70-18,026

KING, James Cornelius, 1940-
A PROGRAM VERIFIER.

Carnegie-Mellon University, Ph.D., 1970
Information Services, data processing

University Microfilms, Inc., Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A Program Verifier

by

James Cornelius King

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

September 1969

Submitted to the Carnegie-Mellon University
In partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; distribution of this document is unlimited.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Form 1-470 2-65-3m

CARNEGIE-MELLON UNIVERSITY

CARNEGIE INSTITUTE OF TECHNOLOGY

COLLEGE OF ENGINEERING AND SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE A Program Verifier

PRESENTED BY James C. King

ACCEPTED BY THE DEPARTMENT OF Computer Science

Robert W Floyd — MAJOR PROFESSOR — Sept 21 1969 — DATE

Alan J. Perlis — DEPARTMENT HEAD — Sept 25 1969 — DATE

APPROVED BY THE COMMITTEE ON GRADUATE DEGREES

[signature] — CHAIRMAN — Nov 5, 1970 — DATE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

This research is a first step toward developing a "verifying compiler". Such a compiler, as well as doing the standard translation of a program to machine executable form, attempts to prove that the program is "correct". In order to do this a program must be annotated with propositions in a mathematical notation which define the "correct" relations among the program variables. The verifying compiler then checks for consistency between the program and its propositions.

The thesis presents the theoretical basis of the method and then describes a prototype verifier in detail. This verifier, running on an IBM 360, operates on programs written in a simple programming language for integer arithmetic. Many programs have been automatically verified by this program. These include a simple sort program, a program which examines a number for the property 'prime', and a rather subtle program which raises an integer to an integral power. The formal analysis of a program produces "verification conditions" which must be proven to be theorems over integers. The verifier proves these theorems by using powerful formula simplification routines and specialized techniques for integer expressions. Ideas for improving this verifier and for building one which will operate on a more complicated programming language are also presented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I would like to acknowledge Professor Robert W. Floyd for being my advisor and for developing the theory on which this work is based. While at CMU I learned a great deal from Professors A. Perlis and A. Newell. Dr. O. W. Rechard has been a valued teacher, advisor, and friend throughout my college career.

Fellow graduate students at CMU provided a stimulating environment and I found discussions with Rudy Krutar, Zohar Manna, and Richard Waldinger particularly helpful in this research. Mr. Krutar also carefully proof-read this written report.

The two ladies in my life, my wife Sandra and IBM 360 - 2067 - 67009, cooperated in typing this document. Sandra has also been a source of encouragement and understanding throughout the work. (This I will not say of the other lady).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

ABSTRACT i

ACKNOWLEDGMENTS ii

INTRODUCTION 1

CHAPTER I: FORMAL BASIS 5
A Program and Its Execution 5
Correctness 9
Verifying a Path 12
Verifying a Program 17
Variations 25

CHAPTER II: A PROTOTYPE VERIFIER 36
The Programming Language 36
System Overview 39
Formula Manipulation 41
Normalized Arithmetic Expressions 44
Normalized Logical Expressions 50
Input and Its Internal Representation 68
Verification Condition Generator 74
Theorem Prover 82
Linear Solver 89
Quantifiers 108

CHAPTER III: SYSTEM PERFORMANCE, LIMITATIONS, AND EXTENSIONS 112
General Discussion 112
Simplification 115
Theorem Prover 120
Assertion Language 127
Human Assistance 130
Debugging 131
Arrays 132
Language Extensions 140

CHAPTER IV: CODING METHOD 151
Internal Formula Representation 151
Use of Macros 158
Debugging Aids 162

CHAPTER V: CONCLUSION 164

BIBLIOGRAPHY 167

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX I: BACKUS-NAUR FORM DEFINITION FOR SIMPLE INTEGER PROGRAMMING LANGUAGE 170

APPENDIX II: EXAMPLES OF PROGRAMS VERIFIED 173

APPENDIX III: FURTHER EXAMPLES..... 249

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

INTRODUCTION

This research has been concerned with proving that computer programs will always execute correctly. Our primary interest is in doing so automatically by a computer itself. The motivation for this project has existed since shortly after people began to write programs for computers. At that time von Neumann and Goldstine presented ideas similar to those used here [34]. Computers and the programs which make them operate are playing a more and more important role in our everyday lives. Some means for assuring that these programs will always run properly is critically needed.

Proofs, in general, may take many different forms. They range from the extreme of "I just know it is so" to the careful step by step application of formal rules in a proof system. A person will first construct a proof to convince himself, beyond some doubt, that his hypothesis is correct. He may then wish to put his convictions to the real test and try to convince others. Proofs about programs must have one additional important property. One is concerned not only with the structure of algorithms but with debugging them as programs as well. For years, numerical analysts have constructed mathematical proofs for the correctness and termination of algorithms. But to be executed correctly on a computer the specific coding of such an algorithm must be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

flawless (at least one would like it to be). This serves as a strong motivation for developing an automated or at least semi-automated verifier.

In [12], Floyd presents a rigorous formal technique for proving the correctness of programs which provides the necessary basis for an automated system. Based on the results in this early paper he developed the notion of a "verifying compiler" which was presented in a brief article in [13]. If one starts with a standard compiler and incorporates into it the ability to prove the correctness of the programs which its compiles (adds to it a "program verifier") the result is a verifying compiler. This is a natural extension to a compiler. Many compilers do admirably well in syntactic error detection and a verifying compiler would complete the job by producing a semantic check as well. A first attempt to build a program verifier is presented here.

In a broad sense, this work may be viewed as an engineering project. The basic formal (mathematical) theory was established in Floyd's first paper and concurrent with our work the theorectical aspects have been extended and amplified by Manna [21,22,23,24,25] and Cooper [5]. Our work has been concerned primarily with the implementation of these theories. As with any first-attempt engineering project, our goals were to explore the practical feasibility of the formal ideas, to give insight into the uses of such a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system, to discover problems inherent in implementation, and to provide a basis for further work.

One can dream of routinely using a verifying compiler as an everyday tool. In the context of this idea our work has been extremely modest and must be considered as a small first step. We only hope that, indeed, this has been a first step of a progression which will allow this dream to come to fruition.

We begin, in Chapter I, by defining a program and its execution. This is done in an elementary yet formal way to serve as a vehicle for then presenting the theory of the correctness of programs. The particular verifier we built is explained at length in Chapter II. This system represents one strategy for employing the formal concepts of Chapter I. The weaknesses of the current system and suggestions for its improvement are considered in Chapter III. In this Chapter we also indicate how the basic ideas may be extended to work for more complex programming languages. Chapter IV is included for the programming oriented reader, giving a closer look at how formulas are stored in the computer memory and giving more details of the coding methods used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NOTATION

This document was produced on the computer itself which did impose a few restrictions on the notation used. Double quotation marks are used to delimit words being defined. They are also used in the customary fashion to mark words used in an unusual context. Single quotes are used for emphasis as one might normally use an underline or italics. We make common use of the symbols ↑, ⊃, ∨, ∧, ~, ≡, ∀, and ∃ which are to be read: to the power, implies, or, and, not, equivalent to, for all, and there exists, respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER I: FORMAL BASIS

A Program and Its Execution

In this chapter we show how to prove that a program will always execute correctly. In order to discuss programs and their executions we turn to an abstract model of a computation. The model provides for the sequential execution of assignment operations which each compute an individual result as a function of the current memory contents (the set of variables). These results are stored in the memory (a variable is assigned a new value) and are available for later computations. The exact sequence of assignments executed can be dynamically controlled by test statements. The model is intended to grossly characterize the way most current digital computers perform their computations.

The correctness of a program is determined by associating predicates, over the program variables, with various points in the program. In particular, a characteristic predicate for the computation (called the 'final predicate' later) is associated with the 'halt' of the program. This predicate is 'true' for correct output and 'false' otherwise. We show how to construct a formal proof that the results of any execution of the program will satisfy this predicate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A "program" is a finite ordered set of "statements" s1,s2,...,sm. A program operates on a fixed set of variables x1,x2,...,xn which may be assigned values from the domains D1,D2,...,Dn, respectively. Each statement of a program can take one of the forms:

1) assign:{xk,f(x1,x2,...,xn),j} or

2) test:{p(x1,x2,...,xn),j1,j2} or

3) halt.

For each assign statement f is some total function which computes a value in domain Dk when supplied with values of the variables x1,x2,...,xn. Each of j, j1, and j2 are fixed integers between 1 and m and, as such, are the names of statements. The expression p(x1,x2,...,xn) is a predicate which results in 'true' or 'false' when evaluated for fixed values of x1,x2,...,xn.

A "state vector" of a program is an (n+1)-tuple of values <N,a1,a2,...,an> where N is an integer $1 \leq N \leq m$ (the program counter) and for all i ($1 \leq i \leq n$) ai is a member of Di. An "execution" of a program P={s1,s2,...,sm} begins with an "initial state vector" v1=<1,a1,a2,...,an> and develops an "execution sequence" of state vectors v1,v2,v3,... . The initial vector v1 is given. The sequence is determined by describing how the (j+1)-th vector is derived from the j-th for any $j \geq 1$. Let vj=<i,b1,b2,...,bn> be an arbitrary vector

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the sequence. Then

1) if si has the form assign:{xk,f(x1,x2,...,xn),N}, the next vector in the sequence is vj+1 = <N,b1,...,f(b1,b2,...,bn),...,bn> where the expression f(b1,b2,...,bn) is a value from Dk and occurs at the (k+1)-th position in the state vector. This new state vector reflects the fact that xk has been assigned a new value and also indicates which statement is to be "executed" next (the N-th one).

2) if si has the form test:{p(x1,x2,...xn),j1,j2}, the next vector depends on the value of p(b1,b2,...,bn). If p(b1,b2,...,bn) = 'true' then vj+1 = <j1,b1,b2,...,bn>, otherwise p(b1,b2,...,bn) = 'false' and vj+1 = <j2,b1,b2,...,bn>. In this case, we choose between two statements to be executed next (j1 or j2) depending on the "current" value of the predicate p.

3) if si is 'halt' there is no next vector and the sequence is finite. The vector vj is denoted as the "final vector" for this execution of the program.

Thus an execution of a program begins with an initial state vector and generates a sequence of state vectors by executing individual statements in the program. If the sequence is finite the last vector is called the final vector and the program is said to "terminate" for this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

initial vector, otherwise it is "non-terminating".

For example, let x1, x2, x3, and x4 be variables over the domain of all signed integers. Define the program P' = {

```
s1:  assign:{x4,x2,2},
s2:  assign:{x3,0,3},
s3:    test:{x4=0,4,5},
s4:    halt,
s5:  assign:{x3,x1+x3,6},
s6:  assign:{x4,x4-1,3}  }
```

The initial state vector <1,3,2,40,0> forms the execution sequence { <1,3,2,40,0>, <2,3,2,40,2>, <3,3,2,0,2>, <5,3,2,0,2>, <6,3,2,3,2>, <3,3,2,3,1>, <5,3,2,3,1>, <6,3,2,6,1>, <3,3,2,6,0>, <4,3,2,6,0> }. The sequence is finite and the final vector is <4,3,2,6,0> giving x1=3, x2=2, x3=6, and x4=0. Notice that in this case the final values satisfy x3=x1*x2. It will be proved later that all final vectors satisfy this equation. On the other hand, the initial state vector <1,3,-1,40,0> forms the infinite execution sequence { <1,3,-1,40,0>, <2,3,-1,40,-1>, <3,3,-1,0,-1>, <5,3,-1,0,-1>, <6,3,-1,3,-1>, <3,3,-1,3,-2>, <5,3,-1,3,-2> <6,3,-1,6,-2>, <3,3,-1,6,-3>,... }. The program is non-terminating for <1,3,-1,40,0>.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Correctness

The "correctness" of a program P={s1,s2,...,sm} is defined with respect to two predicates over the variables of P: the "initial predicate", say I(x1,x2,...,xn), and the "final predicate", say F(x1,x2,...,xn). Assume v1 is an initial state vector for P. P is then "correct for v1" with respect to I and F if either:

1) P is non-terminating for v1, or

2) P terminates for v1, generating the execution sequence {v1,v2,...,vk}, and I(v1) ⊃ F(vk). Note, if v1 = <1,a1,a2,...,an> then I(v1) is used as a convenient notation for I(a1,a2,...,an).

If the initial values of a program execution satisfy the condition I and the execution terminates then the final values must satisfy F. Program P is simply "correct" with respect to I and F if it is correct for v1 with respect to I and F for all possible initial vectors v1.

Note the following three points:

1) The program { assign:{x1,x1,1} } is non-terminating for any initial vector and is, therefore, vacuously 'correct' with respect to 'any' predicates I and F. Although Manna [24] refers to this as "partial correctness" and reserves "correct" for programs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that also terminate for any initial vector v for which I(v) is 'true', we continue to use "correct" as defined above.

2) Any program P is trivially correct with respect to any initial predicate I(v) and the final predicate 'true'.

3) If a) F is identically 'false', b) I(v)='true' for some initial vector v, and c) P is correct for v with respect to this I and F, then P is non-terminating for v. If this were not the case one would have proved I(v) ⊃ 'false', which is impossible since I(v) = 'true'.

This last observation is the basis for results dealing with the termination of programs in [21,23,24].

Given a program P, an initial predicate I, and a final predicate F, the problem is to devise a calculus which would enable one to "verify" P; that is, to construct a rigorous proof that P is correct with respect to I and F. Such a technique is described below and is essentially that developed by Floyd in [12]. Grossly, a proof is a deduction over execution sequences. With each vector in any finite execution sequence, say V = {v1,v2,...,vk}, one associates a predicate, say A1,A2,...,Ak, each over the variables of P such that A1(v1)⊃A2(v2), A2(v2)⊃A3(v3), ..., and Ak-1(vk-1)⊃Ak(vk). If I≡A1 and F≡Ak one may conclude that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$I(v1) \supset F(vk)$ and hence P is correct for v1.

To begin a more complete description of the method, the concept of a control path of a program is useful. A "control path", or simply a "path", is a sequence of statements of the program, say {r1,r2,...} (or, equivalently, a sequence of indices which are interpreted as pointing to statements in the program) with the condition that for any two adjacent statements in the list, ri and ri+1:

1) if ri is of the form assign:{xk,f(x1,x2,...,xn),j} then ri+1 is sj, or

2) if ri is of the form test:{p(x1,x2,...,xn),j,k} then ri+1 may be either sj or sk.

In addition, no ri ($i \geq 1$) may be a 'halt' statement except (possibly) the last member of a finite sequence. Each execution of a program defines a control path if one lists the statements executed in order. On the other hand, there may exist control paths which cannot be associated with any execution. There may be no initial state vector which would cause that sequence of statements to be executed in that order (even though they form a path). For example, in the program { test:{$x1 \geq 0$,2,3}, halt, test:{$x1 \geq 0$,1,2} } there is a path {s1,s3,s1} but no corresponding sequence, since once statement 1 determines that $x1 < 0$, statement 3 could never take the path determined by $x \geq 0$. A sequence of state

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vectors "executes a path" if the sequence of statements executed in forming the vectors is exactly that path. A control path does not necessarily begin with statement 1 nor necessarily end, if at all, with a 'halt' statement.

A program is "loop-free" or has no "loops" if all possible control paths are finite, or equivalently, if no control path has more than one occurrence of the same statement. If a control path exists with statement r occurring twice in it, then an infinite path can be formed by indefinite repetition of the cycle of statements between these two occurrences of r, separating each with an occurrence of r. If a control path is infinite, at least one statement must recur, since there are only a finite number of statements. A loop-free program has only a finite number of distinct control paths. This is true because each path is finite and may have no repetitions within it. A loop-free program may be verified by constructing a proof over each control path which begins with statement 1 and ends with a 'halt' statement. Such proofs are done by application of the following more general procedure.

Verifying a Path

Given: a program P = {s1,s2,...,sN}, an arbitrary finite control path R = {r1,r2,...,rm} of P, and two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

predicates over the variables of the program, say A(x1,x2,...,xn) and B(x1,x2,...,xn). R is "verified" with respect to A and B if for any state vector v1 which, beginning at r1, causes the path R to be executed up to but not including rm, resulting in the vector vm, one can conclude A(v1)⊃B(vm). That is, if A is satisfied by a vector at the beginning of the path and the path is executed then B must be satisfied by the resulting vector.

R can be verified by constructing a sequence of predicates corresponding to the statements in R. Let A1≡A. Then develop, in order, A1,A2,...,Am. Suppose this has been done up to Ai, then Ai+1 can be constructed as follows:

1) If ri is of the form assign:{xk,f(x1,x2,...,xn),j} then Ai+1(x1,x2,...,xn) = ∃xk'[Ai(x1,...,xk',...,xn) ∧ xk=f(x1,...,xk',...,xn)]. Here xk' occurs in the k-th position for Ai and f, and ∃xk' is read "there exists a value in Dk, say xk', such that".

2) If ri is of the form test:{p(x1,x2,...,xn),j1,j2} then
   a) if ri+1 is statement sj1 then Ai+1(x1,x2,...,xn) = [p(x1,x2,...,xn) ∧ Ai(x1,x2,...,xn)], otherwise,
   b) ri+1 is sj2 and then Ai+1(x1,x2,...,xn) = [~p(x1,x2,...,xn) ∧ Ai(x1,x2,...,xn)].

After building the sequence A1,A2,...,Am, we form the expression Am(x1,x2,...,xn) ⊃ B(x1,x2,...,xn) which is called a "verification condition". This name is derived

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the proposition that:

> The path R is verified with respect to A and B if and only if this verification condition is 'true' for all x1, x2, ..., xn in D1, D2, ..., Dn, respectively.

Assume that the verification condition is 'true'. Let v1,v2,...,vm be a sequence of state vectors which execute this path. If A1(v1)='false' R is trivially verified, so assume A1(v1)='true'. From this assumption one can show by induction on i that each Ai is 'true' when evaluated at the corresponding state vector vi ($1 \leq i \leq m$). Then since $Am \supset B$, B(vm) must also be 'true', or $A(v1) \supset B(vm)$ and R is verified. Let vi = <j1,a1,a2,...,an>, vi+1 = <j2,b1,b2,...,bn> and suppose Ai(vi) = 'true'. To show Ai+1(vi+1) = 'true':

1) If ri is assign:{xk,f(x1,x2,...,xn),j} then $Ai+1(vi+1) = \exists xk'[Ai(b1,...,xk',...,bn) \wedge bk=f(b1,...,xk',...,bn)]$. But in this case the execution of ri determines that ai=bi for all $i \neq k$, $1 \leq i \leq n$, and bk = f(a1,...,ak,...,an). By choosing xk'=ak, $Ai+1(vi+1) = [Ai(vi) \wedge (f(vi)=f(vi))] =$ 'true'.

2) If ri is test:{p(x1,x2,...,xn),j1,j2} then $Ai+1(vi+1) = [q(vi+1) \wedge Ai(vi+1)]$ where q(vi+1) = p(vi+1) if ri+1 is sj1 and q(vi+1) = ~p(vi+1) if ri+1 is sj2. But bi=ci for all i, $1 \leq i \leq n$, and, since the vectors

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

were assumed to execute this path, q(vi) = 'true'. So Ai+1(vi+1) = 'true'.

This concludes the proof of the "if" part of the proposition or the proof of the consistency of this verifying method. The "only if" or completeness part follows.

Suppose the verification condition is 'false'. Then there are values d1,d2,...,dn such that Am(d1,d2,...,dn) = 'true' and B(d1,d2,...,dn) = 'false'. Starting with vm = <j,d1,d2,...,dn> (j is determined by rm=sj) and Am(vm) = 'true' one can construct, in reverse order, a set of state vectors v1,v2,...,vm which executes R and for which Ai(vi) = 'true' for $1 \leq i \leq m$. In particular, A(v1) = 'true', exhibiting a sequence of vectors for which R cannot be verified. Assume vi+1, ..., vm has been constructed to execute {ri+1,...,rm}. Let vi+1 = <k2,b1,b2,...,bn> and assume Ai+1(vi+1) = 'true'. To find a vi = <k1,a1,a2,...,an> such that Ai(vi) = 'true' and such that vi,vi+1,...,vm executes {ri,ri+1,...,rm}:

1) If ri is assign:{xk,f(x1,x2,...,xn),j} then Ai+1(vi+1) = $\exists$xk'[Ai(b1,...,xk',...,bn) $\wedge$ bk=f(b1,...,xk',..,bn)] = 'true'. Let ak be a value for xk' which makes Ai+1(vi+1) 'true', and let ai=bi for all $i \neq k$, $1 \leq i \leq n$. Then [Ai(vi) $\wedge$ bk=f(vi)] = 'true' so Ai(vi) is 'true' and vi+1 is derived from vi by execution of ri.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2) If ri is test:(p(x1,x2,...,xn),j1,j2) then let ai=bi for all i, 1≤i≤n. If k2=j1, Ai+1(vi+1) = [p(vi+1) ∧ Ai(vi+1)] = [p(vi) ∧ Ai(vi)] = 'true' so p(vi) = Ai(vi) = 'true' and vi+1 is derived from vi by executing ri. If k2=j2 then Ai+1(vi+1) = [~p(vi+1) ∧ Ai(vi+1)] = [~p(vi) ∧ Ai(vi)] = 'true' so ~p(vi) = Ai(vi) = 'true' and vi+1 is derived from vi by executing ri.

This completes the proof of the proposition.

A statement on a control path defines a transformation of a state vector v1 into a new state vector v2. Also, for statements, rules have just been presented for transforming a predicate A1 into a new predicate A2 such that A1(v1)⊃A2(v2). That is, if one knows A1 about v1 before the statement is executed one can deduce that A2 is known about the resulting v2. The completeness part of the proof also assures us that A2 represents as much as one could learn about v2 from only knowing A1(v1) and understanding what the execution of the statement means. It is in this sense that one may view these transformation rules as giving a "semantic definition" for the execution of statements in the programming language.

Note that once a path is determined the transformation on the predicates is done strictly by examining the program statements. This is important since the results of a single proof may have implications about an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

infinite class of execution sequences, in the same way that a finite program represents (possibly) an infinite number of different computations.

Verifying a Program

This method for verifying a finite path of a program may be applied directly to constructing a proof of correctness for any loop-free program. The finite number of finite paths beginning at statement 1 and ending with a 'halt' statement can each be processed, associating I (the program's initial predicate) with the beginnings of the paths and F (the program's final predicate) with the ends. If all the verification conditions are true the program is correct with respect to I and F. If one or more of the verification conditions is not true there is an execution sequence for which the program is not correct.

Can this same method be applied to an arbitrary program which is not necessarily loop-free? Such a program may have an infinite number of control paths. Directly applying the method used for loop-free programs will not work, but the problem is overcome by introducing "inductive predicates". These predicates are over the variables of the program and each is associated with a distinct statement in the program, "tagging" that statement. The initial

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

predicate, I, tags statement 1 and the final predicate, F, tags all 'halt' statements. The statements of the program must be tagged in such a way that there is no control path which has two occurrences of the same untagged statement that are not separated by at least one tagged statement. This is always possible since the the property would trivially hold if one tagged every statement in the program. These tagged statements essentially "cut" all loops in the program.

Consider all "tagged paths" in the program which take the form R = {r1,r2,...,rk} where statement r1 is tagged by predicate A, rk is tagged by predicate B, and r2,...,rk-1 are untagged statements. Using the method developed earlier, one verifies R with respect to A and B, and does this for all tagged paths in the program. There are only a finite number of these since the tagging was done so that each of r2,..., and rk-1 must be distinct. Each verification produces a verification condition to be proved. If they are all proved 'true' (i.e., the verifications were successful) then the program is correct with respect to the given I and F. This is easily seen by looking at any finite execution sequence {v1,v2,...,vm} and its associated control path R= {r1,r2,...,rm}. If I(v1) = 'false' the program is correct for v1. Suppose I(v1) = 'true'. Let ri be the next tagged statement (rm is 'halt' and is tagged by F). Name that tag A. Since all control paths starting and ending with tagged statements (i.e., all tagged paths) have been

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

verified, I(v1) ⊃ A(vi) and consequently A(vi) = 'true'. The next tagged path of R has A as its initial tag and we apply the same argument to this path, etc., eventually concluding that F(vm) = 'true'. But then I(v1) ⊃ F(vm), and the program is correct for v1.

The method may fail when not all verification conditions are true. This does not necessarily mean that the program is not correct with respect to I and F. If the inductive predicates are not chosen properly a correct program may not be verifiable. The strongest statement that can be made, in this respect, is that if the program is correct there exist inductive predicates which will yield a proof by this technique. (For a proof of this see Theorem 1 of [5].) Suppose that, by some other technique, a given program has been shown to compute correct results for all executions which begin with state vectors satisfying the initial predicate I. Let V = {v1, v2,...,vk} be a finite execution sequence such that I(v1) = 'true'. Then the proper final predicate F, for a proof by our method, would be that one which gave F(vk) = 'true' for all such vk and 'false' for all other values. This comment also applies to the inductive predicates. The proper choice should give 'true' only for the appropriate states of any execution sequence whose first vector satisfies I.

Recall the program P' used as an example before: P' = {

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

s1: assign:{x4,x2,2},

s2: assign:{x3,0,3},

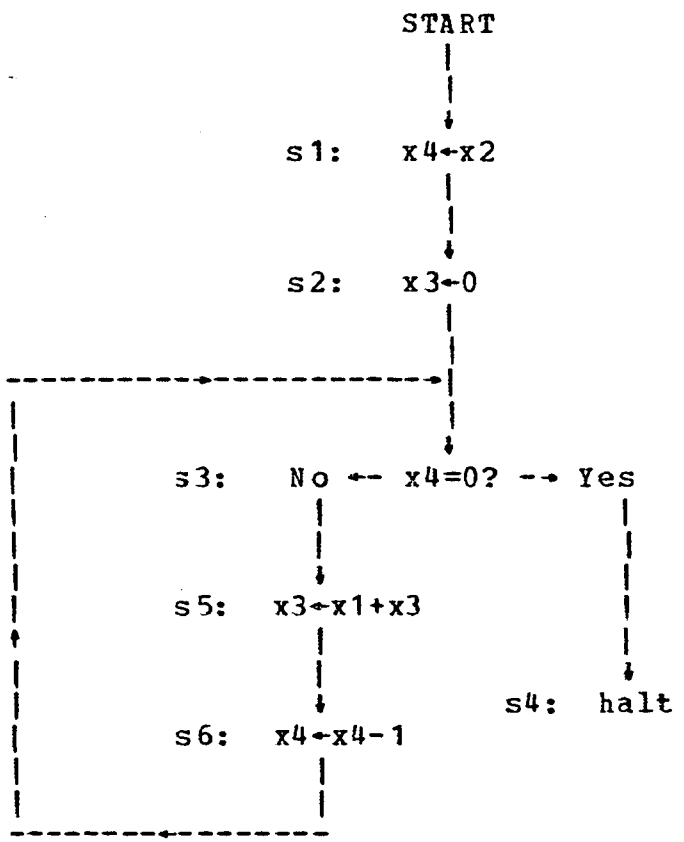
s3: test:{x4=0,4,5},

s4: halt,

s5: assign:{x3,x1+x3,6},

s6: assign:{x4,x4-1,3} }

A flowchart corresponding to P′ is:



Example control paths of P′ are:

cp1 = {s3,s5}, cp2 = {s2,s3,s4},

cp3 = {s1,s2,s3,s5,s6,s3}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Path cp3 shows that P′ is not loop-free since s3 occurs twice. An infinite number of paths as well as an infinite path can be created from the cycle occurring in cp3: {s3,s5,s6,s3}.

P′ is correct with respect to I(x1,x2,x3,x4)≡ 'true' and F(x1,x2,x3,x4,)≡ (x3=x1*x2). To show this, associate the inductive predicate A(x1,x2,x3,x4)≡ (x3+x4*x1=x1*x2) with statement s3. Then the tagged paths to be verified are:

tp1 = {s1,s2,s3}, tp2 = {s3,s4}, tp3 = {s3,s5,s6,s3} .

To verify tp1, begin with I and using statement s1 form the new predicate ∃x4′['true' ∧ (x4=x2)] or more simply (x4=x2). Using this and s2 get ∃x3′[(x4=x2) ∧ x3=0] or [(x4=x2) ∧ (x3=0)]. Next form the verification condition:

vc1 = { [(x4=x2) ∧ (x3=0)] ⊃ (x3+x4*x1=x1*x2) }.

To verify tp2, begin with A and using statement s3 form { (x3+x4*x1=x1*x2) ∧ (x4=0) }. The second verification condition is then:

vc2 = { [(x3+x4*x1=x1*x2) ∧ (x4=0) } ⊃ (x3=x1*x2) }.

To verify tp3, begin again with A and using statement s3 form [(x3+x4*x1=x1*x2) ∧ (x4≠0) }. Using this and s5 get ∃x3′{ (x3′+x4*x1=x1*2) ∧ (x4≠0) ∧ (x3=x1+x3′) }. Then from s6 get ∃x4′∃x3′{ (x3′+x4′*x1=x1*x2) ∧ (x4′≠0) ∧ (x3=x1+x3′) ∧ (x4=x4′-1) }. The final verification condition is:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$vc3 = \{ \exists x4' \exists x3' \{ (x3'+x4'*x1=x1*x2) \wedge (x4' \neq 0) \wedge (x3=x1+x3') \wedge (x4=x4'-1) \} \supset (x3+x4*x1=x1*x2) \}.$$

All three verification conditions can be proved to be 'true' and so P is correct with respect to I and F.

Note that if an equation of the form xk = f(x1,...,xk',...,xn) can be solved for xk', i.e., rewritten as xk' = g(x1,...,xk,...,xn), then in this case (the equation being a conjunct of the predicate) the expression g(x1,...,xk,...,xn) can be substituted throughout the predicate. This eliminates xk' and its associated existential quantifier. So when verifying tp3 one could have simplified

$$\exists x3' [(x3'+x4*x1=x1*x2) \wedge (x4 \neq 0) \wedge (x3=x1+x3')],$$

using x3' = x3-x1, to get

$$(x3-x1+x4*x1=x1*x2) \wedge (x4 \neq 0).$$

In this way, vc3 can be simplified to:

$$vc3 = \{ [(x3-x1+(x4+1)*x1=x1*x2) \wedge (x4+1 \neq 0)] \supset (x3+x4*x1=x1*x2) \}.$$

The inductive predicate A for P' is an 'invariant' statement about the variables of P' with respect to executions of the cycle {s3, s5, s6, s3,}. That is, verification condition vc3 essentially proves that if A is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

'true' of the values of the variables immediately before executing s3 and if the statements s3,s5, and s6 are then executed, A will again be 'true' for the new values. This is a property of inductive predicates. The following observation also lends some intuitive feeling to the inductive predicates: the predicate A in the above example shows the relationship among what has been computed (x3), the desired result (x1*x2), and what has yet to be computed (x4*x1). (i.e., x3+x4*x1=x1*x2).

In the example, the inductive predicate could have been associated with either statement s5 or s6 as well as s3, as was done. Of course, these would not necessarily be the same predicates in each case. For example, let A'(x1,x2,x3,x4)= (x3+(x4-1)*x1=x1*x2) be the inductive predicate associated with statement s6. Then the tagged paths would be:

tp1' = {s1,s2,s3,s4}, tp2' = {s1,s2,s3,s5,s6},
tp3' = {s6,s3,s5,s6}, tp4' = {s6,s3,s4}.

This time, four verification conditions would be generated and proved.

One may also "over-kill" and use both predicates A and A' at the same time. This would result in the tagged paths:

tp1'' = {s1,s2,s3}, tp2'' = {s3,s4},
tp3'' = {s3,s5,s6}, tp4'' = {s6,s3}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

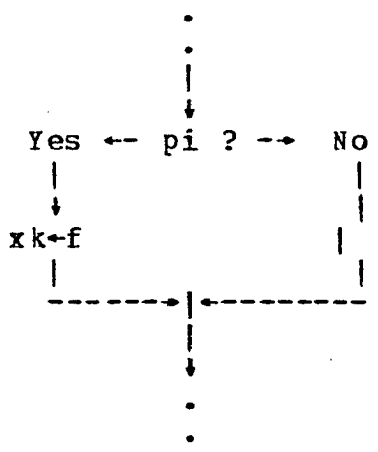
For this simple example, there appears to be no value for introducing more than a minimum number of predicates. However, there are two good reasons one may wish to do so, particularly on larger programs. First, extra predicates may be used to reduce the number of tagged paths and hence the number of verification conditions. To see this, consider the extreme example of the loop-free program having 2m+1 statements:

{ si = test:{pi,i+1,i+2} | for i=1,3,5,7,..,2m-1 },

{ si = assign:{xk,f,i+1} | for i=2,4,6,...,2m },

and s2m+1 = halt.

The program consists of m repetitions of the form:



and, considering only the initial and final predicate tags, has 2↑m distinct tagged paths. If one associates a predicate Ai with each si, i=1,3,5,...,2m-1, the number of tagged paths drops to 2*m.

The second reason one may wish to introduce

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

extraneous predicates is to reduce one large, possibly difficult verification condition to two or more simpler ones. In the example program P', one can associate tp1 with tp1'', tp2 with tp2'', and consider tp3'' and tp4'' as two components of tp3. The original verification of tp3 involved working over {s3,s5,s6,s3} whereas in verifying tp3'', one deals with {s3,s5,s6} and then, in verifying tp4'', continues with {s6,s3}.

Variations

There are variations of the method presented for verifying a path which, while possibly not as intuitive, are quite useful. As before, suppose one has a program P = {s1,s2,...sN}, an arbitrary finite control path of P, R = {r1,r2,...,rm}, and two predicates over the variables of P, say A(x1,x2,...,xn) and B(x1,x2,...,xn). Before, the sequence of predicates A1≡A,A2,...,Am were developed in order, beginning with A. Such a sequence can also be generated by beginning with B and working backward, forming, say B≡Bm,Bm-1,...,B1, in that order. Suppose this has been done for Bm,Bm-1,...,Bi+1. Then construct Bi as follows:

1) When ri is of the form assign:{xk,f(x1,x2,...,xn),j} then Bi(x1,x2,...,xn) = Bi+1(x1,...,f(x1,x2,..., xn),...,xn). Here f occurs in the k-th position.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2) When ri is of the form test:{p(x1,x2,...,xn),j1,j2} then if ri+1 is statement sj1 then Bi(x1,x2,...,xn) = [p(x1,x2,...,xn) ⊃ Bi+1(x1,x2,...,xn)] otherwise ri+1 is sj2 and Bi(x1,x2,...,xn) = [~p(x1,x2,...,xn) ⊃ Bi+1(x1,x2,...,xn)].

The verification condition for this "backward" method is

$$A(x1,x2,\ldots,xn) \supset B1(x1,x2,\ldots,xn).$$

As before, one has the proposition that:

> The path R is verified with respect to A and B if and only if this verification condition is valid.

One can give a proof of this proposition which is analogous to that given for the "forward" method, but more insight is gained by simply observing that both methods generate equivalent verification conditions for any given tagged path of unit length.

Consider the tagged path {r1,r2}, with r1 tagged by A and r2 tagged by B. Suppose r1 has the form:

$$assign:\{xk,f(x1,x2,\ldots,xn),j\}.$$

Then the forward verification condition is

$$vcf = \exists xk'[\, A(x1,\ldots,xk',\ldots,xn) \wedge xk=f(x1,\ldots,xk',\ldots,xn)\,] \supset B(x1,x2,\ldots,xn)$$

and the backward condition is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vcb =

$$A(x1,x2,\ldots,xn) \supset B(x1,\ldots,f(x1,x2,\ldots,xn),\ldots,xn).$$

At first glance, the two expressions appear quite different, especially since vcf has the quantifier $\exists xk'$ and vcb has none. Actually, the quantifier is "ficticious". Upon moving it outside the expression one gets

$$vcf = \forall xk'\{ [A(x1,\ldots,xk',\ldots,xn) \wedge xk=f(x1,\ldots,xk',\ldots,xn)] \supset B(x1,x2,\ldots,xn) \}$$

and in this free variable notation universal quantification is assumed, so the $\forall xk'$ may be dropped:

$$[A(x1,\ldots,xk',\ldots,xn) \wedge xk=f(x1,\ldots,xk',\ldots,xn)] \supset B(x1,x2,\ldots,xn).$$

Next the expression xk=f(x1,...,xk',...,xn) can be used to eliminate xk throughout:

$$vcf = A(x1,x2,\ldots,xk',\ldots,xn) \supset B(x1,\ldots,f(x1,\ldots,xk',\ldots,xn),\ldots,xn).$$

These simplifications result in the same expression as vcb except that the variable xk is renamed xk'. So each equality introduced in going forward corresponds to a substituion done when going backward and the new existential variables are introduced as a means of keeping account of the scope for those delayed substitutions.

For 'test' statements the correspondence is more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

immediate. Suppose r1 has the form:

test:{p(x1,x2,...xn),j,k}.

The verification conditions are therefore:

$$vcf = [\, A(x1,x2,\ldots,xn) \wedge q(x1,x2,\ldots,xn) \,] \supset B(x1,x2,\ldots,xn) \quad \text{and}$$

$$vcb = A(x1,x2,\ldots,xn) \supset [\, q(x1,x2,\ldots,xn) \supset B(x1,x2,\ldots,xn) \,]$$

where q is p or ~p depending on whether r2 is sj or sk, respectively, but the two expressions vcf and vcb are equivalent. This follows from the tautology:

$$[(P \wedge Q) \supset R] \equiv [P \supset (Q \supset R)].$$

Given these observations about a single statement it is left to the reader to convince himself that the verification conditions generated going forward and going backward on an arbitrary tagged path are equivalent.

Both methods can be used together by transforming the first tag, A, forward over the path R={r1,r2,...,rm} to develop Ak for some 1<k<m, then transforming the tag on rm, say B, backward developing Bk, and forming the verification condition

$$Ak(x1,x2,\ldots,xn) \supset Bk(x1,x2,\ldots,xm).$$

One may delay choosing the initial predicate I

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and/or the final predicate F, and after generating all verification conditions in terms of the abstract expressions I(x1,x2,...,xn) and F(x1,x2,...,xn), determine what specific expressions would be appropriate. For the previous example program P', one need supply only the inductive predicate Ä(x1,x2,...,xn)= (x3+x4*x1=x1*x2) associated with s3, getting:

backward:

vc1 = { I(x1,x2,...,xn) ⊃ (0+x2*x1=x1*x2) }

forward:

vc2 = {[(x3+x4*x1=x1*x2) ∧ (x4=0)] ⊃ F(x1,x2,...,xn)}

vc3 = (the same as before)

One must still prove vc3, however, vc1 and vc2 can be "proved" by picking I(x1,x2,...,xn) = 0+x2*x1=x1*x2 or simply 'true', and F(x1,x2,...,xn) = [(x3+x4*x1=x1*x2)∧ (x4=0)] or more simply x3=x1*x2. In this case, these derived predicates just happen to coincide with the choices originally made.

For any loop-free program one may use 'true' for the initial predicate and by conjoining all expressions resulting from going forward on all paths to the 'halt' statements "discover" what the program computes. Alternatively, one may put a final predicate on the program and develop initial conditions which will always assure the results are correct as defined by that predicate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One may also postpone supplying specific inductive predicates by generating the verification conditions in terms of symbolic predicates, say A1(x1,x2,...,xn), A2(x1,x2,...,xn), ..., and Ak(x1,x2,...,xn). Let C1, C2, ..., Cm be the verification conditions so generated, involving the symbolic predicates A1, A2, ..., Ak and the program variables x1, x2, ..., xn. Also let A represent the vector of predicates (A1,A2,...,Ak) and X represent the vector of variables (x1,x2,...,xn). The problem is then to show that one can choose an A such that all the verification conditions are simultaneously true, that is, to prove:

$$\exists A[\ \forall X\ C1(X,A) \wedge \forall X\ C2(X,A) \wedge \ldots \wedge \forall X\ Cm(X,A)\ ].$$

Roughly this formulation, has been explored and developed by both Manna [24] and Cooper [5]. Manna also assumes that the program is completely tagged (i.e., each statement has a symbolic predicate associated with it). One now has the problem of choosing the inductive predicates and proving the verification conditions together in one theorem proving problem. This creates both advantages and disadvantages. One advantage is that every operation in the method, except the proof of this theorem, can be done automatically by following a well defined algorithm. Formally this is a very nice representation for the over-all verification problem. Another possible advantage is that one may avoid actually constructing an A since the theorem only requires that the 'existence' of an A be shown.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

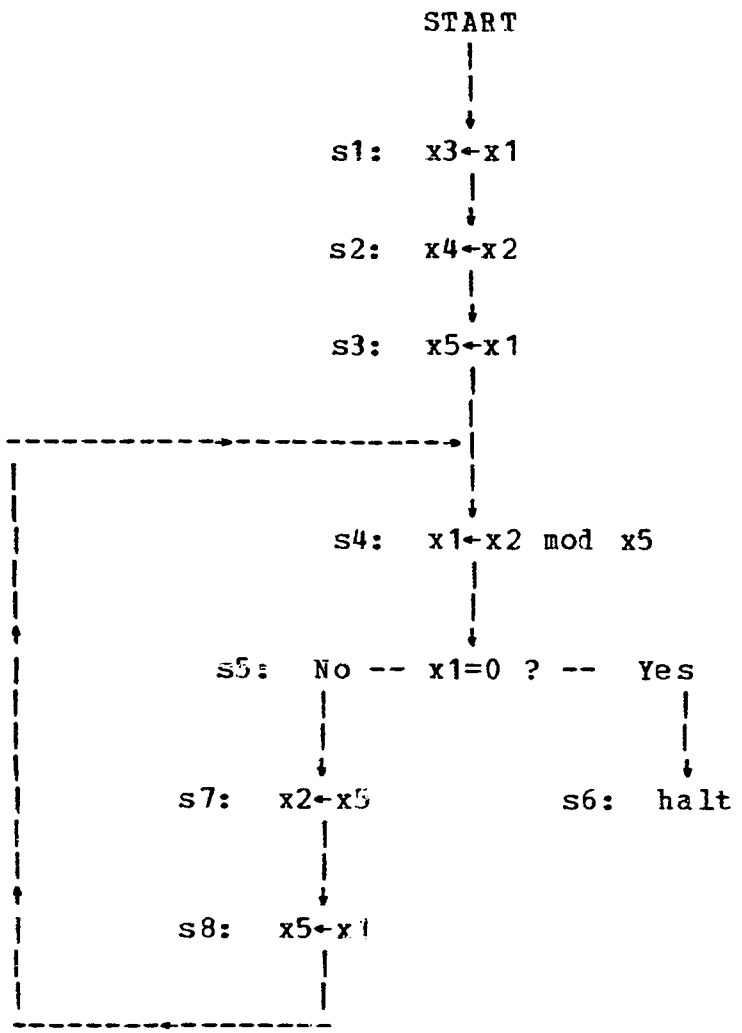
A disadvantage is that most of the intuitive assistance given by the context of the program itself, which may be helpful in determining the proper A, is lost. Another disadvantage when considering an automated verifying compiler is the additional difficulty of the theorems to be proved. Viewing the current state of automatic theorem proving techniques, the proof of the verification conditions, even given the inductive predicates, is an ambitious task. Compounding this task with the problem of choosing the predicates would be ambitious, indeed. Manna [24] has also given an algorithm for producing an expression which captures both correctness and termination together as one theorem proving problem.

