Stanford Artificial Intelligence Laboratory
Memo AIM-255

December 1974

Computer Science Department
Report No. STAN-CS-74-473

Automatic Program Verification II:

VERIFYING PROGRAMS BY ALGEBRAIC AND LOGICAL REDUCTION

by

Norihisa Suzuki

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-255

DECEMBER 1974

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-74-473

AUTOMATIC PROGRAM VERIFICATION II:

VERIFYING PROGRAMS BY ALGEBRAIC AND LOGICAL REDUCTION

by

Norihisa Suzuki

ABSTRACT:
Methods for verifying programs written in a higher level programming language are devised and implemented. The system can verify programs written in a subset of PASCAL, which may have data structures and control structures such as WHILE, REPEAT, FOR, PROCEDURE, FUNCTION and COROUTINE. The process of creation of verification conditions is an extension of the work done by Igarashi, London and Luckham which is based on the deductive theory by Hoare. Verification conditions are proved using specialized simplification and proof techniques, which consist of an arithmetic simplifier, equality replacement rules, fast algorithm for simplifying formulas using propositional truth value evaluation, and a depth first proof search process. The basis of deduction mechanism used in this prover is Gentzen-type formal system. Several sorting programs including Floyd's TREESORT3 and Hoare's FIND are verified. It is shown that the resulting array is not only well-ordered but also a permutation of the input array.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence under contract DAHC 15-73-C-0435.

1. Introduction

Verifying that programs work faultlessly is a necessity. We can test whether they work or not in several cases. But unless we prove the correctness of programs, it is impossible to claim that they endure long lasting usage. Since proving by hand is cumbersome and not always free of errors, mechanization of verification is strongly desired.

Some attempts have been made to verify programs mechanically [1],[2],[10],[11], but there are several problems which must be solved in order to make automatic verification of programs practical.

First, we have to find a way to express assertions more easily. Most of the previous verifiers require assertions to be written in first order predicate sentences with a fixed number of predefined predicate symbols and function symbols. But this is in many cases inconvenient and infeasible. For example, if we have to deal with the correctness of programs with complex data structures, we need to express properties in higher order sentences. Thus, many complex programs have not been verified because the assertions about programs have not been properly stated.

Second, we have to find a better way to prove verification conditions automatically. Proving verification conditions using a general automatic theorem prover is in most of the cases unsatisfactory. If we are verifying programs in specific domains, we can use special properties of functions and predicates to construct fast special purpose provers. King[10] and Deutsch[2] have succeeded by using a built-in simplifier for integer arithmetic, but these programs still cannot cope with other domains.

In most verification systems the user must specify not only input and output conditions but also loop invariants. Although it is an undecidable problem to generate loop invariants, the system should aid the programmer in constructing loop invariants. Also, programs with complex data structures and complex control structures must be verified, including parallel programs.

In this paper we describe a fast simplification and theorem proving facility that is a new component of the Stanford PASCAL Verification System described by Igarashi, London and

Luckham in [9]. This system permits the programmer to formulate the semantics of his data structures, procedures, and functions in simple, natural statements. These statements are used by the system as simplification and special theorem-proving rules during verification. So programs computing over any domain can be dealt with easily.

As an example, automatic verification of a sorting program is studied in detail. It is shown that not only is the resulting array ordered but also it is a permutation of the input array. The verification of Floyd's TREESORT program and Hoare's FIND program are listed, both of which are verified within a reasonable amount of computation time. Because these programs are complex, and use data structures--in this case an array data structure, whose semantics has not been studied well--they have been considered as one of the big challenges for automatic verification. Thus our method of verification is very promising for practical use.

II. Expressing Assertions by Structured Definitions.

Here, we make a few comments about how the user of the system might construct documentation in a way that aids the verification of his program. The main idea is to use defined concepts that are close to the natural concepts employed in creating the program.

As is discussed in the previous section, it is impossible to state all properties of programs in first order sentences with fixed number of predefined function symbols and predicate symbols. As an example let us examine the process of verifying sorting programs. Suppose a program S accepts an array A and sorts it and output it as an array B. Then, the correctness of S is expressed in terms of properties that elements of B are ordered in ascending(or descending)order and B consists of all elements of A and of nothing else. The first property can be stated as

$\forall I.(1\le I\le N-1\supset B[I]\le B[I+1])$.

But one way to describe the second property is to state that there is a one-to-one mapping from elements of A to elements of B. That is the sentence

$\exists F.(\forall I.(1\le I\le N\supset 1\le F(I)\le N)\wedge\forall I,J.(1\le I<J\le N\supset F(I)\ne F(J))\wedge\forall I.(1\le I\le N\supset A[I]=B[F(I)]))$

expresses the second property.

But previous verifications of sorting programs, either manual or automatic, have dealt with only the first property. The detailed study of FIND by Hoare[7] briefly explains that to prove the correctness it is necessary to show that the second property holds, but does not formally verify it. He thought that the assertions were not obvious and the proof would be tedious. It is certainly disadvantageous to introduce second order sentences because they require complicated proof procedures. But since it is essential for the automatic verification to prove the second properties formally, we have to invent a way to verify them.

The way to avoid using second order sentences is to extend the language by introducing new symbols. There is also another nice thing about introducing new symbols. To express that array B is a permutation of array A, we have to employ a rather complex sentence. It might be as difficult to understand what it means as to understand what the program does. Also it is very easy to introduce errors. But we can avoid complexitites by writing

Permutation(B,A).

In general there are two methods to introduce new symbols. The first method is to assume the new symbol as a shorthand representation of a sentence represented by already defined symbols. The second method is to define symbols by axioms stating the properties of these symbols. For example, after defining axioms of propositional calculus consisting of symbols "$\supset$" and "$\neg$", we can introduce "$\wedge$" symbol as a shorthand notation for $\neg(A\supset\neg B)$. But also we can introduce it by axioms,

$A\wedge B\supset A, A\wedge B\supset B$ and $A\supset(B\supset A\wedge B)$.

Assertions describing a program can be structured top-down by using new symbols. Their meanings are refined succesively until everything is well-defined. An analogous concept can be found in programming. We can enrich the language and clarify the meaning by introducing new symbols (operations). These new operations are defined either by macros or by procedures. Macros define new operations by using already defined concepts. So they do not give more computation power but clarify programs. Whereas, procedures can define new operations recursively, so that they give new power.

