (PLISTP S)
                      (LIST.OF.PLISTS (K.COMBS.OVER.S K S)))
             NIL NIL)
(PROVE.LEMMA MEM.DISTRIB1 (REWRITE)
             (IMPLIES (MEMBER X Y)
                      (MEMBER (CONS A X)
                              (DISTRIB1 Y A)))
             NIL NIL)
(PROVE.LEMMA SUB1.LENGTH (REWRITE)
             (IMPLIES (AND (PLISTP X)
                           (LISTP X))
                      (EQUAL (SUB1 (LENGTH X))
                             (LENGTH (CDR X))))
             NIL NIL)
(DEFN ORDERED.SUBSETP (X Y)
      (IF (NLISTP X)
          T
          (IF (NLISTP Y)
              F
              (IF (ORDERED.SUBSETP X (CDR Y))
                  T
                  (IF (EQUAL (CAR X)
                             (CAR Y))
                      (ORDERED.SUBSETP (CDR X)
                                       (CDR Y))
                      F))))
      NIL)
(PROVE.LEMMA ORDERED.SUBSETP.CDR (REWRITE)
             (IMPLIES (AND (PLISTP X)
                           (PLISTP Y)
                           (ORDERED.SUBSETP X Y))
                      (ORDERED.SUBSETP (CDR X)
                                       Y))
             NIL NIL)
```

```
(PROVE.LEMMA ORDERED.SUBSET.K.COMBS (REWRITE)
             (IMPLIES (AND (PLISTP X)
                           (PLISTP Y)
                           (ORDERED.SUBSETP X Y))
                      (MEMBER X (K.COMBS.OVER.S (LENGTH X)
                                                Y)))
             NIL NIL)
(DEFN NON.FAULTY.PROCS (PROCS LIARS)
      (IF (NLISTP PROCS)
          "NIL"
          (IF (MEMBER (CAR PROCS)
                      LIARS)
              (NON.FAULTY.PROCS (CDR PROCS)
                                LIARS)
              (CONS (CAR PROCS)
                    (NON.FAULTY.PROCS (CDR PROCS)
                                      LIARS))))
      NIL)
(PROVE.LEMMA NON.FAULTIES.HAVE.NO.LIARS (REWRITE)
             (HAS.NO.LIARS (NON.FAULTY.PROCS PROCS LIARS)
                           LIARS)
             NIL NIL)
(PROVE.LEMMA PLISTP.NON.FAULTY.PROCS (REWRITE)
             (PLISTP (NON.FAULTY.PROCS PROCS LIARS))
             NIL NIL)
(PROVE.LEMMA ORDERED.SUBSET.NON.FAULTY.PROCS (REWRITE)
             (ORDERED.SUBSETP (NON.FAULTY.PROCS PROCS LIARS)
                              PROCS)
             NIL NIL)
(DEFN TEST.SETS (SETS P Q VALUE LENGTH SUFFIX LIARS)
      (IF (NLISTP SETS)
          F
          (OR (TEST.SET (CAR SETS)
                        P Q VALUE LENGTH SUFFIX LIARS)
              (TEST.SETS (CDR SETS)
                         P Q VALUE LENGTH SUFFIX LIARS)))
      NIL)
```

```
(DEFN
  IC.VECTOR
  (P PROCS PROCSCDRS N M SUFFIX LIARS)
  (IF
    (NLISTP PROCSCDRS)
    "NIL"
    (IF
      (ZEROP N)
      "NIL"
      (CONS
        (CONS
          (CAR PROCSCDRS)
          (IF (TEST.SETS (K.COMBS.OVER.S
                           (ADD1 (QUOTIENT (PLUS N M)
                                           2))
                           PROCS)
                         P
                         (CAR PROCSCDRS)
                         (SCEN (CONS P (CONS (CAR PROCSCDRS)
                                             "NIL"))
                               SUFFIX LIARS)
                         M SUFFIX LIARS)
              (SCEN (CONS P (CONS (CAR PROCSCDRS)
                                  "NIL"))
                    SUFFIX LIARS)
              (VOTE (CEILING.QUOTIENT (PLUS N M)
                                      2)
                    (STRIP (IC.VECTOR P (DELETE (CAR PROCSCDRS)
                                                PROCS)
                                      (DELETE (CAR PROCSCDRS)
                                              PROCS)
                                      (SUB1 N)
                                      (SUB1 M)
                                      (CONS (CAR PROCSCDRS)
                                            SUFFIX)
                                      LIARS)))))
        (IC.VECTOR P PROCS (CDR PROCSCDRS)
                   N M SUFFIX LIARS))))
  NIL)
(PROVE.LEMMA MEMBER.TEST.SETS (REWRITE)
             (IMPLIES (AND (MEMBER X Y)