One important point should be emphasized with respect to constructing proofs of verification conditions: in proving the correctness of a program, one is dealing with its semantic intent. Even if the syntactic expressions allowed in a given programming language are extremely simple, one may be able to express arbitrary computations in that language, (e.g., Turing machines and the simple language presented here). This arbitrariness carries over to program verification. This notion is most clearly presented by example. Let x1, x2, x3, x4, and x5 be variables over the domain of all non-negative integers, and let (b mod c) represent the remainder when b is divided by c. Let the program GCD = {

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
s1:  assign:{x3,x1,2},
s2:  assign:{x4,x2,3},
s3:  assign:{x5,x1,4},
s4:  assign:{x1,x2 mod x5,5},
s5:    test:{x1=0,6,7},
s6:    halt,
s7:  assign:{x2,x5,8},
s8:  assign:{x5,x1,4}   }.
```

A flowchart corresponding to GCD is :



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As one may have guessed from the name GCD, this program will compute the greatest common divisor of the inputs x1 and x2 and leave the result in x5. It is based on the famous Euclidean Algorithm (see [33], page 26 or any book on elementary number theory). We choose the initial predicate $I(x1,...,x5) = (x1 \geq 1 \wedge x2 \geq 1)$ and the final predicate $F(x1,...,x5) = (x5 = gcd(x3,x4))$, where gcd(a,b) stands for the greatest common divisor of a and b. Introduce the inductive predicate $A(x1,...,x5) = (gcd(x3,x4) = gcd(x1,x5))$ and associate it with statement s5. There are three tagged paths which lead to three verification conditions (all developed by going backward):

$$vc1 = \{ (x1 \geq 1 \wedge x2 \geq 1) \supset [gcd(x1,x2) = gcd(x2 \bmod x1, x1)] \}$$

$$vc2 = \{ [gcd(x3,x4) = gcd(x1,x5)] \supset [x1 \neq 0 \supset gcd(x3,x4) = gcd(x5 \bmod x1, x1)] \}$$

$$vc3 = \{ [gcd(x3,x4) = gcd(x1,x5)] \supset [x1 = 0 \supset x5 = gcd(x3,x4)] \}.$$

These conditions can be proved to be valid if the following properties of the function gcd are given:

1) $gcd(a,0) = a$

2) $gcd(a,b) = gcd(b,a)$

3) $gcd(c,b) = gcd(c,b \bmod c)$

for any integers $a \geq 0$, $b \geq 0$, and $c \geq 1$.

The program itself has a very simple structure and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the only function used in assignment statements is (x2 mod x5). The only test predicate is the simple match for zero (x1=0), yet the verification involves the reasonably sophisticated concept of greatest common divisor. Since the program is designed to compute greatest common divisors the verification must involve that concept. The theorems presented in the literature proving the correctness of the technique would essentially have to be re-proved or drawn on as known results when verifying any particular coding of the method as a program.

The three properties of the function gcd given above could be completed to form an axiomatic definition characterizing the function gcd uniquely. The program itself represents a computational definition of the function gcd. Viewed in this way, the method for verifying programs presented here gives a precise technique for demonstrating the equivalence of these two distinctly different representations of the same function. The axiomatic representation may be concise and lend itself to powerful formal analysis but may give only faint clues as to any algorithmic process to compute the function. The program on the other hand completely describes a dynamic computation but in doing so becomes a rather difficult object to analyze formally. The generation and then the proof of the verification conditions based on the axiomatic definitions clearly exhibits the relationship between these two different representations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Before proceeding to describe the prototype verifier which was built based on these ideas we make one more small but interesting point. The example programs given so far are organized with the input values being preserved in some variables, so that those variables can be used in the final predicate which describes the relationship between the program's input and output. For example, in the GCD program x1 and x2 contain the input values which are immediately copied to x3 and x4 by the first two statements of the program. These values in x3 and x4 remain unchanged so that the final predicate (x5 = gcd(x3,x4)) can assert that the output value, in x5, is the gcd of the inputs originally in x1 and x2.

These superfluous variables and assignments can be eliminated, in this case, by deleting the first two statements of the GCD program and then using a new initial predicate:

$$I'(x1,\ldots,x5) = (x1=x3 \wedge x2=x4 \wedge x1\geq 1 \wedge x2\geq 1).$$

Besides what the original predicate, I, asserted, this new one also claims that initially x1 and x3, and x2 and x4 have the same value. The program itself is free of the variables x3 and x4 which are now used just as names for the initial values in the predicates I', A, and F. By still interpreting x3 and x4 as program variables the proof method is completely valid and this modified program is correct with respect to I' and F.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER II: A PROTOTYPE VERIFIER

In this chapter the automated program verifier which we built is described in detail. This is only a verifier as opposed to a verifying compiler in that no attempt was made to compile programs into an executable form. Compiler techniques are well known and the interfacing of a verifier and a compiler would be rather straightforward. As noted in Chapter I, any practical verifier will require a great semantic breadth in order to validate many verification conditions. Human assistance will no doubt be essential. With this in mind, we have built a system to run totally unassisted. When such systems have been examined thoroughly we can begin to see how much the machine can do automatically and in exactly what way the human can yield assistance. After looking at the amount of detailed effort necessary to prove rigorously a simple program as that given as example 8 in Appendix II, one must agree that the machine has to carry the clerical burden in order for this method to work.

The Programming Language

A user of our verifier must know: the programming language in which algorithms to be verified are coded, the language used to express the predicates (assertions), and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the limitations imposed upon the placement of these predicates. The programming language is simple and allows only integer variables and integer vectors (one dimensional arrays). The syntax and basic form for the language is drawn from Algol and a detailed BNF description is given as Appendix I. Simple integer arithmetic expressions can be computed using the standard operators +, -, *, ÷, ↑, mod, and abs. The functions +, -, *, and abs are well defined over all integers. We define a÷b as [a/b], where [x] is taken to mean "the greatest integer less than or equal to x", and /, of course, represents real division. This makes a÷b undefined for b=0. The operation (a mod b) is the remainder of a when divided by b, or [a mod b ≡ a - b*(a÷b)]. This is also undefined for b=0. We define a↑b as the multiplication of a by itself b times. For b≥1 this is valid. For b=0 we let a↑b=1, and for b≤-1 we consider the result undefined. Attempting to evaluate an undefined function during execution of a program will result in an abnormal termination with an appropriate error message.

These arithmetic expressions are used to build assignment statements which assign a value to a simple variable or to a specific element (which is specified by an arithmetic expression) of an integer vector. The Algol notation is used for this (i.e., A[I+1] ← X + B[I];). The arithmetic expressions are also used with the relational operators >, <, ≥, ≤, =, and ≠ to form Boolean primitives. These in turn can be built into more complicated Boolean

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

valued expressions by use of the standard logical connectives ∧ (and), ∨ (or), ~ (not), ⊃ (implies). These Boolean expressions can be used as test predicates for the Algol-like conditional statement: 'IF <Boolean expression> THEN <statement> ELSE <statement>'.

Any statement may be labelled with an identifier as in Algol and the simple 'GO TO <label>' statement is allowed to indicate transfer of control to such a labeled statement. A sequence of statements can be interpreted as a single unit by placing a 'BEGIN-END' around them forming a compound statement, again as in Algol. No declarations are used and no procedures are allowed.

Programs written in this language have an obvious correspondence to flowcharts drawn using five different component types as described by Floyd [12]. These are the simple assignment statement box, the binary test box, the join of two lines of control, and the start and halt.

The predicate tags are supplied along with the program as a separate statement type -- ASSERT. The syntax of this statement is ASSERT(<super Boolean expression>); where a <super Boolean expression> is the same as a Boolean expression in the programming language except that the quantifiers ∀ and ∃ may be used to bind simple integer variables. For example:

$$Y \geq 0 \quad \vee \quad \exists X (X \geq 0 \quad \wedge \quad X+Y \geq 0) \quad \wedge \quad \forall Z (Z-Y>0 \supset \forall W (W \geq 0)).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The interpretation of ∀Z is "for all integers Z" and ∃X is "there exists an integer X".

The program verifier which proves the correctness of programs written in this language requires that one assertion be placed immediately before the final 'END' of the program (this is the standard final predicate) and that inductive predicates be supplied so that each loop in the flow of control of the program has at least one assertion somewhere on its path. A single assertion may serve for more than one loop as long as it falls within the flow of each. Additional assertions may optionally be placed at any other points in a program. In particular, one may wish to supply an initial predicate which would be inserted immediately after the first BEGIN of the program. If no initial predicate is supplied the verifier will derive one and output it.

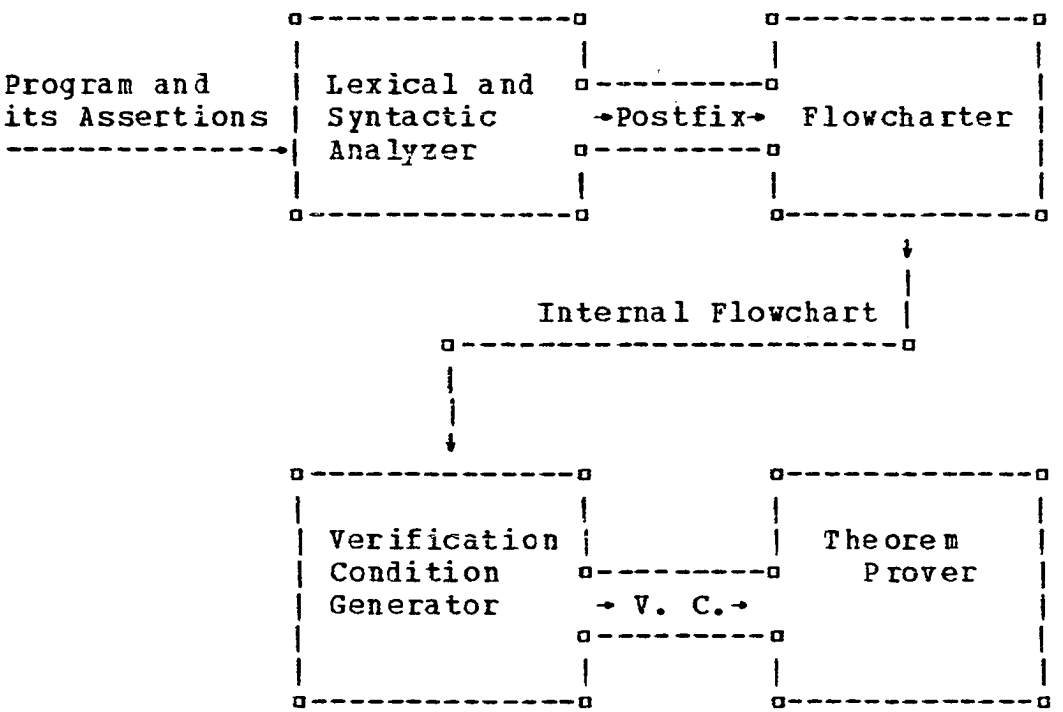
System Overview

The running verifier does automatically verify several interesting programs. Of course, it also handles correctly any number of trivial examples. (See Appendix II.) The most challenging example run successfully to date is example 3 of Appendix II. It computes the positive integer power of another integer by considering the binary expansion of the power. Note that the verification

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conditions are difficult theorems involving ÷ and mod. This current verifier is limited in three areas: 1) theorem proving, 2) the ability to deal with subscripted variables, and 3) the rather limited scope of the assertion language. We will discuss each of these areas after giving a detailed look at the existing program.

The program itself is written in IBM 360 assembly language making extensive use of macros. Specifically it has been run on a model 67 operating in the model 65 mode under OS. It is composed of four major parts: the lexical and syntactic analyzer, the flowcharter, the verification condition generator, and the theorem prover.
A block diagram of the system follows:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The lexical and syntactic analyzer works together with the flowcharter to produce an internal flowchart representation of the given program and its assertions. This also involves converting the text of assignments, conditionals, and assertions into an internal formula representation. Once the program and its assertions are digested into an internal machine form the verification condition generator is called on to process the flowchart and generate the verification conditions. As each verification condition is generated the theorem prover is called to try to establish the validity of the condition.

Formula Manipulation

The flowcharter, verification condition generator, and theorem prover all deal with symbolic formulas or expressions such as $X \geq 0$, $\forall I((1 \leq I \wedge I \leq N) \supset A[I]=0)$, and $X \leftarrow Y \uparrow Z+B$. We have developed a special system to work with such formulas, employing a complete set of basic routines. The ability of these routines to simplify and manipulate such formulas lies at the heart of the whole verifier. Brief definitions of some of the terms we will use in describing the formula manipulation system follow.

Two expressions are "identical" if they are the same string of characters, that is, if they match exactly. Two expressions are "functionally equivalent" or simply

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"equivalent" if for each possible assignment of integer values to the free variables of both expressions the resulting evaluated expressions have the same constant value. For example:

A+B and A+B are identical,

A+B and B+A are not identical but are equivalent, and

A+B and A+C are neither identical nor equivalent.

Identity is a simple syntactic property whereas equivalence is semantic and may be very difficult to establish.

A "strictly syntactic" system is one in which any two expressions are equivalent if and only if they are identical. Our system over integers and truth values is not strictly syntactic. Indeed, we cannot even detect a simple equivalence such as A+B = B+A by syntactic identity. One standard technique for reducing such a system to an equivalent system which is strictly syntactic is the use of "canonical forms". This amounts to determining a subset of the original set of all expressions which functionally spans the original set and which is strictly syntactic. That is, we choose a subset of the original set so that each group of the original functional partitioning is represented by exactly one of its members being included in the subset. This subset is called the set of "canonical expressions" which are all said to be in the "canonical form" (i.e., members of the canonical set).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We also need to be able to map, effectively, any given expression in the original set into its equivalent canonical expression. In this way, we simply put all expressions into canonical form and then work in this strictly syntactic system.

When we cannot devise a canonical form for a system or know that there is none, we try using a "normal form". As with canonical forms we determine a functionally spanning subset of expressions but the resulting system need not be strictly syntactic. That is, we choose a subset of the original set so that each group of the original functional partitioning is represented by at least one of its members being included in the subset. We call this subset the set of "normal expressions" and they are said to be in "normal form". Again we require an equivalence-preserving mapping (normalization) of all members of the original set into members of the subset. To be useful, a normalization process should significantly increase the chances that two equivalent expressions become identical. Normalization can also determine a systematic method for simplifying expressions; they are simplified into the normal form.

In our system, expressions are stored in the computer memory as list structures and a set of utility routines can create new expressions, make copies of expressions, compare two expressions, erase expressions, etc. There is also a routine associated with each operator

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(e.g., +,^,∀,†). We maintain expressions in a normal form and all such routines expect the inputs to be in this form and create outputs in this form. For example, the addition routine (+) expects two arithmetic expressions in the normal form which it adds together symbolically and produces a normal form output. No single routine explicitly puts formulas into the normal form. Formulas are converted from the arbitrary form of the program input to the normal form by the flowcharter and syntax analyzer. They do so by working from the "inside to out" using the operator routines mentioned above. The atomic variables and constants are trivially put in normal form and then more complicated expressions are formed by calling on the appropriate operator routines to operate on existing normalized operands. The input phase of the program verifier not only converts expressions to an internal representation but normalizes them at the same time.

Normalized Arithmetic Expressions

We first examine the normal form as it applies to integer-valued expressions; that is, expressions of the class <arithmetic expression>. A normalized <arithmetic expression> is a sum of "terms" and we refer to it as a "normalized sum" or simply "sum" (i.e., T1+T2+...+Tn where Ti is a term {defined below}). A sum always has one special term--the constant term, which is a signed integer constant.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A sum may have any number of other non-constant terms which each consist of a product of "primaries" (i.e., P1*P2*...Pn where Pi is a primary {defined below}). As with the sum, a term always has one special primary--the constant primary (more commonly--coefficient) which is a signed non-zero integer constant. For ease in reading, we will sometimes omit the redundant constants when writing normalized expressions. Each term also has one or more non-constant primaries. A primary is either a simple variable, an array reference, or a special function. The subscript expression of an array reference must in turn be a normalized sum. The class of special functions is intended to be an open ended catch-all but currently can only be one of ↑, ÷, mod or abs. These each take one or more normalized sums as arguments. Examples of this normalization are:

A*(B-C)	becomes	1*A*B + (-1)*A*C + 0
0*A	"	0
-1	"	-1
-(-(-(-A)))	"	1*A + 0
-4÷2	"	-2
A[I]	"	1*A[1*I + 0] + 0
C*(A÷B)	"	1*C*((1*A + 0)÷(1*B + 0)) + 0
(abs(A)÷B)↑2	"	1*((1*((1*abs(1*A + 0) + 0) ÷(1*B + 0)) + 0)↑2) + 0

As well as having this specific format, normalized expressions must also satisfy some side conditions. One is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that all common expressions must be collected. In a sum, the collection of constant terms is implied by the format allowing for only one integer constant. Other terms which match except for coefficient must be combined as one term with the appropriate coefficient. Thus, although 2*B*A + 3*B*A + 0 is in correct syntactic form it would not be considered completely normalized until it were transformed to 5*B*A + 0.

Within each term, common primaries are also collected. We interpret this in a general way to include the combining of primaries which are the special function ↑ by use of the rule: (a↑b) * (a↑c) = a↑(b+c). Thus

2*(X↑Y)*(X↑Z)	becomes	2*((1*X + 0)↑(1*Y + 1*Z + 0)) + 0
X*X*X	"	1*((1*X + 0)↑3) + 0
abs(X+1)*abs(X+1)	"	1*((1*(abs(1*X + 1)) + 0)↑2) + 0

We also require that additional simplification operations be performed with respect to the special function ↑. Any expression other than a lone primary which is raised to a constant power is to be multiplied out, and anything of the form X↑0 becomes 1, X↑1 becomes X, and 0↑X becomes 0. We also transform (X↑Y)↑Z into X↑(Y*Z). So,

(X+Y)↑3	becomes	1*(1*X + 0)↑3 + 3*Y*(1*X + 0)↑2 + 3*X*(1*Y + 0)↑2 + 1*(1*Y + 0)↑3 + 0
X↑3	"	1*(1*X + 0)↑3 + 0
(X+Y)↑1	"	1*X + 1*Y + 0

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(X-X)↑Y " 0

(X↑2)↑Y " 1*((1*X + 0)↑(2*Y + 0)) + 0

Another powerful and important property also required of all normalized expressions is a complete lexical ordering of sums, terms in sums, and primaries in terms. This ordering makes the normal form more powerful since it causes many equivalent expressions to become identical. Thus A*B*C*D, A*B*D*C , A*D*B*C, ... , D*C*B*A would all become A*B*C*D under normalization (if the lexical ordering of variables were alphabetic, which it isn't--see below). This ordering is essential to the process of collecting common expressions since we must be able to recognize this commoness. In fact, "common" in the broadest sense can be taken to mean functionally equivalent. Another advantage of ordering is that the expression-handling routines can often use the ordering to increase the program's efficiency.

All identifiers are assigned a positive integer value during the input phase and these integers give the ordering over variable names. Thus variable names are ordered according to their first occurrence in the original program input. Array references are ordered first by array name and within that by the ordering of their subscript expressions. Functions are likewise ordered first according to function name and then by the expressions given as arguments (examined left to right). Primaries of different types are sorted: simple variable, array reference, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

special functions, ascending in that order. Terms in a sum are ordered by a left to right comparison of their corresponding primaries ignoring coefficients (since common terms are collected). Sums themselves are ordered by a left to right comparison of their terms including coefficients. This ordering of sums is required for the ordering of array references, functions, and for the logical valued functions discussed later which are built from arithmetic expressions using the relational operators.

If we consider the subset of expressions determined by eliminating all special functions except ↑ and then only allow ↑ to take constant right hand sides (powers) we have the multivariate polynomials. For this set of expressions our normal form is the standard canonical form.

Arithmetic expressions can be combined by the relational operators >, <, ≥, ≤, =, and ≠ to form Boolean primitives. The normal form for such relational expressions is S{R}0 where S is a normalized sum and {R} is one of ≥, ≤, =, or ≠. The relations > and < are eliminated in favor of ≥ and ≤ by use of the rule (true over integers) $X>0 \equiv X-1\geq 0$ and $X<0 \equiv X+1\leq 0$. This rule puts inequalities into the form in which the transitive law is strongest. For example, $X-Y>0$ and $Y-Z+1>0$ would become $X-Y-1\geq 0$ and $Y-Z\geq 0$, respectively, from which it is possible to deduce $X-Z-1\geq 0$ simply by addition. The simple addition of $X-Y>0$ and $Y-Z+1>0$ gives $X-Z+1>0$ which isn't as strong a deduction. We

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

maintain both of the forms ≥ and ≤ for ease at subsequent levels in deducing Y-X=0 from Y-X≥0 and Y-X≤0 by a simple match on the expression Y-X. Otherwise we would have to match -X+Y≥0 and X-Y≥0.

Of course, any relational expressions built from constant arithmetic expressions may be immediately evaluated to form 'true' or 'false'. Two other operations are performed in the normalization of relational expressions. The expressions are reduced by common constant factors and the coefficient of the first non-constant term is made positive by multiplying through by -1 if necessary. The latter operation reduces expressions like X-1≥0 and -X+1≤0 to the common form 1*X + (-1)≥0. By "reduced by common constant factor" we mean the following. Calculate the greatest common divisor, say d, of all the coefficients of non-constant terms in the sum. Note that we do not include the constant term c and it may or may not be evenly divisible by d. If c is evenly divisible by d we divide the sum S by d, getting S', and the expression S'{R} 0 is a reduced form equivalent to the original S {R} 0. If c is not divisible by d then consider S in two components: Q the non-constant part and c the constant part (S=Q+c). We can divide Q by d evenly getting Q' which is always integer valued. Then different reductions take place depending on the particular {R}.

1) {R} is =. We have S=0 or Q+c=0 or Q'+c/d=0. But

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$Q'+c/d=0$ is always 'false' since Q' is always integer valued but c/d never is. We replace $S=0$ by 'false'.

2) {R} is $\neq$. For the same reasons as in (1) we replace $S\neq 0$ by 'true'.

3) {R} is $\geq$. We have $S\geq 0$. But as indicated in (1) the case $S=0$ is not attainable and instead of $Q'+c/d\geq 0$ we use $Q'+(c\div d)\geq 0$.

4) {R} is $\leq$. Analagous to (3) and we use $Q'+((c\div d)+1)\leq 0$.

The use of $\div$ here is the same as that in the programming language, namely: $a\div b=[a/b]$. We have that

$A>B$	becomes	$1*A + (-1)*B + (-1)\geq 0$
$2\geq 3$	"	'false'
$2*(A+B)-1>0$	"	$1*A + 1*B + (-1)\geq 0$
$-6*X+9*Y+(-3)*Z+5\geq 0$	"	$2*X + (-3)*Y + 1*Z + (-1)\leq 0$
$4*X+(-6)*Y+2*Z+(-3)\leq 0$	"	$2*X + (-3)*Y + 1*Z + (-1)\leq 0$
$5*(A-B)-3=0$	"	'false'
$6*(A+B)-2\neq 0$	"	'true'.

Normalized Logical Expressions

Normalized Boolean primitives are used with the logical connectives $\vee$, $\wedge$, $\supset$, and $\sim$ to form more complex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Boolean valued expressions. The well-known "disjunctive normal form" (see [27] page 27) serves as a basis for our normal form for such expressions. As with arithmetic expressions, in addition to the special form, a lexical ordering and considerable simplification is required. The connectives ⊃ and ~ are both eliminated. Implication (⊃) is defined in terms of ∨ and ~ by use of the identity A⊃B ≡ ~A∨B. The ~ is "pushed inside" by use of the identities ~(A∧B) ≡ ~A∨~B and ~(A∨B) ≡ ~A∧~B until it can be applied to the relational expressions. These are then altered to incorporate the ~. Thus ~(X≥0) becomes X+1≤0, ~(X=0) becomes X≠0, ~(X≥0 ∧ X-6≠0) becomes (X+1≤0 ∨ X-6=0), etc.

Disjunctive normal form (d.n.f., for short) is defined as a two level representation where, at the first level, an expression is a set of disjuncts connected by ∨'s, each of which, at the second level, is itself composed of a set of relational expressions connected by ∧'s. We perform simplifications at both levels. First consider the case of one disjunct which is a set of relational expressions connected by ∧'s. Each relational expression, S{R}0 can be considered in three parts:

1) the relational operator (=0, ≠0, ≥0, ≤0),

2) the constant term of the arithmetic expression S, and

3) the remaining non-constant terms of S.

The lexical ordering defined earlier for arithmetic

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expressions is used to impose an ordering on the relational expressions. They are ordered by part (3)--the non-constant terms. This places all expressions involving the same arithmetic expressions (less constant terms) together in one group. Each such group is examined for possible simplification. We have:

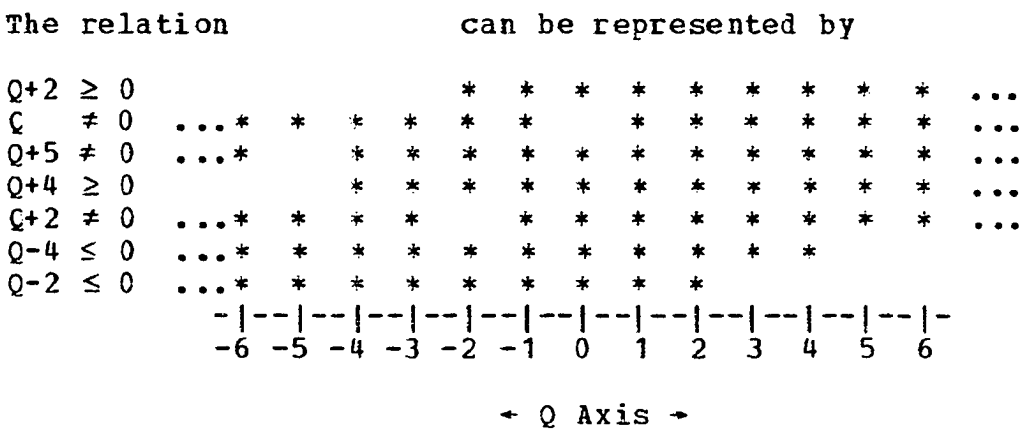
$$
\begin{aligned}
&(Q + c1\ \{R1\}0)\ \wedge \\
&(Q + c2\ \{R2\}0)\ \wedge \\
&\qquad\vdots \\
&(Q + cn\ \{Rn\}0).
\end{aligned}
$$

One interpretation of such a system is to consider each relation as a constraint on the expression Q. Each relation divides the set of all integers into two mutually exclusive subsets. If after substituting a given integer for Q in the relation, it simplifies to 'true', then that integer is "feasible" and in the "feasible set". Otherwise, we get 'false' and it is "non-feasible" and in the "non-feasible set". An integer is feasible with respect to the whole disjunct of relations on Q if it is feasible for all of the relations individually. Of course, this is based on the fact that in order for a disjunct to be 'true' all of its elements must be 'true'. Had the relations been a conjunct, the whole expression is feasible if it is feasible for 'at least one' of the relations.

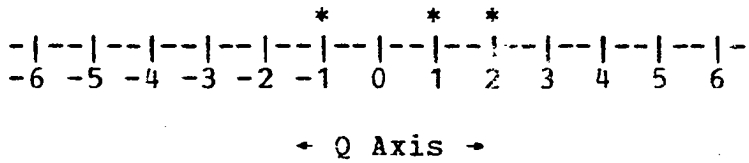
Two systems over the same Q are functionally equivalent if they have the same feasible set. On the other

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

hand, if they are equivalent and the particular Q assumes all possible integer values then they will have the same feasible set. The integer 11 is feasible for X↑2-11≥0 ∧ X↑2-11≤0 but since X↑2 cannot assume the value 11 this system is equivalent to 'false'. Our method for simplifying such systems should be viewed as building a model of the set of feasible points for a system and then reconstructing a simplified system from that model. For example if we use * tc denote feasible points:



The set of points feasible for the whole system consists of those integers whose columns have no blanks:



From this diagram we can construct the simplified system Q+1≥0 ∧ Q≠0 ∧ Q-2≤0 which describes the same feasible set and hence is equivalent to the original system. The actual algorithm for performing such simplifications based on this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

idea operates as follows.

Any relational expression of the form $Q+c=0$ has only one feasible point, namely $-c$. Therefore, the set of feasible points for the whole system can at most contain $-c$. If the system includes an equality we choose it as a hypothetical simplified result. (If more than one equality is in the system, just choose one--say the first.) This is only a hypothetical result since the value $-c$ must also be feasible for all other relations in the system. So we check each of the relations to see if $-c$ is feasible as follows:

1) If the relation is in the form $Q+d\geq 0$, then for $Q=-c$ we get $-c+d\geq 0$. Thus $-c$ is feasible only if $d\geq c$.

2) If it is in the form $Q+d\leq 0$, then for $Q=-c$ we get $-c+1\leq 0$. Thus $-c$ is feasible only if $d\leq c$.

3) If it is in the form $Q+d\neq 0$, then for $Q=-c$ we get $-c+d\neq 0$. Thus $-c$ is feasible only if $c\neq d$.

4) If it is in the form $Q+d=0$, then for $Q=-c$ we get $-c+d=0$. Thus $-c$ is feasible only if $c=d$.

If $-c$ is feasible for all the relations then we output $Q+c=0$ as the simplified result. Otherwise, the feasible set for the whole system is empty and the simplified output is 'false'.

We now turn to the case where there is no equality relation in a system. If more than one $\geq$ relation is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

present, all but one may be dropped by repeated application of the following rule: if we have $Q+c\geq 0 \wedge Q+d\geq 0$, with say $c\leq d$, then $Q+d\geq 0$ is redundant and may be dropped since $Q\geq -c$ and $-c\geq -d$ imply $Q\geq -d$ $(Q+d\geq 0)$. In the analogous fashion we drop all but one $\leq$ relation. This results in an equivalent system with at most one $\geq$ (giving a lower bound on Q), at most one $\leq$ (giving an upper bound on Q), and any number of $\neq$'s. The effect of a relation $Q+c\neq 0$ is to make the integer $-c$ non-feasible with respect to the whole system. If $-c$ falls outside the range specified by the upper and lower bounds then it is already non-feasible and $Q+c\neq 0$ is redundant and may be dropped. If $-c$ coincides with either bound then that bound can be made "one tighter" and then $Q+c\neq 0$ becomes redundant and may be dropped. By "one tighter" we mean to decrease (increase) the upper (lower) bound by one. After adjusting the bound we now re-examine each remaining $\neq$ to see if it may coincide with the new bound.

If the bounds cross, that is , if the upper bound is less than the lower bound, we have an obvious contradiction and simplify the relation to 'false'. If the bounds coincide at, say b, we create the output $Q-b=0$. Of course, if we discover this before we have examined all of the relational expressions we must make sure that b is a feasible value for Q by making the same checks we indicated above in the case the original set contained an equality.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If none of the special cases occur we are left with possibly one upper and one lower bound and any number of $\neq$ relations. These are then output as the simplified result.

This simplification over systems of relational expressions involving the same non-constant terms (the Q part) will always produce the smallest number of relational expressions equivalent to the original system under the following restrictions:

1) the simplified result is a single conjunction of relational expressions, and

2) we assume that the Q part of the expressions can assume all integral values between plus and minus infinity.

The minimality is seen by the fact that each relation in the simplified result is essential. If we output a single $Q+c=0$ equation the only way we could produce a simpler result would be to conclude 'true' or 'false'. But with the assumption that Q assumes all values we have an interpretation which makes $Q+c=0$ 'true' $(-c)$ and all other interpretations make it 'false', so it is not identically 'true' or 'false'. Any $\geq$ $(\leq)$ relation is essential since it denotes an infinite set of integers as non-feasible and could not be replaced by any finite number of $\neq$ or $\leq$ $(\geq)$ relations. Each $\neq$ is also essential, as the point it "marks" non-feasible could not be "covered" by a $\geq$ or $\leq$ since there are feasible points both to its left and to its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

right.

By dropping the above restrictions (1) and (2) we find that simpler formulas exist in some cases. For example,

$$X \geq 2 \wedge X \neq 5 \wedge X \neq 6 \wedge X \neq 7 \wedge X \neq 8 \wedge X \leq 20$$

remains exactly the same after being simplified by our algorithm. Dropping restriction (1) would allow us to use an $\vee$ and simplify this to

$$(X \geq 2 \wedge X \leq 4) \vee (X \geq 9 \wedge X \leq 20).$$

The algorithm simplifies

$$X \uparrow 2 \geq 10 \wedge X \uparrow 2 \neq 10 \wedge X \uparrow 2 \leq 11$$

to

$$X \uparrow 2 = 11.$$

If one is smart enough to realize that $X \uparrow 2$ cannot assume the value 11 (Q does not assume all integer values) this simplifies further to 'false'.

These examples indicate areas in which this stage of the simplification could possibly be improved. One other way would be to consider the interdependence of conjuncts involving distinct yet related expressions. For example,

$$X - Y \geq 0 \wedge Y - Z \geq 0 \wedge X - Z + 1 \leq 0$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

remains exactly the same under our simplification. Yet by considering the transitive property of ≥ we can, in fact, simplify this to 'false'. We do employ a general method in the theorem prover, discussed later, which makes such deductions but it may be fruitful to incorporate it also as part of the simplification.

It was stated earlier that the relational expressions composing each disjunct in the d.n.f. are ordered into groups on their arithmetic expressions (less constant), and now we see that each such group is simplified independently. The simplified groups are further ordered by their relational operator in the order =, ≥, ≠, ≤. The = always occurs alone and there are at most one each of ≤ and ≥, so further ordering within these classes is unnecessary. The ≠ are further ordered on the constant term in descending order.

This method for simplifying something of the form:

A1^A2^...^An

where each Ai is a relational expression can be extended by analogy to work for the case

A1vA2v...vAn

where each Ai is again a relational expression. This remark is made because later levels of simplification do use this analogous technique. In fact, the actual verifier only has

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the routine to handle the conjunctive case. For the disjunctive simplification we use the rule A ≡ ~~A.

A = A1∨A2∨...∨An

is negated to

B = B1∧B2∧...∧Bn ≡ ~A

where Bi is the negated relaticnal expression equivalent to ~Ai (1≤i≤n). This is submitted to the simplifier which produces:

C = C1∧C2∧...∧Cm ≡ B.

By re-negating C we get the simplified form of the original expression:

D = D1∨D2∨...∨Dm ≡ ~C ≡ A

where Di = ~Ci (1≤i≤m).

The next level of simplification considers possible interrelations among the individually simplified disjuncts. Two types of disjuncts are recognized: the "unit disjuncts" and the "non-unit disjuncts". A unit disjunct is the trivial conjunction of one relational expression in contrast to a non-unit one which is the conjunction of two or more. We order the disjuncts in a d.n.f. expression by putting all the units first in order (by the ordering on relational expressions) then following them by the non-units ordered lexicographically by a left to right comparison of their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

relational expressions. We then have an expression of the form:

$$S = A1 \vee A2 \vee \ldots \vee An \vee (b1 \wedge b2 \wedge \ldots \wedge bm) \vee (c1 \wedge c2 \wedge \ldots \wedge ck) \vee \ldots$$

where the Ai's, bi's, and ci's are all relational expressions. The Ai's are the unit disjuncts and $(b1 \wedge b2 \wedge \ldots \wedge bm)$, $(c1 \wedge c2 \wedge \ldots \wedge ck)$, etc. are the non-unit disjuncts.

Taking the unit disjuncts as one group, $A = A1 \vee A2 \vee \ldots \vee An$, we can do a single level simplification directly among relaticnal expressions as we did for each individual disjunct. As noted above, this simplification can be done analogously to that described for conjunctive expressions. Thus after this simplification, we have:

$$A \equiv X = X1 \vee X2 \vee \ldots \vee Xj,$$

and using this in place of A gives a simplified expression S', equivalent to the original S.

$$S \equiv S' = X1 \vee X2 \vee \ldots \vee Xj \vee (b1 \wedge b2 \wedge \ldots \wedge bm) \vee (c1 \wedge c2 \wedge \ldots \wedge ck) \vee \ldots$$

We next do what one might call distribution with respect to unit disjuncts. The disjunction of the units, X, is distributed across each non-unit disjunct, in turn. For example we form

$$X \vee (b1 \wedge b2 \wedge \ldots \wedge bm) \text{ as } (X \vee b1) \wedge (X \vee b2) \wedge \ldots \wedge (X \vee bm).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now if any of these resulting conjuncts, (X∨bi), can be shown to be identically 'true', then that bi in the original expression can be dropped, that is,

$$(W \vee Y) \wedge \text{true} \wedge (W \vee Z) \equiv (W \vee Y) \wedge (W \vee Z) \equiv W \vee (Y \wedge Z).$$

However

$$(X \vee bi) = X1 \vee X2 \vee \ldots \vee Xj \vee bi$$

can be tested for 'true' by submitting it to the same simplification process as before. We do this distribution and testing, for each non-unit disjunct in the expression. If this causes any of the non-unit disjuncts to become units we return to the point of simplifying among units only and again try this distribution process.

We present an example below that demonstrates much of the power of these two simplifications. It is drawn from the verification of example 6 given in Appendix II. At one point (three steps before Theorem 2.1) we have the formula:

$$\forall X\,((A[I-1]-A[I] \leq 0) \vee (N-I \geq 0 \wedge I-K-1=0) \vee$$
$$(N-I \geq 0 \wedge I-K \leq 0) \vee (N-I \geq 0 \wedge K \leq 0) \vee$$
$$(N-I \geq 0 \wedge I-K-1 \neq 0 \wedge A[I-1]-A[K] \geq 0)).$$

We are then required to disjunct $N-I+1 \leq 0$ to this formula. The resulting expression has two unit disjuncts:

$$(A[I-1]-A[I] \leq 0) \vee (N-I+1 \leq 0).$$

There is no immediate simplification between them but by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

distributing these disjuncts over the non-units we cancel the expression N-I≥0 from each non-unit since

$$(A[I-1]-A[I]\leq 0) \vee (N-I+1\leq 0) \vee (N-I\geq 0) \equiv \text{'true'}.$$

This results in

$$\forall K(\ (A[I-1]-A[I]\leq 0) \vee (N-I+1\leq 0) \vee (I-K-1=0) \vee$$
$$(I-K\leq 0) \vee (K\leq 0) \vee$$
$$(I-K-1\neq 0 \wedge A[I-1]-A[K]\geq 0)\ ).$$

All but one of the non-units reduce to units so we again try to simplify among the units alone. This time I-K-1=0 and I-K≤0 combine to form I-K-1≤0. We now have:

$$\forall K(\ (A[I-1]-A[I]\leq 0) \vee (N-I+1\leq 0) \vee$$
$$(I-K-1\leq 0) \vee (K\leq 0) \vee$$
$$(I-K-1\neq 0 \wedge A[I-1]-A[K]\geq 0)\ ).$$

By distributing the units over the single non-unit we find that I-K-1≠0 can be deleted from that disjunct since:

$$(A[I-1]-A[I]\leq 0) \vee (N-I+1\leq 0) \vee (I-K-1\leq 0) \vee$$
$$(K\leq 0) \vee (I-K-1\neq 0) \equiv \text{'true'}.$$

This gives the final simplified result

$$\forall K(\ (A[I-1]-A[I]\leq 0) \vee (N-I+1\leq 0) \vee$$
$$(I-K-1\leq 0) \vee (K\leq 0) \vee$$
$$(A[I-1]-A[K]\geq 0)\ ).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The two simplifications above are simple extensions to the methods used for one level conjuncts of relational expressions. The next and final simplification takes a different form. It is a powerful technique based on a rule usually called "subsumption" (see page 51 of [2]). Consider the tautology:

$$(A \supset B) \supset [(A \vee B) \equiv B].$$

If we have an expression $A \vee B$ and can deduce that every time A is 'true' B must also be 'true' $(A \supset B)$ then we can drop A from $A \vee B$ and simplify it to B. This really amounts to discovering whether $A \supset B$. The most elementary condition implying that $A \supset B$ is that all relational expressions occurring in B also occur in A. If

$A = a1 \wedge a2 \wedge \ldots \wedge an$ and

$B = b1 \wedge b2 \wedge \ldots \wedge bm$ and

each bi $(1 \leq i \leq m)$ is identical to some aj $(1 \leq j \leq n)$ then $A \supset B$. This is the usual subsumption described for simplification of Boolean equations. However, this operates only at the propositional level, matching relational expressions as Boolean primitives. Each relational expression is considered as a Bcolean primitive, say b, or as the negation of some primitive, say ~b. For example, if $b = X \geq 0$ then $X < 0$ would be ~b, but $X \leq 0$ is not ~b although it is very closely related to b and ~b. Much more than the propositional

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information is available in such expressions and we get a stronger form of subsumption by "looking inside" the Boolean primitives just as we did in simplifying within the individual disjuncts. For example, if we examined

$$(X \leq 6 \wedge X \geq 1) \supset (X \leq 6 \wedge X \neq 0)$$

at the propositional level we would have

$$(a \wedge b) \supset (a \wedge c)$$

which cannot be proved to be a tautology. However it is obvious that

$$X \geq 1 \supset X \neq 0$$

and consequently that

$$(X \leq 6 \wedge X \geq 1) \supset (X \leq 6 \wedge X \neq 0)$$

is always 'true'. Armed with this knowledge we can simplify

$$(X \leq 6 \wedge X \geq 1) \vee (X \leq 6 \wedge X \neq 0) \quad \text{to} \quad (X \leq 6 \wedge X \neq 0).$$

We do take advantage of such information and simplify the expressions in d.n.f. by examining each pair of disjuncts for this "relational subsumption". Consider two relational expressions involving the same non-constant terms Q:

$$E1: Q+c1 \{R1\} 0 \quad \text{and} \quad E2: Q+c2 \{R2\} 0 .$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following table gives conditions on c1 and c2, if any, under which E1⊃E2.

{R1} \ E1⊃E2 if {R2}	=	≠	≤	≥
=	c1=c2	c1≠c2	c1≥c2	c1≤c2
≠	c1=c2			
≤	c1>c2	c1≤c2		
≥	c1<c2		c1≥c2	

Assume we have two disjuncts

A = a1∧a2∧...∧an and

B = b1∧b2∧...∧bm.

If for each bi (1≤i≤m) we can find an aj (1≤j≤n) such that aj⊃bi, according to the above table, then we can conclude A⊃B. We use this method to compare each pair of disjuncts A and B to see if A⊃B or B⊃A. If such an implication is discovered we can eliminate one complete disjunct by the subsumption rule. Examples of simplifications by this relational subsumption are:

(Y+Z≥7∧W=0∧X≥6) ∨ (X≠2∧X≠3∧X≠-1) becomes (X≠2∧X≠3∧X≠-1)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$(Y=2 \wedge Z=3) \vee (Y \geq 0) \vee (Z \leq 4 \wedge Y \geq 1)$ becomes $(Y \geq 0)$.

This gives the complete picture of the normal form for non-quantified expressions in our system. The quantifiers $\forall$ and $\exists$ are always "factored outside" of expressions. Thus, the most general form of an expression looks like a list of quantifiers followed by a formula in the simplified d.n.f. (often called "prenex form"). Anytime a quantifier is enccuntered during input, the quantified variable, and all instances of the variable in the quantifier's scope are changed to a different unique variable. Such variable names cannot be input and are distinct from one another so that no confusion of bound variables can later occur.

The above development has been in terms of representing formulae in d.n.f. We also have the capability to deal with formulas in "conjunctive normal form" (c.n.f.) in the system. (The roles of 'and' and 'or' are interchanged in c.n.f.) A control card inserted before the program can set a toggle which causes the system to operate in c.n.f. instead of the assumed d.n.f. The routines 'AND', 'OR', and 'NOT' test this toggle when forming their output and generate their results in the proper form, accordingly. In the theorem prover, discussed later, it is convenient to work with both representations and the prover can also adjust this toggle to change mode.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The system also has facilities for converting from one mode to the other. Suppose one has a formula in d.n.f. like:

$(a \wedge b \wedge c) \vee (d \wedge e) \vee (f \wedge g)$,

then the equivalent expression in c.n.f. would be:

$(a \vee d \vee f) \wedge (a \vee d \vee g) \wedge (a \vee e \vee f) \wedge (a \vee e \vee g) \wedge$
$(b \vee d \vee f) \wedge (b \vee d \vee g) \wedge (b \vee e \vee f) \wedge (b \vee e \vee g) \wedge$
$(c \vee d \vee f) \wedge (c \vee d \vee g) \wedge (c \vee e \vee f) \wedge (c \vee e \vee g)$.

One observes immediately that an explosion of the expression takes place when this conversion is made. In practice, the primitives a, b, c, etc., are not unrelated and the resulting expression may simplify quite a lot, but this is still a dangerous operation to perform. (It is dangerous in either direction as converting $(a \vee b \vee c) \wedge (d \vee e) \wedge (f \vee g)$ to d.n.f. shows.)

Negation has this same "multiplying out" problem as converting from one form to the other. In fact, the system converts from one form to the other by using the negation routine's capability to multiply out expressions. This is done by introducing a special type of negation which we call "frozen negation". One simultaneously replaces each $\wedge$ by an $\vee$, and each $\vee$ by an $\wedge$, and also negates each relational expression in the formula. The over-all shape of the expression is not altered (i.e., no multiplying out occurs); the form of the formula is "frozen". The resulting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expression is equivalent to the negation of the original formula but is in the other normal form. Now, by switching the d.n.f. - c.n.f. toggle and calling the negation routine, we get an expression equivalent to the original but in the opposite form. This "frozen negation" is also used in the theorem prover to convert from one form to the other and negate, in one operation.

Input and Its Internal Representation

We can now return to a more complete discussion of the major components of the program verifier. As mentioned before, the lexical and syntactic analyzer works together with the flowcharter to produce an internal representation of a program and its assertions. This analyzer was taken directly from the CMU Algol-67 compiler -- the version of the Algol-20 compiler developed at CMU for the CDC (Bendix) G-21 computer and rewritten for the IBM 360. This compiler has been documented rather extensively [1,9,10,11]. Its syntax analyzer is controlled by a "production table" generated from a description of the syntax given in a "production language". It is not pertinent to discuss this further except to point out that due to this technique it was easy to make small syntactic changes in the language recognized by the compiler (e.g., adding $\supset$, $\forall$, and $\exists$). We did this by making small changes in the productions which were used to create a new production table thus

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

incorporating the desired changes into the analyzer.

The particular composition of the Algol-67 compiler makes it convient to steal the lexical and syntactic parts. The compiler is grossly divided into two phases. Phase I reads the input program, does the lexical and syntactic analysis, and generates postfix type output which is passed to phase II. Phase II does the semantic analysis and generates the machine code to perform the computations of the given program. In our system phase II is replaced by the flowcharter which builds the internal flowchart instead of generating code.

The lexical scanner breaks the raw input into lexical units consisting primarily of identifiers and operators. Each identifier is converted into an integer "relative address" representing the identifier's position in the "symbol table" of all identifiers. All subsequent routines manipulate the identifier relative address. Two identifiers are the same iff they have the same relative address. The original string of characters representing the identifier, called its "print name", can be retrieved from the symbol table for printing.

The syntax analyzer generates a postfix string of operands (relative addresses) and operators by interpreting the production table with respect to the input stream of lexical units. This postfix string is passed to the flowcharter unit by unit as it is created. The operands of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

any operator immediately precede that operator in the string and are either subordinate postfix strings or relative addresses of identifiers. If the element of the string passed to the flowcharter is an identifier it is placed on the "operand push down stack". If the element is an operator the flowcharter executes the normalization routine for that operator. Most of these operator routines take operands off the top of the push down stack, combine them or examine them, and stack a result in their place.

The routines themselves fall into two categories: ones concerned with mapping the program's flow of control, and ones concerned with building formulas which occur within flow chart boxes. We have already discussed the basic formula manipulation capabilities of the system which are used by the formula building routines. Each formula operator (e.g., +, *, ∨, ~, ∀, ≥, ←) takes normalized operands from the top of the stack, symbolically performs the operation and restacks the normalized result.

The other class of operator routines are concerned with building an internal flowchart representing the input program. These correspond to such things in the program as 'IF', 'THEN', 'GO TO', ':' (after a label), etc. The flowchart is built by interconnecting four different types of "nodes"; each type representing a type of statement in the programming language. These are:

1) Assignment statement

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2) Conditional statement (test)

3) Assertion statement

4) Begin-end statements.

There is only one begin-end node corresponding to both the initial 'BEGIN' and the final 'END'. These two points in the flowchart are important in later processing and are designated by this node. 'BEGIN' and 'END' used as statement parentheses for compound statements simply supply information concerning flow of control and do not explicitly occur in the internal representation.

Each of the four types of node consists of these component parts:

1) Identification--type indication.

2) Text pointer--each node (flow chart box) has a formula associated with it and pointed to by this entry.

3) Forward pointer list--these pointers point to statements which are successors to this statement in the flow of control. All nodes except the test node have only one successor. Test nodes have two: one if the test predicate evaluates as 'true' (true successor) and one if it gives 'false' (false successor).

4) Backward pointer list--these pointers point to statements which precede this one in the flow of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

control. Any statement which is labeled may be designated by any number of go to statements and therefore may have any number of predecessor statements.

5) Label relative address--if the statement corresponding to this node had been preceded by one or more labels, the relative address of the last label identifier will be recorded here.

The nodes represent the boxes of a flowchart and are interconnected by the forward and backward pointers representing the flow of control in the program. Text pointers point to formulas which occur inside the flowchart boxes. Note that there is no node type corresponding to 'GO TO' statements since these indicate flow of control which we represent with pointers between the nodes.

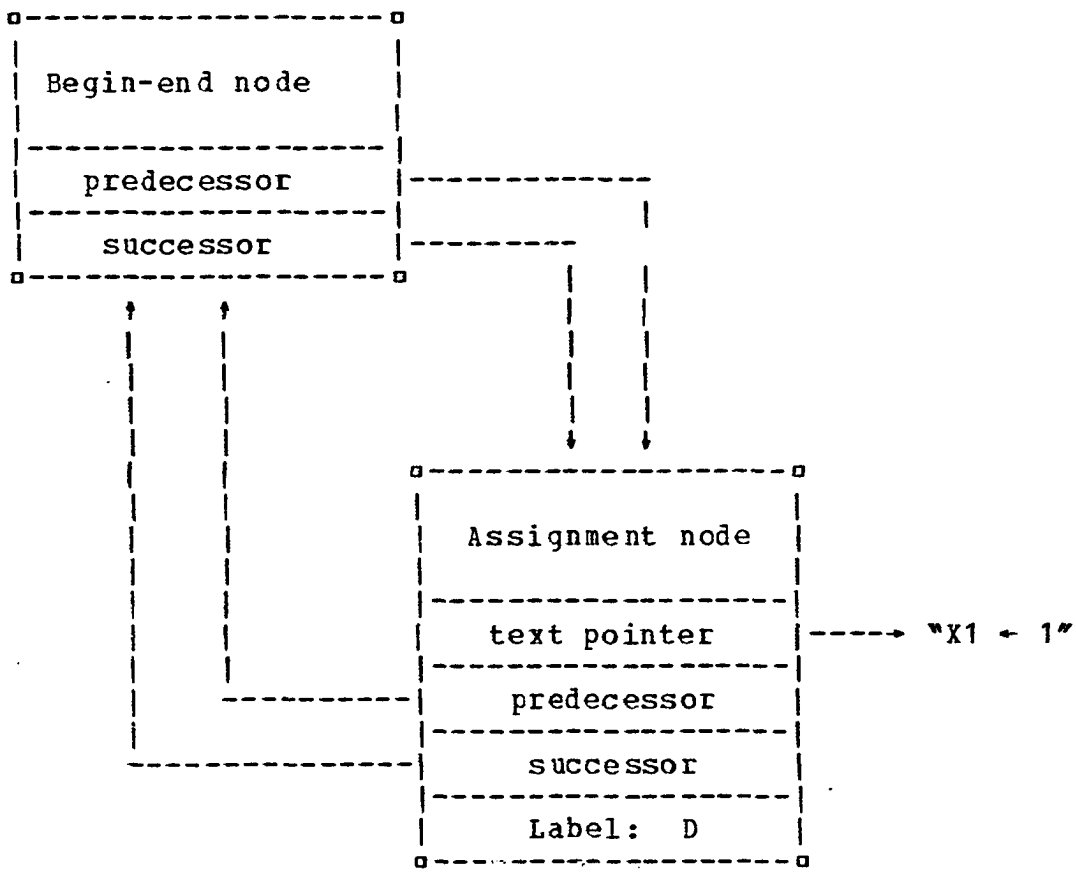
The flowcharter creates each node in the flowchart as the corresponding statement is processed in the input. Their interconnections are developed by remembering predecessors in the same operand push down stack used for saving the operands for formula operators. We also use the symbol table entries of label indentifiers to remember lists of, as yet, unresolved label references, and when they become resolved, to note the node at which the label is defined. These standard compiler building methods allow the resolution of complex flow of control introduced by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

redundant 'GO TO' statements. For example, the program