Following this analogy to programming, we can call the way we write predicate sentences with newly defined symbols a structured way of expressing assertions. A detailed study of how to introduce new symbols is in section V, and also is found in the work by von Henke and Luckham[5].

In the case of "Permutation(B,A)", we could define it as the shorthand representation of the previous sentence. But instead we shall define it by a set of properties (specifications) including the following axiom,

$\forall A,I,J.$Permutation(Exchange(A,I,J),A),

where Exchange(A,I,J) is a function mapping an array A into an array resulting from exchanging I-th element and J-th element of A. In addition, Permutation is an equivalence relation, so we must include axioms for symmetric, reflexive and transitive properties.

We have replaced a second order statement by a relation which has arrays as individuals. Now, arrays are a second sort of individuals.

Thus, we need to have a special semantic definition for array assignment, since **arrays** as well as array elements occur in assertions.

NOTATION <A,I,E>: An array obtained from A by placing E in the i-th position.

ARRAY ASSIGNMENT AXIOM

P(<A,I,E>) {A[I]←E} P(A).

III. Documentation Statements and Their Use,

Introduction of new symbols is essential to verification for ease of both representation and unclerstandrng of assertions. We allow users to introduce new symbols by documentation in the form of three simple kinds of statements. They are used by the prover as (i)rewriting rules to expand new symbols, {ii) reduction strategies which state that some expressions are reduced to others under specified conditions, and (iii)goal-subgoal strategies which state that certain well-formed formulas are true if certain others are true. We found that they are convenient and powerful.

From the method of construction of verification conditions [5],[6],[7],[9], all the verification conditions are of the form

A1∧. , .∧AN → C1∧. . .∧CM.

Since this form of representation is more natural for understanding than disjunctive normal form, we retain this form throughout the proof. The proof procedure is based on Centzen's formal system. Thus, the validity of each CI is proved with the assumption A1∧. . .∧AN.

We first explain a special pattern matching language, in which all the documentation statements are written.

1. Pattern Matching.

A pattern is a string of symbols which match a term or a well-formed formula. Patterns consist of pattern constants and pattern variables. A pattern constant is an identifier and a pattern variable is an identifier preceded by a symbol "@". So @X stands for a pattern variable. Under the pattern matching mechanism, a pattern constant matches only that symbol and an unbound pattern variable matches any term and is bound to that term thereupon. A bound pattern variable matches only the corresponding term.

Higher order pattern matching is undecidable in general. So, in this algorithm a term with unbound pattern variables is not matched to a term with unbound pattern variables. But still this restricted matching algorithm is ambiguous. For example, if a pattern @P(@X) is matched to

Q(F(A)), both @P=Q, @X=F(A) and @P=Q(F()), @X=A are permissible bindings. This ambiguity is costly in computation and should be avoided if possible. Thus, in this system we employ an incomplete but decidable procedure. The matching is done from the outer symbols, and from left to right among parameters. So @P(@X) matches to Q(F(A)) and yields @P=Q and @X=F(A).

The limited facility has not caused much inconvenience. Since higher order sentences can be translated to first order sentences by introducing new symbols, all properties can be expressed in first order sentences. We are going to see that the pattern matching does not cause much inconvenience in the case' of data structures either. Suppose A and B are both arrays. If we match @X[@Y] to A[B[I I 3 , we get @X=A and @Y=B[I I by our matching algorithm. But we do not want the bindings of @X=A[B[3 1 and @Y=I, since A[B[]] is not meaningful.

2. Rewriting Rules.

We can use TEMPLATE statements to introduce new symbols as shorthand representations of already defined expressions.

TEMPLATE <pattern> ↔<expression>.

Then, a rewriting rule is created from this statement. The system replaces every occurrence of <pattern> by <expression> according to the rule.

If we want to introduce

Ordered (A, I, J)

as a shorthand representation of

VX. (I≤X<J ⊃ A[X]≤A[X+1]),

then we can write

TEMPLATE Ordered(@A,@I,@J) ↔ ∀X.(I≤X<J ⊃ A[X]≤A[X+1]).

3. Reduction Strategies.

Also, we can introduce new symbols by a set of axioms. These axioms can be stated by AXIOM statements and COAL statements to produce reduction strategies and goal-subgoal strategies respectively.

We can specify reduction strategies to simplify terms or well-formed formulas. These strategies are of two kinds, one is an unconditional reduction and the other is a conditional reduction.

Unconditional reduction strategies can be fed into the system by statements of the form

AXIOM <pattern> $\leftrightarrow$ <expression>,

The effect of this strategy is to reduce any expression which matches the <pattern> to <expression>. The <expression> may have identifiers which appear in the pattern as pattern variables. They are bound to some forms by matching. For example, one can represent one of the axioms of list data structures,

$\forall X,Y.CAR(CONS(X,Y))=X$,

as a simplification rule,

AXIOM CAR(CONS(@X,@Y)) $\leftrightarrow$ X.

Then P (CAR (CONS (A,B)) is reduced to P(A) since @X is bound to A. Only universally quantified equality or equivalence relations can be represented by this method.

Conditional reduction strategies are specified to the system by statements of the form

AXIOM IF <pattern 1> THEN <pattern> $\leftrightarrow$ <expression>,

The effect is to reduce expressions which match <pattern> to <expression>, if <pat tern 1> is provable by the system. Some pattern variables of the <pat tern 1> become bound when <pat tern> is matched. If the <pat tern 1> does not include unbound pattern variables, the validity of the sentence

$A1\wedge...\wedge AN \rightarrow$ <pattern 1>,

is checked by recursively activating the prover. If the <pat tern 1> includes unbound pattern variables, it is tested whether it matches the antecedent part of the verification condition or not. If it matches then we consider <pat tern 1> to be provable and otherwise not provable.

For example,

. vx, Y $(X\leq Y\wedge Y\leq X\supset X=Y)$

is a valid statement. We want to incorporate this fact into the system by conditional reduction, and reduce $Y\leq X$ to X=Y if $X\leq Y$ holds. The statement we should write is

AXIOM IF $X\leq Y$ THEN $@Y\leq @X \leftrightarrow X=Y$.