                           (TEST.SET X P Q VALUE LENGTH SUFFIX
                                     LIARS))
                      (TEST.SETS Y P Q VALUE LENGTH SUFFIX
                                 LIARS))
             NIL NIL)
(PROVE.LEMMA
  MEM.NON.FAULTY.K.COMBS
  (REWRITE)
  (IMPLIES (PLISTP PROCS)
           (MEMBER (NON.FAULTY.PROCS PROCS LIARS)
                   (K.COMBS.OVER.S
                     (LENGTH (NON.FAULTY.PROCS PROCS LIARS))
                     PROCS)))
  NIL NIL)
```

```
(PROVE.LEMMA TEST.STRINGS.NON.FAULTIES (REWRITE)
             (IMPLIES (AND (PLISTP PROCS)
                           (NOT (MEMBER Q LIARS)))
                      (TEST.STRINGS
                        (K.PERMS.OVER.S
                          LENGTH
                          (NON.FAULTY.PROCS PROCS LIARS))
                        P Q (PV Q)
                        "NIL" LIARS))
             NIL NIL)
(PROVE.LEMMA MEMBER.TEST.SETS.INSTANCE (REWRITE)
             (IMPLIES (AND (MEMBER (NON.FAULTY.PROCS PROCS
                                                     LIARS)
                                   Y)
                           (TEST.SET (NON.FAULTY.PROCS PROCS
                                                       LIARS)
                                     P Q VALUE LENGTH SUFFIX
                                     LIARS))
                      (TEST.SETS Y P Q VALUE LENGTH SUFFIX
                                 LIARS))
             NIL NIL)
(PROVE.LEMMA
  MEMBER.TEST.SETS.INSTANCE.INSTANCE
  (REWRITE)
  (IMPLIES (AND (MEMBER (NON.FAULTY.PROCS PROCS LIARS)
                        (K.COMBS.OVER.S
                          (LENGTH (NON.FAULTY.PROCS PROCS LIARS)
                                  )
                          PROCS))
                (TEST.SET (NON.FAULTY.PROCS PROCS LIARS)
                          P Q VALUE LENGTH SUFFIX LIARS))
           (TEST.SETS (K.COMBS.OVER.S
                        (LENGTH (NON.FAULTY.PROCS PROCS LIARS))
                        PROCS)
                      P Q VALUE LENGTH SUFFIX LIARS))
  NIL NIL)
(PROVE.LEMMA
  TEST.SETS.NON.FAULTIES
  (REWRITE)
  (IMPLIES (AND (PLISTP PROCS)
                (NOT (MEMBER Q LIARS)))
           (TEST.SETS (K.COMBS.OVER.S
                        (LENGTH (NON.FAULTY.PROCS PROCS LIARS))
                        PROCS)
                      P Q (PV Q)
                      K "NIL" LIARS))
  NIL NIL)
(PROVE.LEMMA NEW.MEMBER.TEST.SETS.INSTANCE (REWRITE)
             (IMPLIES
               (AND (MEMBER X (K.COMBS.OVER.S (LENGTH X)
                                              Y))
                    (TEST.SET X P Q (PV Q)
                              LENGTH "NIL" LIARS))
               (TEST.SETS (K.COMBS.OVER.S (LENGTH X)
                                          Y)
                          P Q (PV Q)
                          LENGTH "NIL" LIARS))
             NIL NIL)
```

```
(PROVE.LEMMA TEST.SETS.LEMMA (REWRITE)
             (IMPLIES (AND (PLISTP X)
                           (PLISTP PROCS)
                           (ORDERED.SUBSETP X PROCS)
                           (HAS.NO.LIARS X LIARS)
                           (NOT (MEMBER Q LIARS)))
                      (TEST.SETS (K.COMBS.OVER.S (LENGTH X)
                                                 PROCS)
                                 P Q (PV Q)
                                 LENGTH "NIL" LIARS))
             NIL NIL)
(DEFN TC (X Y Z)
      (IF (NLISTP X)
          (IF (NLISTP Y)
              T
              (IF (NLISTP Z)
                  F
                  (IF (EQUAL (CAR Y)
                             (CAR Z))
                      (TC X (CDR Y)
                          (CDR Z))
                      (TC X Y (CDR Z)))))
          (IF (NLISTP Y)
              F
              (IF (NLISTP Z)
                  F
                  (IF (EQUAL (CAR X)
                             (CAR Y))
                      (IF (EQUAL (CAR Y)
                                 (CAR Z))
                          (TC (CDR X)
                              (CDR Y)
                              (CDR Z))
                          (TC X Y (CDR Z)))