```
BEGIN
     GO TO A;
  C: GO TO D;
  B: GO TO C;
  A: GO TO B;
  D: X1←1;
END;
```

will be transformed into the following internal flowchart:



More complicated examples of the internal flowchart representation can be found in Appendix II where flowchart "dumps" of examples run can be found.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Verification Condition Generator

Once the input program with its associated assertions is digested into an internal form it is processed by the verification condition generator. This generator works by the method of "going backward" discussed in Chapter I. As each assertion statement is encountered and its corresponding assertion node is built, the location of that node is saved on an assertion list. At a later stage, each assertion node on this list is processed by carrying its assertion text backward through all possible control paths until one of the following occurs on each such path:

1) an assertion node (possibly the one where we started),

2) the begin-end node, or

3) a node which we have already encountered while traversing this particular path.

As we pass through each assignment or test node on our backward path we perform the appropriate transformations on the assertion "in hand". The initial assertion in hand is that given in the assertion node where we started. The text of the assignment and test nodes supplies the symbolic formulas needed to perform these transformations. In case (1), (at an assertion node), we build a verification condition which requires that the encountered assertion text

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

imply the assertion in hand. This is then given to the theorem prover. In (2), the assertion in hand represents one conjunct of an initial condition required for complete verification of the program. This may be just part of the initial conditions since other paths may also end at the begin-end node and give additional conjuncts. Case (3) can occur only if there exists a loop in the program which does not have an assertion statement somewhere on its path. This is considered to be an improper input to the verifier and an error message is given.

For each assertion node all possible backward paths are traversed until one of the three conditions is met. This means that whenever a node which has more than one predecessor is encountered there is more than one path to follow. This is done simply by examining the paths serially. We choose the first predecessor as the current path but save the other alternatives by placing them on a push down stack of paths to be traversed later. Of course, we must also copy and stack the assertion in hand so it will be available for processing the remaining paths.

The verification condition generator traverses all possible paths backward from each given assertion statement in the program and submits each resulting verification condition to the theorem prover for validation. Two topics are yet to be discussed: the details of the transformations performed on encountering test and assignment statements on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

our backward traversals, and the theorem prover.

Suppose while moving backward through the program's flow of control we encounter a test statement having the predicate T. Also suppose that the assertion in hand was A. Then the semantic definition for the test statement determines the new assertion in hand after passing through this test statement as either T⊃A, if we had arrived at the test statement from its 'true' successor, or as (~T)⊃A, if we arrived by way of the 'false' successor. We easily determine by way of which successor we arrived at the test node, since it has pointers to both its 'true' and 'false' successors. Routines for symbolically performing the ~ and ⊃ operations are available in the basic set, so these transformations for test's offer no problems.

The semantic definition for assignment statements requires the substitution of an expression for a variable in the assertion in hand. The "assignment expression" is the right side of the assignment statement and the "assignment variable", which may be a simple variable or a subscripted variable, is the left side. These two cases are handled separately. If the assignment variable is a simple variable the substitution is straightforward. In order to keep the assertion in hand in the normal form and hence take advantage of any simplifications which may result from the substitution, this substitution routine must completely "tear apart" and rebuild the assertion during the process.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This is done at each "level" of the expression by first substituting in the "lower level" components one at a time and then reforming them by calling upon one of the basic operator routines. For example, in order to substitute into a normalized sum we consider its component terms. We substitute in each term and then reform the normalized sum by using the basic routine for addition (+) to symbolically add the new terms together. The substitution in terms likewise considers its component primaries, substitutes in them, and reforms itself by using the basic routine for multiplication (*). Similar comments apply for all levels in the expressions (e.g., v, ^, functions, etc.). Function and array reference substitution routines must recursively invoke this whole process to do the substitutions on arguments and subscript expressions, respectively.

Actual substitutions can occur only at the level of a simple variable. If a variable matches the given assignment variable, a copy of the assignment expression is returned as the substituted value. If they do not match, the original variable itself is returned.

More interesting problems occur during a substitution for a subscripted variable. A subscripted variable does not refer to a particular variable but to a whole set of variables depending upon the value of its subscript. That is, A[I] may be the particular variable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A[1], A[20], A[101], etc., depending on the value of I. Two array references to the same array, say A[I] and A[J], may or may not refer to the same variable depending whether I=J or I≠J respectively. Thus on substituting for A[I] in an expression there is a conditional match with any other reference to that same array depending on whether the subscript expressions agree. This property of conditional matching implies that we need conditional substitution. The substitution is done by generating the conditions on the subscripts and including them as additional relational expressions. For example, substituting X for A[J] in the formula

$$\forall I(I\leq 0 \vee I-N-1\geq 0 \vee A[I]-A[I-1]+1\leq 0)$$

without simplification would result in

$$\begin{aligned}\forall I(I\leq 0 \vee I-N-1\geq 0 \vee\ & (J-I=0 \wedge J-I+1=0 \wedge X-X+1\leq 0) \vee \\ & (J-I=0 \wedge J-I+1\neq 0 \wedge X-A[I-1]+1\leq 0) \vee \\ & (J-I\neq 0 \wedge J-I+1=0 \wedge A[I]-X+1\leq 0) \vee \\ & (J-I\neq 0 \wedge J-I+1\neq 0 \wedge A[I]-A[I-1]+1\leq 0)),\end{aligned}$$

which would be simplified by our normalization process to

$$\begin{aligned}\forall I(I\leq 0 \vee I-N-1\geq 0 \vee\ & (J-I=0 \wedge X-A[I-1]+1\leq 0) \vee \\ & (J-I+1=0 \wedge A[I]-X+1\leq 0) \vee \\ & (J-I\neq 0 \wedge J-I+1\neq 0 \wedge A[I]-A[I-1]+1\leq 0)).\end{aligned}$$

The routine which does this substitution is similar to that for the simple variable substitution: the expression

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is completely "broken down" and rebuilt. The differences are in rebuilding the relational expressions. One relational expression may be expanded into many cases each involving some derivation of the original expression conjoined with the conditions determining that case, as in the above example, where A[I]-A[I-1]+1≤0 becomes four disjuncts. This problem is handled by allowing the result of a substitution to be a list of results each prefaced with a condition under which that result was formed. The expression is then rebuilt from such conditional lists by "conditional arithmetic". Suppose there are two conditional lists, say (c1:s1, c2:s2, ..., cn:sn) and (d1:t1, d2:t2, ..., dm:tm), where c's and d's, representing conditions, are separated from their associated expressions, the s's and t's, by colons. Their "conditional sum" would be formed by conjoining conditions and adding expressions in all possible combinations:

((c1∧d1):s1+t1, (c1∧d2):s1+t2, ..., (cn∧dm):sn+tm).

In performing conditional arithmetic other than addition we simply replace the + by the appropriate operator (e.g., *, ↑).

The previous example has three disjuncts developed from A[I]-A[I-1]+1≤0 as follows. To substitute in the relational expression we first substitute in the arithmetic expression from which it is built, namely A[I]-A[I-1]+1. To substitute in this sum we first substitute in each component term. A[I] becomes the conditional list:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(J-I=0:X, J-I≠0:A[I]),

-A[I-1] becomes the conditional list:

(J-I+1=0:-X, J-I+1≠0:-A[I-1])

and +1 becomes the conditional list:

(true:+1).

These lists are now conditionally added:

(J-I=0:X, J-I≠0:A[I]) + (J-I+1=0:-X, J-I+1≠0:-A[I-1]) +
(true: +1)

= (J-I=0:X, J-I≠0:A[I]) + (J-I+1=0:-X+1, J-I+1≠0:-A[I-1]+1)

= ((J-I=0∧J-I+1=0):X-X+1, (J-I=0∧J-I+1≠0):X-A[I-1]+1,
(J-I≠0∧J-I+1=0):A[I]-X+1, (J-I≠0∧J-I+1≠0):A[I]-A[I-1]+1).

As each of the conditions are formed they simplify so we actually get

(J-I=0:X-A[I-1]+1, J-I+1=0:A[I]-X+1,
(J-I ≠0∧J-I+1≠0):A[I]-A[I-1]+1).

Note that since (J-I=0∧J-I+1=0) ≡ 'false', its list element is simply dropped. Now in the next step of the reconstruction the arithmetic expressions become relational expressions by applying the ≤0 operation to each. After this the conditional lists can be built directly into a single expression by conjoining each relation with its condition and then by forming the disjunction of all these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

results. The example now becomes:

(J-I=0 ∧ X-A[I-1]+1≤0) ∨

(J-I+1=0 ∧ A[I]-X+1≤0) ∨

(J-I≠0 ∧ J-I+1≠0 ∧ A[I]-A[I-1]+1≤0).

From this point on, the process is the same as that for the simple variable case. This result is combined, by appropriate ∨'s ∧'s, with the results determined by substituting in the other parts of the expression.

An interesting case occurs when we attempt to substitute in an expression like A[A[J]]=0. Suppose we wish to substitute X for A[I] in A[A[J]]=0. Before we can consider substituting for an array we must resolve the result of substituting in its subscript expression. This gives the subproblem of substituting X for A[I] in A[J]. That, in turn, gives the problem of doing the same substitution in J, but that is just J, so the result of substituting in A[J] gives the conditional list

(I-J=0:X,I-J≠0:A[J]).

We are now faced with an extension of the idea of conditional arithmetic, that of doing conditional subscripting:

A[(I-J=0:X, I-J≠0:A[J])] ≡ (I-J=0:A[X],I-J≠0:A[A[J]])

In the case that I-J=0 we have A[X] and so substituting X for A[I] in this gives

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I-J=0:(I-X=0:X, I-X≠0:A[X]) ≡

((I-J=0 ∧ I-X=0):X, (I-J=0 ∧ I-X≠0):A[X]).

When I-J≠0 the subscript of A is A[J] and therefore we form

I-J≠0:(I-A[J]=0:X, I-A[J]≠0:A[A[J]]) ≡

((I-J≠0 ∧ I-A[J]=0):X, (I-J≠0 ∧ I-A[J]≠0):A[A[J]]).

Substituting X for A[I] in (I-J=0:A[X], I-J≠0:A[A[J]]) gives the conditional list:

((I-J=0∧I-X=0):X, (I-J=0∧I-X≠0):A[X],

(I-J≠0∧I-A[J]=0):X, (I-J≠0∧I-A[J]≠0):A[A[J]]).

This brings us back to the level of relational expressions where we can drop the conditional notation, getting the final result

(I-J=0 ∧ I-X=0 ∧ X=0) ∨

(I-J=0 ∧ I-X≠0 ∧ A[X]=0) ∨

(I-J≠0 ∧ I-A[J]=0 ∧ X=0) ∨

(I-J≠0 ∧ I-A[J]≠0 ∧ A[A[J]]=0).

Theorem Prover

At first the discussion is confined to the case where there are no explicit quantifiers in the theorems to be proved. Later we will discuss quantified expressions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The theorems all arise in the form:

$\vdash P \supset Q$

where both P and Q are in the simplified disjunctive normal form. We can form the universal closure of $P \supset Q$ and get a theorem equivalent to the original:

$\vdash \forall x1 \forall x2 \ldots \forall xn (P \supset Q)$.

where x1,x2,...,xn are assumed to be all the free variables occurring in $P \supset Q$. We do this closure so we can work with the equivalent complimentary problem

$\vdash \sim \exists x1 \exists x2 \ldots \exists xn (\sim (P \supset Q))$.

This amounts to showing that one cannot find a set of integral values to assign to the variables x1,x2,...xn such that $\sim(P \supset Q)$ is 'true'. Although this problem is written as a theorem above, we actually work only with the expression $\sim(P \supset Q)$. The other parts of the theorem (the '~' and the $\exists$'s) can be considered implicit properties which govern the following discussion. The general attack we take is to form deductions from $\sim(P \supset Q)$ which eventually lead to an obvious contradiction. This contradiction shows that $\exists x1 \exists x2 \ldots \exists xn [\sim(P \supset Q)]$ is 'false' and so the original theorem is 'true'.

Eventually the theorem is broken into cases, but before this is done the formula is processed as a whole. The expression is represented in c.n.f. for this processing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the subsequent conversion to d.n.f. causes an explosion of the formula then the steps performed in this parallel phase can be done repeatedly for each case and the phase dropped. None of the simple examples we processed led to any trouble in this respect. Both normal forms for ~(P⊃Q) can be very cumbersome (given that P and Q are in the same form). At least one negation of either P or Q must be done and as we noted in Chapter II this is a dangerous operation.

The first step of the theorem prover is changing the expression ~(P⊃Q) into simplified conjunctive normal form. This is done using the tautology

$$\sim(P \supset Q) \equiv P \wedge \sim Q.$$

Since P and Q are both in d.n.f. we form P∧~Q in c.n.f. by first doing a "frozen negation" on both P and Q. This gives ~P and ~Q in c.n.f. We then do a normal negation on ~P giving P in c.n.f. Now P∧~Q is formed from these by doing the ∧ completely in c.n.f. mode.

At this point, many of the more elementary theorems simplify to 'false' and we are done. The power of the simplification in handling such theorems completely is impressive. On the other hand, many of the theorems are roughly $\vdash A \supset A$ and it is reasonable for simplification to be useful in these cases. This result is to be expected in the case where we traverse a loop in a program, since the assertion on the loop must be invariant with respect to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

number of times the loop is executed.

After forming this complement expression in c.n.f. we examine each unit conjunct to see 1) if it is an equality, 2) if there is a term consisting of a lone simple variable, say x, with a coefficient of +1 or -1, and 3) if that variable occurs nowhere else in the equality. If such an equality is found we can solve for that variable in the expression and put the equality intc the form x=E. Since

$$\exists x[x=E \wedge S(x)] \equiv S(E)$$

we substitute E for x in the rest of the expression and drop the equality. This eliminates the variable x completely and also gives the chance for more simplification to occur. The substitution is performed by using the same routine discussed earlier for handling assignment statements in the verification condition generator. It brings all the simplification techniques to bear on the new expression. We repeat this process of solving for and substituting for lone variables until none remain.

Note two points. First, such solving for variables is done in unit ccnjuncts and not in non-unit conjuncts, since

$$\exists x[(x=E \vee R(x)) \wedge S(x)] \equiv [R(E) \wedge S(E)]$$

is not 'true', in general. (Of course, the substitution is not confined to the units but is done throughout the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

complete expression.) The second point is that the simplification occurring after a substitution may introduce new unit equalities and these are also examined for the elimination of a variable.

The next step is to eliminate the special functions ÷, mod, and abs "by definition". Each occurrence of these functions represents the computation of some integer value and that value has certain properties with respect to the arguments of that computation. (If this sounds a little like the description of a verified program to compute these functions, it should. This is essentially the technique we propose for handling procedures and function calls in the programming language. See Chapter III.) Suppose we have:

$$abs(A+B)+D\neq 0.$$

But abs(A+B) is always some integer value, say C, and so:

$$C+D\neq 0.$$

A complete characterization of that value C is given by:

$$(A+B\geq 0 \supset C=A+B) \wedge (A+B<0 \supset C=-(A+B)),$$

so instead of

$$abs(A+B)+D\neq 0$$

we have:

$$C+D\neq 0 \wedge (A+B\geq 0 \supset C=A+B) \wedge (A+B<0 \supset C=-(A+B)).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the case of abs, C is defined in terms of equalities so we can eliminate it. This gives us:

$(A+B\geq 0 \supset A+B+D\neq 0) \wedge (A+B<0 \supset -A-B+D\neq 0)$.

Since $A+B\geq 0 \equiv \sim(A+B<0)$ this can also be written in either the form

$(A+B\geq 0 \wedge A+B+D\neq 0) \vee (A+B<0 \wedge -A-B+D\neq 0)$ or the form

$(A+B\geq 0 \vee -A-B+D\neq 0) \wedge (A+B<0 \vee A+B+D\neq 0)$.

We are working, at this point, in c.n.f. so we use the latter form.

For defining the special functions we use the same basic mechanism used when substituting for subscripted variables -- that of conditional lists and the rebuilding of relational expressions using conditional arithmetic. We give the definitions with the notation previously used for those conditional lists:

1) Every primary of the form abs(x) in the formula is replaced by $(x\geq 0:x,\ x<0:-x)$,

2) Every primary of the form $(x\div y)$ in the formula is replaced by $((y\geq 0 \wedge x-q*y\geq 0 \wedge y-x+q*y>0):q,\ (y<0 \wedge x-q*y\geq 0 \wedge -y-x+q*y>0):q)$,

3) Every primary of the form (x mod y) in the formula is replaced by $((y\geq 0 \wedge x-q*y\geq 0 \wedge y-x+q*y>0):x-q*y,\ (y<0 \wedge x-q*y\geq 0 \wedge -y-x+q*y>0):x-q*y)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the last two definitions q is, at each application of the definition, a unique variable occurring nowhere in the given formula and different from all other q's. Note that q represents the quotient and the expression x-q*y represents the remainder, in the divisicn of x by y. Using these definitions all special functions except ↑ are eliminated.

As with our other manipulations on formulas, the process of introducing the definitions completely breaks down and rebuilds the expression using the basic routines. The resulting simplifications may introduce new unit conjuncts which are solvable for lone variables. Thus the search for such variables to be eliminated is repeated exactly as in the previous step. If we reduce the expression to 'false' at any stage we have established the proof of the verificatien condition.

This exhausts the list of steps taken on the formula as a whole. The next stage breaks the problem into simpler subproblems each of which must be solved. The subproblems are derived by converting the formula from c.n.f. to d.n.f. and operating on each disjunct separately. This process is valid since

$$\sim\exists x(R(x) \vee S(x)) \equiv (\sim\exists xR(x) \wedge \sim\exists xS(x))$$

and therefore proving the theorem

$$\vdash \sim(\exists xR(x) \vee \exists xS(x))$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is the same as proving

$$\vdash \sim\exists x R(x) \quad \text{and} \quad \vdash \sim\exists x S(x) .$$

Each subproblem is attacked in turn. The subproblems are conjunctions of relational expressions and are therefore much simpler than the original theorem. We need to show that there is no single assignment of integer values to the variables in the expressions such that all relational expressions are simultaneously 'true'. At this point all equality relations are again examined for solvable variables. As before, if any are found we substitute throughout the subsystem for that variable and eliminate it. This time there is no need to worry about being restricted to unit conjuncts since, in effect, all of the relational expressions are units. After eliminating as many variables as possible we turn to the final technique--the "linear solver".

Linear Solver

The linear solver examines interrelations among the expressions. A simple example of the type of deduction possible is to deduce 'false' from the contradictory system mentioned earlier:

$$X-Y\geq 0 \wedge Y-Z\geq 0 \wedge X-Z+1\leq 0.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As the name implies the solver itself deals with "linear" systems. The word "linear" is used here as in "linear programming". Any relational expression whose associated arithmetic expression is composed only of terms (except the constant term), which are all single variables (including array references) with arbitrary coefficients, is considered to be linear. For example,

```
A + 3*B - 1001*C ≥0,
2*X - 5*Z[I] +6 =0,     and
A[X↑2+1] ≠0
```

are all linear relations, but

```
X↑3 + X↑2 + 3 ≥0,
2*X*Y + Z =0,     and
X↑Y ≠0
```

are non-linear.

The restricted format of linear inequalities allows special techniques to be applied to that part of the system. If we can deduce a contradiction from the linear part separately we also have a contradiction for the whole system and the subproblem is solved. When no contradiction in the linear part can be found we attempt to extract information from it in the form of bounds on variables or expressions. This information may be used to simplify the non-linear part and possibly lead to a contradiction or reduce a non-linear

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

formula to a linear one. In the latter case, we can re-examine the new linear part for contradictions or additional bounding information. In coding this part of the theorem prover it was convenient to structure the process in a manner slightly more complicated than that indicated above. So that this process may be explained clearly we begin by discussing the basic notions involved.

As assignment of real (as opposed to integral) values to the set of variables in a system will be called a "point" of that system. When these values are restricted to be integers the assignment is called an "integer point". A point "satisfies" a relational expression or a system of such expressions if it results in the value 'true' when evaluated at that point. If a system has at least one (integer) point which satisfies it we say that the system is (integer) "satisfiable" or that it has a (integer) "solution". If it is not satisfiable it is "unsatisfiable". Our theorem proving task is to show that the given system (both linear and non-linear parts) is integer unsatisfiable.

The linear solver is based on an algorithm for deciding whether a given set of linear inequalities is satisfiable in real numbers. If a set of linear inequalities is (real) unsatisfiable then it is certainly integer unsatisfiable. In [17], Kuhn gives a development of the "real" algorithm which we present briefly.

For simplicity assume that all inequalities are of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the form S≥0. Kuhn considers both forms S≥0 and S>0, but since the integer system need deal only with the form S≥0, just that case is treated here. From the given system one can deduce another system with at least one fewer variable. This deduction preserves satisfiability; any point satisfying the original system must also satisfy the deduced system. The process is repeated getting a sequence of systems, each with fewer variables than the preceding one. Continuing in this way one eventually gets a system with no variables, which can be evaluated to 'true' or 'false'. This final truth value indicates the satisfiability of the system: if it is 'true' the system is satisfiable, if 'false' it is not.

A variable may be eliminated by the repeated application of one basic operation which is called "positive cross multiplication". If

G≥0 and H≥0

then for any two positive constants c and d:

c*G + d*H ≥0.

Suppose the variable x occurs in G with coefficient g and occurs in H with coefficient h. Then provided g and h are of opposite signs x can be eliminated between G and H by the above rule with c=|h| and d=|g|. (|a| is the absolute value of a). That is, c*G would have the term |h|*g*x and d*H would have the term |g|*h*x. Since g and h have opposite

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

signs these two terms would cancel in the sum, (i.e., |h|*g*x + |g|*h*x =0). Thus G and H have been "cross multiplied" to eliminate x.

Given a system P and a variable x, then by eliminating x, a new system Q can be formed from P by the following procedure, denoted by P-x→Q. First include in Q all those inequalities of P which do not involve x. Then include all new inequalities which can be formed by eliminating x between pairs of inequalities of P by cross mulitplication. That is, in every possible way, eliminate x between inequalities of P involving x with opposite signs. Q is then a system, free of x, which is satisfied by at least those points which satisfy P. This fact follows simply because the basic cross multiplication rule is valid.

Note that when all occurrences of x have coefficients of the same sign, the cross multiplication produces nothing, so Q is just that part of P free of x. Note also that other variables which occur in P besides x may happen not to occur in Q. The result of a cross multiplication may have the form n≥0 for some integer constant n. This is immediately evaluated to 'true' or 'false'. Within Q simplify using the rules:

'true' ∧ A ≡ A and

'false' ∧ A ≡ 'false'.

Q may therefore be just 'true' or 'false'. For convenience

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the operation -x→ is defined to take 'true' into 'true' and 'false' into 'false'.

Suppose we have an initial system P involving the variables x1,x2,...,xn. Form the sequence of systems P -x1→ P1 -x2→ P2 ... -xn→ Pn. Pn has no variables and is therefore either 'true' or 'false'. If Pn='false' then Pn is obviously unsatisfiable. Since the -x→ operation preserves satisfiability we know that P is also unsatisfiable. This is also true for integer satisfiability for if P is not satisfied by any real points then it obviously is not satisfied by any integer points. In the case Pn='true' the integer and real systems differ. Pn='true' implies that P is (real) satisfiable but does not necessarily imply that it is integer satisfiable. We first show the completeness for the real case and then proceed to discuss how the basic real method is altered for use in the theorem prover.

If Pn='true' then it is satisfied by all points. Knowing this, one can conclude that P is satisfiable provided:

> for any two systems Q and R such that Q-x→R, Q is satisfiable if R is satisfiable.

If Q = 'true' the property is trivially true. If not then Q has the form:

$$c1*x + A1(Y) \geq 0 \quad \wedge$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{array}{rl}
c2*x + A2(Y) \geq 0 & \wedge \\
\quad\vdots & \\
cn*x + An(Y) \geq 0 & \wedge \\
-d1*x + B1(Y) \geq 0 & \wedge \\
-d2*x + B2(Y) \geq 0 & \wedge \\
\quad\vdots & \\
-dm*x + Bm(Y) \geq 0 & \wedge \\
\quad G1(Y) \geq 0 & \wedge \\
\quad G2(Y) \geq 0 & \wedge \\
\quad\quad\vdots & \\
\quad Gk(Y) \geq 0 &
\end{array}
$$

where ci ($1 \leq i \leq n$) and di ($1 \leq i \leq m$) are positive integer constants and Ai(Y) ($1 \leq i \leq n$), Bi(Y) ($1 \leq i \leq m$), and Gi(Y) ($1 \leq i \leq k$) represent terms involving the remaining variables which are represented by the vector Y. Then R looks like:

$$
\begin{array}{rl}
di*Aj(Y) + cj*Bi(Y) \geq 0 & \wedge \quad \text{(for each } 1 \leq i \leq m \text{ and } 1 \leq j \leq n\text{)} \\
\quad G1(Y) \geq 0 & \wedge \\
\quad G2(Y) \geq 0 & \wedge \\
\quad\quad\vdots & \\
\quad Gk(Y) \geq 0. &
\end{array}
$$

Assuming R is satisfiable, let r be a point satisfying R. This point r can be extended to be a point q over the variables of Q by determining a value for x. If n=m=0 then Q=R and for any value of x, say zero, q satisfies Q. Otherwise evaluate each Ai(Y) ($1 \leq i \leq n$), Bi(Y) ($1 \leq i \leq m$), and Gi(Y) ($1 \leq i \leq k$) at the point r determining integer constants ai ($1 \leq i \leq n$), bi ($1 \leq i \leq m$), and gi ($1 \leq i \leq k$), respectively.

Choose e = minimum over all i ($1 \leq i \leq n$) of {bi/di} and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

f = maximum over all i ($1 \le i \le m$) of $\{-ai/ci\}$. Since r satisfies R, $di*aj + cj*bi \ge 0$ (or rewritten $bi/di \ge -aj/cj$) for all combinations of i and j ($1 \le i \le m$ and $1 \le j \le n$). Therefore $e \ge f$. Now if the point r is extended to q by assigning x the value e (or any value $e \ge x \ge f$) then q satisfies Q. This can be easily seen by considering the inequalities of Q as constraints on x. The Gi's do not contain x and therefore offer no constraints. The expressions $\{bi/di\}$ form upper bounds on x and the expressions $\{-ai/ci\}$ form lower bounds, but e and f are the smallest and largest of these, respectively. Since $e \ge f$ there is a value for x (e.g., e) which satisfies these constraints. In the case that n=0 or m=0, x is unbounded in one direction and so a value can obviously be chosen which makes Q satisfiable.

Thus P is satisfiable if and only if Pn is satisfiable and this technique is a decision procedure for the real case. The proof and, indeed, the method itself fails in the case of an integer valued system since there is no guarantee that there exists an integer value between any two rational numbers, say e=bj/dj and f=-ai/ci, even when $e \ge f$. For example, let P be the system $X-Y \ge 0 \wedge Y-Z \ge 0 \wedge -X+Z \ge 0$, and form P -X→ PX -Y→ PY -Z→ PZ:

PX = $Y-Z \ge 0 \wedge -Y+Z \ge 0$

PY = $0 \ge 0$ ('true')

PZ = 'true'.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since PZ='true' P is satisfiable. Following the proof one can construct the point X=Y=Z=0 which satisfies P. This is an integer point so P is integer satisfiable.

On the other hand, consider P to be

$$-4*X-Y+8\geq 0 \wedge -X+4*Y\geq 0 \wedge 4*X-Y-4\geq 0$$

and form P -X→ PX -Y→ PY:

$$PX = -2*Y+4\geq 0 \wedge 15*Y-4\geq 0$$

$$PY = 52\geq 0 \quad \text{('true')}.$$

Again since PY='true', P is satisfiable. We can exhibit a solution by working backward. The bounds on Y are:

$$e= 2 \geq Y \geq 4/15 =f.$$

Choose, say Y=e=2, and then P becomes

$$-4*X-2+8\geq 0 \wedge -X+4*2\geq 0 \wedge 4*X-2-4\geq 0$$

which forms bounds on X:

$$3/2\geq X \wedge 8\geq X \wedge X\geq 3/2 \quad \text{or}$$

$$e= 3/2 \geq X \geq 3/2 =f.$$

X must equal 3/2. The point X=3/2, Y=2 satisfies P. Suppose we now try to pick an integer solution. Since $2\geq Y\geq 4/15$ there are two possible choices for Y, 1 and 2. Try 2. From before $3/2\geq X\geq 3/2$, so there is no integer value for X when Y=2. Try Y=1. P becomes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$7/4 \geq X \wedge 4 \geq X \wedge X \geq 5/4.$

The bounds on X are then

$$7/4 \geq X \geq 5/4,$$

but there is no integer value for X in this case either so P is satisfiable yet integer unsatisfiable.

This last example suggests a technique to use for the integral problem. As before, suppose P is a system involving x1,x2,...,xn and form P -x1→ P1 -x2→ P2 ... -xn→ Pn. When Pn='false', P is integer unsatisfiable. If Pn='true' one could simply work backward and try to form an integer solution showing P is integer satisfiable or exhaust all possibilities and show it is not. The latter is what was done in the example above. However, "all possibilities" may be difficult to exhaust. Consider the previous example but with the additional inequality $Z \geq 0$. Form P -X→ PX -Y→ PY -Z→ PZ:

$P = -4*X-Y+8 \geq 0 \wedge -X+4*Y \geq 0 \wedge 4*X-Y-4 \geq 0 \wedge Z \geq 0$

$PX = -2*Y+4 \geq 0 \wedge 15*Y-4 \geq 0 \wedge Z \geq 0$

$PY = Z \geq 0$

PZ = empty≡'true'.

Working backward in this case we first find an integer satisfying (+ infinity)≥Z≥0. Pick Z=0 and then by going back to Y and then to X one finds that there are no integer values which can be assigned to them. Try Z=1. Again there

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are no legal X or Y values. Since Z is unbounded one cannot prove that this system is integer unsatisfiable by enumerating in this way, yet it is.

When the set of 'real' points satisfying a system is bounded for each variable then none of the bounds developed in this backward solving will be infinite and all possible combinations could, therefore, be exhausted. Using this hypothesis Cook and Cooper [4] have applied this method to the solution of integer linear programming problems. Our linear solver represents a different method which still suffers from this same restriction but is much better suited to the needs of the theorem prover. It concentrates on extracting as much information as possible from the linear system to be applied to the non-linear part. It can fail to discover that a linear system is contradictory in some unusual cases when the system is satisfied by an unbounded set of real points yet has no integer solutions.

The theorem prover divides the set of relational expressions into two sets: the linear relations and the non-linear ones. This characterization of the division is not precise since all $\neq$ relations are put into the set called non-linear whether they are, in fact, linear or not. The linear solver deals primarily with linear inequalities ($\geq$) and in order to deal with an expression like $S \neq 0$ it would have to be converted to two inequalities by the identity

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$S \neq 0 \equiv (S-1 \geq 0 \vee S+1 \leq 0)$.

This forms two cases for the solver. To avoid a possible proliferation of such cases, these inequalities are simply left out of the linear part and dealt with in the same way as the non-linear formulas.

Relations like $S \neq 0$ are the least significant, individually, with respect to producing contradictions in the sense that only one value for the expression S gives 'false' (namely 0) and an infinite set of values each give 'true'. Equalities, on the other hand, are the most powerful since all but one value of its expression give 'false'. Inequalities ($\leq$ and $\geq$) split the integers in half on the basis of yielding 'true' or 'false'.

Now consider a "linear system" to be a conjunction of relational expressions each of which has one of the forms $S \geq 0$, $S \leq 0$, or $S = 0$. Including both forms $S \geq 0$ and $S \leq 0$ is redundant but they are used by the formula manipulating routines so that the expressions can be normalized by making the leading term positive. The introduction of expressions in the form $S = 0$ is the only difference from the linear systems discussed earlier. We now describe two operations which are denoted by O1 and O2.

O1:

O1 operates on a linear system L and does the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

repeated elimination of variables by the -x→ operation. This is done in the same way as defined earlier except for two important differences:

1) L may contain equalities of the form S=0. These are remembered on a list called 'B' but are otherwise ignored.

2) If the result of a cross multiplication is a constant, we evaluate the resulting relational expression to 'true' or 'false'. If we get 'false', L is integer unsatisfiable and we stop. If the result is 'true', the two relaticnal expressions involved in the cross multiplication form bounds for some linear combination of variables. That is, there must have been expressions of the form:

$$a1*x1+a2*x2+...+an*xn+b\geq 0$$

$$a1*x1+a2*x2+...+an*xn+c\leq 0$$

so that

$$-c\geq a1*x1+a2*x2+...+an*xn\geq -b.$$

When this occurs the expression and its bounds are remembered on the list B mentioned in 1). An equality is a special case of this bounded form when c=b.

The bounds list B is initially empty and accumulates all equalities originally in L as well as all bounded

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expressions discovered in the processing. Note that all systems like L are maintained as a conjunction of relational expressions using the normal form and are processed by the standard basic routines. As a result a certain amount of simplification occurs automatically. An expression like ($S \geq 0 \wedge S \leq 0$) would simplify to $S=0$. The cross multiplication and formation of a bounded expression is performed, in a sense, automatically. For this reason we move equalities to the list B in each new system as well as in the original L. Recall that the simplification also reduces inequalities by constant common factors and eliminates redundant inequalities. One additional note is, that, since both forms $S \geq 0$ and $S \leq 0$ are allowed, when scanning for candidates to cross multiply, one must examine not only the sign of the coefficient of the variable but also the sense of the inequality. We use the word "parity" to refer to these combined properties of a variable. Thus $Y+Z \geq 0$ and $Y-Z \leq 0$ have the same parity for Z but opposite parity for Y.

The operation O1 accepts a linear system L involving inequalities of the form $S \geq 0$, $S \leq 0$, or $S=0$ and from this either:

1) deduces a contradiction proving that L is integer unsatisfiable or

2) fails to produce a contradiction but produces a non-empty list B of bounded linear expressions composed from the variables of L or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3) fails to do either (B is empty).

Case (1) represents a successful conclusion, and the next operation, O2, is designed to work essentially on case (2). Case (3) is discussed later.

O2:

O2 operates on a system composed of three parts: a linear part L, a non-linear part NL, and a linear equality of the form:

a1*x1+a2*x2+...+an*xn+b=0.

This equality can be used as before to eliminate a variable in the system. However, none of the ai ($1 \leq i \leq n$) may happen to be +1 or -1 which would allow the equation to be solved for a lone variable. If there is an ai=+1 or -1 we solve for xi and substitute this expression throughout the system eliminating xi completely. The linear part will remain linear but the simplification of the non-linear part may result in a new linear expression which we then move to the linear part. This completes the operation O2 in this case.

If no coefficient in the equality is +1 or -1 we solve for a1*x1:

a1*x1 = -a2*x2-a3*x3-...-an*xn-b.

This equation can be used to eliminate x1 from all the 'linear' relations by multiplying each through by |a1| and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

then substituting for |a1|*x1 as a unit. Since |a1| is strictly positive the multiplication is valid for expressions in the form S≥0, S≤0, S=0, or S≠0. In general, this kind of substitution cannot be done in non-linear equations since x1 may not occur in a position where it may be converted to the form |a1|*x1. For example, consider X in

Y↑X≥0.

Substitution is done into the expressions in NL for the first level terms in which |a1|*x1 can be formed. This means that x1 is also completely eliminated in the linear ≠ relations (which happen to occur in NL).

The equality not only defines a value for x1 in terms of the other variables but also indicates that -a2*x2-a3*x3-...-an*xn-b must be an integral multiple of a1. If, after the substitution, the equation is discarded this additional information is lost. For that reason the list of "equalities to remember" is created. This list is a special part of NL and any subsequent substitutions made in the system are also made into these equations. Some contradictions to the multiplicative property of these expressions can be detected during such substitutions. For example, if the equalities to remember contain:

5*x1 + 3*x2 + 1 =0

which had been used to eliminate 5*x1, and subsequent

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processing developed the equality:

$$6*x2 + 5*x3 = 0,$$

then using this to eliminate 6*x2 throughout the system would result in substituting in the list of equalities to remember and

$$5*x1 + 3*x2 + 1 = 0$$

would become

$$10*x1 + 5*x3 + 1 = 0.$$

This would be reduced to 'false' by the basic routines since 5 divides (10*x1 + 5*x3) but does not divide 1. If this ever occurs we can immediately stop processing this case, having found a contradiction. Thus the over-all effect of O2 is to take a system and a linear equality and "absorb" the equality into the system by substitution.

How O1 and O2 are combined into an over-all procedure is now discussed. Given the original system composed of L, the linear part, and NL, the non-linear part, first apply O1 to L. If this results in a contradiction the system is unsatisfiable and we quit successfully. If no contradiction is found and no bounds are deduced we quit in failure. When the bounds list B is nonempty it contains expressions in the form:

$$b \geq A \geq c$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where b and c are integer constants and A is some linear expression. An equality is the special case when b=c. Such an expression can be considered in (b-c+1) cases:

$$A=b \vee A=b-1 \vee \ldots \vee A=c+1 \vee A=c.$$

Considering each element on the list B in this way, choose the one which breaks into the fewest cases, (i.e., its (b-c) value is smallest). If two offer the same minimum number of cases just choose the first one.

Subproblems are generated, one for each case of this bounded expression, all of which we must prove to be contradictory in order for the original system to do so. For each subproblem both parts L and NL are inherited from the original system and using the equality defining this subproblem, O2 is applied to get a new linear part L' and a new non-linear part NL'. NL' may also now include a list of equalities to remember. If this substitution causes any of the new component parts of the system to reduce to 'false' then this case is finished, successfully. If not we then repeat the whole process for this new system, submitting L' to O1, and then form a subproblem for each result using O2, etc.

In general, a tree-like structure of systems is formed by applying O2 to a finite set of cases which were determined by O1. Each path in this structure terminates since, in the worst case, O2 reduces the number of variables

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the linear part and as a consequence O1 will eventually be applied to a system with no variables. Each terminal system on the paths of this structure (the leaves) will either be contradictory or represent a "give up" situation. If all the final systems are contradictory then the original system is contradictory, as well, and the given verification condition has been proved.

If any of the systems are terminal because they represent the "give up" output of operator O1 (case 3. above), then the integer satisfiability of the original system may still be unresolved. Such a terminal system, T, is related to the original system, L, by the equalities used at each application of operator O2. Any system, like T, which results in "give up" when submitted to O1, has an infinite number of integer solutions. The trouble arises in now using one of these solutions of T together with the equalities relating it to L to determine an integer solution for L. Each equation in the list of "equalities to remember" requires that a certain linear combination of variables of T be a multiple of a given constant. For example, the system consisting of the single inequality:

$$x3 \geq 0$$

would make O1 give up and has an infinite number of integer solutions. It could have been derived from a larger system L by the equations:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$2*x1 + 5*x3 + 1 = 0$ used to eliminate $2*x1$ and

$2*x2 + 3*x3 + 2 = 0$ used to eliminate $2*x2$.

These equations would be on the list of "equalities to remember". The first equation requires that $[5*x3 + 1]$ be an even number $(-2*x1)$ and the second requires that $[3*x3 + 2]$ be even as well $(-2*x2)$, but no integer x3 can satisfy both of these constraints. Although it is possible to satisfy $x3 \geq 0$, one could not construct an integer point satisfying the original L.

We would expect that the cases which this linear solver leaves unresolved would be rather rare. Alternative methods and improvements are discussed in the next chapter (on limitations and extensions).

Quantifiers

This completes the discussion of the proof method used for quantifier free verification conditions. The bulk of our work has been done on that problem. In this section we discuss one heuristic used for a special case of quantified expressions and suggest how this idea may be generalized. The special case arises from noting that a simple program loop, which forms the iteration of some process, is a very common construct in programs. Many programming languages incorporate particular facilities for this task (e.g., the 'DO' statement of Fortran and the 'FOR'

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

statement of Algol). An inductive predicate for such a loop will usually involve a quantified variable. For example, a simple program to zero the array A is given as example 5 in Appendix II and is reproduced here:

```
BEGIN |      ZERO THE ARRAY A[1:N]
    ASSERT(TRUE);

    I ← 1;
L:
    ASSERT( ∀J( (1≤J ∧ J<I) ⊃ A[J]=0) );

    IF I≤N THEN
    BEGIN
        A[I] ← 0;
        I ← I + 1;
        GO TO L;
    END;

    ASSERT( ∀J( (1≤J ∧ J≤N) ⊃ A[J]=0) );
END;
```

The statement "A[I]←0" could be nearly any statement or sequence of statements which establishes the I-th case of some property which is reflected in the quantified predicates (in this example, the property A[I] = 0). The verification conditions for such a program would have the form:

$$\vdash \forall I\ P(I) \supset \forall J\ Q(J)$$

where no other quantifiers occur in P and Q. This can be rewritten as:

$$\vdash \forall J \exists I (P(I) \supset Q(J))$$

or using the free variable notation, as simply:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\vdash \exists I(P(I) \supset Q(J))$.

The problem is to exhibit an I, say i, for which

$\vdash P(i) \supset Q(J)$.

In the special context in which this formula was derived a reasonable first try would be to choose I=J, so instead of proving

$\vdash \forall I\ P(I) \supset \forall J\ Q(J)$

one tries

$\vdash P(J) \supset Q(J)$

which is a quantifier free problem.

This is a heuristic since there is no guarantee that it will lead to a proof of the original theorem. For a theorem prover in general this would not be a useful heuristic but in our special application it is a good first-try method. This is the only facility for manipulating quantified theorems which is presently coded in the program verifier, yet the above example and several others of a similar form have been successfully verified by it (see examples 4, 5, 6, and 7 in Appendix II). The quantifiers used in assertions for these programs are not a general use of quantification as one might find in a mathematical theory. They usually quantify a variable which is bounded by the program variables, and they correspond to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

iterative events in the program. One would hope that special heuristics could be developed to handle these special uses.

In the previous example, one could change the last assertion

ASSERT(∀J((1≤J∧J≤N) ⊃ A[J]=0))

to the equivalent

ASSERT(∀J((2≤J∧J≤N+1) ⊃ A[J-1]=0))

and then our specific heuristic would fail for the verification condition involving this assertion. One might still consider the over-all approach of finding a suitable i to substitute for I such that

⊢P(i) ⊃ Q(J).

In this case the choice should be I=J-1 instead of I=J. The example suggests generalization of this simple heuristic. Human theorem provers would readily arrive at the choice I=J-1 from the clue given by matching A[I]=0 with A[J-1]=0. One would expect that this sort of insight could be incorporated into an automated theorem prover.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER III: SYSTEM PERFORMANCE, LIMITATIONS, AND EXTENSIONS

General Discussion

The steps necessary to verify a program in the manner discussed here are:

1) A program is written.

2) The programmer must define correctness for that program by supplying the initial and final predicates used in the verification.

3) The programmer must provide a set of inductive predicates and associate each with a statement in the program.

4) The program and its predicates are translated into a form in which the tagged paths can be analyzed.

5) For each tagged path a verification condition is derived.

6) The theorem prover is applied to each verification condition to prove its validity.

In our work, step (1) was not examined except to observe that a programmer will soon determine a style of programming which simplifies the remaining steps. The problem of formally defining the predicates of step (2) and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

particularly those of step (3) is formidable. Being able to devise an adequate language for expressing these predicates and then, for any given program, being able to determine where to assert what, will ultimately determine the feasibility of this approach. The choice of predicate language also determines what formula manipulation facilities are required and establishes the universe of discourse for the theorem prover. Our choice of predicate language drastically limited the number and kind of programs available for study.

A discussion of some ways for increasing the expressive power of the assertions is presented later. Preliminary work has been done by Floyd in order to develop a method for deriving inductive predicates based on their invariant properties. This appears to be a difficult problem and the creation of predicates remains an art learned by experience. Step (4), generating an internal flowchart for a program, is straightforward. This step did not create any significant design or performance problems in our system and we expect this also to be true for future systems.

Generating the verification conditions from the tagged paths (step 5) consists of repeatedly transforming the assertion in hand. These transformations are determined by the semantic definition of the programming language. In our system, they consist of making substitutions and forming

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implications. Implication is a basic operator in the predicate language and gives no trouble. On the other hand, repeated substitutions into an expression can cause it to become more complex. This effects the performance of the system because any subsequent processing of the expression takes longer and the expression consumes more memory space. For this reason our system simplifies expressions with every operation. While this simplification takes time it appears to pay for itself in subsequent processing time, particularly in the theorem proving step.

Our simplifications do not prevent expressions from becoming large and complex after repeated substitutions for array elements. Example 9 in Appendix II exhibits this problem. The difficulty is discussed at length later in this chapter. We also discuss means of increasing the power of the simplification process.

Adding more sophisticated features to the programming language (e.g., recursive procedures) requires more complex transformations to be performed on the assertion in hand but should not significantly decrease the performance of the system. This is also discussed later.

Proof of the verification conditions (in step 6) is dependent on the program being verified. The previous steps performed by the automatic verifier (4 and 5) are well-defined once the programming language and predicate language are determined. The requirements of the theorem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

prover are highly dependent upon the computation to be performed by the specific program being verified. Genuine open-endedness must be provided here. Our theorem prover does not satisfy this criterion and that severly limits the systems applicability. It is specifically tailored to handle elementary integer functions well, but has only limited power beyond that. The subsequent discussion also suggests means for generalizing the theorem prover.

In general, our system performs extremely well on the small class of programs to which it is applicable. The remaining part of this chapter discusses how its applicability can be increased while trying to maintain high efficiency. We see no major hurdle which prevents the system from being continually upgraded by removing the limitations.

Simplification

We begin by discussing an approach for improving the simplification of logical expressions. Formula simplification contributes much to the capabilities of the theorem prover. If a canonical form were available for our general expressions there would be no need for a theorem prover at all, since all theorems would reduce to the canonical form 'true'. The normal form of the current system is quite good but any additional improvements would

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

upgrade the capabilities of the theorem prover directly. The class of expressions dealt with by the system encourages one to focus on two aspects of their representation: the form for arithmetic expressions which compose the relational expressions and the form in which these are combined to make logical expressions.

Some recent work ([3],[31]) has been done in finding canonical forms for different classes of arithmetic expressions as well as proving that no effective canonical forms are possible for other classes. More work in this area needs to be done. Our normal form for arithmetic expressions is canonical for the subclass comprising polynomial expressions but we were unable to show that it is canonical for arbitrary expressions involving +, -, * and ↑. The special functions ÷, mod, and abs are eliminated in the theorem prover resulting in expressions involving only +, -, * and ↑, so a canonical form for this class of expressions would be desirable.

For the logical expressions, at least one more simplification operation seems feasible. Considerable space in the literature has been devoted to minimization of logical expressions of propositional calculus. (For example, see the bibliography at the end of Chapter 4 in [2]). Most of this work was done in the framework of designing minimum electronic circuits. Our subsumption operation is an adaption of that used in these minimization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

procedures and the complete minimization methods should be adaptable in an analogous way. Instead of considering each distinct relational expression to be a unique Boolean primitive, the subsumption operation deals with them in the form (S + c {R} 0) where (S+c) is a normalized sum, c is the constant term, and {R} is $\geq$, $\leq$, =, or $\neq$. Expressions with the same S parts interact even though they may not be identical or complementary Boolean primitives. This same approach could be used to minimize our d.n.f. expressions. For example, one operation basic to some of these minimization methods is to determine whether any disjunct is redundant, and consequently may be dropped. The subsumption operation detects this redundancy when considering pairs of disjuncts, say b and c, by testing whether $b \supset c$ or $c \supset b$. However, if one has the expression:

$$(\sim c \wedge d \wedge e) \vee (a \wedge b \wedge d \wedge e) \vee (a \wedge b \wedge c)$$

no redundancy can be detected by the pairwise examination of the three disjuncts. Although neither

$$(a \wedge b \wedge d \wedge e) \supset (\sim c \wedge d \wedge e) \quad \text{nor}$$

$$(a \wedge b \wedge d \wedge e) \supset (a \wedge b \wedge c)$$

is a tautology, the expression

$$(a \wedge b \wedge d \wedge e) \supset [(\sim c \wedge d \wedge e) \vee (a \wedge b \wedge c)]$$

is a tautology. As in subsumption, this tautology implies that the sub-expression $(a \wedge b \wedge d \wedge e)$ is redundant and may be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

dropped from the original expression. One method used to show that the last implication is tautologous is by truth value analysis. Suppose (a∧b∧d∧e) were 'true', then a, b, d, and e must individually be 'true'. With this assumption, the expression [(~c∧d∧e) ∨ (a∧b∧c)] reduces to (~c∨c) which, after examining the two possible values of c, is found to be 'true'. Thus, the implication itself is always 'true'.

A corresponding process can be done on our logical expressions built from relaticnal expressions. For example, consider the expression

$$(x\geq 0 \wedge y\geq 0) \vee (x+1\geq 0 \wedge y\neq 0 \wedge y-1\neq 0) \vee (x+5\geq 0 \wedge y-4\leq 0)$$

where x and y are any normalized sums. This is not reduced any by our present normalization process, but the disjunct (x≥0 ∧ y≥0) is, in fact, redundant and could be eliminated. This can be detected by forming

$$(x\geq 0 \wedge y\geq 0) \supset [(x+1\geq 0 \wedge y\neq 0 \wedge y-1\neq 0) \vee (x+5\geq 0 \wedge y-4\leq 0)]$$

An analysis similar to that just demonstrated for the Boolean case shows this to be a tautology. Assume (x≥0 ∧ y≥0) is 'true'. These inequalities determine a domain for the expressions x:[0, +infinity] and y:[0, +infinity] over which the right hand formula [(x+1≥0 ∧ y≠0 ∧ y-1≠0) ∨ (x+5≥0 ∧ y-4≤0)] should be 'true'. Both x+1≥0 and x+5≥0 are universally 'true' over this domain but none of y≠0, y-1=0, or y-4≤0 is a constant value for y:[0, +infinity]. Using the constant values of those expressions which are constant

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

over the domain, the right hand expression becomes

$$[(y \neq 0 \wedge y-1 \neq 0) \vee (y-4 \leq 0)].$$

The truth of this can be resolved by considering cases of the domain for which each relation is a constant value. That is, consider the domain Y:[0, + infinity] in component parts: [0,0], [1,1], [2,4], and [5, +infinity]. In the first case, y=0 and $[(0 \neq 0 \wedge -1 \neq 0) \vee (0-4 \leq 0)]$='true'. For y=1, $[(1 \neq 0 \wedge 1-1 \neq 0) \vee (1-4 \leq 0)]$='true'. For y:[2,4] the expression is [('true' ∧ 'true') ∨ 'true'] = 'true', and finally, for y:[5, +infinity] the expression becomes [('true' ∧ 'true') ∨ 'false'] = 'true'. For the domain determined by the left side of the implication the right hand side is always 'true' and therefore so is the implication itself.

Note that what was a truth value analysis in the Boolean case becomes a matter of considering subdomains over which fixed truth values can again be determined. This discussion was meant to indicate an approach to altering the known Boolean minimization methods to work with relational expressions. What remains to be done is to find a method which can be adapted into an 'efficient' counterpart. Any significant improvement in the basic formula simplification process may also automatically improve the theorem prover.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem Prover

One thing to note about the theorem prover is that it involves no heuristics. It operates as a fixed sequence of steps. Each step is performed in turn and either the theorem is successfully proved at some step or else the list of steps is exhausted, resulting in failure. More complex theorems can be proved by adding new processes to handle them to the end of the list. This approach can be very successful provided the class of theorems one can prove is sufficiently rich.

Although the linear solver coded here will not decide the integer satisfiability question of all linear systems, the question is decidable. Presburger [30] designed a decision procedure for this class of problems; in fact, his procedure also operates on expressions whose variables may be quantified either existentially or universally. To do this it operates by the same principle as Tarski's decision method for real algebra [32, 28]. The idea is simple and could be used in proving quantified theorems. All $\forall$ quantifiers are converted to $\exists$ form by use of the identity $\vdash \forall x\, A(x) \equiv \vdash \sim\exists x \sim A(x)$. The process is applied from the innermost quantifier to the outermost. If we have $\exists x A(x,y,z)$ where A is quantifier free, the problem is to find rules which will produce a $B(y,z)$ such that B is quantifier free and $\vdash B(y,z) \equiv \vdash \exists x A(x,y,z)$. That is, can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we devise conditions (namely B(y,z)) on y and z which are true if and only if there is an x satisfying the original conditions (namely A(x,y,z))? By successively reducing the number of variables we eventually get a variable-free expression which is either 'true' or 'false'. Presburger's procedure is discussed in [14] (pages 359 - 366), and Davis [8] programmed a version of it for a computer. However, the methods, as they stand, are not very practical to code for efficient use.

The linear solver described in Chapter II could be made into a decision procedure simply by "backing it up" with Presburger's method. This is not as redundant as it may first appear. The linear solver is rather efficient and straightforward. In the case that it fails to make a decision, the system has been reduced to the special form:

$$g_i(x_k,\ldots,x_n) \geq 0 \quad \text{one for each } i \ (1 \leq i \leq m)$$

which is an integer satisfiable system over the variables xk, ..., xn. We also have a set of "equalities to remember" which must be shown to be satisfiable:

$$a_i * x_i + f_i(x_k,\ldots,x_n) = 0 \quad \text{one for each } i \ (1 \leq i \leq k-1)$$

where $|a_i|>1$ and gi and fi are linear expressions over the variables xk, ..., xn. A specialized version of Presburger's algorithm may prove efficient for this class of problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

From a practical viewpoint it may be wasteful to spend effort adding code to the linear solver just to make it complete for cases which may never arise in practice. It is perhaps wiser to consider broadening the "attackable" set of problems.

There are two additional ways for bringing the linear techniques to bear on the non-linear problem neither of which has been coded in the current system. The first is to treat all terms of arithmetic expressions as if they were individual variables and then interpret the whole problem as if it were strictly linear. For example,

$$x\uparrow y-1\geq 0 \wedge x\uparrow y+z\leq 0 \wedge z\geq 0$$

could be interpreted as

$$b=x\uparrow y \wedge [b-1\geq 0 \wedge b+z\leq 0 \wedge z\geq 0].$$

By considering the term $(x\uparrow y)$ as the simple variable b, the expression in brackets is a linear system which can be shown to be contradictory by the current prover.

The second method is to decompose non-linear expressions into an equivalent set of linear expressions by factoring out common variables, or more generally, common linear expressions. If one has an expression in which a variable (or linear expression), say x, occurs in all terms except, of course, the constant term, then x can be factored out to give an equation of the form:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$S*x + c \{R\} 0$

where S is a normalized sum, c an integer constant, and {R} one of ≥, ≤, =, or ≠. The general schemes are:

1) Suppose {R} is ≥. (Similar results hold for ≤.)

Testing the integer constant c, we perform one of two transformations. When $c \geq 0$ then $S*x+c \geq 0$ becomes the (2*c+3) cases on x:

$(x > c \wedge S \geq 0) \vee$

$(x < -c \wedge S \leq 0) \vee$

$(x = k \wedge k*S+c \geq 0)$ (for each integer k, $-c \leq k \leq c$)

When $c < 0$ then $S*x+c \geq 0$ becomes (2*c+1) cases on x:

$(x \geq -c \wedge S \geq 1) \vee$

$(x \leq c \wedge S \leq -1) \vee$

$(x = k \wedge k*S+c \geq 0)$ (for each integer k, $c < k < -c$)

2) Suppose {R} is ≠. Consider the set consisting of pairs of positive integer constants $\{(ni, mi) \mid i=1,2,\ldots,t\}$ which are all the positive integer factorizations of $|c|$, that is, for all i $(1 \leq i \leq t)$ $ni*mi = |c|$. If $c > 0$ then $S*x+c \neq 0$ becomes the (2*t+1) cases on x:

$(x = ni \wedge S \neq -mi)$ $i=1,2,\ldots,t,$

$(x = -ni \wedge S \neq mi)$ $i=1,2,\ldots,t,$

$(x \neq n1 \wedge x \neq -n1 \wedge x \neq n2 \wedge \ldots \wedge x \neq nt \wedge x \neq -nt).$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If c=0 then S*x≠0 beccmes one case:

$(x \neq 0 \wedge S \neq 0)$.

If c<0 then S*x+c≠0 becomes (2*t+1) cases on x:

$(x = ni \wedge S \neq mi) \quad i=1,2,\ldots,t,$

$(x = -ni \wedge S \neq -mi) \quad i=1,2,\ldots,t,$

$(x \neq n1 \wedge x \neq -n1 \wedge x \neq n2 \wedge \ldots \wedge x \neq nt \wedge x \neq -nt)$.

3) Suppose {R} is =. Let (ni,mi), i=1,2,...,t be defined as in 2). If c>0 then S*x+c=0 becomes (2*t) cases on x:

$(x = ni \wedge S = -mi) \quad i=1,2,\ldots,t,$

$(x = -ni \wedge S = mi) \quad i=1,2,\ldots,t.$

If c=0 then S*x=0 becomes two cases:

$(x=0) \vee$

$(S=0)$.

If c<0 then S*x+c=0 becomes (2*t) cases on x:

$(x = ni \wedge S = mi) \quad i=1,2,\ldots,t,$

$(x = -ni \wedge S = -mi) \quad i=1,2,\ldots,t.$

One important question is "how far can one extend the method like this to solve non-linear problems?" We now show that if one were able to extend the linear techniques

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to deal with arbitrary theorems involving polynomial expressions then we would have a solution to Hilbert's tenth problem, and conversely. Hilbert's tenth problem refers to one of the problems presented by David Hilbert in 1900. The problem is to decide whether an arbitrary polynomial (arbitrary with respect to degree and number of variables) with integral coefficients has integral roots. The problem is still unresolved (see [6,7] for a detailed discussion).

Any theorem at all can be generated as the verification condition of a "do nothing" program:

```
BEGIN
    ASSERT(TRUE);
    ASSERT(S);
END;
```

The verification condition for this program is

$\vdash$'true' $\supset$ S or simply $\vdash$S.

If S is drawn from the class of expressions of the form $Q \neq 0$, where Q is an arbitrary polynomial over integers, then any program verifier which could decide the correctness of 'all' such do nothing programs would also be a solution for Hilbert's tenth problem. $\vdash Q \neq 0$ is a theorem if there are no integral roots of Q and not a theorem, otherwise.

Conversely, if one had a decision procedure for Hilbert's problem then it could be used to verify programs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

whose verification conditions involve only polynomials. This can be seen by reducing the complemented form of any given expression to a single polynomial of the form (S=0) such that a solution to that equation would also give a solution to the original expression. The reduction can be done by repeated application of the rules:

1) Any expression $S \neq 0$ is equivalent to $(S-1 \geq 0 \wedge -S-1 \geq 0)$.

2) Any expression $S(X) \geq 0$ is satisfiable if and only if $[S(X) - a*a - b*b - c*c - d*d = 0]$ is satisfiable (a, b, c and d are new integer variables). If $S(X) \geq 0$ is satisfied by X0 then S(X0) is an integer and there exist four integers a0, b0, c0, and d0 such that $S(X0) = a0*a0 + b0*b0 + c0*c0 + d0*d0$. (See [33], page 379 for a proof that any positive integer can be expressed as the sum of the squares of four integers.) On the other hand, if there are values X0, a0, b0, c0, and d0 such that $S(X0) - a0*a0 - b0*b0 - c0*c0 - d0*d0 = 0$ then certainly $S(X0) \geq 0$.

3) Any expression $(S=0 \vee R=0)$ is satisfiable if and only if $S*R=0$ is satisfiable. Note, too, that S*R is still a polynomial.

4) Any expression $(S=0 \wedge R=0)$ is satisfiable if and only if $S*S + R*R = 0$ is satisfiable. Again, note that this expression is a polynomial.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using these rules one may, in principle, reduce the complemented form of any verification condition involving polynomial expressions to one (usually much more complex) equation of the form S=0. One could then apply a decision procedure for Hilbert's tenth problem to that equation.

This gives a feel for exactly how hard it is to design a decision procedure for more general classes of formulas. After incorporating a few more general methods into the prover one should then consider how a procedure using heuristic methods could be developed with this system as a basis. One will most likely want to build a theorem prover which has some ability to deal with functions defined by axioms. With as much simplification as is done here, one may be able to reserve the more general axiomatic method for the points where simplification is no longer fruitful. One strong motivation to have some facility with functions defined by axioms comes from trying to improve the expressive power of the assertion language. We turn to this next.

Assertion Language

Another point noted earlier as a major limitation of the current verifier was the assertion language: the notation for writing assertion statements. A person using the program verifier should be able to use functions like

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"summation", "factorial", "greatest common divisor", etc. The current system has much of the knowledge useful for simplifying expressions involving +, -, *, and ↑ built-in in the form of operator routines. It is also able to "know" about ÷, mod, and abs because each has two routines associated with it. The first routine for each composes the function from its arguments. In this routine any special relationships between the given arguments which may result in simplifications can be detected. (e.g., in forming abs(-3) the fact that -3 is a constant is detected and the resultant output is simply +3 instead of abs(-3)). The second routine which each function has associated with it is a routine to eliminate an occurrence of the function by defining it in terms of the primitive operators +, -, * and ↑. These routines are used at one point in the theorem prover to eliminate ÷, mod, and abs.

It would be a straightforward process for someone familiar with the system to write this pair of routines for other functions and cause them to be built-in as well. The difficulty with this approach would be that in order to give the system a general scope one would have to have hundreds of functions built-in. In fact, it would probably be impossible to keep up with the demand for new functions since they are used to express the open-ended semantics of programs. The natural solution to this problem is to supply the user of the program verifier with a simple notation in which he could describe what the two routines for each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

function should look like. Then the system would absorb these into itself and operate as if they were built-in for that run.

As long as the routines for defining the functions always generate expressions only involving +, -, *, and ↑, no major changes need be made in the theorem prover. Let us call this the "macro definition" approach since one could consider the functions as a macro shorthand notation for functions actually definable in terms of the more primitive operators. Each occurrence of ↑ can be thought of as a concise abreviation of the cumbersome set of equations which really define it.

Such macro definitions for functions are very convenient but not really sufficient. Many useful functions are not easily expressed, if at all, in terms of +, -, *, and ↑. One needs the ability to define functions axiomatically. That is, rather than define functions in terms of a set of basic operations, they are simply defined in terms of their properties. This approach does imply that the theorem prover must be modified in a fundamental way.

Instead of just forging ahead following a specific algorithm designed to deal with a specific set of operators, the theorem prover must decide exactly where, when, and how to apply the axioms to the theorem candidate. Avoiding this problem is, perhaps, one of the virtues of the existing theorem prover. It operates directly and efficiently on the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

theorems it can prove. It seems wasteful to approach the theorem proving problem with general axiomatic methods at the onset when so much can be done by powerful simplification and some specific algorithms. After having developed a system around specialized techniques it appears appropriate (and perhaps necessary) at this point to incorporate some heuristic axiomatic methods.

Human Assistance

One way to simplify some aspects of program verification is to involve the user more directly in the processing. This would seem to be a natural application for the current widely discussed "conversational programs". Although this allows one to avoid many of the more complex issues by simply "letting the human solve the problems which require genuine intellegence", a new class of problems arises which may be just as difficult. How does one design a program to cooperate with and to converse with a man?

The simplest concept of a conversational verifier would be for it to generate the verification conditions and then simply present them to the human component for proof. A glance at some of the more complex conditions generated by the examples in Appendix II will quickly discourage one from this simple approach. One can imagine the shock to a user who, sitting at his remote computer terminal, recieves the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

message "Is the following statement true:" followed by 30 lines of formula.

In a conversational verifier the machine should do just as much of the tedious clerical work as possible. For this reason, we feel that our system offers a good basis for this area of development as well. The simplifications and other well-defined operations should be handled by the machine. The human component of the system may be just the element necessary to deal with the proper application of axioms. Of course, as indicated above, just how this help can be used most efficiently and most conveniently for the human is an open problem.

Debugging

When one considers the procedure of verifying a program as a debugging process, a different set of extensions to the system come to mind. The user is interested in the failures of the verification process. Our discussion and indeed the prototype verifier itself emphasizes the positive aspect of the process. The work should be reviewed to discover what additions or changes could be made to aid in debugging programs.

When a verification condition is found not to be a theorem, one usually is able to exhibit a set of values for the variables which make it evaluate to 'false'. The linear

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

solver in our prover should be modified to produce a counter-example set of values whenever the proof fails. These values can be used to form a particular state vector for some point in the program where the program and assertions disagree. A verifier which was able to construct such counter-examples for erroneous programs would be an extremely useful debugging aid. Other useful aids would also evolve from careful consideration of the whole process with debugging in mind.

Arrays

Another weakness of the current verifier is its treatment of programs involving arrays. In theory this is a solved problem but in practice it offers difficulties. The way our verification condition generator deals with assignment statements, where the left hand side is an array reference, is unsatisfactory. This is shown vividly in example 10 of Appendix II. The difficulty occurs because such an assignment statement is actually a scheme representing a (possibly infinite) class of simple assignment statements.

For example, the assignment statement

```
A[I] ← E;
```

represents a different assignment depending on the value of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

I. It represents all of the simpler statements:

```
A[1] ← E;
A[100] ← E;
A[-4] ← E;
```

depending on the particular value of I. This additional degree of variability requires special techniques. The method the verifier employs (explained previously in Chapter II) generates cases based on the different subscript expressions of that same array occuring in the "current" assertion. Two cases are created for each such occurrence. If an assertion has say 4 occurrences, this results in 2↑4 or 16 cases being generated. Most of these 16 cases will, in turn, contain references to that same array and if another assignment statement for that array occurs we then get, in the worst case, 2↑16 new cases.

Each case must be included for the method to be complete but many of them represent situations which could never occur, and may be dropped as redundant. For example, consider the statement sequence of example 6 in Appendix II which exchanges the values of the array elements A[I] and A[I-1]:

```
X ← A[I];
A[I] ← A[I-1];
A[I-1] ← X;
```

We are also given the assertion

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\forall K\,(\,(1 \le K \wedge K < I) \supset A[I] \ge A[K]\,) \wedge I \le N$$

which is assumed to be true immediately after the last statement of the sequence. By simply realizing that the statements exchange the values of A[I] and A[I-1] we know that

$$\forall K\,(\,((1 \le K \wedge K < I-1) \supset A[I-1] \ge A[K]) \wedge I \le N \qquad (*)$$
$$\wedge\ A[I-1] \ge A[I]\,)$$

must have been true before these three statements were executed.

This is not a difficult derivation to do by hand but when one considers how to accomplish an equivalent result by an automated technique he realizes how much processing a human can do so easily. Doing the same example in a careful step by step manner is enlightening. Starting with the assertion

$$\forall K\,(\,(1 \le K \wedge K < I) \supset A[I] \ge A[K]\,) \wedge I \le N$$

and moving backward over the statement

$$A[I-1] \leftarrow X;$$

requires the substitution of X for A[I-1] in the assertion. There are two possible places for substitution to occur, namely A[I] and A[K]. The human processor would immediately dismiss the case A[I] since he "knows" that A[I-1] and A[I] can never match. It is more difficult for a program to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"know" such information. Carefully applying the rules one would get:

$$\forall K(\ ((1\le K \wedge K<I \wedge I=I-1 \wedge K=I-1) \supset X\ge X)\ \wedge$$
$$((1\le K \wedge K<I \wedge I=I-1 \wedge K\ne I-1) \supset X\ge A[K])\ \wedge$$
$$((1\le K \wedge K<I \wedge I\ne I-1 \wedge K=I-1) \supset A[I]\ge X)\ \wedge$$
$$((1\le K \wedge K<I \wedge I\ne I-1 \wedge K\ne I-1) \supset A[I]\ge A[K])\ )\ \wedge$$
$$I\le N.$$

The first two cases have I=I-1 which simplifies to 'false' and would be dropped. The remaining two cases simplify to:

$$\forall K(\ ((1\le K \wedge K=I-1) \supset A[I]\ge X)\ \wedge$$
$$((1\le K \wedge K<I-1) \supset A[I]\ge A[K])\ )\ \wedge\ I\le N.$$

The next statement A[I] ← A[I-1] dictates the substitution of A[I-1] for A[I]. This is simpler since two references to the array A match A[I] exactly. The third occurrence, A[K], could not possibly match A[I] since K<I-1, but proceding mechanically one gets:

$$\forall K(\ ((1\le K \wedge K=I-1) \supset A[I-1]\ge X)\ \wedge$$
$$((1\le K \wedge K<I-1 \wedge K=I) \supset A[I-1]\ge A[I-1])\ \wedge$$
$$((1\le K \wedge K<I-1 \wedge K\ne I) \supset A[I-1]\ge A[K])\ )\ \wedge$$
$$I\le N.$$

Again, this simplifies to

$$\forall K(\ ((1\le K \wedge K=I-1) \supset A[I-1]\ge X)\ \wedge$$
$$((1\le K \wedge K<I-1) \supset A[I-1]\ge A[K])\ )\ \wedge\ I\le N.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The statement X ← A[I] indicates the straightforward substitution of A[I] for X getting:

$$\forall K(\ ((1 \leq K \wedge K = I-1) \supset A[I-1] \geq A[I])\ \wedge$$
$$((1 \leq K \wedge K < I-1) \supset A[I-1] \geq A[K])\ )\ \wedge\ I \leq N.$$

One might note that since the formulas here were not in d.n.f. the result looks slightly different from that developed by the program itself. We chose not to give the example in d.n.f. because the notation involving implication is easier to follow.

All but the most astute reader will have missed the fact that the original expression (*) which was simply written down and asserted to be true is, in fact, incorrect. One can see this by comparing it with the expression created more rigorously. This is one good reason why a mechanization of this process is desirable. Simple, yet critical errors are very easy to make by hand.

This example was presented to show in detail what is involved in dealing with arrays. We wish to emphasize the complexity caused by examining the cases over subscript expressions as well as the fact that (at least for this example) roughly one half of those cases are trivial and are dropped during simplification. The problem, which the current system handles by introducing the cases, is inherent in this operation and any alternative method must also cope

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with it.

This is possibly the most serious weakness of the whole system. Many interesting programs involving arrays simply could not be handled even though most verification conditions which would have been generated, could be proved by the existing theorem prover. There appear to be two avenues for attacking this problem. One is to improve the simplification techniques so that as many redundant cases as possible are eliminated. The improvements for formula simplification suggested above should help. The other approach is a notational convenience which eliminates the immediate proliferation of cases by pushing the problem off to the simplification routines and theorem prover.

Introduce the two functions "acc" (access) and "chng" (change) used by McCarthy [26]. An array, say A, is considered to be a single value which represents some coding of all the information contained in the array in such a way that the function acc(A,I) represents extracting the value A[I] (i.e., A[I] ≡ acc(A,I)) and the function chng(A,I,X) results in a new coded value representing the array A with the change A[I] ← X incorporated into it.

In this notation the statement

```
A[I] ← A[J] + 2;
```

would be written

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A ← chng(A,I,acc(A,J)+2);

This results in assignment statements over simple variables and eliminates the problem of array assignments in favor of introducing these new functions chng and acc. Of course, the user of the system would still be allowed to write

A[I] ← A[J] + 2;

and the processor for converting the program to an internal representation would introduce the functions chng and acc in the appropriate way.

Now there is a new problem: how does one deal with chng and acc? The following axioms allow us to work with these functions. Let A be an array, X an expression, and i and j distinct integer constants, then

1) acc(chng(A,j,X),i) = acc(A,i)

2) acc(chng(A,j,X),j) = X.

Suppose we re-do the previous example using this notation. The program segment would then be:

X ← acc(A,I);
A ← chng(A,I,acc(A,I-1));
A ← chng(A,I-1,X);

The given assertion would be written:

$\forall K ((1 \leq K \wedge K < I) \supset acc(A,I) \geq acc(A,K)) \wedge I \leq N.$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Performing the substitution indicated by the statement A←chng(A,I-1,X) we get:

$$\forall K((1 \leq K \wedge K < I) \supset acc(chng(A,I-1,X),I) \geq acc(chng(A,I-1,X),K)) \wedge I \leq N.$$

Since I and I-1 are always distinct integers, acc(chng(A,I-1,X),I) can be simplified to acc(A,I) by using axiom 2). The expression is then

$$\forall K((1 \leq K \wedge K < I) \supset acc(A,I) \geq acc(chng(A,I-1,X),K)) \wedge I \leq N.$$

Passing back over the statement A ← chng(A,I,acc(A,I-1)) we have:

$$\forall K((1 \leq K \wedge K < I) \supset acc(chng(A,I,acc(A,I-1)),I) \geq acc(chng(chng(A,I,acc(A,I-1)),I-1,X),K)) \wedge I \leq N.$$

Again using the axioms to simplify:

$$\forall K((1 \leq K \wedge K < I) \supset acc(A,I-1) \geq acc(chng(chng(A,I,acc(A,I-1)),I-1,X),K)) \wedge I \leq N.$$

The statement X ← acc(A,I) finally gives:

$$\forall K((1 \leq K \wedge K < I) \supset acc(A,I-1) \geq acc(chng(chng(A,I,acc(A,I-1)),I-1,acc(A,I)),K)) \wedge I \leq N.$$

Except for the continued nesting of functions this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

notation keeps the expressions simpler; at least the exponential growth exhibited in the other method does not occur. The inherent problem is still present, since if one tries to apply the axioms to simplify, say

acc(chng(Q,I-1,acc(A,I)) ,K)

one must again do the same case analysis:

[I-1=K ⊃ acc(A,I)] ∧
[I-1≠K ⊃ acc(Q,K)].

This notation does allow one to choose exactly when he wishes to apply the axioms and to what expressions, whereas the other method requires that the cases be generated immediately. Note that the same simplifications were performed as the expressions were processed. If one devises techniques to cope with the new functions acc and chng then this method would seem to offer more hope than that presently coded in the verifier.

Language Extensions

In the previous discussion we have been concerned primarily with improvements to the current system. This system operates on programs written in a simple integer programming language. It is also of interest to explore extending these techniques to more complex programming languages. We feel that the basic ideas are general enough

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to allow one, at least in principle, to prove the correctness of programs in, say, Algol 60. Most programming language features like real variables and double precision variables carry over exclusively to problems in the theorem prover. If there are different types of variables then the theorem prover must be cognizant of this when it performs deductions on them. This is not to say that these theorems can be proved as easily as those dealing with a single type but at least, in principle, what has to be done is clear.

Another similar problem is to relate carefully any operators in the programming language (e.g.,+, -, *) which indicate the performance of some basic machine operations to the corresponding functions (e.g., +, -, *) used in assertions to denote relations among variables. For example, in Algol 60, if one has an integer variable I and a real variable R and performs the assignment:

```
I := R;
```

the assignment operator (:=) has an implicit rounding function associated with it in this case. When this statement is used to determine a substitution in an assertion statement this rounding must be made explicit by substituting rnd(R) for I instead of simply R. The function rnd(R) is the result of rounding its argument R, and its interpretation is either built into the verifier or defined by axioms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the prototype verifier described here, assuming the programs are executed on a machine of infinite precision as we did, the relationship between these two notations is so immediate that it is no issue. If this assumption is dropped or if more complex constructs are allowed these notational inconsistencies must be made explicit.

Assuming all such issues of a notational nature have been resolved, one is still faced with the problem of devising appropriate "semantic definitions" for the various statement types. Our verifier deals with two statement types: assignment statements and if statements. How does one handle other constructs? If one can define the execution of a statement in terms of state vectors then he can also devise a definition in terms of the statement's effect on assertions. To see this, suppose the state of the machine is represented by the state vector v. Also suppose s is the statement of interest and P(v) is the assertion given before s. We want to find the rules for developing an assertion R(v) such that $P(v1) \Rightarrow R(v2)$, where v2 is the constant vector resulting from the execution of s on v1. In general, R(v) can always take the form:

$$R(v) = \exists v'[P(v') \wedge Cs(v,v')]$$

where v' represents the "old" values of the state vector and v represents the "new" values resulting from executing statement s. The component P(v') simply reflects the information known before the execution of s, and Cs(v,v')

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

must characterize the relationship between the old values and the new ones.

For example, suppose one has the assignment statement

x1 ← f(x1,x2,x3);

and is given P(x1,x2,x3) as the assertion 'true' immediately before the statement is executed, then

$$R(x1,x2,x3) = \exists x1'\exists x2'\exists x3'[P(x1',x2',x3') \wedge x1=f(x1',x2',x3') \wedge x2=x2' \wedge x3=x3'].$$

One may solve for and eliminate x2' and x3' getting

$$R(x1,x2,x3) = \exists x1'[P(x1',x2,x3) \wedge x1=f(x1',x2,x3)]$$

which is the expression one would have gotten from applying the definition for assignment statements given earlier.

If the language allowed a slightly more complicated expression like

x1 ← x2 ← f(x1,x2,x3);

which is interpreted to mean "assign the current value of f(x1,x2,x3) to both x1 and x2", then Cs(v,v'), in this case, would be

$$(x1=f(x1',x2',x3') \wedge x2=f(x1',x2',x3') \wedge x3=x3').$$

The even more complex construction like

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
x1 ← g(x1,x2,x3) + (x2←f(x1,x2,x3)) + h(x1,x2,x3);
```

is also handled correctly. Intrepret this statement as being executed from left to right so that g(x1,x2,x3) is computed based on the old value of x2 and h(x1,x2,x3) is computed from the new value of x2 just assigned by the imbedded statement x2 ← f(x1,x2,x3). In this case Cs(v,v') would be:

$$(x1=g(x1',x2',x3') + x2 + h(x1',x2,x3') \wedge x2=f(x1',x2',x3') \wedge x3=x3').$$

Note that the "side effect" of the imbedded assignment on the value of x2 is handled by using both x2' and x2 in the appropriate places in the equation defining x1. The rules determining which value to use at each point (x2 or x2') are those same semantic rules defining the language which one would use in compiling the proper machine instructions for this statement. Other side effect problems such as that caused by global variables in procedures can also be handled in this way.