Then if we are to reduce the statement

$A\leq B\wedge B\leq A\wedge P(A)\supset P(B)$, the pattern matches to $A \le B$ to get bindings @Y=A and @X=B. Since there is no unbound pattern variable, the system sets up a subgoal $B \le A$, and tries to prove

$B \le A \wedge P(A) \supset B \le A$,

which is valid. So the statement is reduced to

$B = A \wedge B \le A \wedge P(A) \supset P(B)$,

which will be proved to be valid by equality substitution. As the previous example shows, universally quantified theorems can be represented by this statement. But also some existentially quantified theorems can be represented,

For example

$\forall X(\exists Y.P(X,Y) \supset F(X) = G(X))$

can be represented by a statement

AXIOM IF P(X,@Y) THEN F(@X)↔G(X).

4. Coal-Subgoal Strategies.

Reduction strategies turn out to be important components of proof. It is a frequently used proof step. However, we rely heavily on additional goal-subgoal strategies to complete many verification proofs. Verification conditions are of the form

$A1 \wedge ... \wedge AN \rightarrow C1 \wedge ... \wedge CM$.

The problem is to prove each CI . If we can prove $BI \supset CI$ and $A1 \wedge ... \wedge AN \rightarrow BI$, we can deduce $A1 \wedge ... \wedge AN \rightarrow CI$ by modus ponens. Thus, if we have an axiom $BI \supset CI$ the subproblem we have to solve is

$A1 \wedge ... \wedge AN \rightarrow BI$.

This fact is the motivation for employing goal-subgoal strategies.

Statements to specify strategies are of the form

GOAL <pattern> SUB <pattern 1>,...,<pattern n>.

The strategy constructed from this statement works as follows. If <pattern> matches to the consequent CI, each <pattern j> is tested successively until one of them is provable. If <pattern j> has unbound pattern variables it is tested to determine whether it matches one of the conjuncts of the antecedent. If <pattern j> has no unbound pattern variables, a new subproblem

$A1\wedge\ldots\wedge AN \rightarrow$ <pattern j>

is tested by recursively activating the prover.

For example, the transitivity of "≤" is defined by an axiom

$\forall X, Y. (\exists Z. (X\leq Z\wedge Z\leq Y) \supset X\leq Y).$

This is represented by a goal-subgoal strategy,

GOAL @X≤@Y SUB X≤@Z∧@Z≤Y.

In order to prove a sentence

$A\leq B\wedge B\leq C\wedge C\leq D\rightarrow A\leq D$

using this goal, first @X≤@Y is matched to A≤D to obtain @X=A and @Y=D. Then, the antecedent is searched whether A≤@Z matches one of the conjuncts. In this case the search is successful and yields @Z=B. Thus, the remaining subgoal is @Z≤D, which is now B≤D. So the new subproblem

$A\leq B\wedge B\leq C\wedge C\leq D\rightarrow B\leq D$

is set up. This can be proved by using the same goal one more time. These strategies can also represent universally or existentially quantified theorems.

Everything which goal-subgoal strategies can express can be expressed by conditional reduction strategies, since we can express the statement

GOAL A SUB B,

by the statement

AXIOM IF B THEN A↔TRUE.

However, the system uses these statements in different ways. Conditional reduction strategies are used to reduce expressions in both the consequent and the antecedent of verification conditions. For example, suppose we have a conditional reduction strategy specified by

IF Al THEN A2 ↔ C.

then

$A1\wedge A2\rightarrow B$

is reduced to

$A1\wedge C \rightarrow B,$

and

$Al \rightarrow AZ$

is reduced to

$A1 \rightarrow C.$

Goal-subgoal strategies are used only to make reduction in the consequent.

The reason why we have goal-subgoal strategies is that because they are more efficient than conditional reduction strategies. Most of the time we are interested in proving the validity of a statement of the form $A \rightarrow B$. Thus, we are interested in how B can be proved from A, Also the antecedent A is usually more complex than the consequent B because the antecedent contains all the information about data structures and control structures. So the goal-subgoal strategy gains efficiency by limiting the reduction to the consequent part.

IV. Implementation.

This verification system is built upon the PASCAL verification condition generator VCGEN[9]. First, files of the user's Axioms and Goal statements are input to the system, and the corresponding reduction rules and goal-subgoaling strategies are constructed. This yields a special reduction and proof system for the data structures and functions described by these statements. The system is extensible, since strategies can be added to handle larger domain of programs. Next, a file containing the program with assertions is processed by VCCEN to produce verification conditions. These are passed to the proving system. The proving system is divided into several functions. They are (i)the arithmetic simplifier, (ii)the equality substitution algorithm, (iii)the truth value substitution algorithm, (iv)the unconditional simplifier, (v)the conditional simplifier, (vi)the goal-subgoaler, and (vii)the logic symbol elimination algorithm.

Gentzen-type inference rule notations are used to express the effects of functions.

NOTATION :
$$\frac{A \qquad B}{C}$$

,where C is the goal and A and B are subgoals both of which must be proved in order to prove C.

(i) The arithmetic simplifier transforms arithmetic expressions into standard representations, and simplifies them. The standard representation is a sum of products of simple factors. A simple factor is an arithmetic expression which is neither a sum nor a product. Then each product consists of a coefficient(if not equal to 1) followed by simple factors which are ordered by system-defined orderings, And the sum consists of the ordered products followed by a constant(if not equal to 0).

(ii) The equality substitution algorithm handles verification conditions of the form

$A \wedge (\alpha=\beta) \wedge B \rightarrow C.$

CASE I. Suppose one of a or β is a variable. Without loss of generality we can suppose a to be a variable. If β is a constant, a variable, or an expression with α not appearing free, then all the occurrences of a in A, B and C are replaced by β.

CASE 2. Suppose one of a or β is a variable. Without loss of generality we suppose a to be a variable. If β is an expression containing a, then all the occurrences of β in A, B and C are replaced by a.

CASE 3. If α and β do not satisfy cases 1 or 2 then all the occurrences of a are replaced by β.

