HEURISTIC METHODS FOR MECHANICALLY DERIVING INDUCTIVE ASSERTIONS

Ben Hegbreitt
Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

ABSTRACT

Current methods for mechanical program verification require a complete predicate specification on each loop. Because this is tedious and error-prone, producing a program with complete, correct predicates is reasonably difficult and would be facilitated by machine assistance. This paper discusses heuristic methods for mechanically deriving loop predicates from their boundary conditions and for mechanically completing partially specified loop predicates.

Introduction

Mechanical verification of program correctness is desirable and possible.[1] Given a program, a first-order axiotnatization of its semantics, and predicates on the input, output, and each loop, verification of the output predicate is a mechanical process, (c.f. [2] and [3] for recent surveys).

Input and output predicates are necessary and natural for a programmer to supply. However, completely specifying the predicates on loops is tedious, error-prone, and redundant. It is tedious due to the large amount of stereotyped detail required. It is error-prone* partly because of the tedium and partly because the notation is less natural than that for procedural steps. It is redundant since the predicates repeat information which is manifest in the program. The purpose of this paper is to show that loop predicates can be derived mechanically** and that partially specified loop predicates can be completed mechanically.

*An example may lend some weight of experience. In his thesis[4], King presents nine programs submitted to the verifier; the most complex of these (Example 9) has an incorrect loop predicate, i.e. the inductive assertion is too weak to be consistent *or* to imply the desired output predicate. (Since the theorem prover rejected the loop predicate due to an inability to handle multiple quantification, the predicate error was overlooked.)

**In one sense, this is trivial. All well-formed predicate expressions for each loop can be enumerated and proofs of correctness dovetailed until one succeeds. If every valid theorem of the subject domain is provable, this will eventually verify the program; otherwise, mechanical verification is not possible in general. Such a procedure is, however, computationally intractible.

tAlso at Harvard University, Cambridge, Mass.

Elspas, Green, Levitt, and Waldinger[5] have independently worked on this problem using difference equations as an aid *to* specifying assertions. Cooper[6] has previously studied the problem and observed that an inductive assertion can be obtained by hand by constructing the first few terms in the loop expansion, which generally shows what the infinite union must be.

Our method uses a different approach. To generate loop predicates where none are supplied, the output predicate is dragged backward through the program and modified when passing through program units, to produce trial loop predicates. Trial loop predicates which are loop inconsistent are modified according to various heuristics, to generate better trial predicates. Hence, it is also possible to accept a programmer-supplied inductive assertion which gives the "essential" idea of some loop and mechanically fill in the details to arrive at a complete, correct loop predicate. Many of the heuristics are domain specific, this paper uses integers and integer arrays as the subject domain.

The paper is divided into five sections. Section 2 illustrates our approach with two simple examples. Section 3 discusses the general method, domain-independent heuristics, and heuristics specific to the integers. Section 4 treats a number of complex examples to show how the heuristics are used and exhibit their coupling. Section 5 discusses implementation and application of this method.

Notation. Throughout, a simple flowchart language is used. The input predicate is denoted by o; the output predicate by w. Unprimed (primed) variables and predicates denote values and predicates on these values before (after) control flows through a set of flowchart boxes. The transformation due to a flowchart path A^i A£2-'*Ain is denoted by 6{ii|„i2,...£).

Simple Examples

The flowchart of Figure 1 (taken from [1]) computes the quotient Q and remainder R of integer X divided by integer Y. $={X>0 A Y>0), *={X-QY+R A 0<R A R<Y>. * at the entrance implies that at arc A_0 the predicate P«-{X>0 A Y>0 A Q«0 A R-X} holds. To verify the flowchart, it suffices to find a loop predicate P^ at arc A^ such that

(E1)
(E2) A 6(1,2,3,1) - P_1
(E3) P_2 A 6(1,4)

The standard means for generating a loop predicate is to use E3 and start with trial choice of P_t»{6(l,4) ■* *'}. Here, this gives P_1-{R<Y •+ (X-QY+R A 0sR A R<Y)}. Converting

to disjunctive form and simplifying, $P_1=\{R\geq Y \vee (X=QY+R \wedge \emptyset\leq R)\}$. To verify the flowchart, it suffices to prove that with this choice of P_1, E1 and E2 are each valid. E1 is

$$\{X\geq\emptyset \wedge Y>\emptyset \wedge Q=\emptyset \wedge R=X\} \rightarrow \{R\geq Y \vee (X=QY+R \wedge \emptyset\leq R)\}$$

which is valid. However, E2 is

$$\{R\geq Y \vee (X=QY+R \wedge \emptyset\leq R)\} \wedge R\geq Y \wedge R'=R-Y \wedge Q'=Q+1 \rightarrow \{R'\geq Y \vee (X=Q'Y+R' \wedge \emptyset\leq R')\}$$

which, while satisfiable, is not valid. Since E2 is satisfiable, this suggests that the trial choice for P_1 be replaced by a stronger one. Dropping a disjunct is a possible strengthening transformation; plausibility arguments suggest that the disjunct to drop is the one arising from $\delta(1,4)$. Hence consider the next trial choice $P_1=\{X=QY+R \wedge \emptyset\leq R\}$. E1 remains valid; E2 becomes

$$\{X=QY+R \wedge \emptyset\leq R\} \wedge R\geq Y \wedge R'=R-Y \wedge Q'=Q+1 \rightarrow \{X=Q'Y+R' \wedge \emptyset\leq R'\}$$

which is also valid. Hence, this choice of P_1 is said to validate E1 and E2, and the flowchart is verified.