Function subroutines and procedures can be dealt with by this same method of relating old and new values of the state vector. Suppose 1) that one has the assignment statement

```
x1 ← f(x1,r(x1,x2));
```

where r(x,y) is a previously verified function routine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

having an initial predicate I(x,y), and a final predicate F(x,y,z) relating the parameters x and y to the final result of the function, z, and 2) that P(x1,x2) is the assertion which is assumed to be 'true' before the execution of the statement. The assertion R(x1,x2), 'true' after the execution, would be:

$$R(x1,x2) = \exists x1' \exists x2' [P(x1',x2') \wedge x1=f(x1',r(x1',x2')) \wedge x2=x2' \wedge F(x1',x2',r(x1',x2'))].$$

The notation r(x1',x2') has made a subtle change when used in the assertions as compared to its use in the assignment statement. It is now the abstract function r(x,y) which characterizes the computation done by the program r(x,y). The expression F(x1',x2',r(x1',x2')) is thought of as defining this abstract function r for this case. A slightly simpler notation can be used if one considers that the procedure call r(x1',x2') results in the computation of some fixed value say r', then

$$R(x1,x2) = \exists x1' \exists x2' \exists r' [P(x1',x2') \wedge x1=f(x1',r') \wedge x2=x2' \wedge F(x1',x2',r')].$$

A "verification side effect" also occurs when one derives an assertion from a statement involving procedure calls. One must guarantee that the initial conditions of the procedure are satisfied by generating the "side verification condition":

$$\vdash P(x1,x2) \supset I(x1,x2).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The correctness of the program depends on the proof of this condition as well as all other verification conditions.

Recursive procedures can be verified using this method. This use is justified by the same argument which applies to the normal use of the method. If any execution path through a program as a whole (including any procedures) is broken down into a sequence of tagged paths then if the paths are each verified so is the program. To demonstrate this we use the standard example: factorial (denoted by !). Let Fac(N) be the recursive procedure coded in Algol 60:

```
PROCEDURE Fac(N);
    VALUE N; INTEGER N;
    IF N≠0 THEN Fac ← N*Fac(N-1) ELSE Fac ← 1;
```

Let $N \geq 0$ be the initial predicate and $(Fac=N!)$ be the final predicate. There are two control paths in the program. First verify the path which executes Fac←1. This involves no recursion and can be verified directly. The verification condition is

$$\vdash (N \geq 0 \wedge N=0 \wedge Fac=1) \supset Fac=N!$$

Next verify the control path involving the recursion getting the verification condition:

$$\vdash (N \geq 0 \wedge N \neq 0 \wedge Fac=N*Fac' \wedge Fac'=(N-1)!) \supset Fac=N!$$

with the side condition:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\vdash (N \geq 0 \wedge N \neq 0) \supset N-1 \geq 0.$

Note that in generating the condition for the recursive path the final assertion (Fac=N!) was used even though it had not yet been proved to be correct. This, however, is no different than the way inductive predicates were used before; they too were assumed to be correct as part of an inductive argument. In this case we see that the final predicate serves double duty by being an inductive one as well.

This leads to an interesting insight into the relationship between iterative and recursive programs. For example, if one writes an equivalent iterative program to compute factorial without recourse to recursion he must supply an inductive predicate as well as the initial and final ones. In the recursive case the final predicate also serves as an inductive one and this imposes a rigid inductive property on the program. For the iterative program the induction is determined by the particular looping structure. This is why recursive programs are usually easier to understand; a person need keep in mind only the final predicate with the induction being a fixed notion for all such programs. For iterative programs, in addition to the final predicate, one must also consciously concern himself with the induction. Another example program which computes factorial employing both recursion and iteration together is given in Appendix III.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algol procedures involving "call by name" parameters are particularly messy (e.g., see [16]). The problem here is that the procedure is in actual fact a procedure scheme. It represents a whole set of procedures depending on which actual parameters are passed to the procedure. For example, the procedure

```
PROCEDURE SP(Q,E,I,N);
    INTEGER Q,E,I,N;
    BEGIN
        I←1;
    L:  IF I≤N THEN
        BEGIN
            Q←E;
            I←I+1;
            GO TO L;
        END;
    END;
```

will compute the sum S= A[1] + A[2] + ... + A[M] if called by {R←0; SP(R,R+A[J],J,M)}, but will compute the product P = A[1] * A[2] * ... * A[M] if called by {P←1; SP(P,P*A[J],J,M)}. There are no doubt many more exotic computations it will do given different calls.

The problem is how can one verify such a procedure whose semantic results are so dependent on its parameters. The easiest solution is not to verify it, once for all, in isolation but to verify it in context of its use, once for each specific call made to it. This can be done by substituting the actual by name parameters throughout the procedure. A more complicated solution would be to devise a notation for assertions which would allow verification of such a procedure scheme. Exactly how this may be done

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

escapes us. A practical compromise would be to verify the procedure once for each class of anticipated interpretations by assuming certain properties of the actual parameters. For example, the above procedure could be verified twice, once for sums and a second time for products. Any other uses of the procedure could be verified "in-line".

One essential property that a program must have in order to be proved correct by these methods is a control structure which can be analyzed. For example, in a machine language program one may have a branch instruction which transfers the machine control to an arbitrary next instruction depending on some register value. Potentially the control could pass next to any other instruction in the program (perhaps even to some location containing data, but that would usually be considered an error). This makes control path analysis infinite in any practical sense, unless the assertions contain enough information about the variables effecting such a branch to determine some small number of successor instructions.

The intent of this section has not been to describe explicitly how this method can be used to prove correctness of Algol 60 programs, but rather to share our conviction that the method can be extended in very general ways. A precise set of definitions for a language as semanticly rich as Algol would in itself be an extensive work. On the other hand, enough different concepts have been presented here so

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

one may see directly how to extend the ideas to, say, Fortran.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER IV: CODING METHOD

This chapter relates in more detail how the system described in Chapter II was coded. The programs were written for the IBM 360 and in particular for a Model 65 (really a Model 67 that thinks it's a 65) running IBM's O.S. (Operating System). The standard 360 Level F Assembler was used, with approximately 60 user defined macros forming the basis for most of the coding. No attempt was made to make the programs machine independent, and when desirable, assembly code was freely interspersed among macro calls. With our machine configuration the user's partition of main high speed memory is 64K 32 bit words (where K=1024). Approximately 24K of this is consumed by the program and local data storage, and the remaining 40K is available for formula storage. There are roughly 10,000 source cards for the whole program of which one half is that portion borrowed from the CMU Algol 67 compiler.

Internal Formula Representation

A large part of the system is concerned with formula manipulation. The formulas themselves are composed of "cells" which are linked together by pointers to form general list structures. This list structure processing is much in the spirit of IPL-V [29] but is designed as a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

special purpose system. This special purpose nature plus the fact that the coding was done in assembly language results in a very efficient system. Most examples took a total run time of from 12 to 40 seconds, about 8 to 10 of which can be directly attributed to system overhead.

The cells which make up the list structures are each 4 contiguous 32 bit words. There are many different types of cells, each one containing an identifying code (e.g., +, *, ∀, simple variable). The features and exact format for any particular cell depends upon its type, but all cells do have some common properties. Cells of any type may be composed to form lists. Each cell has one word called the "link" which points to the next element of the list of which it is a member. A 32 bit word can be interpreted as an absolute memory address and a word "points" to a cell by containing the address of its first word. If the link is zero this cell is the last (or possibly only) cell of its list. Most cells also contain a pointer called the "list" which points to a sublist of this cell. With the link and list pointers, arbitrary tree-like list structures can be formed. In addition to type, link, and list fields, each cell contains an additional field which is called the "data" field. Its particular contents depend on the cell type.

Initially the area reserved for formula storage (up to 40K words or 10K cells) is linked together to form one long list of "available cells". Each time a cell is needed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for construction of a formula an available cell is removed from the list, and each time a cell or set of cells are no longer needed they are returned to the available status by being linked onto the list. This is the same storage management technique used by IPL-V. Each cell is accounted for at all times by either being "available" or "in use". This method of using storage does impose the burden on the programmer to continually return "used" cells to the available list. To aid in this process a count of available cells is maintained and can be easily printed out. At times, the fact that the wrong number of cells are in the available list, can lead to discovering obscure program errors.

Other storage management techniques are commonly used for list handling systems, but we found this method extremely simple to set up and practical to use. One popular method is based on "garbage collection" (see pages 406 - 420 of [15] for a more complete discussion). One simply uses available cells but never returns any to the available status. As soon as there are no more available cells the "garbage collector" routine is called upon to collect all discarded "garbage" cells. This is usually done by first marking all cells that are "in use" and then forming a new available cell list from those remaining unmarked. This imposes the burden on the programmer of somehow recording pointers to all structures which are "in use". We have no strong feelings as to desirable methods to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

manage storage but found the method we did use quite satisfactory.

Formulas in our normalized representation, explained in detail in Chapter II, are stored quite naturally as list structures. The cells which are operator types (e.g.,+) are considered to be n-ary operators and their list pointers point to a list of operands. For example, a normalized sum is a cell of type <SUM> (thought of as an n-ary +) whose list field points to the terms composing the sum. The following table shows all the types of cells and to what type cell their list may point as well as the contents of their data field. The cells types are also grouped together into classes. In general, the link will always point to another cell of the same type or general class.

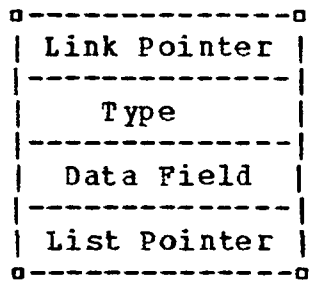
Type	Name	List Points To	Data Field Contains
Arithmetic			
Sum (+)	<SUM>	List of <TERM>'s	Signed integer constant term of the sum
Term (*)	<TERM>	List of <PRIM>'s	Signed integer constant coefficient of the term
Primaries (<PRIM>)			
Simple Variable	<SVAR>	(not used)	Relative address of variable name
Array Variable	<AVAR>	Subscript-<SUM>	Relative address of variable name
Special Functions	<FUNC>	List of arguments-<SUM>'s	Code to distinguish particular functions.
Relationals (<REL>)			
≥0	<GTEQ>	Same as <SUM>	Same as <SUM>
≤0	<LTEQ>	" " "	" " "
=0	<EQ>	" " "	" " "
≠0	<NOTE>	" " "	" " "

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Logical (<LOG>)			
'true'	<TRUE>	(not used)	(not used)
'false'	<FALS>	" "	" "
∧	<AND>	List of <REL>'s or List of <OR>'s	" "
∨	<OR>	List of <REL>'s or List of <AND>'s	" "
∀	<FALL>	<LOG>	Relative address of quantified variable name
∃	<EXIS>	<LOG>	Relative address of quantified variable name
Miscellaneous			
←	<LEFT>	List:(<SVAR>or <AVAR>,<SUM>)	(not used)

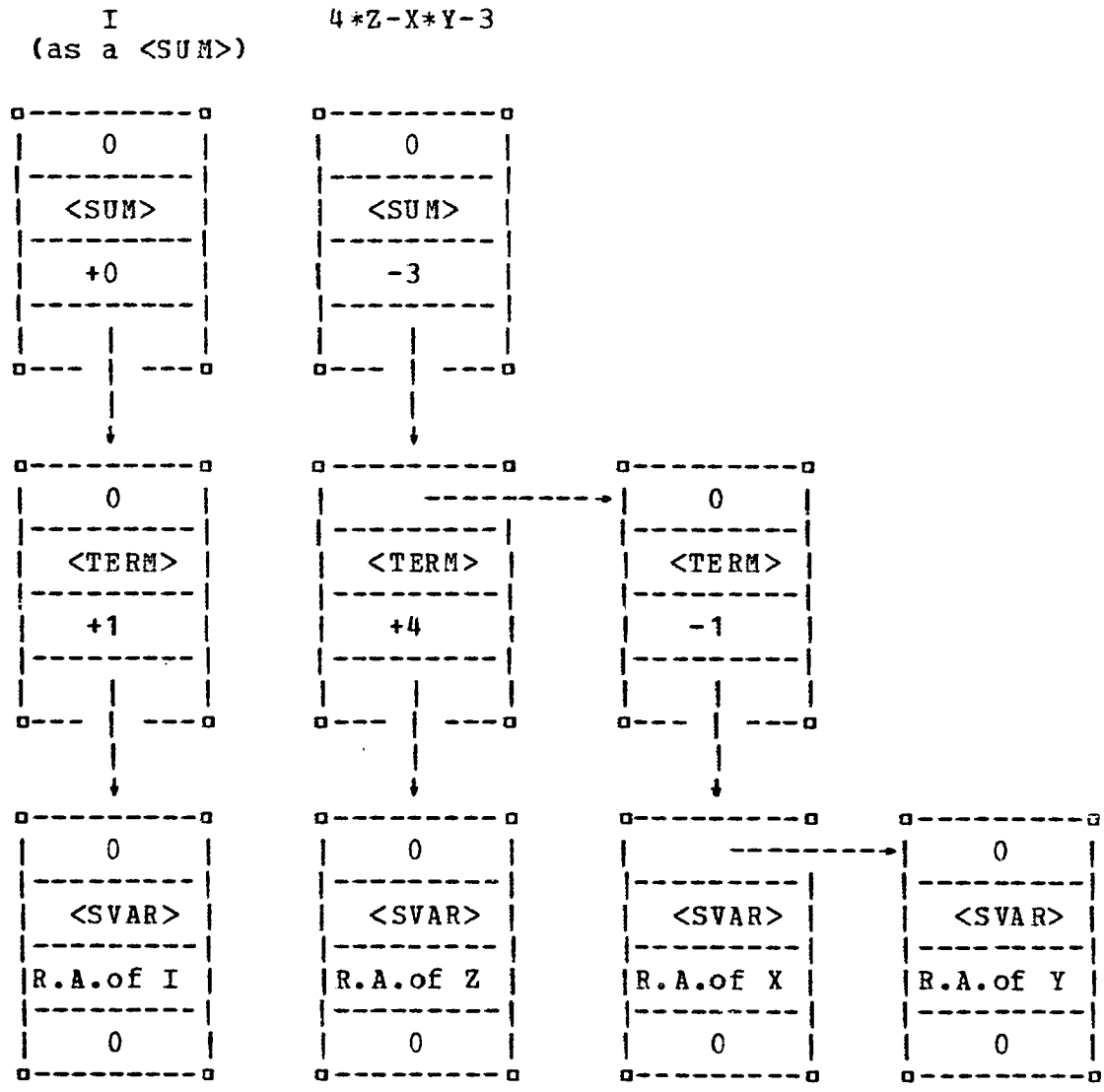
Note that those integer constants which are special terms and primaries in the normalized form are also represented in memory in a special way, being the data fields of <SUM> and <TERM> cells, respectively.

By representing a cell with the notation:



specific examples of the way formulas are stored in the computer memory can be shown. Example <SUM>'s are:

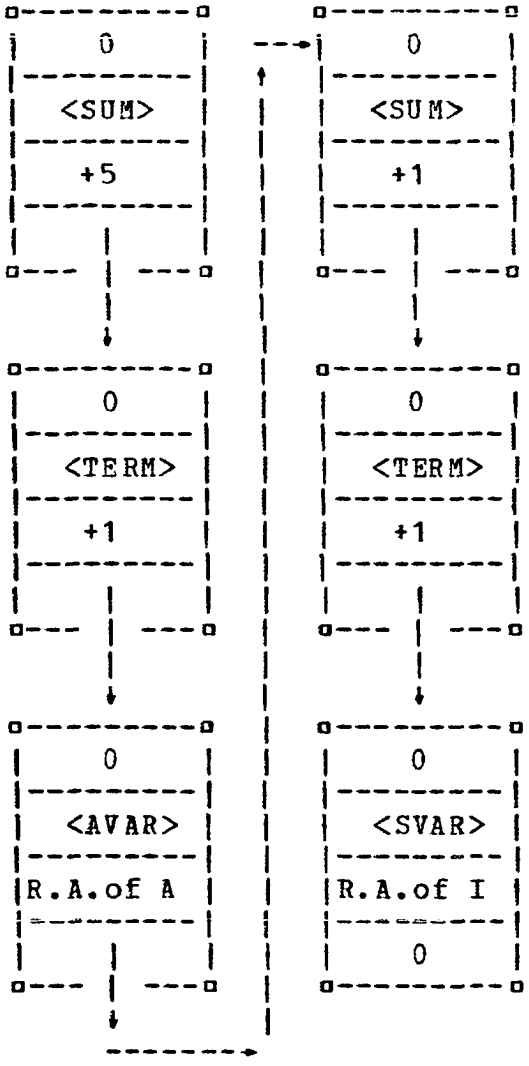
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



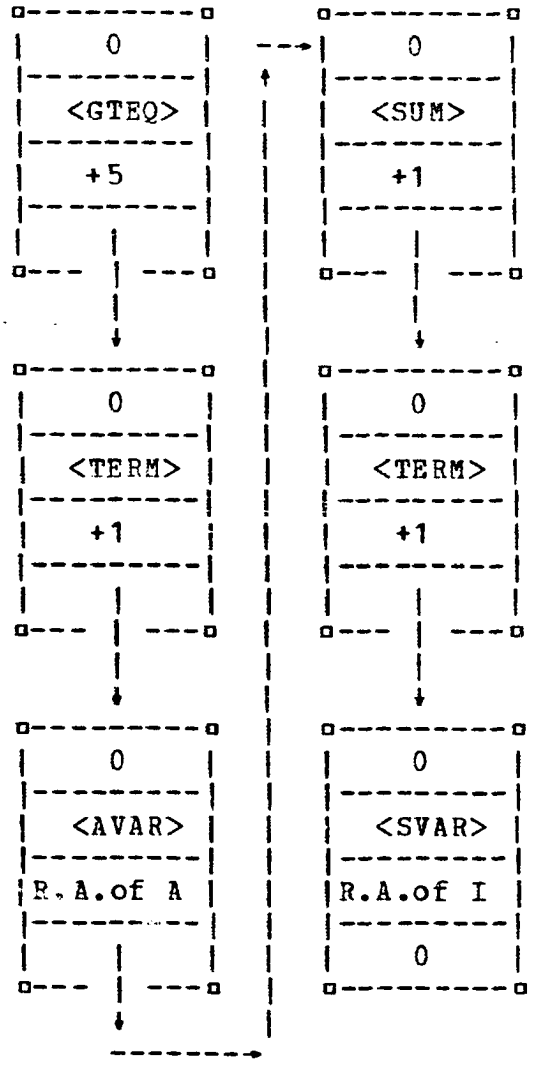
A <SUM> becomes a <REL> by simply replacing the type field by the type <REL>. Of course, the other normalizations are also done (e.g., factoring out greatest common divisor). For example:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A[I+1]+5



A[I+1]+5 ≥ 0

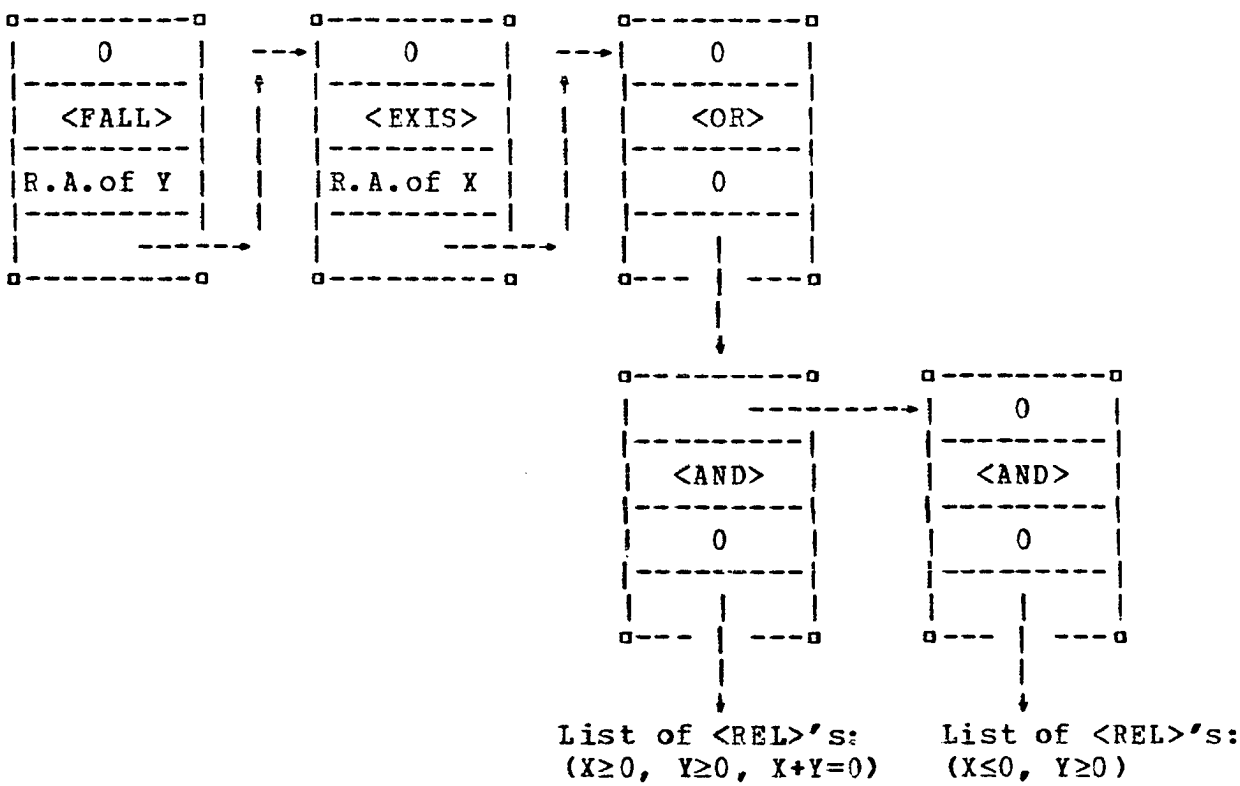


<REL>'s are composed by <AND>'s and <OR>'s to form logical expressions in d.n.f. or c.n.f. Quantifiers may be prefixed at the outermost level. For example,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\forall Y \exists X ((X \geq 0 \wedge Y \leq 0 \wedge X+Y=0) \vee (X \leq 0 \wedge Y \geq 0))$$

would be stored in c.n.f. in the form:



For the examples given in Appendix II, the integer numbers occurring in several places adjacent to the formulas gives the number of cells which that formula occupies.

Use of Macros

Macros were designed to allow symbolic references to the different fields of the cells. The IBM 360 has 16 general purpose 32 bit registers which are used for addressing memory and doing integer arithmetic. Our system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

considers seven of these (symbolically named R0, R1, ..., R6) as working registers for holding integer constants and pointers to list structures. If R1 contains an address pointing to a given cell then the macro 'LLINK R2,R1' will cause the link pointer of that cell (pointed to by R1) to be loaded into register R2. The complimentary macro 'STLINK R2,R1' will store the value of register R2 as the link field of the cell pointed to by register R1. The complete set of these primitives to access and store into the fields of cells were put into the macro library. (They include, for example, LIN - Load INteger constant, LVAR - Load VARiable relative address, STLIST - STore LIST, etc.)

The macros in this set each usually generate one machine instruction to perform the desired service. For example, the macro call 'LLINK R2,R1' would generate the single instruction 'L R2,0(R1)'. One could just as easily have coded the machine instruction directly, but the symbolic value of the macro when reading a program listing is very helpful. When one reads the machine instruction 'L R2,0(R1)' he understands that register R2 is being loaded by a word at the zero-th position from the address in register R1. On the other hand, when reading the macro form 'LLINK R2,R1' we understand that R2 is being set to the "link" of "cell" R1. Just from this extremely trivial use of macros we found we had developed a special purpose problem oriented language. When using these macros one soon begins to think of the registers R0 to R6 as variables. The other

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

facilities for coding routines also encourage this idea.

A set of three, more complex macros enables one to write recursive procedures. One routine may call another routine, including itself, by means of the linkage macro RCALL which has the general form:

```
RCALL    <Routine name>,(<Input>),(<Output>)
```

The fields <Input> and <Output> are lists of register R0 to R6. For example, there is a routine named ADD which accepts two formulas of type <SUM> and symbolically adds them to form a resultant <SUM>. A typical call on this routine would look like:

```
RCALL    ADD,(R3,R2),(R0)
```

Registers R3 and R2 are assumed to be each pointing to a <SUM> and a pointer to the resulting <SUM> is returned by ADD in register R0.

A second macro named RENTER is used to denote a routine name, entry point, and to indicate into which registers the parameters should be placed. The first instruction of the ADD routine is

```
ADD    RENTER    (R0,R1)
```

This is the heading for the procedure and is followed by the code for forming the sum of formulas pointed to by R0 and R1. When control is passed from one routine to another the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

values of registers R0 through R6 are stored on a push down stack and the parameters are moved to the proper registers. This allows each routine to make full use of these registers without effecting their values for any other active routine. They are "local variables". Someplace in the code of the routine ADD the result will have been formed, pointed to by, say, register R2. At this point the macro 'REXIT' is used to return to the routine which called ADD:

```
REXIT    (R2)
```

This macro also causes the calling routine's register values to be retrieved from the push down stack and the values of the result registers (in this case just R2) are moved to the registers indicated in the RCALL as output registers.

In most simple routines the seven registers are ample for storage of all pointers and for simple integer computations. As long as there are no references to data except via these registers and as long as no instructions themselves are altered during execution (which is clumsy and unatural to do on the 360, anyway) the routines are reenterable (recursive) because the registers are saved on the push down stack.

Some routines do require more local memory than afforded by the seven registers. For these routines, local storage areas may be declared. These areas are allocated dynamically in the register push down stack and in that way

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

each time a routine is called new local storage is also provided. The need for local storage in a routine is indicated by placing the letter "S" after the parameters on the RENTER statement. The local storage can be declared by following the RENTER by any number of the standard assembler "define storage" operators. The end of this local storage is then denoted by the macro 'EDB'. Thus if the routine ADD required two additional full words (denoted in assembly language by "F") of local storage, named X and Y, then one would code:

```
ADD     RENTER    (R0,R1),S
X       DS        F
Y       DS        F
        EDB

   {body of code for routine which may freely reference
     X and Y as words in memory}
```

One fixed section of memory is reserved for data, common to all routines, such as standard tables and constants. This is a "global" storage area.

Debugging Aids

The use of these linkage macros for routines makes it very easy to incorporate routine tracing for debugging. A simple trace was built into these macros so that upon each entry to a routine its name, the input parameter values, and the number of cells currently in the available list are printed out. Each exit from a routine prints the values of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the output parameters and the current number of available cells. This tracing is under control of a toggle which can be turned on and off within any program as well as by control cards included in the input. Routines which are debugged can be isolated from the tracing by placing the additional parameter (TRC=OFF) in the RENTER macro call. It would be nearly impossible to find obscure coding errors without this tracing feature.

Two additional debugging aids are the "print" and "dump" routines. The print routines convert the list structured formulas into parenthesized character strings in the conventional infix notation. The formula output of the examples in Appendix II is formed by these print routines. During test runs the well-formedness of various expressions is an important issue. The exact internal form cannot always be determined by the print routine output, so dump routines output the formulas, in a form similar to the diagrams used here, showing each cell with its pointers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER V: CONCLUSION

Perhaps the most important question to ask at this point is: are we any closer to realization of a general purpose verifying compiler. The program verifier we built does verify many small yet interesting examples. The formula manipulation, simplification, and theorem proving capabilities of the system are quite interesting independently. But can this verifier be modified to deal with large real programs? One may feel that in this attempt we have simply climbed a tree as a first step toward getting to the moon. That is a possibility. By constricting a problem one runs the risk of having changed its fundamental characteristics and the solution for limited cases may shed no light on the original problem. We do not believe this to be true for the program verifier.

Just what are the differences between large real programs and the examples worked with here? Of course, size is the most obvious distinction. One expects the amount of processing to increase with size but the important issue is the rate of the increase. One measure of the amount of processing required to verify a program is the number of verification conditions generated. How this can be kept in check by the appropriate placement of inductive predicates was discussed at the end of Chapter I. If all statements in a program are "tagged" by predicates then the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

verification conditions is proportional to the number of statements in the program. Number 8 of Appendix II was included as an example of a program "larger" in this respect.

If growth of the number of conditions generated is not a potential problem then what are the differences in generating and proving the conditions for "real" programs. These problems do not seem to be of a magnitude greater than those encountered in this work. This is not to say that it is a trivial matter to extend this work to a general system, but that, as the limitations and problems discussed in Chapter III are eliminated, the ideas can grow smoothly into a usable system. The great bulk of the statements of large real programs are clerical in nature. Not many real programs are more subtle than example 3 for raising a number to a power. Programs may be complicated in their dynamic aspect and yet not truly sophisticated. When a program is based on some powerful mathematical results those results are known to the programmer and could be supplied to the verifier.

One philosophical note that gives confidence to the approach, is that Floyd appears to have formally captured the casual way a programmer "understands" a program. A programmer usually considers the relationships among the program variables at different points in the program and follows what "happens" along alternate control paths. It is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a natural technique, and for that reason should turn out to be extremely practical. Our emphasis has been on working within the precise formal system. Other people have concentrated on the more informal approach to proving correctness of programs. London and Good [20] have created rigorous, yet not strictly formal, proofs of correctness by hand. Computer assistance to this tedious work appears to be as necessary for them as human assistance is for our theorem prover. Some workable compromise between the efforts of man and machine seems to offer the most hope.

It is also interesting to note the diversity of subject matter touched upon in this work. In particular, the following areas have been important: theory of computation, coding methods and programming language, compiler design, formula manipulation and list processing, canonical forms and simplification, number theory, integer linear programming, mathematical logic and theorem proving, and circuit (boolean) minimization. This made the research very stimulating. While we have not contributed any startling results to any of the fields individually, we hope the synthesis of ideas does make a contribution to computer science.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

1. Bajzek, T., et al. Algol 67 internal specifications. Carnegie - Mellon Univ. Comp. Center Rep., 1966.

2. Bartee, T. C., Leblow, I. L., and Reed, I. S. "Theory and Design of Digital Machines". McGraw-Hill, New York, N. Y., 1962.

3. Caviness, B. F. On canonical forms and simplification. Ph.D. Thesis, Computer Science Dept., Carnegie - Mellon Univ., Pittsburgh, Pa., May 1968.

4. Cook, R. A. and Cooper, L. An algorithm for integer linear programming. Report No. AM65-2, School of Engineering and Applied Science, Washington Univ., St. Louis, Missouri, 1965.

5. Cooper, D. C. Program scheme equivalences and second order logic. Fourth Ann. Machine Intelligence Workshop, Univ. of Edinburgh, Aug. 1968.

6. Davis, M. Diophantine equations and recursively enumerable sets. Automata Theory, E. R. Caianiello (ed.), Academic Press, 1966, 146 - 152.

7. Davis, M. Extensions and corrollaries of recent work on Hilbert's tenth problem. Illinois J. Math. 7, 1963, 246 - 250.

8. Davis, M. A program for Presburger's algorithm. Summer Inst. for Symbolic Logic, Cornell Univ., 1957, 215 - 223.

9. Evans, A. An algol 60 compiler. Annual Rev. in Auto. Prog., Vol. 4, Pergamon Press, 1964.

10. Evans, A. Syntax analysis by a production language. Ph. D. Thesis, Carnegie - Mellon Univ., 1965.

11. Fierst, J. W., et al. Algol - 20 Language Manual. Cargegie - Mellon Univ. Comp. Center, 1965.

12. Floyd, R. W. Assigning meanings to programs. Proc. Symp. Appl. Math., Amer. Math. Soc., Vol. 19, 1967, 19 - 32.

13. Floyd, R. W. The verifying compiler. Computer Science Research Review, Carnegie - Mellon Univ. Annual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Report, 1967, 18 - 19.

14. Hilbert, D. and Bernays, P. "Grundlagen Der Mathematik". Vol. 1, Julius Springer, Berlin, 1934.

15. Knuth, D. E. "The Art of Computer Programming", Vol. 1. Addison - Wesley Publishing Co., Reading, Mass., 1968.

16. Knuth, D. E. and Merner, J. N. Algol-60 confidential. Comm. ACM 4-6 (June 1961), 268-272.

17. Kuhn, H. W. Solvability and consistency for linear equations and inequalities. Amer. Math. Monthly, 63, April 1956.

18. London, R. L. Correctness of the Algol procedure askforhand. Comp. Sci. Tech. Rep. No. 50, Comp. Sci. Dept., Univ. of Wisc., 1968.

19. London, R. L. Computer programs can be proved correct. Fourth Sys. Sym. Formal Sys. and Non-Num. Prob. Solving by Computer, Case Western Reserve Univ., Cleveland, Ohio, 1968.

20. London, R. L. and Good, D. I. Interval arithmetic for the Burroughs B5500: Four Algol procedures and proofs of their correctness. Comp. Sci. Tech. Rep. No. 26, Comp. Sci. Dept., Univ. of Wisc., 1968.

21. Manna, Z. Termination of algorithms. Ph.D. Thesis, Computer Science Dept., Carnegie - Mellon Univ., Pittsburgh, Pa., April 1968.

22. Manna, Z. Formalization of properties of programs. Memo. No. AI - 64, Stanford Artificial Intelligence Report, Stanford, Calif., July 1968.

23. Manna, Z. Properties of programs and the first - order predicate calculus. JACM, Vol. 16, No. 2, April 1969, 244 - 255.

24. Manna, Z. The correctness of programs. J. of Comp. and Sys. Sciences, May 1969.

25. Manna, Z., and Pnueli, A. Formalization of properties of recursively defined functions. ACM Sym. on Theory of Comp., Marina del Ray, Calif., May 1969.

26. McCarthy, J. and Painter, J. A. Correctness of a compiler for arithmetic expressions. Proc. Symp. Appl. Math., Amer. Math. Soc., Vol. 19, 1967, 33 - 41.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

27. Mendelson, E. "Introduction to Mathematical Logic". D. Van Nostrand Co., Inc., Princeton, N. J., 1964.

28. Meserve, B. E. Decision methods for elementary algebra. Amer. Math. Monthly, 62, 1965, 1 - 8.

29. Newell, A., et al. "Information Processing Language - V Manual", second edition. Prentice - Hall, Inc., Englewood Cliffs, N. J., 1964.

30. Presburger, M. Uber die Vollstandigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in whelchem die Addition als einzige Operation hervortritt. Comptes - Rend. du Premier Congr. d. Math. des Pays Slaves, 1929 (Warschau 1930), 92 - 101.

31. Richardson, D. Some unsolvable problems involving functions of a real variable. Notices of the Amer. Math. Soc., 13, 1966, 135.

32. Tarski, A. "A Decision Method for Elementary Algebra and Geometry", second edition. Univ. of Calif. Press, 1951.

33. Uspensky, J. V. and Heaslet, M. A. "Elementary Number Theory". McGraw - Hill Book Co., Inc., N. Y., 1939.

34. von Neumann, J., and Goldstine, H. H. Planning and coding problems for an electronic computing instrument. "Collected Works of John von Neumann", Vol. 5. A. H. Taub (ed.), Pergamon Press, N. Y., 1961, 80 - 235.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix I

Backus-Naur Form Definition for Simple Integer Programming Language

The program verifier accepts, as input, character strings in the class <Program>. A complete description of that class follows:

<Program> ::= <unlabeled compound statement>

<unlabeled compound statement> ::= BEGIN<compound tail>

<compound statement> ::= <unlabeled compound statement>|
<label>:<compound statement>

<compound tail> ::= <statement>END|<statement>;<compound tail>

<statement> ::= <unconditional statement>|<conditional statement>

<unconditional statement> ::= <basic statement>|
<compound statement>

<basic statement> ::= <unlabeled basic statement>|
<label>:<basic statement>

<unlabeled basic statement> ::= <assignment statement>|
<go to statement>|<dummy statement>|<assertion statement>

<assignment statement> ::= <variable>←<arithmetic expression>

<go to statement> ::= GO TO <label>

<dummy statement> ::= <empty>

<assertion statement> ::= ASSERT(<super Boolean expression>)

<conditional statement> ::= <if statement>|<if statement>
ELSE<statement>|<label>:<conditional statement>

<if statement> ::= <if clause><unconditional statement>

<if clause> ::= IF<Boolean expression>THEN

<arithmetic expression> ::= <term>|+<term>|-<term>|
<arithmetic expression>+<term>|

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        <arithmetic expression>-<term>

<term> ::= <factor>|<term>*<factor>|<term>÷<factor>

<factor> ::= <primary>|<factor>↑<primary>

<primary> ::= <unsigned integer>|<variable>|<function>|
                                    (<arithmetic expression>)

<function> ::= ABS(<arithmetic expression>)|<primary>
                                    MOD<primary>

<Boolean expression> ::= <Boolean term>|
                         <Boolean expression>⊃<Boolean term>

<Boolean term> ::= <Boolean factor>|<Boolean term>∨
                                    <Boolean factor>

<Boolean factor> ::= <Boolean sum>|<Boolean factor>∧
                                    <Boolean sum>

<Boolean sum> ::= <Boolean primary>|¬<Boolean primary>

<Boolean primary> ::= TRUE|FALSE|<relational expression>|
                                    (<Boolean expression>)

<relational operator> ::= =|≠|>|<|≥|≤

<relational expression> ::= <arithmetic expression>
      <relational expression><arithmetic expression>

<super Boolean expression> ::= <s-Boolean term>|
      <super Boolean expression>⊃<s-Boolean term>

<s-Boolean term> ::= <s-Boolean factor>|<s-Boolean term>∨
                                    <s-Boolean factor>

<s-Boolean factor> ::= <s-Boolean sum>|<s-Boolean factor>∧
                                    <s-Boolean sum>

<s-Boolean sum> ::= <s-Boolean primary>|¬<s-Boolean primary>

<s-Boolean primary> ::= <Boolean primary>|
      ∀<simple variable>(<super Boolean expression>)|
      ∃<simple variable>(<super Boolean expression>)

<variable> ::= <simple variable>|<array reference>

<simple variable> ::= <identifier>

<array reference> ::= <identifier>[<arithmetic expression>]

<label> ::= <identifier>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<identifier> ::= <letter>|<identifier><letter>|
                                    <identifier><digit>

<letter> ::= A|B|C|...|Z

<digit> ::= 0|1|2|...|9
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX II

Examples of Programs Verified

The following is the actual computer output from the program verifier described here. Each sample begins with a listing of the source program (printed as it is read by the verifier). This is followed by a "map" of the internal flowchart representation of the program. It is a sequence of "node descriptions", each of which is labeled by a hexidecimal number which represents the actual location of that node in the computer memory. The nodes cross reference each other as "successors" or "predecessors" by means of these hexidecimal labels. The next part of each sample output is a step by step tracing of the verification condition generator and the theorem prover. The hexadecimal numbers occurring here, again refer to the location of the nodes in memory and can be used to index to the preceding map. The small decimal numbers associated with some of the formulas represent the number of memory "cells" that that formula occupies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 1.