(iii) The truth value substitution algorithm evaluates logical sentences. The grand rule of the truth value substitution is

$$\frac{\text{Tsubst}(A,\alpha)\wedge\alpha\wedge\text{Tsubst}(B,\alpha)\rightarrow \text{Tsubst}(C,\alpha)}{A\wedge\alpha\wedge B \rightarrow C,}$$

where both A and B may be null expressions and a is not a conjunction. Tsubst (A, α) is defined by the following see of functions, which give the value of A assuming a is true.

```
Tsubst$(A,\alpha)$=if a is of the form $\neg\beta$ then Fsubst$(A,\beta)$ else
            if $\alpha$ is of the form $\beta\wedge\epsilon$ then
                 Tsubst(Tsubst$(A,\beta)$,$\epsilon$) else
                 replace al l occurences of a in A by "True".
 Fsubst$(A,\beta)$=if $\beta$ is of the form $\neg\alpha$ then Tsubst$(A,\alpha)$ else
            if $\beta$ is of the form $ax$ then
                 Fsubst(Tsubst$(A,\alpha)$,$\epsilon$) else
            if $\beta$ is of the form $\alpha\vee\epsilon$ then
                 Fsubst (Fsubst$(A,\alpha)$,$\epsilon$) else
            replace al l occurences of $\beta$ in A by "False",
```

(iv) The **unconditional** simplifier applies all unconditional reduction **strategies**.

The algorithm works from inside out. Thus if we want to simplify

```
R(P1,...,PN),
```

first all P1, . . . , PN are simplified to Q1,...,QN respectively. Then R(Q1,...,QN) is simplified.

(v) The conditional simplifier applies all conditional reduction strategies. The treatment is different according to the position of the expression--in the antecedent or consequent of the verification condition. Suppose a conditional reduction strategy is given to the system by a statement

```
AXIOM IF <pattern 1> THEN <pattern> ↔ <expression>,
```

and the verification condition to be proved is

```
A1∧...∧AM → C1∧...∧CN.
```

If <pattern> matches a subexpression of CI, then

```
A1∧...∧AM → <pattern 1>
```

becomes the subproblem to be solved.

Nest, suppose <pattern> matches a subexpression of the antecedent say AI. Then

```
A1∧...∧AI-1∧AI+1∧...∧AM → <pattern 1>
```

becomes the subproblem to be solved. If it is valid then the replacement takes place as before.

The validity is checked by recursively activating the prover. So this is a depth first search, and it might go into a wrong direction infinitely. So the system allows the user to specify the search depth. If the search reaches this limit, it is backed up until the last decision point.

(vi) The **goal-subgoaler** incorporates all goal-subgoal strategies. Suppose a goal-subgoal strategy is given to the system by a statement

```
GOAL <pattern> SUB <pattern 1>,...,<pattern N>,
```

and the verification condition to be proved is

```
A1∧...∧AN → C1∧...∧CM.
```

If CI matches to <pattern>, then

```
A1∧...∧AN → <pattern 1>,....., A1∧...∧AN → <pattern N>
```

are set up as a disjunction of subproblems successively, until one of them is proved to be “True”. If the proof is successful the problem is reduced to

$A1 \wedge \ldots \wedge AN \rightarrow C1 \wedge \ldots \wedge CI\text{-}1 \wedge CI\text{+}1 \wedge \ldots \wedge CM.$

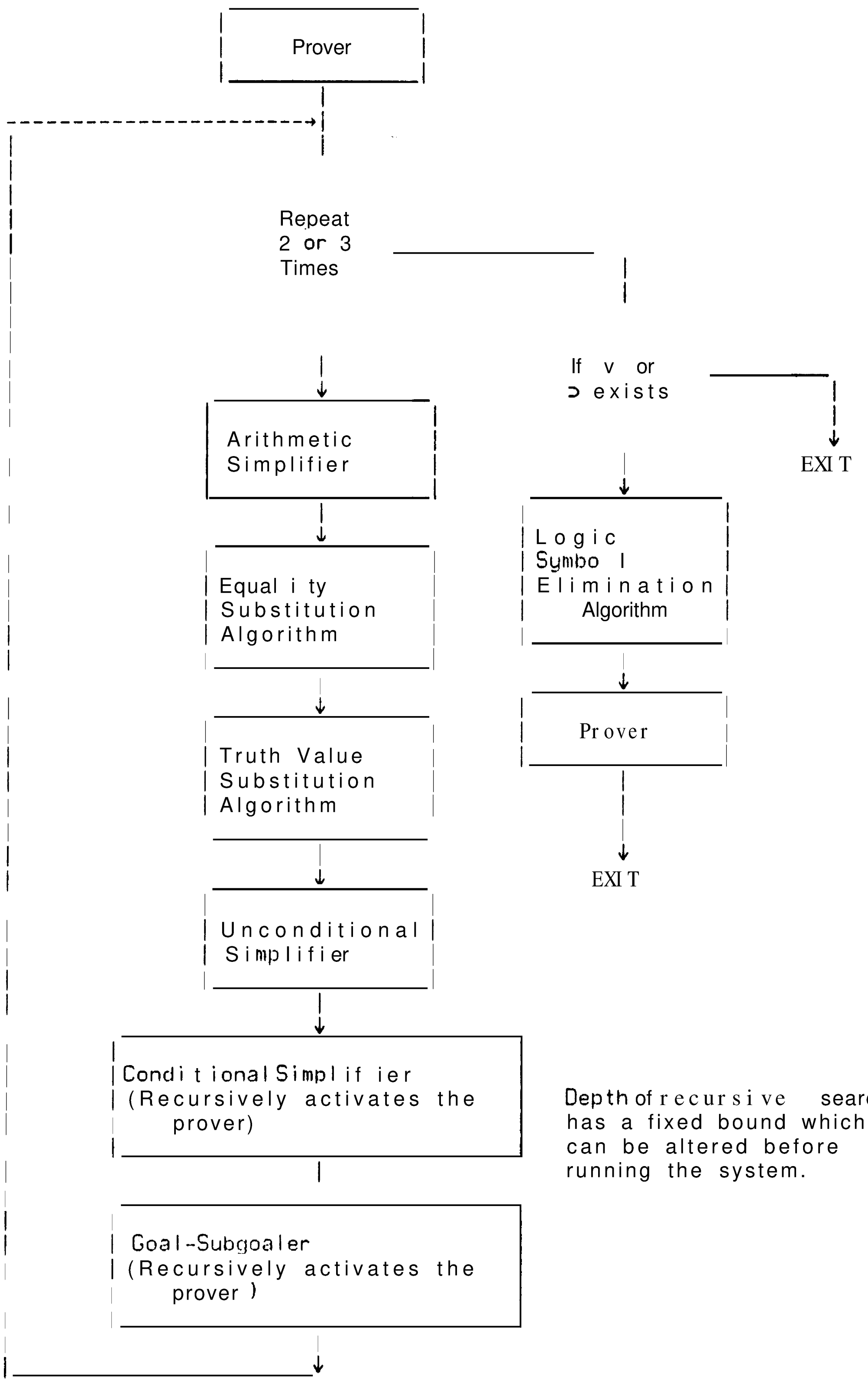
(vii) The logic symbol elimination algorithm works on elimination of logic symbols "$\vee$" and "$\supset$" from the antecedent of the statement. Their functions are explained by inference rules as shown below.