                      (IF (EQUAL (CAR Y)
                                 (CAR Z))
                          (TC X (CDR Y)
                              (CDR Z))
                          (TC X Y (CDR Z)))))))
      NIL)
```

```
(DEFN UC (X Y Z)
      (IF (NLISTP Y)
          (IF (NLISTP X)
              T F)
          (IF (NLISTP Z)
              F
              (IF (EQUAL (CAR Y)
                         (CAR Z))
                  (IF (NLISTP X)
                      (UC X (CDR Y)
                          (CDR Z))
                      (IF (EQUAL (CAR X)
                                 (CAR Y))
                          (UC (CDR X)
                              (CDR Y)
                              (CDR Z))
                          (UC X (CDR Y)
                              (CDR Z))))
                  (UC X Y (CDR Z))))))
      NIL)
(DEFN REST (X Y)
      (IF (NLISTP X)
          Y
          (REST (CDR X)
                (CDR Y)))
      NIL)
(DEFN N.SUBSET.OF.S (N S)
      (IF (ZEROP N)
          "NIL"
          (IF (NLISTP S)
              "NIL"
              (CONS (CAR S)
                    (N.SUBSET.OF.S (SUB1 N)
                                   (CDR S))))))
      NIL)
(PROVE.LEMMA PLISTP.N.SUBSET.OF.S (REWRITE)
             (PLISTP (N.SUBSET.OF.S N S))
             NIL NIL)
(PROVE.LEMMA ORDERED.SUBSETP.N.SUBSET.OF.S (REWRITE)
             (IMPLIES (AND (NUMBERP N)
                           (LESSEQP N (LENGTH S))
                           (PLISTP S))
                      (ORDERED.SUBSETP (N.SUBSET.OF.S N S)
                                       S))
             NIL NIL)
(PROVE.LEMMA N.SUBSET.APPEND (REWRITE)
             (IMPLIES (PLISTP X)
                      (EQUAL (APPEND
                               (N.SUBSET.OF.S N X)
                               (REST (N.SUBSET.OF.S N X)
                                     X))
                             X))
             NIL NIL)
```

```
(PROVE.LEMMA OS.APPEND (REWRITE)
             (IMPLIES (AND (EQUAL Z Z)
                           (ORDERED.SUBSETP (APPEND X Y)
                                            Z))
                      (ORDERED.SUBSETP X Z))
             NIL NIL)
(PROVE.LEMMA
  OS.APPEND.INSTANCE
  (REWRITE)
  (IMPLIES
    (AND (NOT (LESSP (LENGTH (NON.FAULTY.PROCS PROCS LIARS))
                     N))
         (PLISTP X)
         (ORDERED.SUBSETP
           (APPEND X
                   (REST (N.SUBSET.OF.S
                           N
                           (NON.FAULTY.PROCS PROCS LIARS))
                         (NON.FAULTY.PROCS PROCS LIARS)))
           Z))
    (ORDERED.SUBSETP X Z))
  NIL NIL)
(PROVE.LEMMA
  OS.NSUBSET
  (REWRITE)
  (IMPLIES (AND (NUMBERP N)
                (PLISTP PROCS)
                (LESSEQP N (LENGTH (NON.FAULTY.PROCS PROCS
                                                     LIARS))))
           (ORDERED.SUBSETP (N.SUBSET.OF.S
                              N
                              (NON.FAULTY.PROCS PROCS LIARS))
                            PROCS))
  NIL NIL)
(PROVE.LEMMA LENGTH.N.SUBSET (REWRITE)
             (IMPLIES (AND (NUMBERP N)
                           (NOT (LESSP (LENGTH S)
                                       N)))
                      (EQUAL (LENGTH (N.SUBSET.OF.S N S))
                             N))
             NIL NIL)
```

```
(PROVE.LEMMA HAS.NO.LIARS.N.SUBSET (REWRITE)
             (IMPLIES (PLISTP PROCS)
                      (HAS.NO.LIARS
                        (N.SUBSET.OF.S N
                                       (NON.FAULTY.PROCS PROCS
                                                         LIARS))
                        LIARS))
             NIL NIL)
(PROVE.LEMMA NEW.TEST.SETS.LEMMA (REWRITE)
             (IMPLIES (AND (EQUAL N (LENGTH X))
                           (PLISTP PROCS)
                           (ORDERED.SUBSETP X PROCS)
                           (HAS.NO.LIARS X LIARS)
                           (NOT (MEMBER Q LIARS))
                           (PLISTP X))
                      (TEST.SETS (K.COMBS.OVER.S N PROCS)
                                 P Q (PV Q)
                                 K "NIL" LIARS))