This program computes X = A*B by adding A to itself B times. The example is simple enough that each step in the verification can be followed. Note how the formula simplifications allow the theorems to be proved by just solving for lone variables and substituting.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN |      MULTIPLY BY SUCCESSIVE ADDITION  (X←A*B)

         ASSERT( Y=B ∧ B≥0 );

         X ← 0;
   L:
         IF  Y ≠ 0  THEN
         BEGIN
              X ← X + A;
              Y ← Y - 1;

              ASSERT( X=A*(B-Y) ∧ Y≥0 );

              GO TO L;
         END;

         ASSERT( X=A*B );

   END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
HEXADECIMAL PROGRAM MAP


000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 000471DC
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    10--00018320     ((Y-B=0∧B≥0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018370     X ← 0
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: IF STATEMENT
          LABEL: L
          TEXT:    5--00018470     ((Y≠0))
          PREDECESSORS: 00047134, 000471B4
          SUCCESSORS:   00047174, 000471DC

00047174: ASSIGNMENT STATEMENT
          TEXT:    7--00018560     X ← X+A
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194

00047194: ASSIGNMENT STATEMENT
          TEXT:    5--00018620     Y ← Y- 1
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4

000471B4: ASSERTION STATEMENT
          TEXT:    14--00018E50     ((Y≥0∧Y*A-B*A+X=0))
          PREDECESSORS: 00047194
          SUCCESSORS:   00047154

000471DC: ASSERTION STATEMENT
          TEXT:    8--000191C0     ((B*A-X=0))
          PREDECESSORS: 00047154
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 GENERATE VERIFICATION CONDITIONS

ASSERTION  1:  000471DC  ((B*A-X=0))

            MOVE BACK TO NODE:  00047154  ((Y≠0))
            IF STATEMENT, FROM FALSE BRANCH
             12--00019340: ((Y≠0)∨(B*A-X=0))


            MOVE BACK TO NODE:  00047134  X ← 0
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             10--00019A20: ((Y≠0)∨(B*A=0))

            MOVE BACK TO NODE:  00047114  ((Y-B=0∧B≥0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((Y-B=0∧B≥0)) ⊃ ((Y≠0)∨(B*A=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((Y=0)∧(Y-B=0)∧(B≥0)∧(B*A≠0))

    SOLVE FOR Y IN Y=0
    SUBSTITUTE  0 FOR Y
    GETTING...((B=0)∧(B*A≠0))

    SOLVE FOR B IN B=0
    SUBSTITUTE  0 FOR B
    GETTING... f
        Q.E.D.



            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((Y≠0)∨(B*A-X=0))
            MOVE BACK TO NODE:  000471B4  ((Y≥0∧Y*A-B*A+
    X=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.2: ((Y≥0∧Y*A-B*A+X=0)) ⊃ ((Y≠0)∨(B*A-X=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((Y=0)∧(Y*A-B*A+X=0)∧(B*A-X≠0))

    SOLVE FOR Y IN Y=0
    SUBSTITUTE  0 FOR Y
    GETTING... f
        Q.E.D.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ASSERTION  2:  000471B4  ((Y≥0∧Y*A-B*A+X=0))

               MOVE BACK TO NODE:  00047194  Y ← Y- 1
               ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                16--0001CA10:  ((Y- 1≥0∧Y*A-B*A+X-A=0))

               MOVE BACK TO NODE:  00047174  X ← X*A
               ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                14--0001D6F0:  ((Y- 1≥0∧Y*A-B*A+X=0))

               MOVE BACK TO NODE:  00047154  ((Y≠0))
               IF STATEMENT, FROM TRUE BRANCH
                18--0001DA60:  ((Y=0)∨(Y- 1≥0∧Y*A-B*A+X=0))


               MOVE BACK TO NODE:  00047134  X ← 0
               ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                16--0001E9E0:  ((Y=0)∨(Y- 1≥0∧Y*A-B*A=0))

               MOVE BACK TO NODE:  00047114  ((Y-B=0∧B≥0))
               ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1:  ((Y-B=0∧B≥0)) ⊃ ((Y=0)∨(Y- 1≥0∧Y*A-B*A=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((Y≠0)∧(Y-B=0)∧(B≥0)∧(Y≤0∨Y*
    A-B*A≠0))

    SOLVE FOR Y IN Y-B=0
    SUBSTITUTE B FOR Y
    GETTING... f
        Q.E.D.


               RETURN TO ALTERNATE BACKWARD PATH
               ASSERTION WAS:  ((Y=0)∨(Y- 1≥0∧Y*A-B*A+X=0))
               MOVE BACK TO NODE:  000471B4  ((Y≥0∧Y*A-B*A+
    X=0))
               ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.2:  ((Y≥0∧Y*A-B*A+X=0)) ⊃ ((Y=0)∨(Y- 1≥0∧Y*A-
    B*A+X=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 2.

This program computes the integer quotient (Q) and remainder (R) of the integer division A÷B. In this case, the theorems are proved solely by simplification.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |     DIVIDE BY SUCCESSIVE SUBTRACTIONS

       ASSERT(A≥0 ∧ B≥0);

       Q ← 0;
       R ← A;
   L:
       ASSERT(A=Q*B+R  ∧ R≥0);

       IF R ≥ B  THEN
       BEGIN
           Q ← Q + 1;
           R ← R - B;
           GO TO L;
       END;

       ASSERT(A=Q*B+R  ∧  R≥0  ∧  R<B);

   END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 000471FC
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    8--000181F0     ((A≥0∧B≥0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018240     Q ← 0
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: ASSIGNMENT STATEMENT
          TEXT:    5--000182B0     R ← A
          PREDECESSORS: 00047134
          SUCCESSORS:   00047174

00047174: ASSERTION STATEMENT
          LABEL: L
          TEXT:    13--00018990     ((A-B*Q-R=0∧R≥0))
          PREDECESSORS: 00047154, 000471D4
          SUCCESSORS:   00047194

00047194: IF STATEMENT
          TEXT:    7--00018AE0     ((B-R≤0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4, 000471FC

000471B4: ASSIGNMENT STATEMENT
          TEXT:    5--00018B90     Q ← Q+ 1
          PREDECESSORS: 00047194
          SUCCESSORS:   000471D4

000471D4: ASSIGNMENT STATEMENT
          TEXT:    7--00018CB0     R ← -B+R
          PREDECESSORS: 000471B4
          SUCCESSORS:   00047174

000471FC: ASSERTION STATEMENT
          TEXT:    18--00019710     ((A-B*Q-R=0∧B-R- 1≥0∧R≥0))
          PREDECESSORS: 00047194
          SUCCESSORS:   000470F4
```

END HEX PROGRAM MAP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  000471FC  ((A-B*Q-R=0∧B-R- 1≥0∧R≥0))

            MOVE BACK TO NODE:  00047194  ((B-R≤0))
            IF STATEMENT, FROM FALSE BRANCH
             19--00019BC0: ((B-R≤0)∨(A-B*Q-R=0∧R≥0))

            MOVE BACK TO NODE:  00047174  ((A-B*Q-R=0∧R≥0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((A-B*Q-R=0∧R≥0)) ⊃ ((B-R≤0)∨(A-B*Q-R=0∧R≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.



ASSERTION  2:  00047174  ((A-B*Q-R=0∧R≥0))

            MOVE BACK TO NODE:  00047154  R ← A
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             9--0001AAF0: ((A≥0∧B*Q=0))

            MOVE BACK TO NODE:  00047134  Q ← 0
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             5--0001ADE0: ((A≥0))

            MOVE BACK TO NODE:  00047114  ((A≥0∧B≥0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1: ((A≥0∧B≥0)) ⊃ ((A≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.



            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A-B*Q-R=0∧R≥0))
            MOVE BACK TO NODE:  000471D4  R ← -B+R
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             17--0001BCC0: ((A+B-B*Q-R=0∧B-R≤0))

            MOVE BACK TO NODE:  000471B4  Q ← Q+ 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                 15--0001C930: ((A-B*Q-R=0∧B-R≤0))

                MOVE BACK TO NODE:  00047194  ((B-R≤0))
                IF STATEMENT, FROM TRUE BRANCH
                 16--0001CCA0: ((A-B*Q-R=0)∨(B-R- 1≥0))

                MOVE BACK TO NODE:  00047174  ((A-B*Q-R=0∧R≥0))
                ASSERTION STATEMENT-FORM VER. COND.


 THEOREM  2.2: ((A-B*Q-R=0∧R≥0)) ⊃ ((A-B*Q-R=0)∨(B-R- 1≥0))

     ENTER THEOREM PROVER

     SHOW TO BE ALWAYS FALSE: f
         Q.E.D.



 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 3.

This program computes Z as A↑B by considering the binary representation of B. It repeatedly squares A (as X) giving a sequence of values of the form (A↑2)↑i, and forms the product of those values of X which correspond to 1-bits in B. The theorems are harder to prove than those in the previous examples. This is the first example which uses the special function definitions in the proofs. It also resorts to case analysis.

```
*  SYNTAX ANALYSIS   *
   BEGIN  |     COMPUTE Z = A↑B

        ASSERT(X=A ∧ Y=B ∧ B≥0);

        Z ← 1;
    E:

        ASSERT(Y≥0 ∧ Z*(X↑Y)=A↑B);

        IF Y ≠ 0 THEN
        BEGIN
            IF (Y MOD 2) = 1 THEN
            BEGIN
                Z ← Z*X;
            END;

            Y ← Y÷2;
            X ← X*X;
            GO TO E;
        END;

        ASSERT(Z=A↑B);

    END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 0004722C
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    15--000186F0     ((X-A=0∧Y-B=0∧B≥0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018740     Z ← 1
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: ASSERTION STATEMENT
          LABEL: E
          TEXT:    23--00019380     ((((X)↑(Y))*Z-((A)↑(B))=0∧
      Y≥0))
          PREDECESSORS: 00047134, 00047204
          SUCCESSORS:   00047174

00047174: IF STATEMENT
          TEXT:    5--00019450     ((Y≠0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194, 0004722C

00047194: IF STATEMENT
          TEXT:    9--000196B0     ((((Y) mod ( 2))- 1=0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4, 000471D4

000471B4: ASSIGNMENT STATEMENT
          TEXT:    6--00019830     Z ← X*Z
          PREDECESSORS: 00047194
          SUCCESSORS:   000471D4

000471D4: ASSIGNMENT STATEMENT
          TEXT:    9--00019920     Y ← ((Y)÷( 2))
          PREDECESSORS: 000471B4, 00047194
          SUCCESSORS:   00047204

00047204: ASSIGNMENT STATEMENT
          TEXT:    9--00019B80     X ← ((X)↑( 2))
          PREDECESSORS: 000471D4
          SUCCESSORS:   00047154

0004722C: ASSERTION STATEMENT
          TEXT:    13--00019FE0     (((A)↑(B))-Z=0))
          PREDECESSORS: 00047174
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SUCCESSORS: 000470F4

END HEX PROGRAM MAP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  0004722C  ((((A)↑(B))-Z=0))

           MOVE BACK TO NODE:  00047174  ((Y≠0))
           IF STATEMENT, FROM FALSE BRANCH
            17--0001A200: ((((A)↑(B))-Z=0)∨(Y≠0))

           MOVE BACK TO NODE:  00047154  ((((X)↑(Y))*Z-((
    A)↑(B))=0∧Y≥0))
           ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((((X)↑(Y))*Z-((A)↑(B))=0∧Y≥0)) ⊃ ((((A)↑(
    B))-Z=0)∨(Y≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((((X)↑(Y))*Z-((A)↑(B))=0)∧(((
    A)↑(B))-Z≠0)∧(Y=0))

    SOLVE FOR Y IN Y=0
    SUBSTITUTE  0 FOR Y
    GETTING... f
        Q.E.D.



ASSERTION  2:  00047154  (((X)↑(Y))*Z-((A)↑(B))=0∧Y≥0))


           MOVE BACK TO NODE:  00047134  Z ← 1
           ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
            22--0001D110: ((((X)↑(Y))-((A)↑(B))=0∧Y≥0))

           MOVE BACK TO NODE:  00047114  ((X-A=0∧Y-B=0∧
    B≥0))
           ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1: ((X-A=0∧Y-B=0∧B≥0)) ⊃ ((((X)↑(Y))-((A)↑(B))=0∧
    Y≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((X-A=0)∧(Y-B=0)∧(B≥0)∧(((X)↑(
    Y))-((A)↑(B))≠0∨Y+ 1≤0))

    SOLVE FOR X IN X-A=0
    SUBSTITUTE A FOR X
    GETTING...((Y-B=0)∧(B≥0)∧(((A)↑(Y))-((A)↑(B))≠0∨Y+
     1≤0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
SOLVE FOR Y IN Y-B=0
SUBSTITUTE B FOR Y
GETTING... f
    Q.E.D.


        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ((((X)↑(Y))*Z-((A)↑(B))=0∧Y≥0))
        MOVE BACK TO NODE:  00047204  X ← ((X)↑( 2))
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         23--000225A0: ((((X)↑( 2*Y))*Z-((A)↑(B))=0∧
Y≥0))

        MOVE BACK TO NODE:  000471D4  Y ← ((Y)÷( 2))
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         31--00023EF0: ((((Y)÷( 2))≥0∧((X)↑( 2*((Y)÷(
 2))))*Z-((A)↑(B))=0))


        MOVE BACK TO NODE:  000471B4  Z ← X*Z
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         31--00025DA0: ((((Y)÷( 2))≥0∧((X)↑( 2*((Y)÷(
 2))+ 1))*Z-((A)↑(B))=0))

        MOVE BACK TO NODE:  00047194  ((((Y) mod (
 2))- 1=0))
        IF STATEMENT, FROM TRUE BRANCH
         39--00026570: ((((Y) mod ( 2))- 1≠0)∨(((Y)÷(
 2))≥0∧((X)↑( 2*((Y)÷( 2))+ 1))*Z-((A)↑(B))=0))

        MOVE BACK TO NODE:  00047174  ((Y≠0))
        IF STATEMENT, FROM TRUE BRANCH
         43--00026DC0: ((((Y) mod ( 2))- 1≠0)∨(Y=0)∨(((
Y)÷( 2))≥0∧((X)↑( 2*((Y)÷( 2))+ 1))*Z-((A)↑(B))=0))

        MOVE BACK TO NODE:  00047154  ((((X)↑(Y))*Z-((
A)↑(B))=0∧Y≥0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.2: ((((X)↑(Y))*Z-((A)↑(B))=0∧Y≥0)) ⊃ ((((Y) mod (
 2))- 1≠0)∨(Y=0)∨(((Y)÷( 2))≥0∧((X)↑( 2*((Y)÷( 2))+
 1))*Z-((A)↑(B))=0))

ENTER THEOREM PROVER

SHOW TO BE ALWAYS FALSE:((((Y) mod ( 2))- 1=0)∧(((X)↑(
Y))*Z-((A)↑(B))=0)∧(Y- 1≥0)∧(((Y)÷( 2))+ 1≤0∨((X)↑(
 2*((Y)÷( 2))+ 1))*Z-((A)↑(B))≠0))

  DEFINE SPECIAL FUNCTIONS
SHOW TO BE ALWAYS FALSE:((((X)↑(Y))*Z-((A)↑(B))=0)∧(
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Y- 1≥0)∧(Y- 2*$01- 1=0)∧(((X)↑( 2*$03+ 1))*Z-((A)↑(
B))≠0∨Y- 2*$02- 1≤0)∧(((X)↑( 2*$03+ 1))*Z-((A)↑(B))≠0∨
$02+ 1≤0)∧(((X)↑( 2*$03+ 1))*Z-((A)↑(B))≠0∨Y- 2*$02≥0)∧(
Y- 2*$02- 1≤0∨Y- 2*$03- 1≤0)∧(Y- 2*$02- 1≤0∨Y- 2*$03≥0)∧(
Y- 2*$03- 1≤0∨$02+ 1≤0)∧(Y- 2*$02≥0∨Y- 2*$03- 1≤0)∧(
Y- 2*$02≥0∨Y- 2*$03≥0)∧(Y- 2*$03≥0∨$02+ 1≤0))

SOLVE FOR Y IN Y- 2*$01- 1=0
SUBSTITUTE  2*$01+ 1 FOR Y
GETTING...((((X)↑( 2*$01+ 1))*Z-((A)↑(B))=0)∧($01≥0)∧(((
X)↑( 2*$03+ 1))*Z-((A)↑(B))≠0∨$01-$02≤0)∧(((X)↑( 2*
$03+ 1))*Z-((A)↑(B))≠0∨$02+ 1≤0)∧(((X)↑( 2*$03+ 1))*
Z-((A)↑(B))≠0∨$01-$02≥0)∧($01-$03≤0∨$01-$03≤0)∧($01-
$02≤0∨$01-$03≥0)∧($01-$03≤0∨$02+ 1≤0)∧($01-$02≥0∨$01-
$03≤0)∧($01-$02≥0∨$01-$03≥0)∧($01-$03≥0∨$02+ 1≤0))

  PROOF BY CASES: (EACH MUST BE FALSE)

CASE 1...(((X)↑( 2*$01+ 1))*Z-((A)↑(B))=0∧$01≥0∧$01-
$02=0∧$02+ 1≤0)

SOLVE FOR $01 IN $01-$02=0
SUBSTITUTE $02 FOR $01
GETTING... f
  THIS CASE -- Q.E.D.

CASE 2...(((X)↑( 2*$01+ 1))*Z-((A)↑(B))=0∧((X)↑( 2*
$03+ 1))*Z-((A)↑(B))≠0∧$01≥0∧$01-$03=0)

SOLVE FOR $01 IN $01-$03=0
SUBSTITUTE $03 FOR $01
GETTING... f
  THIS CASE -- Q.E.D.



        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ((((Y)÷( 2))≥0∧((X)↑( 2*((Y)÷(
 2))))*Z-((A)↑(B))=0))
        MOVE BACK TO NODE:  00047194  ((((Y) mod (
 2))- 1=0))
        IF STATEMENT, FROM FALSE BRANCH
         39--00020650: ((((Y) mod ( 2))- 1=0)∨(((Y)÷(
 2))≥0∧((X)↑( 2*((Y)÷( 2))))*Z-((A)↑(B))=0))

        MOVE BACK TO NODE:  00047174  ((Y≠0))
        IF STATEMENT, FROM TRUE BRANCH
         43--0001E860: ((((Y) mod ( 2))- 1=0)∨(Y=0)∨(((
Y)÷( 2))≥0∧((X)↑( 2*((Y)÷( 2))))*Z-((A)↑(B))=0))

        MOVE BACK TO NODE:  00047154  ((((X)↑(Y))*Z-((
A)↑(B))=0∧Y≥0))
        ASSERTION STATEMENT-FORM VER. COND.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
THEOREM  2.3: ((((X)↑(Y))*Z-((A)↑(B))=0∧Y≥0)) ⊃ ((((Y) mod (
     2))- 1=0)∨(Y=0)∨(((Y)÷( 2))≥0∧((X)↑( 2*((Y)÷( 2))))*
    Z-((A)↑(B))=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((((Y) mod ( 2))- 1≠0)∧(((X)↑(
    Y))*Z-((A)↑(B))=0)∧(Y- 1≥0)∧(((Y)÷( 2))+ 1≤0∨((X)↑(
     2*((Y)÷( 2))))*Z-((A)↑(B))≠0))

      DEFINE SPECIAL FUNCTIONS
    SHOW TO BE ALWAYS FALSE:((((X)↑(Y))*Z-((A)↑(B))=0)∧(
    Y- 1≥0)∧(Y- 2*$04=0)∧(((X)↑( 2*$06))*Z-((A)↑(B))≠0∨
    Y- 2*$05- 1≤0)∧(((X)↑( 2*$06))*Z-((A)↑(B))≠0∨$05+
     1≤0)∧(((X)↑( 2*$06))*Z-((A)↑(B))≠0∨Y- 2*$05≥0)∧(Y-
     2*$05- 1≤0∨Y- 2*$06- 1≤0)∧(Y- 2*$05- 1≤0∨Y- 2*$06≥0)∧(
    Y- 2*$06- 1≤0∨$05+ 1≤0)∧(Y- 2*$05≥0∨Y- 2*$06- 1≤0)∧(
    Y- 2*$05≥0∨Y- 2*$06≥0)∧(Y- 2*$06≥0∨$05+ 1≤0))

    SOLVE FOR Y IN Y- 2*$04=0
    SUBSTITUTE  2*$04 FOR Y
    GETTING...((((X)↑( 2*$04))*Z-((A)↑(B))=0)∧($04- 1≥0)∧(((
    X)↑( 2*$06))*Z-((A)↑(B))≠0∨$04-$05≤0)∧(((X)↑( 2*$06))*
    Z-((A)↑(B))≠0∨$05+ 1≤0)∧(((X)↑( 2*$06))*Z-((A)↑(B))≠0∨
    $04-$05≥0)∧($04-$05≤0∨$04-$06≤0)∧($04-$05≤0∨$04-$06≥0)∧(
    $04-$06≤0∨$05+ 1≤0)∧($04-$05≥0∨$04-$06≤0)∧($04-$05≥0∨
    $04-$06≥0)∧($04-$06≥0∨$05+ 1≤0))

      PROOF BY CASES: (EACH MUST BE FALSE)

    CASE 1...(((X)↑( 2*$04))*Z-((A)↑(B))=0∧$04- 1≥0∧$04-
    $05=0∧$05+ 1≤0)

    SOLVE FOR $04 IN $04-$05=0
    SUBSTITUTE $05 FOR $04
    GETTING... f
      THIS CASE -- Q.E.D.

    CASE 2...(((X)↑( 2*$04))*Z-((A)↑(B))=0∧((X)↑( 2*$06))*
    Z-((A)↑(B))≠0∧$04- 1≥0∧$04-$06=0)

    SOLVE FOR $04 IN $04-$06=0
    SUBSTITUTE $06 FOR $04
    GETTING... f
      THIS CASE -- Q.E.D.



 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 4.

This example exhibits a simple use of quantification in the assertions. The heuristic for quantified theorems allows the verification conditions to be easily proved.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS    *
   BEGIN  |      IS 'A' A PRIME?


        ASSERT(A≥2);

        I ← 2;
    L:
        ASSERT(  ∀K(  (2≤K  ∧  K<I)  ⊃  A MOD K ≠0)  ∧  I≤A);

        IF I < A THEN
        BEGIN
            IF A MOD I ≠ 0  THEN
            BEGIN
                I ← I + 1;
                GO TO L;
            END  ELSE

                J ← 1;
        END  ELSE

            J ← 0;

        ASSERT(  (J=0  ⊃ ∀K(  (2≤K ∧ K<A) ⊃ A MOD K ≠ 0)  ) ∧
                 (J=1  ⊃   A MOD I = 0)  );

    END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 0004722C
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    5--00018030     ((A- 2≥0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018080     I ← 2
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: ASSERTION STATEMENT
          LABEL: L
          TEXT:    37--000198F0     ∀K01((A-I≥0∧K01- 1≤0)∨(
      A-I≥0∧I-K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))
          PREDECESSORS: 00047134, 000471B4
          SUCCESSORS:   00047174

00047174: IF STATEMENT
          TEXT:    7--0001A4F0     ((A-I- 1≥0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194, 00047204

00047194: IF STATEMENT
          TEXT:    11--0001A740     ((((A) mod (I))≠0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4, 000471DC

000471B4: ASSIGNMENT STATEMENT
          TEXT:    5--0001A7F0     I ← I+ 1
          PREDECESSORS: 00047194
          SUCCESSORS:   00047154

000471DC: ASSIGNMENT STATEMENT
          TEXT:    3--0001A870     J ← 1
          PREDECESSORS: 00047194
          SUCCESSORS:   0004722C

0004722C: ASSERTION STATEMENT
          TEXT:    98--0001C8E0     ∀K02((((A) mod (I))=0∧K02-
       1≤0)∨(((A) mod (I))=0∧A-K02≤0)∨(((A) mod (I))=0∧((
      A) mod (K02))≠0)∨(((A) mod (I))=0∧J≠0)∨(A-K02≤0∧J-
       1≠0)∨(J- 1≠0∧K02- 1≤0)∨(((A) mod (K02))≠0∧J- 1≠0)∨(
      J≠0∧J- 1≠0))
          PREDECESSORS: 00047204, 000471DC
          SUCCESSORS:   000470F4
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
00047204: ASSIGNMENT STATEMENT
          TEXT:    3--0001A8C0     J ← 0
          PREDECESSORS: 00047174
          SUCCESSORS:   0004722C


     END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  0004722C  ∀K02((((A) mod (I))=0∧K02- 1≤0)∨(((
    A) mod (I))=0∧A-K02≤0)∨(((A) mod (I))=0∧((A) mod (K02))≠0)∨(
    ((A) mod (I))=0∧J≠0)∨(A-K02≤0∧J- 1≠0)∨(J- 1≠0∧K02-
     1≤0)∨(((A) mod (K02))≠0∧J- 1≠0)∨(J≠0∧J- 1≠0))


          MOVE BACK TO NODE:  00047204  J ← 0
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           22--00021820: ∀K02((((A) mod (K02))≠0)∨(A-
    K02≤0)∨(K02- 1≤0))

          MOVE BACK TO NODE:  00047174  ((A-I- 1≥0))
          IF STATEMENT, FROM FALSE BRANCH
           28--00027870: ∀K02((((A) mod (K02))≠0)∨(A-
    I- 1≥0)∨(A-K02≤0)∨(K02- 1≤0))

          MOVE BACK TO NODE:  00047154  ∀K01((A-I≥0∧K01-
     1≤0)∨(A-I≥0∧I-K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ∀K01((A-I≥0∧K01- 1≤0)∨(A-I≥0∧I-K01≤0)∨(((
    A) mod (K01))≠0∧A-I≥0)) ⊃ ∀K02((((A) mod (K02))≠0)∨(
    A-I- 1≥0)∨(A-K02≤0)∨(K02- 1≤0))

    ENTER THEOREM PROVER


    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((((A) mod (K02))=0)∧(A-I=0)∧(
    A-K02- 1≥0)∧(I-K02≤0)∧(K02- 2≥0))

    SOLVE FOR A IN A-I=0
    SUBSTITUTE I FOR A
    GETTING... f
        Q.E.D.



          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀K02((((A) mod (I))=0∧K02-
     1≤0)∨(((A) mod (I))=0∧A-K02≤0)∨(((A) mod (I))=0∧((
    A) mod (K02))≠0)∨(((A) mod (I))=0∧J≠0)∨(A-K02≤0∧J-
     1≠0)∨(J- 1≠0∧K02- 1≤0)∨(((A) mod (K02))≠0∧J- 1≠0)∨(
    J≠0∧J- 1≠0))
          MOVE BACK TO NODE:  000471DC  J ← 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           12--0002E7E0: ∀K02((((A) mod (I))=0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            MOVE BACK TO NODE:  00047194  ((((A) mod (I))≠0))
            IF STATEMENT, FROM FALSE BRANCH
             1--000336D0:  t

            MOVE BACK TO NODE:  00047174  ((A-I- 1≥0))
            IF STATEMENT, FROM TRUE BRANCH
             1--00033750:  t

            MOVE BACK TO NODE:  00047154  ∀K01((A-I≥0∧K01-
     1≤0)∨(A-I≥0∧I-K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.2: ∀K01((A-I≥0∧K01- 1≤0)∨(A-I≥0∧I-K01≤0)∨(((
    A) mod (K01))≠0∧A-I≥0)) ⊃  t

    ENTER THEOREM PROVER

     Q.E.D.



ASSERTION  2:  00047154  ∀K01((A-I≥0∧K01- 1≤0)∨(A-I≥0∧I-
    K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))


            MOVE BACK TO NODE:  00047134  I ← 2
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             29--00033C00: ∀K01((A- 2≥0∧K01- 2≥0)∨(A-
     2≥0∧K01- 1≤0)∨(((A) mod (K01))≠0∧A- 2≥0))

            MOVE BACK TO NODE:  00047114  ((A- 2≥0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1: ((A- 2≥0)) ⊃ ∀K01((A- 2≥0∧K01- 2≥0)∨(A-
     2≥0∧K01- 1≤0)∨(((A) mod (K01))≠0∧A- 2≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ∀K01((A-I≥0∧K01- 1≤0)∨(A-I≥0∧
    I-K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))
            MOVE BACK TO NODE:  000471B4  I ← I+ 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             37--000357D0: ∀K01((A-I- 1≥0∧K01- 1≤0)∨(A-
    I- 1≥0∧I-K01+ 1≤0)∨(((A) mod (K01))≠0∧A-I- 1≥0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          MOVE BACK TO NODE:  00047194  ((((A) mod (I))≠0))
          IF STATEMENT, FROM TRUE BRANCH
           47--00037700: ∀K01((((A) mod (I))=0)∨(A-I-
     1≥0∧K01- 1≤0)∨(A-I- 1≥0∧I-K01+ 1≤0)∨(((A) mod (K01))≠0∧
    A-I- 1≥0))

          MOVE BACK TO NODE:  00047174  ((A-I- 1≥0))
          IF STATEMENT, FROM TRUE BRANCH
           38--00038270: ∀K01((((A) mod (I))=0)∨(((A) mod (
    K01))≠0)∨(A-I≤0)∨(I-K01+ 1≤0)∨(K01- 1≤0))

          MOVE BACK TO NODE:  00047154  ∀K01((A-I≥0∧K01-
     1≤0)∨(A-I≥0∧I-K01≤0)∨(((A) mod (K01))≠0∧A-I≥0))
          ASSERTION STATEMENT-FORM VER. COND.


 THEOREM  2.2: ∀K01((A-I≥0∧K01- 1≤0)∨(A-I≥0∧I-K01≤0)∨(((
     A) mod (K01))≠0∧A-I≥0)) ⊃ ∀K01((((A) mod (I))=0)∨(((
     A) mod (K01))≠0)∨(A-I≤0)∨(I-K01+ 1≤0)∨(K01- 1≤0))

     ENTER THEOREM PROVER

     SHOW TO BE ALWAYS FALSE:((((A) mod (I))≠0)∧(((A) mod (
     K01))=0)∧(A-I- 1≥0)∧(I-K01=0)∧(K01- 2≥0))

     SOLVE FOR I IN I-K01=0
     SUBSTITUTE K01 FOR I
     GETTING... f
         Q.E.D.



 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 5.

An array is used for the first time in this example. Most assertions involving arrays need some form of quantification to express facts true over more than one array element. This example also makes use of the linear solver for proof of one theorem.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |     ZERO THE ARRAY A[1:N]

       ASSERT(TRUE);

       I ← 1;
   L:
       ASSERT( ∀J( (1≤J ∧ J<I) ⊃ A[J]=0) );

       IF I≤N THEN
       BEGIN
           A[I] ← 0;
           I ← I + 1;
           GO TO L;
       END;

       ASSERT( ∀J( (1≤J ∧ J≤N) ⊃ A[J]=0) );
   END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    HEXADECIMAL PROGRAM MAP


000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 000471DC
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    1--00017F70      t
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00017FC0    I ← 1
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: ASSERTION STATEMENT
          LABEL: L
          TEXT:    19--00019400     ∀J01((I-J01≤0)∨(J01≤0)∨(
      A[J01]=0))
          PREDECESSORS: 00047134, 000471B4
          SUCCESSORS:   00047174

00047174: IF STATEMENT
          TEXT:    7--00019550     ((I-N≤0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194, 000471DC

00047194: ASSIGNMENT STATEMENT
          TEXT:    6--00019600    A[I] ← 0
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4

000471B4: ASSIGNMENT STATEMENT
          TEXT:    5--000196B0    I ← I+ 1
          PREDECESSORS: 00047194
          SUCCESSORS:   00047154

000471DC: ASSERTION STATEMENT
          TEXT:    19--0001AAB0     ∀J02((N-J02+ 1≤0)∨(J02≤0)∨(
      A[J02]=0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS

ASSERTION  1:  000471DC  ∀J02((N-J02+ 1≤0)∨(J02≤0)∨(A[J02]=0))

          MOVE BACK TO NODE:  00047174  ((I-N≤0))
          IF STATEMENT, FROM FALSE BRANCH
           25--0001ABF0:  ∀J02((I-N≤0)∨(N-J02+ 1≤0)∨(J02≤0)∨(
    A[J02]=0))

          MOVE BACK TO NODE:  00047154  ∀J01((I-J01≤0)∨(
    J01≤0)∨(A[J01]=0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ∀J01((I-J01≤0)∨(J01≤0)∨(A[J01]=0)) ⊃ ∀J02((
    I-N≤0)∨(N-J02+ 1≤0)∨(J02≤0)∨(A[J02]=0))

    ENTER THEOREM PROVER


    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((I-N- 1≥0)∧(I-J02≤0)∧(N-J02≥0)∧(
    J02- 1≥0)∧(A[J02]≠0))

      DEFINE SPECIAL FUNCTIONS
    SHOW TO BE ALWAYS FALSE:((I-N- 1≥0)∧(I-J02≤0)∧(N-J02≥0)∧(
    J02- 1≥0)∧(A[J02]≠0))

      TRY LINEAR SOLVER


    LINEAR PART:I-N- 1≥0∧I-J02≤0∧N-J02≥0∧J02- 1≥0
     ;  NON-LINEAR PART:A[J02]≠0 ;  EQUALITIES TO REMEMBER: nil
    ELIMINATE VARIABLE: I
    BETWEEN:I-N- 1≥0::I-J02≤0
    FORMING...((N-J02+ 1≤0))
    END ELIMINATION -- CONCLUDE:  f
          f -- Q.E.D.



ASSERTION  2:  00047154  ∀J01((I-J01≤0)∨(J01≤0)∨(A[J01]=0))

          MOVE BACK TO NODE:  00047134  I ← 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           1--0001F460:  t

          MOVE BACK TO NODE:  00047114  t
          ASSERTION STATEMENT-FORM VER. COND.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
THEOREM  2.1:  t  ⊃  t

     ENTER THEOREM PROVER

     SHOW TO BE ALWAYS FALSE: f
         Q.E.D.


               RETURN TO ALTERNATE BACKWARD PATH
               ASSERTION WAS: ∀J01((I-J01≤0)∨(J01≤0)∨(A[J01]=0))
               MOVE BACK TO NODE:  000471B4  I ← I+ 1
               ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                19--0001F480: ∀J01((I-J01+ 1≤0)∨(J01≤0)∨(A[
     J01]=0))

               MOVE BACK TO NODE:  00047194  A[I] ←  0
               ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
                19--0001FFD0: ∀J01((I-J01≤0)∨(J01≤0)∨(A[J01]=0))

               MOVE BACK TO NODE:  00047174  ((I-N≤0))
               IF STATEMENT, FROM TRUE BRANCH
                25--00021790: ∀J01((I-J01≤0)∨(I-N- 1≥0)∨(J01≤0)∨(
     A[J01]=0))

               MOVE BACK TO NODE:  00047154  ∀J01((I-J01≤0)∨(
     J01≤0)∨(A[J01]=0))
               ASSERTION STATEMENT-FORM VER. COND.


 THEOREM  2.2: ∀J01((I-J01≤0)∨(J01≤0)∨(A[J01]=0)) ⊃ ∀J01((
     I-J01≤0)∨(I-N- 1≥0)∨(J01≤0)∨(A[J01]=0))

     ENTER THEOREM PROVER

     SHOW TO BE ALWAYS FALSE: f
         Q.E.D.


 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 6.

At one point in this example the values of two array elements are interchanged. The transformation of the assertion over the three assignment statements which perform the exchange is complex but the simplification routines are able to make impressive reductions. This part of the example is described in detail in Chapter II.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |      MOVE LARGEST ELEMENT OF ARRAY A TO A[N]

        ASSERT(N>0);

        I ← 2;
   L:
        IF  I ≤ N  THEN
        BEGIN
             IF  A[I-1] > A[I]  THEN
             BEGIN
                  X ← A[I];
                  A[I] ← A[I-1];
                  A[I-1] ← X;
             END;

             ASSERT( ∀K( (1≤K ∧ K<I) ⊃ A[I] ≥ A[K])  ∧ I≤N);

             I ← I + 1;
             GO TO L;
        END;

        ASSERT( ∀L( (1≤L ∧ L<N) ⊃ A[N] ≥ A[L]) );

    END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 0004724C
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    5--00018030     ((N- 1≥0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018080     I ← 2
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: IF STATEMENT
          LABEL: L
          TEXT:    7--00018200     ((N-I≥0))
          PREDECESSORS: 00047134, 00047224
          SUCCESSORS:   00047174, 0004724C

00047174: IF STATEMENT
          TEXT:    13--00018550     ((A[I- 1]-A[I]- 1≥0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194, 000471F4

00047194: ASSIGNMENT STATEMENT
          TEXT:    8--00018620     X ← A[I]
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4

000471B4: ASSIGNMENT STATEMENT
          TEXT:    11--000187A0     A[I] ← A[I- 1]
          PREDECESSORS: 00047194
          SUCCESSORS:   000471D4

000471D4: ASSIGNMENT STATEMENT
          TEXT:    8--000188C0     A[I- 1] ← X
          PREDECESSORS: 000471B4
          SUCCESSORS:   000471F4

000471F4: ASSERTION STATEMENT
          TEXT:    39--0001A2B0     ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(
      N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))
          PREDECESSORS: 000471D4, 00047174
          SUCCESSORS:   0004722C

00047224: ASSIGNMENT STATEMENT
          TEXT:    5--0001AD70     I ← I+ 1
          PREDECESSORS: 000471F4
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          SUCCESSORS:   00047154

0004724C: ASSERTION STATEMENT
          TEXT:   24--0001C630     ∀L02((N-L02≤0)∨(L02≤0)∨(
      A[N]-A[L02]≥0))
          PREDECESSORS: 00047154
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  0004724C  ∀L02((N-L02≤0)∨(L02≤0)∨(A[N]-A[
    L02]≥0))

          MOVE BACK TO NODE:  00047154  ((N-I≥0))
          IF STATEMENT, FROM FALSE BRANCH
           30--0001C7C0: ∀L02((N-I≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))


          MOVE BACK TO NODE:  00047134  I ← 2
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           28--0001CC00: ∀L02((N- 2≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))

          MOVE BACK TO NODE:  00047114  ((N- 1≥0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((N- 1≥0)) ⊃ ∀L02((N- 2≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((N- 1=0)∧(N-L02- 1≥0)∧(L02-
     1≥0)∧(A[N]-A[L02]+ 1≤0))

    SOLVE FOR N IN N- 1=0
    SUBSTITUTE  1 FOR N
    GETTING... f
        Q.E.D.



          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀L02((N-I≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))
          MOVE BACK TO NODE:  00047224  I ← I+ 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           30--0001EE00: ∀L02((N-I- 1≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))

          MOVE BACK TO NODE:  000471F4  ∀K01((N-I≥0∧A[
    I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.2: ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(
    N-I≥0∧K01≤0)) ⊃ ∀L02((N-I- 1≥0)∨(N-L02≤0)∨(L02≤0)∨(
    A[N]-A[L02]≥0))

    ENTER THEOREM PROVER
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ENTER THEOREM PROVER

SHOW TO BE ALWAYS FALSE:((N-I=0)∧(N-L02- 1≥0)∧(L02-
 1≥0)∧(A[N]-A[L02]+ 1≤0)∧(I-L02≤0∨A[I]-A[L02]≥0))

SOLVE FOR N IN N-I=0
SUBSTITUTE I FOR N
GETTING... f
    Q.E.D.



ASSERTION  2:  000471F4  ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(N-
    I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))


          MOVE BACK TO NODE:  000471D4  A[I- 1] ← X
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           63--00028110: ∀K01((N-I≥0∧I-K01- 1=0∧X-A[I]≤0)∨(
    N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0)∨(N-I≥0∧I-K01- 1≠0∧A[I]-
    A[K01]≥0))

          MOVE BACK TO NODE:  000471B4  A[I] ← A[I-
 1]
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           63--0002C4E0: ∀K01((N-I≥0∧I-K01- 1=0∧X-A[I-
 1]≤0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0)∨(N-I≥0∧I-K01-
 1≠0∧A[I- 1]-A[K01]≥0))

          MOVE BACK TO NODE:  00047194  X ← A[I]
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           66--00032000: ∀K01((N-I≥0∧I-K01- 1=0∧A[I-
 1]-A[I]≥0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0)∨(N-I≥0∧I-
    K01- 1≠0∧A[I- 1]-A[K01]≥0))

          MOVE BACK TO NODE:  00047174  ((A[I- 1]-A[I]-
 1≥0))
          IF STATEMENT, FROM TRUE BRANCH
           67--00036200: ∀K01((A[I- 1]-A[I]≤0)∨(N-I≥0∧
    I-K01- 1=0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0)∨(N-I≥0∧I-
    K01- 1≠0∧A[I- 1]-A[K01]≥0))

          MOVE BACK TO NODE:  00047154  ((N-I≥0))
          IF STATEMENT, FROM TRUE BRANCH
           42--000373F0: ∀K01((N-I+ 1≤0)∨(I-K01- 1≤0)∨(
    K01≤0)∨(A[I- 1]-A[I]≤0)∨(A[I- 1]-A[K01]≥0))


          MOVE BACK TO NODE:  00047134  I ← 2
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          1--000398D0:  t

          MOVE BACK TO NODE:  00047114  ((N- 1≥0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1: ((N- 1≥0)) ⊃  t

    ENTER THEOREM PROVER

     Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀K01((N-I+ 1≤0)∨(I-K01- 1≤0)∨(
    K01≤0)∨(A[I- 1]-A[I]≤0)∨(A[I- 1]-A[K01]≥0))
          MOVE BACK TO NODE:  00047224  I ← I+ 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           42--000398E0: ∀K01((N-I≤0)∨(I-K01≤0)∨(K01≤0)∨(
    A[I]-A[K01]≥0)∨(A[I]-A[I+ 1]≤0))

          MOVE BACK TO NODE:  000471F4  ∀K01((N-I≥0∧A[
    I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.2: ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(
    N-I≥0∧K01≤0)) ⊃ ∀K01((N-I≤0)∨(I-K01≤0)∨(K01≤0)∨(A[I]-
    A[K01]≥0)∨(A[I]-A[I+ 1]≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(
    N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))
          MOVE BACK TO NODE:  00047174  ((A[I- 1]-A[I]-
     1≥0))
          IF STATEMENT, FROM FALSE BRANCH
           51--0001CFD0: ∀K01((A[I- 1]-A[I]- 1≥0)∨(N-
    I≥0∧A[I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))

          MOVE BACK TO NODE:  00047154  ((N-I≥0))
          IF STATEMENT, FROM TRUE BRANCH
           42--0001CA40: ∀K01((N-I+ 1≤0)∨(I-K01≤0)∨(K01≤0)∨(
    A[I- 1]-A[I]- 1≥0)∨(A[I]-A[K01]≥0))


          MOVE BACK TO NODE:  00047134  I ← 2
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           28--0001DCD0: ∀K01((N- 1≤0)∨(K01- 1≠0)∨(A[
    K01]-A[ 2]≤0)∨(A[ 1]-A[ 2]- 1≥0))

          MOVE BACK TO NODE:  00047114  ((N- 1≥0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.3: ((N- 1≥0)) ⊃ ∀K01((N- 1≤0)∨(K01- 1≠0)∨(A[
    K01]-A[ 2]≤0)∨(A[ 1]-A[ 2]- 1≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((N- 2≥0)∧(K01- 1=0)∧(A[K01]-
    A[ 2]- 1≥0)∧(A[ 1]-A[ 2]≤0))

    SOLVE FOR K01 IN K01- 1=0
    SUBSTITUTE  1 FOR K01
    GETTING... f
        Q.E.D.