(v-elimination)
$$\frac{A \wedge \alpha \wedge B \rightarrow C \qquad A \wedge \beta \wedge B \rightarrow C}{A \wedge (\alpha \vee \beta) \wedge B \rightarrow C}$$

($\supset$-elimination)
$$\frac{A \wedge \neg \alpha \wedge B \rightarrow C \qquad A \wedge \beta \wedge B \rightarrow C}{A \wedge (\alpha \supset \beta) \wedge B \rightarrow C}$$

These seven functions are applied serially. But the simplification may be applicable after reduction by goal-subgoaling. So these functions are iterated several times. The user can specify the number of iterations.

The overall structure of the prover is as follows.



Prover
Repeat 2 or 3 Times
If v or ⊃ exists
EXIT
Arithmetic Simplifier
Logic Symbol Elimination Algorithm
Equality Substitution Algorithm
Prover
Truth Value Substitution Algorithm
EXIT
Unconditional Simplifier
Conditional Simplifier (Recursively activates the prover)
Goal-Subgoaler (Recursively activates the prover)

Depth of recursive search has a fixed bound which can be altered before running the system.

V. Application to Sorting Programs

As the first example, the verification of a simple sorting program which successively finds the largest element among the unordered part of the array and puts it at the end of the ordered part is considered. This program is the one considered by King[10]. The program with input and output conditions and an assertions about loop invariants is shown below. This is the actual input form for the system.

```
PASCAL

TYPE SARRAY=ARRAY [1:L] OF INTEGER;

PROCEDURE EXCHANGESORT (VAR A: SARRAY;L: INTEGER) ;
INITIAL A=A0;
ENTRY 1≤L;
EXIT Issortedarrayof(A,A0);

VAR X:REAL;VAR K,I, J: INTEGER:

          BEGIN
          I-L;
          INVARIANT Permutation(A,A0)∧Ordered(A,I+1,L)∧Partitioned(A,I)∧(I≥1)
          WHILE I>1 00
                    BEGIN
                    J←2;X←A[1];K←1;
                    INVARIANT Biggest(A,J-1,K)∧(1≤K)∧(K≤J-1)∧(J-1≤I)∧(X=A[K])
                    WHILE J≤I DO
                              BEGIN
                              IF X≥A[J] THEN GOT0 3;
                              X←A[J];
                              K←J;
                         3: J←J+1
                              END:
                    A[K]←A[I];
                    A[I]←X;
                    ICI-1
                    END;
          END;.;
```

We are going to explain the intended interpretation of symbols and the set of axioms defining them. When we express axioms, we have to be careful not to introduce an inconsistent set. Since a consistent set of axioms has a model, we can avoid introducing an inconsistent set by defining an interpretation and justifying axioms by showing validity relative to that interpretation.

Inputs to this program are an array A and an integer parameter L defining the upper bound of the array. Since we have an array with at least one element, the input condition is

$L \geq 1$.

The output condition is

Issortedarrayof(A,A0),

where A0 is the initial value of A at the entrance to the procedure and Issortedarrayof(A, A0) means that A0 is sorted to become A.

1. Issortedarrayof(A,B).

In order A to be a sorted array of B, it must be ordered in ascending order and it must consist of all the elements of B and nothing else. We describe the two facts by introducing additional predicates. The axiom is,

$Ordered(A,1,L) \wedge Permutation(A,B) \supset Issortedarrayof(A,B)$.

2. Ordered(A, J,L).

The interpretation of Ordered(A,J,L) is that the subarray A[J:L] is ascendingly ordered. Thus,

$Ordered(A,J,L) \ast\!\!\rightarrow \forall X.\ (J \leq X \leq L-1 \supset A[X] \leq A[X+1])$,

where $\ast\!\!\rightarrow$ means that the left-hand side is the shorthand notation of the right-hand side.

Three axioms are necessary to specify the predicate. The first one specifies the boundary case when J is equal to L+l. Then there is no element in the subarray and an empty array is ordered. So

Ordered(A,L+1,L)

is true.

The next axiom is an induction axiom which state that if the property holds for a smaller subarray it holds for a larger subarray under certain conditions. It is

$Ordered(A,J,L) \wedge Partitioned(A,J-1) \supset Ordered(A,J-1,L)$.

This axiom enables the property to be extended to the whole array. The meaning of Partitioned(A, J-l) is that the array A is partitioned between J-l and J such that all the elements in the upper half are larger than or equal to all the elements in the lower half.

The last axiom states that changing elements outside of the concerned subarray will not change the property. The operation on the array in this program is Exchange (A, I, J) , which is an array obtained by exchanging I-th and J-th element of A, thus

$\text{Ordered}(A, J, L) \wedge (I \le J) \wedge (K \le J) \wedge \text{Partitioned}(A, J)$

$\supset \text{Ordered}(\text{Exchange}(A, I, K), J, L).$

3. Partitioned(A,J).

The meaning of this predicate has been stated before as

$\text{Partitioned}(A, J) \leftrightarrow \forall X, Y.(1 \le X \le J < Y \le L \supset A[X] \le A[Y]).$

There are also three axioms to specify this predicate with the same nature as those of Ordered(A, J,L).

When J is equal to L, there is no element in the upper half of the array, so the property holds. Thus, the boundary property is

$\text{Partitioned}(A, L).$

The axiom about induction is

$\text{Partitioned}(A, J) \wedge \text{Biggest}(A, J, J) \supset \text{Partitioned}(A, J-1).$

Since Biggest (A, J, J) means that A[J] is the biggest element among elements of the subarray A[1: J], there is a separation between J- l and J.

Also if we exchange elements of the lower half of the array the property remains valid. So,

$\text{Partitioned}(A, J) \wedge (I \le J) \wedge (K \le J) \supset \text{Partitioned}(\text{Exchange}(A, I, K), J).$

4. Biggest(A,I, J).

The meaning of this predicate is that, A[J] is the biggest element among the elements of the subarray A[1:I].