             NIL NIL)
(PROVE.LEMMA
  TEST.SETS.N
  (REWRITE)
  (IMPLIES (AND (EQUAL N
                       (LENGTH (N.SUBSET.OF.S
                                 N
                                 (NON.FAULTY.PROCS PROCS LIARS))
                               ))
                (NUMBERP N)
                (NOT (LESSP (LENGTH (NON.FAULTY.PROCS PROCS
                                                      LIARS))
                            N))
                (PLISTP PROCS)
                (NOT (MEMBER Q LIARS)))
           (TEST.SETS (K.COMBS.OVER.S N PROCS)
                      P Q (PV Q)
                      LENGTH "NIL" LIARS))
  NIL NIL)
(PROVE.LEMMA
  STRONG.TEST.SETS.N
  (REWRITE)
  (IMPLIES (AND (NUMBERP N)
                (NOT (LESSP (LENGTH (NON.FAULTY.PROCS PROCS
                                                      LIARS))
                            N))
                (PLISTP PROCS)
                (NOT (MEMBER Q LIARS)))
           (TEST.SETS (K.COMBS.OVER.S N PROCS)
                      P Q (PV Q)
                      LENGTH "NIL" LIARS))
  NIL NIL)
(PROVE.LEMMA QUO.TIMES (REWRITE)
             (IMPLIES (NUMBERP N)
                      (NOT (LESSP N (TIMES 2 (QUOTIENT N 2)))))
             NIL NIL)))
```

Appendix D

PROJECT ACTIVITIES

Appendix D

PROJECT ACTIVITIES

A number of outside activities were undertaken on project. They consisted of attendance at conferences or workshops related to this project (in many cases presentations were made at these gatherings), and publication of papers closely related to the project. The essential features of these activities are summarized below.

1. Conferences Attended and Papers Presented

Fifth International Joint Conference on Artificial Intelligence - 1977 (IJCAI-77), Massachusetts Institute of Technology, Cambridge, Mass., 22-25 August 1977. Robert E. Shostak presented his paper, "An Algorithm for Reasoning About Equality," covering portions of his research on the decision algorithm for Presburger arithmetic carried out under this AFOSR contract. His attendance at IJCAI-77 was also supported in part by this contract.

AFOSR/ARO/ONR Conference on Research Directions in Software Technology, Providence, RI, 10-12 October 1977. Bernard Elspas attended this conference and submitted a "Discussant Contribution" at the invitation of Prof. Jack Dennis, an Associate Chairman. The "Discussant Contribution" concerned one of the formal conference papers, "Program Verification," by Ralph London. This contribution is to be published in the book, Research Directions in Software Technology, now being prepared under the general editorship of P. Wegner with Tri-Service support.

Third International Conference on Software Engineering, Atlanta, GA, 10-12 May, 1978. Bernard Elspas attended this conference for the purpose of acting as "coordinator" (discussion leader) for an evening session on "Prospects for Program Verification." At this well-attended session a lively discussion focused on these questions: (1) whether formal correctness proof is practical, (2) when such techniques might be

expected to come into wider use outside of academic and laboratory environments, and (3) which basic problems currently inhibit this technology transfer.

2. Papers Published

Robert E. Shostak, "An Algorithm for Reasoning About Equality," Proc. IJCAI-77 Conference, Vol. 1, pp. 526-527 (August 1977). Scheduled for publication in J.ACM.

Robert S. Boyer and J Strother Moore, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory," Proc. IJCAI-77 Conference, Vol. 1, pp. 511-519 (August 1977). Support for the preparation of this paper was shared among the present AFOSR contract, ONR Contract N00014-75-C-0816, and NSF Grant DCR72-0373A01.

Robert E. Shostak, "Deciding Linear Inequalities by Computing Loop Residues," submitted for publication to C.ACM (10 March 1978).

Bernard Elspas, "Program Verification," a discussion of the paper by Ralph L. London of the same name, to appear in Research Directions in Software Technology, P. Wegner (ed.), MIT Press, Cambridge, Massachusetts (forthcoming).