          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀K01((N-I+ 1≤0)∨(I-K01≤0)∨(K01≤0)∨(
    A[I- 1]-A[I]- 1≥0)∨(A[I]-A[K01]≥0))
          MOVE BACK TO NODE:  00047224  I ← I+ 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           42--00021580: ∀K01((N-I≤0)∨(I-K01+ 1≤0)∨(K01≤0)∨(
    A[I]-A[I+ 1]- 1≥0)∨(A[K01]-A[I+ 1]≤0))

          MOVE BACK TO NODE:  000471F4  ∀K01((N-I≥0∧A[
    I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(N-I≥0∧K01≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.4: ∀K01((N-I≥0∧A[I]-A[K01]≥0)∨(N-I≥0∧I-K01≤0)∨(
    N-I≥0∧K01≤0)) ⊃ ∀K01((N-I≤0)∨(I-K01+ 1≤0)∨(K01≤0)∨(
    A[I]-A[I+ 1]- 1≥0)∨(A[K01]-A[I+ 1]≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((N-I- 1≥0)∧(I-K01≥0)∧(K01-
     1≥0)∧(A[I]-A[I+ 1]≤0)∧(A[K01]-A[I+ 1]- 1≥0)∧(I-K01≤0∨
    A[I]-A[K01]≥0))

      DEFINE SPECIAL FUNCTIONS
    SHOW TO BE ALWAYS FALSE:((N-I- 1≥0)∧(I-K01≥0)∧(K01-
     1≥0)∧(A[I]-A[I+ 1]≤0)∧(A[K01]-A[I+ 1]- 1≥0)∧(I-K01≤0∨
    A[I]-A[K01]≥0))

      PROOF BY CASES: (EACH MUST BE FALSE)

    CASE 1...(N-I- 1≥0∧I-K01=0∧K01- 1≥0∧A[I]-A[I+ 1]≤0∧
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
A[K01]-A[I+ 1]- 1≥0)

SOLVE FOR I IN I-K01=0
SUBSTITUTE K01 FOR I
GETTING... f
  THIS CASE -- Q.E.D.

CASE 2...(N-I- 1≥0^I-K01≥0^K01- 1≥0^A[I]-A[K01]≥0^A[
I]-A[I+ 1]≤0^A[K01]-A[I+ 1]- 1≥0)

  TRY LINEAR SOLVER


LINEAR PART:N-I- 1≥0^I-K01≥0^K01- 1≥0^A[I]-A[K01]≥0^
A[I]-A[I+ 1]≤0^A[K01]-A[I+ 1]- 1≥0 ;  NON-LINEAR PART: nil
 ;  EQUALITIES TO REMEMBER: nil
ELIMINATE VARIABLE: N

NEW SUBSYSTEM:I-K01≥0^K01- 1≥0^A[I]-A[K01]≥0^A[I]-A[
I+ 1]≤0^A[K01]-A[I+ 1]- 1≥0
ELIMINATE VARIABLE: I

NEW SUBSYSTEM:K01- 1≥0^A[I]-A[K01]≥0^A[I]-A[I+ 1]≤0^
A[K01]-A[I+ 1]- 1≥0
ELIMINATE VARIABLE: K01

NEW SUBSYSTEM:A[I]-A[K01]≥0^A[I]-A[I+ 1]≤0^A[K01]-A[
I+ 1]- 1≥0
ELIMINATE VARIABLE: A[I]
BETWEEN:A[I]-A[K01]≥0::A[I]-A[I+ 1]≤0
FORMING...((A[K01]-A[I+ 1]≤0))
END ELIMINATION -- CONCLUDE:  f
END ELIMINATION -- CONCLUDE:  f
END ELIMINATION -- CONCLUDE:  f
END ELIMINATION -- CONCLUDE:  f
      f -- Q.E.D.



 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 7.

This is one of the most "real" examples. Such a program could be the result of an assignment in an introductory programming class. Note that no initial predicate is given with the input and the verifier generates the proper one ('true') automatically.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN |     SIMPLE EXCHANGE SORT


   B:
        I ← 2;
        J ← 0;
   L:

        ASSERT( ∀L( (J=0 ∧ 2≤L ∧ L<I) ⊃ A[L-1] ≤ A[L]) );

        IF  I ≤ N  THEN
        BEGIN
             IF  A[I-1] > A[I]  THEN
             BEGIN
                  X ← A[I-1];
                  A[I-1] ← A[I];
                  A[I] ← X;
                  J ← 1;
             END;
             I ← I + 1;
             GO TO L;
        END;
        IF  J ≠ 0  THEN  GO TO B;

        ASSERT( ∀M( (2≤M ∧ M≤N) ⊃ A[M-1] ≤ A[M]) );

   END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 0004729C
          SUCCESSORS:   00047114

00047114: ASSIGNMENT STATEMENT
          LABEL: B
          TEXT:    3--00017FB0     I ← 2
          PREDECESSORS: 000470F4, 0004726C
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018000     J ← 0
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: ASSERTION STATEMENT
          LABEL: L
          TEXT:    28--0001A140     ∀L01((I-L01≤0)∨(J≠0)∨(L01-
       1≤0)∨(A[L01- 1]-A[L01]≤0))
          PREDECESSORS: 00047134, 00047234
          SUCCESSORS:   00047174

00047174: IF STATEMENT
          TEXT:    7--0001A290     ((I-N≤0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194, 0004726C

00047194: IF STATEMENT
          TEXT:    13--0001A5E0     ((A[I- 1]-A[I]- 1≥0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4, 00047234

000471B4: ASSIGNMENT STATEMENT
          TEXT:    8--0001A700     X ← A[I- 1]
          PREDECESSORS: 00047194
          SUCCESSORS:   000471D4

000471D4: ASSIGNMENT STATEMENT
          TEXT:    11--0001A880     A[I- 1] ← A[I]
          PREDECESSORS: 000471B4
          SUCCESSORS:   000471F4

000471F4: ASSIGNMENT STATEMENT
          TEXT:    8--0001A950     A[I] ← X
          PREDECESSORS: 000471D4
          SUCCESSORS:   00047214

00047214: ASSIGNMENT STATEMENT
          TEXT:    3--0001A9A0     J ← 1
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          PREDECESSORS: 000471F4
          SUCCESSORS:   00047234

00047234: ASSIGNMENT STATEMENT
          TEXT:   5--0001AA50     I ← I+ 1
          PREDECESSORS: 00047214, 00047194
          SUCCESSORS:   00047154

0004726C: IF STATEMENT
          TEXT:   5--0001AB50     ((J≠0))
          PREDECESSORS: 00047174
          SUCCESSORS:   00047114, 0004729C

0004729C: ASSERTION STATEMENT
          TEXT:   24--0001C4D0     ∀M02((N-M02+ 1≤0)∨(M02-
       1≤0)∨(A[M02- 1]-A[M02]≤0))
          PREDECESSORS: 0004726C
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  0004729C  ∀M02((N-M02+ 1≤0)∨(M02- 1≤0)∨(
    A[M02- 1]-A[M02]≤0))

          MOVE BACK TO NODE:  0004726C  ((J≠0))
          IF STATEMENT, FROM FALSE BRANCH
           28--0001C660: ∀M02((J≠0)∨(N-M02+ 1≤0)∨(M02-
     1≤0)∨(A[M02- 1]-A[M02]≤0))

          MOVE BACK TO NODE:  00047174  ((I-N≤0))
          IF STATEMENT, FROM FALSE BRANCH
           34--0001C8A0: ∀M02((I-N≤0)∨(J≠0)∨(N-M02+
     1≤0)∨(M02- 1≤0)∨(A[M02- 1]-A[M02]≤0))

          MOVE BACK TO NODE:  00047154  ∀L01((I-L01≤0)∨(
    J≠0)∨(L01- 1≤0)∨(A[L01- 1]-A[L01]≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ∀L01((I-L01≤0)∨(J≠0)∨(L01- 1≤0)∨(A[L01-
     1]-A[L01]≤0)) ⊃ ∀M02((I-N≤0)∨(J≠0)∨(N-M02+ 1≤0)∨(M02-
     1≤0)∨(A[M02- 1]-A[M02]≤0))

    ENTER THEOREM PROVER


    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((I-N- 1≥0)∧(I-M02≤0)∧(J=0)∧(
    N-M02≥0)∧(M02- 2≥0)∧(A[M02- 1]-A[M02]- 1≥0))

    SOLVE FOR J IN J=0
    SUBSTITUTE  0 FOR J
    GETTING...((I-N- 1≥0)∧(I-M02≤0)∧(N-M02≥0)∧(M02- 2≥0)∧(
    A[M02- 1]-A[M02]- 1≥0))

      DEFINE SPECIAL FUNCTIONS
    SHOW TO BE ALWAYS FALSE:((I-N- 1≥0)∧(I-M02≤0)∧(N-M02≥0)∧(
    M02- 2≥0)∧(A[M02- 1]-A[M02]- 1≥0))

      TRY LINEAR SOLVER


    LINEAR PART:I-N- 1≥0∧I-M02≤0∧N-M02≥0∧M02- 2≥0∧A[M02-
     1]-A[M02]- 1≥0 ;  NON-LINEAR PART: nil
     ;  EQUALITIES TO REMEMBER: nil
    ELIMINATE VARIABLE: I
    BETWEEN:I-N- 1≥0::I-M02≤0
    FORMING...((N-M02+ 1≤0))
    END ELIMINATION -- CONCLUDE:  f
          f -- Q.E.D.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ASSERTION  2:  00047154  ∀L01((I-L01≤0)∨(J≠0)∨(L01- 1≤0)∨(
    A[L01- 1]-A[L01]≤0))


          MOVE BACK TO NODE:  00047134  J ← 0
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           24--00023D10: ∀L01((I-L01≤0)∨(L01- 1≤0)∨(A[
    L01- 1]-A[L01]≤0))

          MOVE BACK TO NODE:  00047114  I ← 2
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           1--000258F0:  t


          MOVE BACK TO NODE:  000470F4   nil

   BEGIN -- FORM INITIAL CONDITIONS
           t


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS:  t
          MOVE BACK TO NODE:  0004726C  ((J≠0))
          IF STATEMENT, FROM TRUE BRANCH
           1--00025960:  t

          MOVE BACK TO NODE:  00047174  ((I-N≤0))
          IF STATEMENT, FROM FALSE BRANCH
           1--00025970:  t

          MOVE BACK TO NODE:  00047154  ∀L01((I-L01≤0)∨(
    J≠0)∨(L01- 1≤0)∨(A[L01- 1]-A[L01]≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.1: ∀L01((I-L01≤0)∨(J≠0)∨(L01- 1≤0)∨(A[L01-
     1]-A[L01]≤0)) ⊃  t

    ENTER THEOREM PROVER

     Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀L01((I-L01≤0)∨(J≠0)∨(L01-
     1≤0)∨(A[L01- 1]-A[L01]≤0))
          MOVE BACK TO NODE:  00047234  I ← I+ 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           28--00025980: ∀L01((I-L01+ 1≤0)∨(J≠0)∨(L01-
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1≤0)∨(A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047214  J ← 1
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         1--00027B00:  t

        MOVE BACK TO NODE:  000471F4  A[I] ← X
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         1--00027B10:  t

        MOVE BACK TO NODE:  000471D4  A[I- 1] ← A[I]
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         1--00027B20:  t

        MOVE BACK TO NODE:  000471B4  X ← A[I- 1]
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         1--00027B30:  t

        MOVE BACK TO NODE:  00047194  ((A[I- 1]-A[I]-
1≥0))
        IF STATEMENT, FROM TRUE BRANCH
         1--00027C10:  t

        MOVE BACK TO NODE:  00047174  ((I-N≤0))
        IF STATEMENT, FROM TRUE BRANCH
         1--00027C90:  t

        MOVE BACK TO NODE:  00047154  ∀L01((I-L01≤0)∨(
J≠0)∨(L01- 1≤0)∨(A[L01- 1]-A[L01]≤0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.2: ∀L01((I-L01≤0)∨(J≠0)∨(L01- 1≤0)∨(A[L01-
1]-A[L01]≤0)) ⊃  t

ENTER THEOREM PROVER

Q.E.D.


        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ∀L01((I-L01+ 1≤0)∨(J≠0)∨(L01-
1≤0)∨(A[L01- 1]-A[L01]≤0))
        MOVE BACK TO NODE:  00047194  ((A[I- 1]-A[I]-
1≥0))
        IF STATEMENT, FROM FALSE BRANCH
         40--00027CA0: ∀L01((I-L01+ 1≤0)∨(J≠0)∨(L01-
1≤0)∨(A[I- 1]-A[I]- 1≥0)∨(A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047174  ((I-N≤0))
        IF STATEMENT, FROM TRUE BRANCH
         46--00028030: ∀L01((I-L01+ 1≤0)∨(I-N- 1≥0)∨(
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
J≠0)∨(L01- 1≤0)∨(A[I- 1]-A[I]- 1≥0)∨(A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047154  ∀L01((I-L01≤0)∨(
J≠0)∨(L01- 1≤0)∨(A[L01- 1]-A[L01]≤0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.3: ∀L01((I-L01≤0)∨(J≠0)∨(L01- 1≤0)∨(A[L01-
     1]-A[L01]≤0)) ⊃ ∀L01((I-L01+ 1≤0)∨(I-N- 1≥0)∨(J≠0)∨(
     L01- 1≤0)∨(A[I- 1]-A[I]- 1≥0)∨(A[L01- 1]-A[L01]≤0))

     ENTER THEOREM PROVER

     SHOW TO BE ALWAYS FALSE:((I-L01=0)∧(I-N≤0)∧(J=0)∧(L01-
      2≥0)∧(A[I- 1]-A[I]≤0)∧(A[L01- 1]-A[L01]- 1≥0))

     SOLVE FOR I IN I-L01=0
     SUBSTITUTE L01 FOR I
     GETTING... f
         Q.E.D.



 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 8.

This program exhibits a complicated control structure. Sixteen verification conditions result. The program multiplies A*B as does example (1), but this time signed inputs are accepted and all additive operations are restricted to incrementing and decrementing by one.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |     MULTIPLY BY ADDING ONES  (Y←A*B),
          |     WHERE BOTH A AND B MAY BE SIGNED INTEGERS.
       ASSERT( A=DA );

       Y ← 0;
   ALPHA:
       IF  A ≠ 0  THEN
       BEGIN

           ASSERT(Y=(DA-A)*B);

           IF A > 0 THEN
           BEGIN
               XB ← B;
     BETA:
               IF  XB≠0  THEN
               BEGIN
                   IF XB>0 THEN
                   BEGIN
                       Y ← Y + 1;
                       XB ← XB -1;
                   END  ELSE

                   BEGIN
                       Y ← Y - 1;
                       XB ← XB + 1;
                   END;

                   ASSERT(Y=(DA-A)*B + B-XB);

                   GO TO BETA;
               END;
               A ← A - 1;
               GO TO ALPHA;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        END ELSE
        BEGIN
            XB ← B;
 GAMMA:
            IF XB ≠ 0 THEN
            BEGIN
                IF XB > 0 THEN
                BEGIN
                    Y ← Y -1;
                    XB ← XB -1;
                END ELSE
                BEGIN
                    Y ← Y + 1;
                    XB ← XB +1;
                END;

                ASSERT(Y = (DA-A)*B - B + XB);

                GO TO GAMMA;
            END;
            A ← A + 1;
            GO TO ALPHA;
        END;
    END;

    ASSERT(Y = DA*B);

END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 0004744C
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    7--00018120      ((A-DA=0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSIGNMENT STATEMENT
          TEXT:    3--00018170      Y ← 0
          PREDECESSORS: 00047114
          SUCCESSORS:   00047154

00047154: IF STATEMENT
          LABEL: ALPHA
          TEXT:    5--00018270      ((A≠0))
          PREDECESSORS: 00047134, 000472CC, 0004741C
          SUCCESSORS:   00047174, 0004744C

00047174: ASSERTION STATEMENT
          TEXT:    11--000187F0     ((A*B-DA*B+Y=0))
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194

00047194: IF STATEMENT
          TEXT:    5--000188C0      ((A- 1≥0))
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4, 000472FC

000471B4: ASSIGNMENT STATEMENT
          TEXT:    5--00018900      XB ← B
          PREDECESSORS: 00047194
          SUCCESSORS:   000471D4

000471D4: IF STATEMENT
          LABEL: BETA
          TEXT:    5--00018A30      ((XB≠0))
          PREDECESSORS: 000471B4, 0004729C
          SUCCESSORS:   000471F4, 000472CC

000471F4: IF STATEMENT
          TEXT:    5--00018B00      ((XB- 1≥0))
          PREDECESSORS: 000471D4
          SUCCESSORS:   00047214, 00047254

00047214: ASSIGNMENT STATEMENT
          TEXT:    5--00018BB0      Y ← Y+ 1
          PREDECESSORS: 000471F4
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          SUCCESSORS:   00047234

00047234: ASSIGNMENT STATEMENT
          TEXT:   5--00018C70     XB ← XB- 1
          PREDECESSORS: 00047214
          SUCCESSORS:   0004729C

0004729C: ASSERTION STATEMENT
          TEXT:   15--00019630     ((A*B-DA*B+Y-B+XB=0))
          PREDECESSORS: 0004727C, 00047234
          SUCCESSORS:   000471D4

00047254: ASSIGNMENT STATEMENT
          TEXT:   5--00018D30     Y ← Y- 1
          PREDECESSORS: 000471F4
          SUCCESSORS:   0004727C

0004727C: ASSIGNMENT STATEMENT
          TEXT:   5--00018DE0     XB ← XB+ 1
          PREDECESSORS: 00047254
          SUCCESSORS:   0004729C

000472CC: ASSIGNMENT STATEMENT
          TEXT:   5--00019720     A ← A- 1
          PREDECESSORS: 000471D4
          SUCCESSORS:   00047154

000472FC: ASSIGNMENT STATEMENT
          TEXT:   5--000197C0     XB ← B
          PREDECESSORS: 00047194
          SUCCESSORS:   00047324

00047324: IF STATEMENT
          LABEL: GAMMA
          TEXT:   5--000198C0     ((XB≠0))
          PREDECESSORS: 000472FC, 000473EC
          SUCCESSORS:   00047344, 0004741C

00047344: IF STATEMENT
          TEXT:   5--00019990     ((XB- 1≥0))
          PREDECESSORS: 00047324
          SUCCESSORS:   00047364, 000473A4

00047364: ASSIGNMENT STATEMENT
          TEXT:   5--00019A50     Y ← Y- 1
          PREDECESSORS: 00047344
          SUCCESSORS:   00047384

00047384: ASSIGNMENT STATEMENT
          TEXT:   5--00019B10     XB ← XB- 1
          PREDECESSORS: 00047364
          SUCCESSORS:   000473EC

000473EC: ASSERTION STATEMENT
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          TEXT:    15--0001A4C0     ((A*B-DA*B+Y+B-XB=0))
          PREDECESSORS: 000473CC, 00047384
          SUCCESSORS:   00047324

000473A4: ASSIGNMENT STATEMENT
          TEXT:    5--00019BC0     Y ← Y+ 1
          PREDECESSORS: 00047344
          SUCCESSORS:   000473CC

000473CC: ASSIGNMENT STATEMENT
          TEXT:    5--00019C70     XB ← XB+ 1
          PREDECESSORS: 000473A4
          SUCCESSORS:   000473EC

0004741C: ASSIGNMENT STATEMENT
          TEXT:    5--0001A5A0     A ← A+ 1
          PREDECESSORS: 00047324
          SUCCESSORS:   00047154

0004744C: ASSERTION STATEMENT
          TEXT:    8--0001A910     ((DA*B-Y=0))
          PREDECESSORS: 00047154
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS

ASSERTION  1:  0004744C  ((DA*B-Y=0))

           MOVE BACK TO NODE:  00047154  ((A≠0))
           IF STATEMENT, FROM FALSE BRANCH
            12--0001AA90: ((A≠0)∨(DA*B-Y=0))


           MOVE BACK TO NODE:  00047134  Y ← 0
           ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
            10--0001B170: ((A≠0)∨(DA*B=0))

           MOVE BACK TO NODE:  00047114  ((A-DA=0))
           ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((A-DA=0)) ⊃ ((A≠0)∨(DA*B=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A=0)∧(A-DA=0)∧(DA*B≠0))

    SOLVE FOR A IN A=0
    SUBSTITUTE  0 FOR A
    GETTING...((DA=0)∧(DA*B≠0))

    SOLVE FOR DA IN DA=0
    SUBSTITUTE  0 FOR DA
    GETTING... f
        Q.E.D.


           RETURN TO ALTERNATE BACKWARD PATH
           ASSERTION WAS: ((A≠0)∨(DA*B-Y=0))

           MOVE BACK TO NODE:  000472CC  A ← A- 1
           ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
            12--0001C600: ((A- 1≠0)∨(DA*B-Y=0))

           MOVE BACK TO NODE:  000471D4  ((XB≠0))
           IF STATEMENT, FROM FALSE BRANCH
            16--0001C760: ((A- 1≠0)∨(DA*B-Y=0)∨(XB≠0))


           MOVE BACK TO NODE:  000471B4  XB ← B
           ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
            16--0001D290: ((A- 1≠0)∨(DA*B-Y=0)∨(B≠0))

           MOVE BACK TO NODE:  00047194  ((A- 1≥0))
           IF STATEMENT, FROM TRUE BRANCH
            16--0001D450: ((A- 1≠0)∨(DA*B-Y=0)∨(B≠0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.2: ((A*B-DA*B+Y=0)) ⊃ ((A- 1≠0)∨(DA*B-Y=0)∨(
    B≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A- 1=0)∧(A*B-DA*B+Y=0)∧(DA*
    B-Y≠0)∧(B=0))

    SOLVE FOR A IN A- 1=0
    SUBSTITUTE  1 FOR A
    GETTING...((DA*B-Y≠0)∧(DA*B-Y-B=0)∧(B=0))

    SOLVE FOR Y IN DA*B-Y-B=0
    SUBSTITUTE DA*B-B FOR Y
    GETTING... f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A- 1≠0)∨(DA*B-Y=0)∨(XB≠0))
            MOVE BACK TO NODE:  0004729C  ((A*B-DA*B+Y-
    B+XB=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.3: ((A*B-DA*B+Y-B+XB=0)) ⊃ ((A- 1≠0)∨(DA*B-Y=0)∨(
    XB≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A- 1=0)∧(A*B-DA*B+Y-B+XB=0)∧(
    DA*B-Y≠0)∧(XB=0))

    SOLVE FOR A IN A- 1=0
    SUBSTITUTE  1 FOR A
    GETTING...((DA*B-Y≠0)∧(DA*B-Y-XB=0)∧(XB=0))

    SOLVE FOR Y IN DA*B-Y-XB=0
    SUBSTITUTE DA*B-XB FOR Y
    GETTING... f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A≠0)∨(DA*B-Y=0))
            MOVE BACK TO NODE:  0004741C  A ← A+ 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 12--00022B80: ((A+ 1≠0)∨(DA*B-Y=0))

MOVE BACK TO NODE:  00047324  ((XB≠0))
IF STATEMENT, FROM FALSE BRANCH
 16--00022CE0: ((A+ 1≠0)∨(DA*B-Y=0)∨(XB≠0))


MOVE BACK TO NODE:  000472FC  XB ← B
ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
 16--00023810: ((A+ 1≠0)∨(DA*B-Y=0)∨(B≠0))

MOVE BACK TO NODE:  00047194  ((A- 1≥0))
IF STATEMENT, FROM FALSE BRANCH
 16--00023980: ((A+ 1≠0)∨(DA*B-Y=0)∨(B≠0))

MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
ASSERTION STATEMENT-FORM VER. COND.


THEOREM 1.4: ((A*B-DA*B+Y=0)) ⊃ ((A+ 1≠0)∨(DA*B-Y=0)∨(
    B≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A+ 1=0)∧(A*B-DA*B+Y=0)∧(DA*
    B-Y≠0)∧(B=0))

    SOLVE FOR A IN A+ 1=0
    SUBSTITUTE -1 FOR A
    GETTING...((DA*B-Y≠0)∧(DA*B-Y+B=0)∧(B=0))

    SOLVE FOR Y IN DA*B-Y+B=0
    SUBSTITUTE DA*B+B FOR Y
    GETTING... f
        Q.E.D.



        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ((A+ 1≠0)∨(DA*B-Y=0)∨(XB≠0))
        MOVE BACK TO NODE:  000473EC  ((A*B-DA*B+Y+
    B-XB=0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM 1.5: ((A*B-DA*B+Y+B-XB=0)) ⊃ ((A+ 1≠0)∨(DA*B-Y=0)∨(
    XB≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A+ 1=0)∧(A*B-DA*B+Y+B-XB=0)∧(
    DA*B-Y≠0)∧(XB=0))

    SOLVE FOR A IN A+ 1=0
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SUBSTITUTE -1 FOR A
GETTING... ((DA*B-Y≠0)∧(DA*B-Y+XB=0)∧(XB=0))

SOLVE FOR Y IN DA*B-Y+XB=0
SUBSTITUTE DA*B+XB FOR Y
GETTING... f
Q.E.D.

ASSERTION 2: 000473EC ((A*B-DA*B+Y+B-XB=0))

MOVE BACK TO NODE: 000473CC XB ← XB+ 1
ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
15--000296B0: ((A*B-DA*B+Y+B-XB- 1=0))

MOVE BACK TO NODE: 000473A4 Y ← Y+ 1
ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
15--0002A200: ((A*B-DA*B+Y+B-XB=0))

MOVE BACK TO NODE: 00047344 ((XB- 1≥0))
IF STATEMENT, FROM FALSE BRANCH
19--0002A370: ((A*B-DA*B+Y+B-XB=0)∨(XB- 1≥0))

MOVE BACK TO NODE: 00047324 ((XB≠0))
IF STATEMENT, FROM TRUE BRANCH
19--0002A540: ((A*B-DA*B+Y+B-XB=0)∨(XB≥0))

MOVE BACK TO NODE: 000472FC XB ← B
ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
15--0002B360: ((A*B-DA*B+Y=0)∨(B≥0))

MOVE BACK TO NODE: 00047194 ((A- 1≥0))
IF STATEMENT, FROM FALSE BRANCH
19--0002B4F0: ((A- 1≥0)∨(A*B-DA*B+Y=0)∨(B≥0))

MOVE BACK TO NODE: 00047174 ((A*B-DA*B+Y=0))
ASSERTION STATEMENT-FORM VER. COND.

THEOREM 2.1: ((A*B-DA*B+Y=0)) ⊃ ((A- 1≥0)∨(A*B-DA*B+Y=0)∨(B≥0))

ENTER THEOREM PROVER

SHOW TO BE ALWAYS FALSE: f
Q.E.D.

RETURN TO ALTERNATE BACKWARD PATH

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          ASSERTION WAS: ((A*B-DA*B+Y+B-XB=0)∨(XB≥0))
          MOVE BACK TO NODE:  000473EC  ((A*B-DA*B+Y+
    B-XB=0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.2: ((A*B-DA*B+Y+B-XB=0)) ⊃ ((A*B-DA*B+Y+B-XB=0)∨(
    XB≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ((A*B-DA*B+Y+B-XB=0))
          MOVE BACK TO NODE:  00047384  XB ← XB- 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           15--0002C4A0: ((A*B-DA*B+Y+B-XB+ 1=0))

          MOVE BACK TO NODE:  00047364  Y ← Y- 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           15--0002CFF0: ((A*B-DA*B+Y+B-XB=0))

          MOVE BACK TO NODE:  00047344  ((XB- 1≥0))
          IF STATEMENT, FROM TRUE BRANCH
           19--0002D1B0: ((A*B-DA*B+Y+B-XB=0)∨(XB≤0))

          MOVE BACK TO NODE:  00047324  ((XB≠0))
          IF STATEMENT, FROM TRUE BRANCH
           19--0002D380: ((A*B-DA*B+Y+B-XB=0)∨(XB≤0))


          MOVE BACK TO NODE:  000472FC  XB ← B
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           15--0002E1C0: ((A*B-DA*B+Y=0)∨(B≤0))

          MOVE BACK TO NODE:  00047194  ((A- 1≥0))
          IF STATEMENT, FROM FALSE BRANCH
           19--0002E350: ((A- 1≥0)∨(A*B-DA*B+Y=0)∨(B≤0))

          MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.3: ((A*B-DA*B+Y=0)) ⊃ ((A- 1≥0)∨(A*B-DA*B+Y=0)∨(
    B≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A*B-DA*B+Y+B-XB=0)∨(XB≤0))
            MOVE BACK TO NODE:  000473EC  ((A*B-DA*B+Y+
    B-XB=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.4: ((A*B-DA*B+Y+B-XB=0)) ⊃ ((A*B-DA*B+Y+B-XB=0)∨(
    XB≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.



ASSERTION  3:  0004729C  ((A*B-DA*B+Y-B+XB=0))


            MOVE BACK TO NODE:  0004727C  XB ← XB+ 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--0002F4E0: ((A*B-DA*B+Y-B+XB+ 1=0))

            MOVE BACK TO NODE:  00047254  Y ← Y- 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--00030030: ((A*B-DA*B+Y-B+XB=0))

            MOVE BACK TO NODE:  000471F4  ((XB- 1≥0))
            IF STATEMENT, FROM FALSE BRANCH
             19--000301A0: ((A*B-DA*B+Y-B+XB=0)∨(XB- 1≥0))

            MOVE BACK TO NODE:  000471D4  ((XB≠0))
            IF STATEMENT, FROM TRUE BRANCH
             19--00030370: ((A*B-DA*B+Y-B+XB=0)∨(XB≥0))


            MOVE BACK TO NODE:  000471B4  XB ← B
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--00031190: ((A*B-DA*B+Y=0)∨(B≥0))

            MOVE BACK TO NODE:  00047194  ((A- 1≥0))
            IF STATEMENT, FROM TRUE BRANCH
             19--00031370: ((A≤0)∨(A*B-DA*B+Y=0)∨(B≥0))

            MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  3.1: ((A*B-DA*B+Y=0)) ⊃ ((A≤0)∨(A*B-DA*B+Y=0)∨(
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
B≥0))

ENTER THEOREM PROVER

SHOW TO BE ALWAYS FALSE: f
    Q.E.D.


        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ((A*B-DA*B+Y-B+XB=0)∨(XB≥0))
        MOVE BACK TO NODE:  0004729C  ((A*B-DA*B+Y-
B+XB=0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM 3.2: ((A*B-DA*B+Y-B+XB=0)) ⊃ ((A*B-DA*B+Y-B+XB=0)∨(
    XB≥0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A*B-DA*B+Y-B+XB=0))
            MOVE BACK TO NODE:  00047234  XB ← XB- 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--00032320: ((A*B-DA*B+Y-B+XB- 1=0))

            MOVE BACK TO NODE:  00047214  Y ← Y+ 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--00032E70: ((A*B-DA*B+Y-B+XB=0))

            MOVE BACK TO NODE:  000471F4  ((XB- 1≥0))
            IF STATEMENT, FROM TRUE BRANCH
             19--00033030: ((A*B-DA*B+Y-B+XB=0)∨(XB≤0))

            MOVE BACK TO NODE:  000471D4  ((XB≠0))
            IF STATEMENT, FROM TRUE BRANCH
             19--00033200: ((A*B-DA*B+Y-B+XB=0)∨(XB≤0))


            MOVE BACK TO NODE:  000471B4  XB ← B
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             15--00034040: ((A*B-DA*B+Y=0)∨(B≤0))

            MOVE BACK TO NODE:  00047194  ((A- 1≥0))
            IF STATEMENT, FROM TRUE BRANCH
             19--00034220: ((A≤0)∨(A*B-DA*B+Y=0)∨(B≤0))

            MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                ASSERTION STATEMENT-FORM VER. COND.

THEOREM  3.3: ((A*B-DA*B+Y=0)) ⊃ ((A≤0)∨(A*B-DA*B+Y=0)∨(
    B≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A*B-DA*B+Y-B+XB=0)∨(XB≤0))
            MOVE BACK TO NODE:  0004729C  ((A*B-DA*B+Y-
    B+XB=0))
            ASSERTION STATEMENT-FORM VER. COND.

THEOREM  3.4: ((A*B-DA*B+Y-B+XB=0)) ⊃ ((A*B-DA*B+Y-B+XB=0)∨(
    XB≤0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE: f
        Q.E.D.


ASSERTION  4:  00047174  ((A*B-DA*B+Y=0))

            MOVE BACK TO NODE:  00047154  ((A≠0))
            IF STATEMENT, FROM TRUE BRANCH
             15--000348B0: ((A=0)∨(A*B-DA*B+Y=0))


            MOVE BACK TO NODE:  00047134  Y ← 0
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             13--00035340: ((A=0)∨(A*B-DA*B=0))

            MOVE BACK TO NODE:  00047114  ((A-DA=0))
            ASSERTION STATEMENT-FORM VER. COND.

THEOREM  4.1: ((A-DA=0)) ⊃ ((A=0)∨(A*B-DA*B=0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A≠0)∧(A-DA=0)∧(A*B-DA*B≠0))

    SOLVE FOR A IN A-DA=0
    SUBSTITUTE DA FOR A
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GETTING... f
    Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ((A=0)∨(A*B-DA*B+Y=0))

          MOVE BACK TO NODE:  000472CC  A ← A- 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           17--00036C70: ((A- 1=0)∨(A*B-DA*B+Y-B=0))

          MOVE BACK TO NODE:  000471D4  ((XB≠0))
          IF STATEMENT, FROM FALSE BRANCH
           21--00036E20: ((A- 1=0)∨(A*B-DA*B+Y-B=0)∨(
XB≠0))


          MOVE BACK TO NODE:  000471B4  XB ← B
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           21--00037F60: ((A- 1=0)∨(A*B-DA*B+Y-B=0)∨(
B≠0))

          MOVE BACK TO NODE:  00047194  ((A- 1≥0))
          IF STATEMENT, FROM TRUE BRANCH
           21--00038170: ((A- 1≤0)∨(A*B-DA*B+Y-B=0)∨(
B≠0))

          MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  4.2: ((A*B-DA*B+Y=0)) ⊃ ((A- 1≤0)∨(A*B-DA*B+Y-
    B=0)∨(B≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A- 2≥0)∧(A*B-DA*B+Y=0)∧(A*
    B-DA*B+Y-B≠0)∧(B=0))

    SOLVE FOR Y IN A*B-DA*B+Y=0
    SUBSTITUTE -A*B+DA*B FOR Y
    GETTING... f
        Q.E.D.


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ((A- 1=0)∨(A*B-DA*B+Y-B=0)∨(
    XB≠0))
          MOVE BACK TO NODE:  0004729C  ((A*B-DA*B+Y-
    B+XB=0))
          ASSERTION STATEMENT-FORM VER. COND.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
THEOREM  4.3: ((A*B-DA*B+Y-B+XB=0)) ⊃ ((A- 1=0)∨(A*B-DA*
    B+Y-B=0)∨(XB≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A- 1≠0)∧(A*B-DA*B+Y-B≠0)∧(
    A*B-DA*B+Y-B+XB=0)∧(XB=0))

    SOLVE FOR Y IN A*B-DA*B+Y-B+XB=0
    SUBSTITUTE -A*B+DA*B+B-XB FOR Y
    GETTING... f
        Q.E.D.


            RETURN TO ALTERNATE BACKWARD PATH
            ASSERTION WAS: ((A=0)∨(A*B-DA*B+Y=0))
            MOVE BACK TO NODE:  0004741C  A ← A+ 1
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             17--0001BA30: ((A+ 1=0)∨(A*B-DA*B+Y+B=0))

            MOVE BACK TO NODE:  00047324  ((XB≠0))
            IF STATEMENT, FROM FALSE BRANCH
             21--0001BCA0: ((A+ 1=0)∨(A*B-DA*B+Y+B=0)∨(
    XB≠0))


            MOVE BACK TO NODE:  000472FC  XB ← B
            ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
             21--0001C9E0: ((A+ 1=0)∨(A*B-DA*B+Y+B=0)∨(
    B≠0))

            MOVE BACK TO NODE:  00047194  ((A- 1≥0))
            IF STATEMENT, FROM FALSE BRANCH
             25--0001CFC0: ((A- 1≥0)∨(A+ 1=0)∨(A*B-DA*B+
    Y+B=0)∨(B≠0))

            MOVE BACK TO NODE:  00047174  ((A*B-DA*B+Y=0))
            ASSERTION STATEMENT-FORM VER. COND.


THEOREM  4.4: ((A*B-DA*B+Y=0)) ⊃ ((A- 1≥0)∨(A+ 1=0)∨(A*
    B-DA*B+Y+B=0)∨(B≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A≤0)∧(A+ 1≠0)∧(A*B-DA*B+Y=0)∧(
    A*B-DA*B+Y+B≠0)∧(B=0))

    SOLVE FOR Y IN A*B-DA*B+Y=0
    SUBSTITUTE -A*B+DA*B FOR Y
    GETTING... f
        Q.E.D.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ((A+ 1=0)∨(A*B-DA*B+Y+B=0)∨(
    XB≠0))
          MOVE BACK TO NODE:  000473EC  ((A*B-DA*B+Y+
    B-XB=0))
          ASSERTION STATEMENT-FORM VER. COND.

 THEOREM  4.5: ((A*B-DA*B+Y+B-XB=0)) ⊃ ((A+ 1=0)∨(A*B-DA*
    B+Y+B=0)∨(XB≠0))

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((A+ 1≠0)∧(A*B-DA*B+Y+B≠0)∧(
    A*B-DA*B+Y+B-XB=0)∧(XB=0))

    SOLVE FOR Y IN A*B-DA*B+Y+B-XB=0
    SUBSTITUTE -A*B+DA*B-B+XB FOR Y
    GETTING... f
        Q.E.D.


 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 9.