The axiom of the boundary case states when I is equal to 1. Then, there is one element in the subarray which is the biggest element. Thus,

$\text{Biggest}(A, 1, 1).$

The axioms about the induction are

$\text{Biggest}(A, I, J) \wedge (A[J] \ge A[I+1]) \supset \text{Biggest}(A, I+1, J)$

and

$\text{Biggest}(A,I,J) \wedge (A[I+1] \geq A[J]) \supset \text{Biggest}(A,I+1,I+1)$.

The next axiom states that if we move the biggest element by Exchange, then the place of the biggest clement changes. The objective of the program is to move the biggest element of **subarray** A[1:I] to A[I]. Thus, the axiom

$\text{Biggest}(A,I,J) \supset \text{Biggest}(\text{Exchange}(A,J,I),I,I)$,

is sufficient.

5. Permutation(A,B).

The meaning is that the array A is a permutation of the array B.

If we exchange elements of an array, this is a permutation of the array.

Thus,

$\text{Permutation}(\text{Exchange}(A,I,J),A)$

is an axiom. Also $\text{Permutation}(A,B)$ is an equivalence relation, so

$\text{Permutation}(A,A)$, and

$\text{Permutation}(A,B) \supset \text{Permutation}(B,A)$, and

$\text{Permutation}(A,B) \wedge \text{Permutation}(B,C) \supset \text{Permutation}(A,C)$,

are axioms. Since any permutation can be obtained by repeated operations of Exchange, these are sufficient axioms to prove the property.

6. Exchange(A,I,J).

The axiom sufficient to represent that any N-place cycle is decomposable into N Exchanges is

$Y=A[J] \supset \langle\langle A,I,Y\rangle,J,X\rangle = \text{Exchange}(\langle A,I,X\rangle,I,J)$.

The following listing is the goalfile which is supplied to the system along with the program. This shows how simplification and goal-subgoaling rules are selected to represent axioms.

```
GOALFILE
          GOAL Issortedarrayof(@A,@B) SUB Permutation(A,B)∧Ordered(A,1,L);

          AXIOM Ordered(A,L+1,L)↔TRUE;
          GOAL Ordered(@A,@P1,L) SUB Ordered(A,P1+1,L)∧Partitioned(A,P1);
          GOAL Ordered(Exchange(@A,@P1,@P2),@P3,L)
                    SUB (P1≤P3)∧(P2≤P3)∧Ordered(A,P3,L)∧Partitioned(A,P3);

          AXIOM Partitioned(A,L)↔TRUE;
          GOAL Partitioned(@A,@P1) SUB Biggest(A,P1+1,P1+1)∧Partitioned(A,P1+1);
          GOAL Partitioned(Exchange(@A,@P1,@P2),@P3)
                    SUB (P1≤P3)∧(P2≤P3)∧Partitioned(A,P3);

          AXIOM Biggest(A,1,1)↔TRUE;
          GOAL Biggest(Exchange(@A,@P1,@P2),@P2,@P2) SUB Biggest(A,P2,P1);
          GOAL Biggest(@A,@P2,@P1) SUB (A[P1]≥A[P2])∧Biggest(A,P2-1,P1);
          GOAL Biggest(@A,@P2,@P2) SUB (A[P2]≥A[P1])∧Biggest(A,P2-1,@P1);

          AXIOM Permutation(@I,@I)↔TRUE;
          AXIOM Permutation(Exchange(@I1,@I2,@I3),@I1)↔TRUE;
          GOAL Permutation(@A,@B) SUB Permutation(A,@C)∧Permutation(@C,B);

          AXIOM IF Y=P1[P3] THEN
                    <<@P1,@P2,@Y>,@P3,@P4>↔Exchange(<P1,P2,P4>,P2,P3);

          GOAL 0 ≤@P1+@P2 SUB (0≤P1)∧(0≤P2);
          GOAL @P1≤@P2 SUB(P1≤@P3)∧(@P3≤P2);
          AXIOM @P1<@P2↔P1+1≤P2;
```

This is the output of computation which verified the program in 19 seconds.

```
THERE ARE 4 VERIFICATION CONDITIONS

# 1
(1≤L
→
 Permutation(A,A) &
 Ordered(A,L+1,L) &
 Partitioned(A,L) &
 1≤L &
 (¬1<I#1 &
  Permutation(A#1,A) &
  Ordered(A#1,I#1+1,L) &
  Partitioned(A#1, I#1) &
  1≤I#1

 -+I ssortedarrayof(A#1,A)))

# 2
(1<I &
 Permutation(A,A0) &
 Ordered(A,I+1,L) &
 Partitioned(A,I) &
 1≤I
→
 Biggest(A,2-1,1) &
 1≤I &
 1≤2-1 &
 2-1≤I &
 A [1] =A [1]&
 (¬J#3≤I &
  Biggest(A,J#3-1,K#3) &
  1≤K#3 &
  K#3≤J#3-1 &
  J#3-1≤I &
  X#3=A [K#3]
 →
  Permutation(<<A,K#3,A[I]>,I,X#3>,A0) &
  Ordered(<<A,K#3,A[I]>,I,X#3>,I-1+1,L) &
  Partitioned(<<A,K#3,A[I]>,I,X#3>,I-1) &
  1≤I-1))

# 3
(¬A [J] ≤X &
 J≤I &
 Biggest(A,J-1,K) &
 1≤K &
 KrJ-1 &
 J-1≤I &
 X=A [K]
3
 Biggest(A,J+1-1,J) &
 1≤J &
 J≤J+1-1 &
 J+1-1≤I &
 A [J] =A [J])
```

```
# 4
(A[J]≤X &
 J≤I &
 Biggest(A,J-1,K) &
 1≤K &
 K≤J-1 &
 J-1≤I &
 X=A[K]
→
 Biggest(A,J+1-1,K) &
 1≤K &
 K≤J+1-1 &
 J+1-1≤I &
 X=A[K])

AFTER SOME SIMPLIFICATION, YOU CAN GET

# 1
TRUE

# 2
TRUE

# 3
TRUE

# 4
TRUE

*****

TIME: 19 CPU SECS, 21 REAL SECS

778 STATE STACK CELLS USED
136 TOKEN STACK CELLS USED

958 DECISION POINTS
1947 FAILURES
3 SECS GC TIME
```

Here is another sorting program which has been verified. This is Floyd's TREE SORT program[4] with assertions and the goalfile. This is verified with 142 seconds of computation time. Most of the previously defined predicates are used in the goalfile with the same set of axioms. **Thus** there is a possibility of forming a standard set of symbols and axioms.