DISTRIBUTION LIST

Rome/Air Development Center
RADC/IS
Griffiss AFB, NY 13441
[one copy]

Department of Electrical Engineering
New York University
University Heights
Bronx, N.Y. 10453
Professor Herbert Freeman
[one copy]

Department of Industrial
and Operations Engineering
University of Michigan
Ann Arbor, MI 48104
Professor Alan G. Merten
[one copy]

Air Force Avionics Laboratory
AFAL/AA
Wright-Patterson AFB, OH 45433
[one copy]

Information Systems Sciences
McDonnell Douglas Astronautics Company
Huntington Beach, CA 92647
Dr. Paul B. Moranda
[one copy]

Electrical Engineering & Computer
Sciences
University of California, Berkeley
Berkeley, CA 94720
Professor Herbert B. Baskin
[one copy]

Department of the Army
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, N.C. 27709
Mr. Paul Boggs
[two copies]

RADC/ISIS
Griffis AFB, N.Y. 13441
Dr. Northrup Fowler
[two copies]

Program Director
Information Systems
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Mr. Marvin Denicoff Code 437
[two copies]

Directorate of Mathematical and
Information Sciences
Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332
George W. McKemie, Lt. Col. USAF
[sixteen copies]

SFH

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR- 78-1491
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

THE SEMIAUTOMATIC GENERATION OF INDUCTIVE ASSERTIONS FOR PROVING PROGRAM CORRECTNESS

5. TYPE OF REPORT & PERIOD COVERED: Final

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Bernard Elspas and Robert E. Shostak

8. CONTRACT OR GRANT NUMBER(s)

AFOSR
F44620-73-C-0068

9. PERFORMING ORGANIZATION NAME AND ADDRESS

SRI International
333 Ravenswood Avenue
Menlo Park, CAlifornia 94025

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

61102F 2304/A2

11. CONTROLLING OFFICE NAME AND ADDRESS

Air Force Office of Scientific Research/NM
Bolling AFB, Washington, DC 20332

12. REPORT DATE: August 1978

13. NUMBER OF PAGES: 113

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

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)

18. SUPPLEMENTARY NOTES

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

Software verification
Inductive assertions
Verification condition generation
Decision procedures for Presburger arithmetic
Synchronization algorithms

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

This final report describes progress over the period 1 July 1977 through 30 June 1978 on a five-year project aimed at the problem of synthesizing so-called inductive assertions to support the Floyd-Hoare method for proving correctness of computer programs. → next page

During the first few years of the project, the emphasis was on more or less direct approaches toward alleviating this problem. Initially, we

DD FORM 1 JAN 73 1473

20. Abstract continued.

concentrated on building and using a mechanical solver for finite difference equations to synthesize inductive assertions. This approach had limited success. Unfortunately, neither this approach nor the allied approaches pursued simultaneously by Katz, Manna, Wegbreit, and German have proved to be adequate in any general sense. Therefore, during the period 1975-77 we explored alternatives that gave promise of at least alleviating the problem or of bypassing it entirely. These alternatives encompassed the transformation of programs into primitive recursive form prior to verification, the method of generator induction for proof of properties of complex data structures, the use of a hierarchical design methodology (HDM) to structure programs so as to minimize the need for loop-cutting assertions, and the methods allied to subgoal induction and computational induction. The general tenor of these alternative schemes is that to facilitate program verification considerable care must be taken in properly structuring both the programs to be proved and their specifications. An ideal situation would be one in which all the specifications are written in a formal language that can be processed by a powerful theorem prover. For the Boyer-Moore theorem prover recursive function theory is such a language.

In the fifth year of our research, reported here, we concentrated on using the Boyer-Moore system to prove several quite different kinds of programs. The first set of programs verified here form a system of LISP functions implementing a verification condition generator (VCG) for a simple block-structured language. The specifications for this VCG are given as standard Hoare axioms and rules. The second set of programs are algorithms for achieving synchronization among several clocks. This application arose in connection with the design of an operating system for a fault-tolerant avionics computer (SIFT) with hardware and software redundancy features.

A separate problem addressed during the fifth year is the application of directed graph theory to the design of an efficient algorithm for deciding inequalities for sets of integer variables. This work represents a further extension of a series of efficient decision algorithms for Presburger arithmetic (under various restrictions). Most of these algorithms have been implemented (in LISP) as part of experimental program verifiers built at SRI during the past few years.