The verification of this program is unsuccessful. The assertions involve more than one quantifier and the theorem prover just rejects the verification conditions with the response "NO HELP." The size of some of the verification conditions is alarming. An alternative method for dealing with arrays and more powerful simplification may help reduce these formulas to a managable size. This is discussed in Chapter III.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |     SORT BY SUCCESSIVELY FINDING THE LARGEST

       I ← 1;

    L1:
       IF I < N THEN
       BEGIN
           J ← I + 1;
           X ← A[I];
           K ← I;
    L2:
           ASSERT(∀L(  (2≤L ∧ L<I)  ⊃ A[L-1] ≤ A[L] )   ∧
                  ∀M(  (I ≤ M  ∧  M < J) ⊃  X≤ A[M] )  );

          IF J ≤ N THEN
          BEGIN
               IF X ≤ A[J] THEN GO TO L3;
               X ← A[J];
               K ← J;
      L3:
               J ← J + 1;
              GO TO L2;
           END;

           A[K] ← A[I];
           A[I] ← X;
           I ← I + 1;
           GO TO L1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
   END;

ASSERT( ∀L(  (2 ≤ L ∧ L ≤ N)  ⊃  A[L-1] ≤ A[L] )  );
END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HEXADECIMAL PROGRAM MAP

```
000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 000472FC
          SUCCESSORS:   00047114

00047114: ASSIGNMENT STATEMENT
          TEXT:    3--00017F80     I ← 1
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: IF STATEMENT
          LABEL: L1
          TEXT:    7--00018100     ((I-N+ 1≤0))
          PREDECESSORS: 00047114, 000472D4
          SUCCESSORS:   00047154, 000472FC

00047154: ASSIGNMENT STATEMENT
          TEXT:    5--000181B0     J ← I+ 1
          PREDECESSORS: 00047134
          SUCCESSORS:   00047174

00047174: ASSIGNMENT STATEMENT
          TEXT:    8--00018280     X ← A[I]
          PREDECESSORS: 00047154
          SUCCESSORS:   00047194

00047194: ASSIGNMENT STATEMENT
          TEXT:    5--000182F0     K ← I
          PREDECESSORS: 00047174
          SUCCESSORS:   000471B4

000471B4: ASSERTION STATEMENT
          LABEL: L2
          TEXT:    123--0001B470     ∀L01∀M02((I-M02- 1≥0∧L01-
        1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02-
        1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧
       L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧
       L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))
          PREDECESSORS: 00047194, 0004725C
          SUCCESSORS:   000471D4

000471D4: IF STATEMENT
          TEXT:    7--00020190     ((N-J≥0))
          PREDECESSORS: 000471B4
          SUCCESSORS:   000471F4, 0004728C

000471F4: IF STATEMENT
          TEXT:    10--000203A0     ((X-A[J]≤0))
          PREDECESSORS: 000471D4
          SUCCESSORS:   0004725C, 00047214
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
0004725C: ASSIGNMENT STATEMENT
          LABEL: L3
          TEXT:   5--000205F0    J ← J+ 1
          PREDECESSORS: 0004723C, 000471F4
          SUCCESSORS:   000471B4

00047214: ASSIGNMENT STATEMENT
          TEXT:   8--000204A0    X ← A[J]
          PREDECESSORS: 000471F4
          SUCCESSORS:   0004723C

0004723C: ASSIGNMENT STATEMENT
          TEXT:   5--00020510    K ← J
          PREDECESSORS: 00047214
          SUCCESSORS:   0004725C

0004728C: ASSIGNMENT STATEMENT
          TEXT:   11--00020750    A[K] ← A[I]
          PREDECESSORS: 000471D4
          SUCCESSORS:   000472B4

000472B4: ASSIGNMENT STATEMENT
          TEXT:   8--00020820    A[I] ← X
          PREDECESSORS: 0004728C
          SUCCESSORS:   000472D4

000472D4: ASSIGNMENT STATEMENT
          TEXT:   5--000208D0    I ← I+ 1
          PREDECESSORS: 000472B4
          SUCCESSORS:   00047134

000472FC: ASSERTION STATEMENT
          TEXT:   24--00022250    ∀L03((N-L03+ 1≤0)∨(L03-
       1≤0)∨(A[L03- 1]-A[L03]≤0))
          PREDECESSORS: 00047134
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  000472FC  ∀L03((N-L03+ 1≤0)∨(L03- 1≤0)∨(
    A[L03- 1]-A[L03]≤0))

          MOVE BACK TO NODE:  00047134  ((I-N+ 1≤0))
          IF STATEMENT, FROM FALSE BRANCH
           30--000223E0: ∀L03((I-N+ 1≤0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(A[L03- 1]-A[L03]≤0))


          MOVE BACK TO NODE:  00047114  I ← 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           28--00022820: ∀L03((N- 2≥0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(A[L03- 1]-A[L03]≤0))

          MOVE BACK TO NODE:  000470F4   nil

   BEGIN -- FORM INITIAL CONDITIONS
          ∀L03((N- 2≥0)∨(N-L03+ 1≤0)∨(L03- 1≤0)∨(A[L03-
     1]-A[L03]≤0))


          RETURN TO ALTERNATE BACKWARD PATH
          ASSERTION WAS: ∀L03((I-N+ 1≤0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(A[L03- 1]-A[L03]≤0))
          MOVE BACK TO NODE:  000472D4  I ← I+ 1
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           30--00023B20: ∀L03((I-N+ 2≤0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(A[L03- 1]-A[L03]≤0))

          MOVE BACK TO NODE:  000472B4  A[I] ← X
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           68--00024F90: ∀L03((I-N+ 2≤0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(I-L03=0∧X-A[L03- 1]≥0)∨(I-L03+ 1=0∧X-A[L03]≤0)∨(
    I-L03+ 1≠0∧I-L03≠0∧A[L03- 1]-A[L03]≤0))

          MOVE BACK TO NODE:  0004728C  A[K] ← A[I]
          ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
           180--000297C0: ∀L03((I-N+ 2≤0)∨(N-L03+ 1≤0)∨(
    L03- 1≤0)∨(I-L03=0∧X-A[L03- 1]≥0∧K-L03+ 1≠0)∨(I-L03=0∧
    X-A[I]≥0∧K-L03+ 1=0)∨(I-L03+ 1=0∧X-A[I]≤0∧K-L03=0)∨(
    I-L03+ 1=0∧X-A[L03]≤0∧K-L03≠0)∨(I-L03+ 1≠0∧I-L03≠0∧
    K-L03=0∧A[L03- 1]-A[I]≤0)∨(I-L03+ 1≠0∧I-L03≠0∧K-L03+
     1=0∧A[I]-A[L03]≤0)∨(I-L03+ 1≠0∧I-L03≠0∧K-L03+ 1≠0∧
    K-L03≠0∧A[L03- 1]-A[L03]≤0))

          MOVE BACK TO NODE:  000471D4  ((N-J≥0))
          IF STATEMENT, FROM FALSE BRANCH
           186--000395D0: ∀L03((I-N+ 2≤0)∨(N-J≥0)∨(N-
    L03+ 1≤0)∨(L03- 1≤0)∨(I-L03=0∧X-A[L03- 1]≥0∧K-L03+
     1≠0)∨(I-L03=0∧X-A[I]≥0∧K-L03+ 1=0)∨(I-L03+ 1=0∧X-A[
    I]≤0∧K-L03=0)∨(I-L03+ 1=0∧X-A[L03]≤0∧K-L03≠0)∨(I-L03+
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1≠0∧I-L03≠0∧K-L03=0∧A[L03- 1]-A[I]≤0)∨(I-L03+ 1≠0∧
I-L03≠0∧K-L03+ 1=0∧A[I]-A[L03]≤0)∨(I-L03+ 1≠0∧I-L03≠0∧
K-L03+ 1≠0∧K-L03≠0∧A[L03- 1]-A[L03]≤0))

        MOVE BACK TO NODE:  000471B4  ∀L01∀M02((I-M02-
 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧
I-M02- 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(
J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[
M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))
        ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧
    A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧J-
    M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧
    A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
    A[L01- 1]-A[L01]≤0)) ⊃ ∀L03((I-N+ 2≤0)∨(N-J≥0)∨(N-L03+
     1≤0)∨(L03- 1≤0)∨(I-L03=0∧X-A[L03- 1]≥0∧K-L03+ 1≠0)∨(
    I-L03=0∧X-A[I]≥0∧K-L03+ 1=0)∨(I-L03+ 1=0∧X-A[I]≤0∧K-
    L03=0)∨(I-L03+ 1=0∧X-A[L03]≤0∧K-L03≠0)∨(I-L03+ 1≠0∧
    I-L03≠0∧K-L03=0∧A[L03- 1]-A[I]≤0)∨(I-L03+ 1≠0∧I-L03≠0∧
    K-L03+ 1=0∧A[I]-A[L03]≤0)∨(I-L03+ 1≠0∧I-L03≠0∧K-L03+
     1≠0∧K-L03≠0∧A[L03- 1]-A[L03]≤0))

    ENTER THEOREM PROVER

    NO HELP


ASSERTION  2:  000471B4  ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(
    I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(
    I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01-
     1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01-
     1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))


        MOVE BACK TO NODE:  00047194  K ← I
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         123--0001C860: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02-
 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧
L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧
L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))


        MOVE BACK TO NODE:  00047174  X ← A[I]
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         132--00027230: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧A[I]-
A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-
L01≤0∧A[I]-A[M02]≤0)∨(J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧A[
L01- 1]-A[L01]≤0)∨(A[L01- 1]-A[L01]≤0∧A[I]-A[M02]≤0))
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        MOVE BACK TO NODE:  00047154  J ← I+ 1
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         132--00031CF0: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧A[I]-
A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧A[I]-A[M02]≤0)∨(
I-L01≤0∧I-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧A[I]-A[M02]≤0)∨(
I-M02+ 1≤0∧L01- 1≤0)∨(I-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047134  ((I-N+ 1≤0))
        IF STATEMENT, FROM TRUE BRANCH
         138--00019FB0: ∀L01∀M02((I-N≥0)∨(I-M02- 1≥0∧
L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧
A[I]-A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧A[I]-A[
M02]≤0)∨(I-L01≤0∧I-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧A[
I]-A[M02]≤0)∨(I-M02+ 1≤0∧L01- 1≤0)∨(I-M02+ 1≤0∧A[L01-
 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047114  I ← 1
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         112--0001D840: ∀L01∀M02((N- 1≤0)∨(M02- 2≥0∧
A[L01- 1]-A[L01]≤0)∨(L01- 1≥0∧M02- 2≥0)∨(L01- 1≥0∧A[
M02]-A[ 1]≥0)∨(L01- 1≥0∧M02≤0)∨(L01- 1≤0∧M02- 2≥0)∨(
L01- 1≤0∧A[M02]-A[ 1]≥0)∨(L01- 1≤0∧M02≤0)∨(M02≤0∧A[
L01- 1]-A[L01]≤0)∨(A[L01- 1]-A[L01]≤0∧A[M02]-A[ 1]≥0))

        MOVE BACK TO NODE:  000470F4  nil

BEGIN -- FORM INITIAL CONDITIONS
        ∀L01∀M02((N- 1≤0)∨(M02- 2≥0∧A[L01- 1]-A[L01]≤0)∨(
 L01- 1≥0∧M02- 2≥0)∨(L01- 1≥0∧A[M02]-A[ 1]≥0)∨(L01-
  1≥0∧M02≤0)∨(L01- 1≤0∧M02- 2≥0)∨(L01- 1≤0∧A[M02]-A[
  1]≥0)∨(L01- 1≤0∧M02≤0)∨(M02≤0∧A[L01- 1]-A[L01]≤0)∨(
 A[L01- 1]-A[L01]≤0∧A[M02]-A[ 1]≥0))

        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ∀L01∀M02((I-N≥0)∨(I-M02- 1≥0∧
L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧
A[I]-A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧A[I]-A[
M02]≤0)∨(I-L01≤0∧I-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧A[
I]-A[M02]≤0)∨(I-M02+ 1≤0∧L01- 1≤0)∨(I-M02+ 1≤0∧A[L01-
 1]-A[L01]≤0))
        MOVE BACK TO NODE:  000472D4  I ← I+ 1
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         138--0002A790: ∀L01∀M02((I-N+ 1≥0)∨(I-M02≥0∧
L01- 1≤0)∨(I-M02≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧A[
M02]-A[I+ 1]≥0)∨(A[L01- 1]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(
I-L01+ 1≤0∧I-M02≥0)∨(I-L01+ 1≤0∧A[M02]-A[I+ 1]≥0)∨(
I-L01+ 1≤0∧I-M02+ 2≤0)∨(I-M02+ 2≤0∧L01- 1≤0)∨(I-M02+
 2≤0∧A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  000472B4  A[I] ← X
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

251--00019730: ∀L01∀M02((I-N+ 1≥0)∨(I-L01=0∧
I-M02≥0∧X-A[L01- 1]≥0)∨(I-L01=0∧I-M02+ 2≤0∧X-A[L01-
1]≥0)∨(I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧A[M02]-A[I+
1]≥0)∨(I-M02≥0∧L01- 1≤0)∨(I-L01+ 1≤0∧I-M02≥0)∨(I-L01+
1≤0∧I-M02+ 2≤0)∨(I-L01+ 1≤0∧I-M02≠0∧A[M02]-A[I+ 1]≥0)∨(
I-M02+ 2≤0∧L01- 1≤0)∨(I-M02≠0∧L01- 1≤0∧A[M02]-A[I+
1]≥0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02≥0∧A[L01- 1]-A[L01]≤0)∨(
I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧A[L01- 1]-A[L01]≤0)∨(
I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧A[L01- 1]-A[L01]≤0∧A[M02]-
A[I+ 1]≥0))

MOVE BACK TO NODE: 0004728C A[K] ← A[I]
ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
1534--0001D0B0: ∀L01∀M02((I-N+ 1≥0)∨(I-L01=0∧
I-M02≥0∧X-A[L01- 1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧I-M02≥0∧
X-A[I]≥0∧K-L01+ 1=0)∨(I-L01=0∧I-M02+ 2≤0∧X-A[L01-
1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧I-M02+ 2≤0∧X-A[I]≥0∧K-L01+
1=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧K-L01+
1≠0∧K-M02=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧
K-L01+ 1≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01=0∧
I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02=0)∨(I-K+ 1=0∧I-L01=0∧
I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧A[I]-A[M02]≤0)∨(
I-K+ 1=0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0)∨(I-K+ 1=0∧I-L01+
1≤0∧I-M02≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-M02≠0∧
K-M02=0∧L01- 1≤0)∨(I-K+ 1=0∧I-M02≠0∧K-M02≠0∧L01- 1≤0∧
A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01=0∧K-M02=0∧A[L01- 1]-A[I]≤0)∨(I-K+ 1=0∧I-L01+
1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧A[L01- 1]-A[I]≤0∧
A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0)∨(I-K+ 1=0∧I-L01+
1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02≠0∧A[I]-A[L01]≤0∧
A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01+ 1≠0∧K-L01≠0∧K-M02=0∧A[L01- 1]-A[L01]≤0)∨(I-K+
1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧
K-M02≠0∧A[L01- 1]-A[L01]≤0∧A[I]-A[M02]≤0)∨(I-M02≥0∧
L01- 1≤0)∨(I-L01+ 1≤0∧I-M02≥0)∨(I-L01+ 1≤0∧I-M02+
2≤0)∨(I-M02+ 2≤0∧L01- 1≤0)∨(I-K+ 1≠0∧I-L01=0∧I-M02≠0∧
X-A[L01- 1]≥0∧K-L01+ 1≠0∧K-M02=0∧A[I]-A[I+ 1]≥0)∨(I-
K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧K-L01+ 1≠0∧K-M02≠0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[I]≥0∧
K-L01+ 1=0∧K-M02=0∧A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01=0∧
I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧A[M02]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0∧A[I]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02≠0∧A[M02]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-M02≠0∧K-M02=0∧L01- 1≤0∧A[I]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-M02≠0∧K-M02≠0∧L01- 1≤0∧A[M02]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02=0∧
A[L01- 1]-A[I]≤0∧A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+
1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧A[L01- 1]-A[I]≤0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0∧A[I]-A[I+ 1]≥0)∨(I-
K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02≠0∧
A[I]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧
I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02=0∧A[L01-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 1]-A[L01]≤0∧A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-
L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02≠0∧A[L01- 1]-
A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02≥0∧
K-L01=0∧A[L01- 1]-A[I]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02≥0∧
K-L01+ 1=0∧A[I]-A[L01]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02≥0∧
K-L01+ 1≠0∧K-L01≠0∧A[L01- 1]-A[L01]≤0)∨(I-L01+ 1≠0∧
I-L01≠0∧I-M02+ 2≤0∧K-L01=0∧A[L01- 1]-A[I]≤0)∨(I-L01+
 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01+ 1=0∧A[I]-A[L01]≤0)∨(
I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01+ 1≠0∧K-L01≠0∧A[
L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  000471D4  ((N-J≥0))
        IF STATEMENT, FROM FALSE BRANCH
         1540--0002F290: ∀L01∀M02((I-N+ 1≥0)∨(N-J≥0)∨(
I-L01=0∧I-M02≥0∧X-A[L01- 1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧
I-M02≥0∧X-A[I]≥0∧K-L01+ 1=0)∨(I-L01=0∧I-M02+ 2≤0∧X-
A[L01- 1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧I-M02+ 2≤0∧X-A[I]≥0∧
K-L01+ 1=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧
K-L01+ 1≠0∧K-M02=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01-
 1]≥0∧K-L01+ 1≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-
L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02=0)∨(I-K+ 1=0∧
I-L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧A[I]-A[
M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0)∨(I-K+
 1=0∧I-L01+ 1≤0∧I-M02≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+
 1=0∧I-M02≠0∧K-M02=0∧L01- 1≤0)∨(I-K+ 1=0∧I-M02≠0∧K-
M02≠0∧L01- 1≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧
I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02=0∧A[L01- 1]-A[I]≤0)∨(
I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧
A[L01- 1]-A[I]≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧
I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0)∨(
I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02≠0∧
A[I]-A[L01]≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-
L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02=0∧A[L01- 1]-
A[L01]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+
 1≠0∧K-L01≠0∧K-M02≠0∧A[L01- 1]-A[L01]≤0∧A[I]-A[M02]≤0)∨(
I-M02≥0∧L01- 1≤0)∨(I-L01+ 1≤0∧I-M02≥0)∨(I-L01+ 1≤0∧
I-M02+ 2≤0)∨(I-M02+ 2≤0∧L01- 1≤0)∨(I-K+ 1≠0∧I-L01=0∧
I-M02≠0∧X-A[L01- 1]≥0∧K-L01+ 1≠0∧K-M02=0∧A[I]-A[I+
 1]≥0)∨(I-K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧K-L01+
 1≠0∧K-M02≠0∧A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01=0∧I-
M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02≠0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-M02≠0∧K-M02=0∧L01- 1≤0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-M02≠0∧K-M02≠0∧L01- 1≤0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01=0∧K-M02=0∧A[L01- 1]-A[I]≤0∧A[I]-A[I+ 1]≥0)∨(I-
K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧A[
L01- 1]-A[I]≤0∧A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+
 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01+ 1=0∧K-M02≠0∧A[I]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

K-M02=0∧A[L01- 1]-A[L01]≤0∧A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧
I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02≠0∧
A[L01- 1]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(I-L01+ 1≠0∧I-
L01≠0∧I-M02≥0∧K-L01=0∧A[L01- 1]-A[I]≤0)∨(I-L01+ 1≠0∧
I-L01≠0∧I-M02≥0∧K-L01+ 1=0∧A[I]-A[L01]≤0)∨(I-L01+
1≠0∧I-L01≠0∧I-M02≥0∧K-L01+ 1≠0∧K-L01≠0∧A[L01- 1]-A[
L01]≤0))∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01=0∧A[L01-
1]-A[I]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01+
1=0∧A[I]-A[L01]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧
K-L01+ 1≠0∧K-L01≠0∧A[L01- 1]-A[L01]≤0))

MOVE BACK TO NODE: 000471B4 ∀L01∀M02((I-M02-
1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧
I-M02- 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(
J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[
M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))

ASSERTION STATEMENT-FORM VER. COND.

THEOREM 2.1: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧
A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧J-
M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧
A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
A[L01- 1]-A[L01]≤0)) ⊃ ∀L01∀M02((I-N+ 1≥0)∨(N-J≥0)∨(
I-L01=0∧I-M02≥0∧X-A[L01- 1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧
I-M02≥0∧X-A[I]≥0∧K-L01+ 1=0)∨(I-L01=0∧I-M02+ 2≤0∧X-
A[L01- 1]≥0∧K-L01+ 1≠0)∨(I-L01=0∧I-M02+ 2≤0∧X-A[I]≥0∧
K-L01+ 1=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧
K-L01+ 1≠0∧K-M02=0)∨(I-K+ 1=0∧I-L01=0∧I-M02≠0∧X-A[L01-
1]≥0∧K-L01+ 1≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-
L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02=0)∨(I-K+ 1=0∧
I-L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧A[I]-A[
M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0)∨(I-K+
1=0∧I-L01+ 1≤0∧I-M02≠0∧K-M02≠0∧A[I]-A[M02]≤0)∨(I-K+
1=0∧I-M02≠0∧K-M02=0∧L01- 1≤0)∨(I-K+ 1=0∧I-M02≠0∧K-
M02≠0∧L01- 1≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧
I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02=0∧A[L01- 1]-A[I]≤0)∨(
I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧
A[L01- 1]-A[I]≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧
I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0)∨(
I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02≠0∧
A[I]-A[L01]≤0∧A[I]-A[M02]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-
L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02=0∧A[L01- 1]-
A[L01]≤0)∨(I-K+ 1=0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+
1≠0∧K-L01≠0∧K-M02≠0∧A[L01- 1]-A[L01]≤0∧A[I]-A[M02]≤0)∨(
I-M02≥0∧L01- 1≤0)∨(I-L01+ 1≤0∧I-M02≥0)∨(I-L01+ 1≤0∧
I-M02+ 2≤0)∨(I-M02+ 2≤0∧L01- 1≤0)∨(I-K+ 1≠0∧I-L01=0∧
I-M02≠0∧X-A[L01- 1]≥0∧K-L01+ 1≠0∧K-M02=0∧A[I]-A[I+
1]≥0)∨(I-K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[L01- 1]≥0∧K-L01+
1≠0∧K-M02≠0∧A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01=0∧I-
M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01=0∧I-M02≠0∧X-A[I]≥0∧K-L01+ 1=0∧K-M02≠0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02=0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≤0∧I-M02≠0∧K-M02≠0∧

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-M02≠0∧K-M02=0∧L01- 1≤0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-M02≠0∧K-M02≠0∧L01- 1≤0∧
A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01=0∧K-M02=0∧A[L01- 1]-A[I]≤0∧A[I]-A[I+ 1]≥0)∨(I-
K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01=0∧K-M02≠0∧A[
L01- 1]-A[I]≤0∧A[M02]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+
 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1=0∧K-M02=0∧A[I]-A[L01]≤0∧
A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧
K-L01+ 1=0∧K-M02≠0∧A[I]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(
I-K+ 1≠0∧I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧
K-M02=0∧A[L01- 1]-A[L01]≤0∧A[I]-A[I+ 1]≥0)∨(I-K+ 1≠0∧
I-L01+ 1≠0∧I-L01≠0∧I-M02≠0∧K-L01+ 1≠0∧K-L01≠0∧K-M02≠0∧
A[L01- 1]-A[L01]≤0∧A[M02]-A[I+ 1]≥0)∨(I-L01+ 1≠0∧I-
L01≠0∧I-M02≥0∧K-L01=0∧A[L01- 1]-A[I]≤0)∨(I-L01+ 1≠0∧
I-L01≠0∧I-M02≥0∧K-L01+ 1=0∧A[I]-A[L01]≤0)∨(I-L01+
 1≠0∧I-L01≠0∧I-M02≥0∧K-L01+ 1≠0∧K-L01≠0∧A[L01- 1]-A[
L01]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01=0∧A[L01-
 1]-A[I]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧K-L01+
 1=0∧A[I]-A[L01]≤0)∨(I-L01+ 1≠0∧I-L01≠0∧I-M02+ 2≤0∧
K-L01+ 1≠0∧K-L01≠0∧A[L01- 1]-A[L01]≤0))

ENTER THEOREM PROVER

NO HELP


        RETURN TO ALTERNATE BACKWARD PATH
        ASSERTION WAS: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(
I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(
I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01-
 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01-
 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))
        MOVE BACK TO NODE:  0004725C  J ← J+ 1
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         123--00027FC0: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02-
 1≥0)∨(I-L01≤0∧X-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(X-
A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0)∨(
J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))


        MOVE BACK TO NODE:  0004723C  K ← J
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         123--0001E5F0: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02-
 1≥0)∨(I-L01≤0∧X-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(X-
A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0)∨(
J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))

        MOVE BACK TO NODE:  00047214  X ← A[J]
        ASSIGNMENT STATEMENT-MAKE SUBSTITUTION
         132--00034160: ∀L01∀M02((I-M02- 1≥0∧L01-
 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01- 1≤0∧A[J]-
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧A[J]-A[M02]≤0)∨(
I-L01≤0∧J-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧A[J]-A[M02]≤0)∨(
J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))

MOVE BACK TO NODE: 000471F4 ((X-A[J]≤0))
IF STATEMENT, FROM FALSE BRANCH
141--0001CCF0: ∀L01∀M02((X-A[J]≤0)∨(I-M02-
1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(L01-
1≤0∧A[J]-A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧A[
J]-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧
A[J]-A[M02]≤0)∨(J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[
L01- 1]-A[L01]≤0))

MOVE BACK TO NODE: 000471D4 ((N-J≥0))
IF STATEMENT, FROM TRUE BRANCH
147--000290C0: ∀L01∀M02((N-J+ 1≤0)∨(X-A[J]≤0)∨(
I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(
L01- 1≤0∧A[J]-A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧
A[J]-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧
A[J]-A[M02]≤0)∨(J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[
L01- 1]-A[L01]≤0))

MOVE BACK TO NODE: 000471B4 ∀L01∀M02((I-M02-
1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧
I-M02- 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(
J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[
M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))
ASSERTION STATEMENT-FORM VER. COND.

THEOREM 2.2: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧
A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧J-
M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧
A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
A[L01- 1]-A[L01]≤0)) ⊃ ∀L01∀M02((N-J+ 1≤0)∨(X-A[J]≤0)∨(
I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(
L01- 1≤0∧A[J]-A[M02]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧
A[J]-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(A[L01- 1]-A[L01]≤0∧
A[J]-A[M02]≤0)∨(J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[
L01- 1]-A[L01]≤0))

ENTER THEOREM PROVER

NO HELP

RETURN TO ALTERNATE BACKWARD PATH
ASSERTION WAS: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(
I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(
I-L01≤0∧X-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(X-A[M02]≤0∧
L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0)∨(J-M02+
1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))
MOVE BACK TO NODE: 000471F4 ((X-A[J]≤0))

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
          IF STATEMENT, FROM TRUE BRANCH
           132--0001E260: ∀L01∀M02((X-A[J]- 1≥0)∨(I-M02-
 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧
I-M02- 1≥0)∨(I-L01≤0∧X-A[M02]≤0)∨(I-L01≤0∧J-M02+ 1≤0)∨(
X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0)∨(
J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+ 1≤0∧A[L01- 1]-A[L01]≤0))

          MOVE BACK TO NODE:  000471D4  ((N-J≥0))
          IF STATEMENT, FROM TRUE BRANCH
           138--00029130: ∀L01∀M02((N-J+ 1≤0)∨(X-A[J]-
 1≥0)∨(I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[
L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧X-A[M02]≤0)∨(
I-L01≤0∧J-M02+ 1≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
A[L01- 1]-A[L01]≤0)∨(J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+
 1≤0∧A[L01- 1]-A[L01]≤0))

          MOVE BACK TO NODE:  000471B4  ∀L01∀M02((I-M02-
 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧
I-M02- 1≥0)∨(I-L01≤0∧J-M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(
J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧A[L01- 1]-A[L01]≤0)∨(X-A[
M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧A[L01- 1]-A[L01]≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  2.3: ∀L01∀M02((I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧
     A[L01- 1]-A[L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧J-
     M02≤0)∨(I-L01≤0∧X-A[M02]≤0)∨(J-M02≤0∧L01- 1≤0)∨(J-M02≤0∧
     A[L01- 1]-A[L01]≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
     A[L01- 1]-A[L01]≤0)) ⊃ ∀L01∀M02((N-J+ 1≤0)∨(X-A[J]-
      1≥0)∨(I-M02- 1≥0∧L01- 1≤0)∨(I-M02- 1≥0∧A[L01- 1]-A[
     L01]≤0)∨(I-L01≤0∧I-M02- 1≥0)∨(I-L01≤0∧X-A[M02]≤0)∨(
     I-L01≤0∧J-M02+ 1≤0)∨(X-A[M02]≤0∧L01- 1≤0)∨(X-A[M02]≤0∧
     A[L01- 1]-A[L01]≤0)∨(J-M02+ 1≤0∧L01- 1≤0)∨(J-M02+
      1≤0∧A[L01- 1]-A[L01]≤0))

     ENTER THEOREM PROVER

     NO HELP


 END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 10.

The problem of finding feasible solutions to a linear system is presented to the theorem prover by using this null program. The linear solver concludes that the two points (3,3,0) and (1,4,0) are feasible by reporting "GIVE UP" for those assignments to the variables.

```
** PROGRAM VERIFIER **
*  SYNTAX ANALYSIS   *
   BEGIN  |     LINEAR SYSTEM

          |     HAS TWO FEASIBLE POINTS: (3,3,0) AND (1,4,0).

    ASSERT(  17*X1 +  3*X2 + 11*X3 ≤ 68  ∧
              5*X1 + 10*X2 +  5*X3 ≤ 45  ∧
             10*X1 +  5*X2 +    X3 ≤ 50  ∧
              4*X1 +  8*X2 +  3*X3 ≥ 36  ∧
             X1 ≥ 0  ∧  X2 ≥ 0  ∧  X3 ≥ 0  );

    ASSERT(FALSE);

    END;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    HEXADECIMAL PROGRAM MAP


000470F4: BEGIN-END DUMMY NODE
          TEXT:    0--00000000      nil
          PREDECESSORS: 00047134
          SUCCESSORS:   00047114

00047114: ASSERTION STATEMENT
          TEXT:    39--0001A8A0     ((X1≥0∧X1+ 2*X2+X3- 9≤0∧
      X2≥0∧X3≥0∧ 4*X1+ 8*X2+ 3*X3- 36≥0∧ 10*X1+ 5*X2+X3-
       50≤0∧ 17*X1+ 3*X2+ 11*X3- 68≤0))
          PREDECESSORS: 000470F4
          SUCCESSORS:   00047134

00047134: ASSERTION STATEMENT
          TEXT:    1--0001A8E0      f
          PREDECESSORS: 00047114
          SUCCESSORS:   000470F4


    END HEX PROGRAM MAP
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 GENERATE VERIFICATION CONDITIONS


ASSERTION  1:  00047134   f

          MOVE BACK TO NODE:  00047114  ((X1≥0∧X1+ 2*
    X2+X3- 9≤0∧X2≥0∧X3≥0∧ 4*X1+ 8*X2+ 3*X3- 36≥0∧ 10*X1+
     5*X2+X3- 50≤0∧ 17*X1+ 3*X2+ 11*X3- 68≤0))
          ASSERTION STATEMENT-FORM VER. COND.


THEOREM  1.1: ((X1≥0∧X1+ 2*X2+X3- 9≤0∧X2≥0∧X3≥0∧ 4*X1+
     8*X2+ 3*X3- 36≥0∧ 10*X1+ 5*X2+X3- 50≤0∧ 17*X1+ 3*X2+
     11*X3- 68≤0)) ⊃  f

    ENTER THEOREM PROVER

    SHOW TO BE ALWAYS FALSE:((X1≥0)∧(X1+ 2*X2+X3- 9≤0)∧(
    X2≥0)∧(X3≥0)∧( 4*X1+ 8*X2+ 3*X3- 36≥0)∧( 10*X1+ 5*X2+
    X3- 50≤0)∧( 17*X1+ 3*X2+ 11*X3- 68≤0))

      DEFINE SPECIAL FUNCTIONS
    SHOW TO BE ALWAYS FALSE:((X1≥0)∧(X1+ 2*X2+X3- 9≤0)∧(
    X2≥0)∧(X3≥0)∧( 4*X1+ 8*X2+ 3*X3- 36≥0)∧( 10*X1+ 5*X2+
    X3- 50≤0)∧( 17*X1+ 3*X2+ 11*X3- 68≤0))

      TRY LINEAR SOLVER


    LINEAR PART:X1≥0∧X1+ 2*X2+X3- 9≤0∧X2≥0∧X3≥0∧ 4*X1+
     8*X2+ 3*X3- 36≥0∧ 10*X1+ 5*X2+X3- 50≤0∧ 17*X1+ 3*X2+
     11*X3- 68≤0 ;  NON-LINEAR PART: nil
     ;  EQUALITIES TO REMEMBER: nil
    ELIMINATE VARIABLE: X1
    BETWEEN:X1≥0::X1+ 2*X2+X3- 9≤0
    FORMING...(( 2*X2+X3- 9≤0))
    BETWEEN:X1≥0:: 10*X1+ 5*X2+X3- 50≤0
    FORMING...(( 5*X2+X3- 50≤0))
    BETWEEN:X1≥0:: 17*X1+ 3*X2+ 11*X3- 68≤0
    FORMING...(( 3*X2+ 11*X3- 68≤0))
    BETWEEN: 4*X1+ 8*X2+ 3*X3- 36≥0::X1+ 2*X2+X3- 9≤0
    FORMING...((X3≤0))
    BETWEEN: 4*X1+ 8*X2+ 3*X3- 36≥0:: 10*X1+ 5*X2+X3-
     50≤0
    FORMING...(( 30*X2+ 13*X3- 80≥0))
    BETWEEN: 4*X1+ 8*X2+ 3*X3- 36≥0:: 17*X1+ 3*X2+ 11*X3-
     68≤0
    FORMING...(( 124*X2+ 7*X3- 340≥0))

    NEW SUBSYSTEM:X2≥0∧X3=0∧ 2*X2+X3- 9≤0∧ 3*X2+ 11*X3-
     68≤0∧ 5*X2+X3- 50≤0∧ 30*X2+ 13*X3- 80≥0∧ 124*X2+
     7*X3- 340≥0
    ELIMINATE VARIABLE: X2
    BETWEEN:X2≥0:: 2*X2+X3- 9≤0
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
FORMING...((X3- 9≤0))
BETWEEN:X2≥0:: 3*X2+ 11*X3- 68≤0
FORMING...((X3- 6≤0))
BETWEEN:X2≥0:: 5*X2+X3- 50≤0
FORMING...((X3- 50≤0))
BETWEEN: 30*X2+ 13*X3- 80≥0:: 2*X2+X3- 9≤0
FORMING...((X3- 27≤0))
BETWEEN: 30*X2+ 13*X3- 80≥0:: 3*X2+ 11*X3- 68≤0
FORMING...((X3- 6≤0))
BETWEEN: 30*X2+ 13*X3- 80≥0:: 5*X2+X3- 50≤0
FORMING...((X3+ 31≥0))
BETWEEN: 124*X2+ 7*X3- 340≥0:: 2*X2+X3- 9≤0
FORMING...((X3- 3≤0))
BETWEEN: 124*X2+ 7*X3- 340≥0:: 3*X2+ 11*X3- 68≤0
FORMING...((X3- 5≤0))
BETWEEN: 124*X2+ 7*X3- 340≥0:: 5*X2+X3- 50≤0
FORMING...((X3- 50≤0))

NEW SUBSYSTEM:X3+ 31≥0^X3- 3≤0
ELIMINATE VARIABLE: X3
BETWEEN:X3+ 31≥0::X3- 3≤0
FORMING... t
END ELIMINATION -- CONCLUDE: X3+ 31≥0^X3- 3≤0
END ELIMINATION -- CONCLUDE: X3=0^X3+ 31≥0^X3- 3≤0
END ELIMINATION -- CONCLUDE: X3=0^X3+ 31≥0^X3- 3≤0
SUBSTITUTE  0 FOR  1*X3


LINEAR PART:X1≥0^X1+ 2*X2- 9=0^X2≥0^ 2*X1+X2- 10≤0^
 17*X1+ 3*X2- 68≤0 ;  NON-LINEAR PART: nil
 ;  EQUALITIES TO REMEMBER: nil
ELIMINATE VARIABLE: X1
BETWEEN:X1≥0:: 2*X1+X2- 10≤0
FORMING...((X2- 10≤0))
BETWEEN:X1≥0:: 17*X1+ 3*X2- 68≤0
FORMING...((X2- 22≤0))

NEW SUBSYSTEM:X2≥0^X2- 10≤0
ELIMINATE VARIABLE: X2
BETWEEN:X2≥0::X2- 10≤0
FORMING... t
END ELIMINATION -- CONCLUDE: X2≥0^X2- 10≤0
END ELIMINATION -- CONCLUDE: X1+ 2*X2- 9=0^X2≥0^X2-
 10≤0
SUBSTITUTE - 2*X2+ 9 FOR  1*X1


LINEAR PART:X2- 3≥0^X2- 4≤0 ;  NON-LINEAR PART: nil
 ;  EQUALITIES TO REMEMBER: nil
ELIMINATE VARIABLE: X2
BETWEEN:X2- 3≥0::X2- 4≤0
FORMING... t
END ELIMINATION -- CONCLUDE: X2- 3≥0^X2- 4≤0
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
TRY CASES ON: 3≤+ X2≤ 4
SUBSTITUTE  3 FOR  1*X2
    GIVE UP
SUBSTITUTE  4 FOR  1*X2
    GIVE UP
```

END VERIFICATION CONDITION GENERATION

*** END PROGRAM VERIFICATION ***

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX III

Further Examples

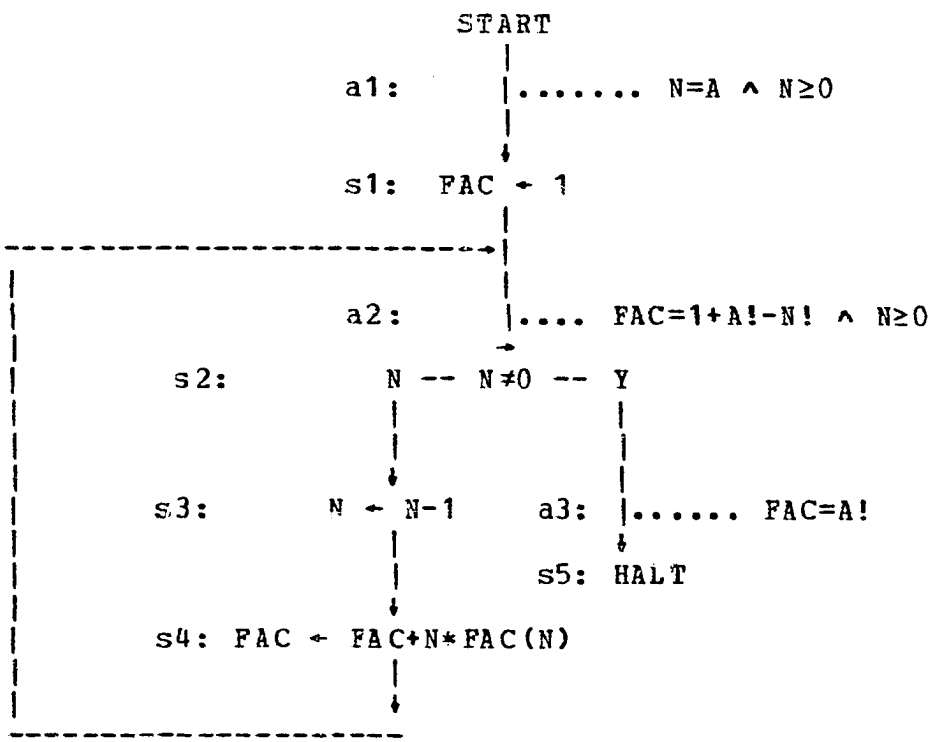
The first example is the manual verification of a program to compute factorial. It involves both iteration and recursion. The automatic program verifier does not currently handle recursion but could be upgraded to verify this example without too much effort. The next example suggests how global variables may be processed using a slightly modified version of the first example.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example A.

```
PROCEDURE FAC(N);
VALUE N; INTEGER N;
BEGIN
     ASSERT(N=A ∧ N≥0);

     FAC ← 1;
L:
     ASSERT(FAC=1+A!-N! ∧ N≥0);

     IF N≠0 THEN
     BEGIN
         N ← N - 1;
         FAC ← FAC + N*FAC(N);
         GO TO L;
     END;

     ASSERT(FAC=A!);

END;
```

A flowchart corresponding to this program is :



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The proof of correctness with respect to these assertions is given by generating and proving four verification conditions. Start with assertion a1 and "move over" statement s1 getting the verification condition:

1. $\vdash (N=A \wedge N\geq 0 \wedge FAC=1) \supset (FAC=1+-N! \wedge N\geq 0))$

which is obviously 'true'. Next take assertion a2 around the path {s2, s3, s4, s2} getting successively:

s2: $N\neq 0 \wedge FAC=1+A!-N! \wedge N\geq 0$

s3: $N+1\neq 0 \wedge FAC=1+A!-(N+1)! \wedge N+1\geq 0$

s4: $\exists f \exists g[(N+1\neq 0 \wedge f=1+A!-(N+1)! \wedge N+1\geq 0) \wedge$
$FAC=f+N*g \wedge (g=N!)]$

or more simply

$N+1\neq 0 \wedge FAC-N*N!=1+A!-(N+1)! \wedge N+1\geq 0.$

The recursive call on FAC produces the side verification condition:

2. $\vdash (N+1\neq 0 \wedge FAC=1+A!-(N+1)! \wedge N+1\geq 0) \supset N\geq 0,$

which is 'true'. Moving from s4 back to s2 produces the verification condition:

3. $\vdash (N+1\neq 0 \wedge FAC-N*N!=1+A!-(N+1)! \wedge N+1\geq 0) \supset$
$(FAC=1+A!-N! \wedge N\geq 0).$

This is seen to be 'true' by simplifying:

$FAC-N*N! = 1+A!-(N+1)!$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to

FAC = 1+A!-N!.

The final verification condition is derived by taking assertion a2 to the halt:

4. $\vdash (N=0 \wedge FAC=1+A!-N! \wedge N\geq 0) \supset FAC=A!$

which is also 'true'. Thus this program does compute factorial correctly.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example B.

The processing of global variables can also be demonstrated by modifying Example A. slightly. We introduce a "main program" which calls the procedure FAC, and add the global variable I which counts the number of times FAC is entered (exited):

```
BEGIN
    INTEGER I, N, K;
    PROCEDURE FAC(N);
        INTEGER N; VALUE N;
        BEGIN

            ASSERT(N=A ∧ N≥0 ∧ I=J);

            FAC ← 1;
    L:
            ASSERT(FAC=1+A!-N! ∧ N≥0 ∧ I=J+2↑A-2↑N);

            IF N≠0 THEN
            BEGIN
                N ← N - 1;
                FAC ← FAC + N*FAC(N);
                GO TO L;
            END;
            I ← I + 1;

            ASSERT(FAC=A! ∧ I=J+2↑A);

        END PROCEDURE FAC;


    ASSERT(I=0 ∧ N≥0);

    K ← FAC(N) + I;

    K ← I + FAC(N);

END;
```

Begin by verifying the procedure. This time the four verification conditions corresponding to those of Example

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A. are:

1. $\vdash (N=A \wedge N\geq 0 \wedge I=J \wedge FAC=1) \supset$
$(FAC=1*A!-N! \wedge N\geq 0 \wedge I=J+2\uparrow A-2\uparrow N)$

2. $\vdash (N+1\neq 0 \wedge FAC=1*A!-(N+1)! \wedge N+1\geq 0 \wedge$
$I=J+2*A-2\uparrow(N+1)) \supset N\geq 0$

3. $\vdash \exists f \exists g \exists i[(N+1\neq 0 \wedge f=1*A!-(N+1)! \wedge N+1\geq 0 \wedge$
$i=J+2\uparrow A-2\uparrow(N+1)) \wedge (FAC=f+N*g) \wedge$
$(g=N! \wedge I=i+2\uparrow N)] \supset$
$[FAC=1*A!-N! \wedge N\geq 0 \wedge I=J+2\uparrow A-2\uparrow N]$

or more simply

$\vdash [N+1\neq 0 \wedge FAC-N*N!=1*A!-(N+1)! \wedge N+1\geq 0 \wedge$
$I-2\uparrow N=J+2\uparrow A-2\uparrow(N+1)] \supset$
$[FAC=1*A!-N! \wedge N\geq 0 \wedge I=J+2\uparrow A-2\uparrow N]$

4. $\vdash (N=0 \wedge FAC=1*A!-N! \wedge N\geq 0 \wedge I-1=J+2\uparrow A-2\uparrow N) \supset$
$(FAC=A! \wedge I=J+2\uparrow A).$

These can all be shown to be 'true'. Note that this verification of the procedure includes the verification of its effect on the global variable I. Now consider the main program. After the statement:

K ← FAC(N) + I;

we have:

$\exists i \exists f[(i=0 \wedge N\geq 0) \wedge K=f+I \wedge (f=N! \wedge I=i+2\uparrow N)]$

or more simply:

$(N\geq 0 \wedge K=N!+2\uparrow N \wedge I=2\uparrow N).$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the second case after statement:

```
K ← I + FAC(N);
```

we then have:

$$\exists i \exists f [(N \geq 0 \wedge i = 2 \uparrow N) \wedge K = i + f \wedge (f = N! \wedge I = i + 2 \uparrow N)]$$

or more simply:

$$N \geq 0 \wedge K = 2 \uparrow N + N! \wedge I = 2 \uparrow (N+1).$$

The side effects are accounted for by using i and I appropriately. (Note the difference of (K=f+I) and (K=i+f).)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.