PASCAL

```
PROCEDURE TREESORT3(VAR A: TREEARRAY;L: INTEGER) ;
INITIAL A=A0;
ENTRY L≥2;
EXIT Issortedarrayof(A,A0);

         PROCEDURE SIFTUP(VAR M:REAL ;I,N:INTEGER);
         INITIAL I=I0,M=M0;
         ENTRY Treeordered(M,I+1,N)∧(I≥1);
         EXIT Treeordered(M,I0,N)∧Permutation(M,M0)∧
                  Unchanged(M,M0,1,I0-1)∧Unchanged(M,M0,N+1,L);

         VAR COPY: REAL: J: INTEGER:

                BEGIN
                        COPY ←M[I];
                    10: J ← 2 * I;
                        IF J ≤ N THEN
                                BEGIN
                                IF J < N THEN   IF  M[J+1]>M[J] THEN J ←J+1;
                                IF M [J]> COPY THEN
                                       BEGIN
                                        M[I] ← M[J];
                                ASSERT  Treeordered(M,I0,N)∧(COPY≤M[J DIV 2])∧
                                        Permutation(<M,J,COPY>,M0)∧
                                        Unchanged(M,M0,1, I0-1)∧
                                        Unchanged(M,M0,N+1,L)∧
                                        (N≥J)∧(J≥I0)∧(I0≥1);
                                        I ← J;
                                        GO TO 10
                                        END;
                                END;
                      M[I] ← COPY;
                ENO;

VAR WORK:REAL;I: INTEGER:

         BEGIN
                I←L DIV 2 ;
                INVARIANT Treeordered(A,I+1,L)∧(I≥1)∧Permutation(A,A0)
                WHILE I≥2 00
                        BEGIN SIFTUP(A,I,L);I←I-1 END:
                I←L;
                INVARIANT Ordered(A,I+1,L)∧Partitioned(A,I)∧Treeordered(A,2,I)
                        ∧(I≥1)∧Permutation(A,A0)
                WHILE I≥2 0 0
                        BEGIN
                        SIFTUP(A,1,I);
                        WORK←A[1]; A[1]←A[I]; A[I]←WORK;
                        I←I-1
                        END
         END;.;
```

```
GOALFILE
          GOAL Issortedarrayof(@A,@B) SUB Permutation(A,B)∧Ordered(A,1,L);

          AXIOM Permutation(@I,@I)↔TRUE;
          AXIOM Permutation(Exchange(@I1,@I2,@I3),@I1)↔TRUE;
          GOAL Permutation(@A,@B) SUB Permutation(A,@C)∧Permutation(@C,B);

          AXIOM IF Y=P1[P2]
                    THEN <<@P1,@P2,@Y>,@P3,@P4>↔Exchange(<P1,P2,P4>,P2,P3);

          AXIOM Ordered(A,L+1,L)↔TRUE;
          GOAL Ordered(@A,@P1,L) SUB Ordered(A,P1+1,L)∧Partitioned(A,P1);
          GOAL Ordered (Exchange (@A,@P1,@P2),@P3,L)
                    SUB Ordered(A,P3,L)∧(P1≤P3)∧(P2≤P3)∧Partitioned(A,P3);
          SUB Ordered(@Y,P,L)∧Unchanged(X,Y,@Q,L)∧(Q≤P+1);
          AXIOM Unchanged(@X,@X,@I,@J)↔TRUE;
          GOAL Unchanged(<@X,@I,@J>,@Y,@K,@L)
                    SUB Unchanged(X,Y,K,L)∧Outofrange(K,I,L);

          GOAL Biggest(@A,@I,1) SUB Treeordered(A,1,I);
          GOAL Biggest(Exchange(@A,@I,@J),@J,@J) SUB Biggest(A,J,I);

          GOAL Treeordered (@P1,@P2,@P3)
                    SUB Treeordered (P1,P2+1,P3)
                              ∧Biggerthanchildren(P1,P3,P2,P1[P2]);
          GOAL Treeordered(<@M,@J,@K>,@I,@N)
                    SUB Treeordered(M,I,N)∧Outofrange(I,J,N);
          GOAL Treeordered (@M,@I,@N) SUB N<2*I;
          GOAL Treeordered(<@M,@J,@K>,@I,@N)
                    SUB Treeordered(M,I,N)∧Smallerthanparent(M,I,J,K)∧
                              Biggerthanchildren(M,N,J,K);
          GOAL Treeordered (Exchange (@A,@I,@J),@K,@L)
                    SUB Treeordered(A,@M,@N)∧(K=I+1)∧(L=J-1)∧(M≤K)∧(N≥L);

          GOAL Outofrange(@I,@J,@N) SUB J<I, N<J;

          GOAL Smallerthanparent (@M,@I,@J,@K)
                    SUB J<2*I,(K≤M[J OIV 2]),K=M[2*J],K=M[2*J+1];

          GOAL Biggerthanchildren(@M,@N,@J,@K)
                    SUB N<2*J,  (N=2*J)∧(K≥M[N]) ,  (K≥M[2*J])∧(K≥M[2*J+1]);


          AXIOM Partitioned(A,L)↔TRUE;
          GOAL Partitioned(@A,@P1) SUB Partitioned(A,P1+1)∧Biggest(A,P1+1,P1+1);
          GOAL Partitioned(Exchange(@A,@P1,@P2),@P3)
                    SUB Partitioned(A,P3)∧(P1≤P3)∧(P2≤P3);
          GOAL Partitioned(@X,@P)
                    SUB Partitioned(@Y,P)∧Unchanged(X,Y,@Q,L)∧(Q≤P+1);

          AXIOM (@K*@L)DIV @K ↔ L;
          AXIOM @K*(@L DIV @K)↔ L;
          AXIOM IF M+1≤K THEN ((@K*@L)+@M)DIV @K ↔ L;
          GOAL @P1≤@P2 OIV @P3 SUB P1*P3≤ P2;
          GOAL 5 ≤@P1+@P2 SUB (0≤P1)∧(0≤P2);
          GOAL @P1≤@P2 SUB (P1≤@P3)∧(@P3≤P2);
          AXIOM @P1<@P2 ↔ P1+1≤P2;
```

This is Hoar-e's FIND program[7] and goalfile. This program is verified with 53 seconds of computation time.

```
PASCAL

PROCEDURE FIND(VAR A:FARRAY;F,K:INTEGER);
INITIAL A=A0;
ENTRY 1≤F&F≤K;
EXIT PARTITIONED(A,F)∧PERMUTATION(A,A0);

VAR M,K: INTEGER;VAR R:REAL;

BEGIN
M←1; N←K;
INVARIANT MINVARIANT(A,M)∧NINVARIANT(A,N)∧PERMUTATION(A,A0)
          ∧(M≤F)∧(F≤N)
WHILE  M  < N DO
           BEGIN
           R←A[F];I←M; J - N ;
           INVARIANT MINVARIANT(A,M)∧NINVARIANT(A,N)∧I INVARIANT(A,I,R)
                     ∧JINVARIANT(A,J,R)∧PERMUTATION(A,A0)∧(M≤I)∧(N≥J)
           WHILE I≤J DO
                     BEGIN
                     INVARIANT IINVARIANT(A,I,R)∧(M≤I)
                     WHILE A[I] < R DO  I←I+1;
                     INVARIANT JINVARIANT(A,J,R)∧(N≥J)
                     WHILE R<A[J] DO J ← J-l;
                     IF I≤ J THEN
                               BEGIN
                               W←A[I]; A[I]←A[J];  A[J]←W;
                               I←I+1; J←J-1
                               END
                     END:
           IF F ≤J THEN N←J ELSE IF I≤F THEN M←I ELSE GO TO 10
           END;
10:
END:.:
```

```
GOALFJLE
          AXIOM PERMUTATION(@I,@I)↔TRUE;
          AXIOM PERMUTATION(EXCHANGE(@I1,@I2,@I3),@I1)↔TRUE;
          GOAL PERMUTATION(@A,@B) SUB PERMUTATION(A,@C)∧PERMUTATION(@C,B);

          AXIOM  IF Y=P1[P2]
                    THEN <<@P1,@P2,@Y>,@P3,@P4>↔Exchange(<P1,P2,P4>,P2,P3);

          GOAL PARTITIONED(@A,@I) SUB MINVARIANT(A,I)∧NINVARIANT(A,I);
          AXIOM MINVARIANT(@A,1)↔ TRUE;'
          GOAL MINVARIANT(@A,@M)
                    SUB IINVARIANT(A,@I,@X)∧JINVARIANT(A,@J,@X)∧(I≥J+1)∧(I≥M)∧(M≥J);
          GOAL MINVARIANT(EXCHANGE(@A,@I,@J),@M) SUB MINVARIANT(A,M)∧(I≥M)∧(J≥M);
          AXIOM NINVARIANT(@A,K)↔ TRUE:
          GOAL NINVARIANT(@A,@N)
                    SUB IINVARIANT(A,@I,@X)∧JINVARIANT(A,@J,@X)∧(I≥J+1)∧(I≥N)∧(N≥J);
          GOAL NINVARIANT(EXCHANGE(@A,@I,@J),@N) SUB NINVARIANT(A,N)∧(I≤N)∧(J≤N);
GOAL IINVARIANT(@A,@I,@A[@J]) SUB MINVARIANT(A,I)∧(J≥I);
GOAL IINVARIANT(@A,@I,@J) SUB IINVARIANT(A,@K,J)∧(@K=I-1)∧(J≥A[@K]);
GOAL IINVARIANT(EXCHANGE(@A,@I,@J),@I+1,@R)
         SUB IINVARIANT(A,I,R)∧(I≤J)∧(R≥A[J]);
GOAL JINVARIANT(@A,@I,@A[@J]) SUB NINVARIANT(A,I)∧(J≤I);
GOAL JINVARIANT(@A,@I,@J) SUB JINVARIANT(A,@K,J)∧(@K=I+1)∧(J≤A[@K]);
GOAL JINVARIANT (EXCHANGE (@A,@I,@J),@J-1,@R)
         SUB JINVARIANT(A,J,R)∧(I≤J)∧(R≤A[I]);


AXIOM @A<@B ↔ A+1≤B;
GOAL @P1≤@P2 SUB(P1≤@P3)∧(@P3≤P2);
AXIOM IF P1≤P2 THEN @P2≤@P1 ↔ P1=P2;
```

Von Henke and Luckham have verified other programs using this system. Also a detailed study of the verification method has been performed.[5]

Acknowledgements. The author is grateful to David C. Luckham, who helped a great deal in writing this paper and in developing the system. Also, he is grateful to Friedrich W. von Henke and Jorge Morales for their advice and comments during the development of the system.

References

1. R.S. Boyer and J.S. Moore, *"Proving Theorems about LISP Problems"*, Third 1 JCAI Proceedings, 1973.

2. Deutsch, L. P., *An Interactive Program Verifier,* Ph. D. thesis, University of California, Berkeley, 1973.

3. R. W. Floyd, *"Assigning Meanings to Programs"*, Proc. Symp. Appl. Math., Amer. Math. Soc., Vol. 19, 1967, 19-32.

4. R. W. Floyd, *"Algorithms 245. TREESORT 3"*, CACM, Vol 7, 1964, Dec., 701.

5. F.W. von Henke and D.C. Luckham, *A Methodology for Verifying Programs,* forthcoming AIMEMO, Stanford Artificial Intelligence Project, Stanford University, 1974.

6. C.A.R. Hoare, *"An Axiomatic Basis for Computer Programming"*, CACM, Vol. 12, 1969, Oct., 576-580.

C.A.R. Hoare, *"Proof of a Program: FIND"*, CACM, Vol. 14, 1971, Jan., 39-45.

8. C.A.R. Hoare and N. Wirth, *"An Axiomatic Definition of the Programming Lang uage PASCAL"*, Acta Informatica 2, 19'73, 335-355.

9. S. Igarashi, R.L. London, and D.C. Luckham, *"Automatic Program Verification I: Logical Basis and Its Implementation"*, AIM-200, Stanford Artificial Intelligence Project, Stanford University, 1972.

10. J.C. King, *A* Program *Verifier*, Ph.D. thesis, Carnegie-Mellon University, 1969.

11. R. Milt-let-, *Logic for Computable Functions: Description of a Machine Machine Implementation,* AIM-200, Stanford Artificial Intelligence Project, Stanford University, 1972.