The flowchart of Figure 2 (taken from [4]) computes A^B by a binary decomposition of B. $\phi=\{X=A \wedge Y=B \wedge Y\geq\emptyset\}$, $\psi=\{Z=A\uparrow B\}$. From ϕ it follows that $P_o=\{X=A \wedge Y=B \wedge Y\geq\emptyset \wedge Z=1\}$ holds at arc A_o. Using the exit condition E3, the first trial choice for P_1 is $\{Y=\emptyset \rightarrow Z=A\uparrow B\}$. This validates the entrance condition E1. The loop condition E2 generates an implication for each loop path. The one on $A_1A_2A_5A_6A_7A_1$ is valid. The one on $A_1A_2A_3A_4A_6A_7A_1$ requires

$$\{Y\neq\emptyset \vee Z=A\uparrow B\} \wedge Y\neq\emptyset \wedge Y \text{ MOD } 2=1 \wedge Z'=Z\cdot X \wedge Y'=Y\div 2 \wedge X'=X\cdot X \rightarrow \{Y'\neq\emptyset \vee Z'=A\uparrow B\}$$

which is not valid but is, however, satisfiable.

The occurrence of an invalid but satisfiable E2 implication indicates that the first trial loop predicate is too weak. Hence, heuristics are used to generate a stronger P_1 from the approximation $\{Y=\emptyset \rightarrow Z=A\uparrow B\}$. The appearance of antecedent $Y=\emptyset$ with a consequent equality suggests three likely candidates for a stronger P_1: $\{Z=A\uparrow B+K\cdot Y\}$, $\{Z=(A\uparrow B)\cdot(K\uparrow Y)\}$, and $\{Z\cdot(K\uparrow Y)=A\uparrow B\}$, where K is unknown. E1 and E2 are used to decide which, if any, of the three are consistent loop predicates and to solve for K in the consistent cases. Using the third choice for P_1 in E_1 gives

$$\{X=A \wedge Y=B \wedge Y\geq\emptyset \wedge Z=1\} \rightarrow \{Z\cdot(K\uparrow Y)=A\uparrow B\}$$

which is valid if K=A or K=X. Using the choice K=X in P_1, E2 gives

$$\{Z\cdot(X\uparrow Y)=A\uparrow B\} \wedge Y\neq\emptyset \wedge Y \text{ MOD } 2\neq 1 \wedge Y'=Y\div 2 \wedge X'=X\cdot X \rightarrow \{Z\cdot(X'\uparrow Y')=A\uparrow B\}$$

and

$$\{Z\cdot(X\uparrow Y)=A\uparrow B\} \wedge Y\neq\emptyset \wedge Y \text{ MOD } 2=1 \wedge Z'=Z\cdot X \wedge Y'=Y\div 2 \wedge X'=X\cdot X \rightarrow \{Z'\cdot(X'\uparrow Y')=A\uparrow B\}$$

both of which are valid. Hence, $P_1=\{Z\cdot(X\uparrow Y)=A\uparrow B\}$ verifies the flowchart.

Determining The Loop Predicates

General Considerations for Single-Loop Flowcharts

Consider a flowchart belonging to the schema of Figure 3. The initialization, loop-body, and terminal-computation are arbitrary loop-free computations. From ϕ and the initialization, the strongest predicate P_o on arc A_o can be derived mechanically. To verify the flowchart, it suffices to find a loop predicate P_1 at arc A_1 such that

(E1) $P_o \rightarrow P_1'$

(E2) $P_1 \wedge \delta(1,2,3,1) \rightarrow P_1'$

(E3) $P_1 \wedge \delta(1,4,5) \rightarrow \psi'$

In view of E3, the weakest possible choice for P_1 is $\hat{P}_1=\{\delta(1,4,5) \rightarrow \psi'\}$. Should E1 be invalid with this choice of P_1, the flowchart is not verifiable. E2 generates an implication for each path between A_2 and A_3: if all are valid, the flowchart is verified; if any are unsatisfiable, the flowchart is not verifiable; if all are satisfiable but some are invalid, then $\hat{P}_1$ must be replaced by a stronger inductive assertion.

There are two domain-independent heuristics for obtaining a stronger loop predicate from a previous approximation.

(G1) Convert to disjunctive form and drop some of the disjuncts which come from the exit predicate D. Typically, drop those disjuncts which cannot be used in any of the domain-specific heuristics described in the next section.

(G2) If $\{P_1 \wedge \delta_i \rightarrow P_1' \mid i=k_1,\ldots k_n\}$ are the invalid but satisfiable E2 implications, then replace P_1 by $P_1 \wedge (\delta_{k_1} \rightarrow P_1') \wedge \ldots \wedge (\delta_{k_n} \rightarrow P_1')\}$. If applied blindly and repeatedly, this can lead to an infinite sequence of trial P_1's; hence, this must be used to limited depth, in concert with heuristics for "simplifying" the expressions. (e.g. I1 through I4 below)

Whenever a new choice for P_1 is obtained by some heuristic from a previous approximation, it is tested in E1, E2, and E3. E1 not valid indicates that the choice is too strong. This is generally a good filter giving a quick reject to incorrect predicates; since the set of trial predicates should be pruned whenever possible, this is useful. E3 not valid indicates that the choice is too weak. An unsatisfiable implication due to E2 indicates that the choice is too strong. An invalid but satisfiable E2 implication indicates only that the choice is wrong.

There are three other useful domain-independent heuristics:

(G3) To verify a predicate, convert to conjunctive form and verify each conjunct separately. This may be a losing strategy if two or more conjuncts are coupled and can only be proved in concert, but when this works, it reduces the size of the formulas to be manipulated and hence cuts the computation for a proof and the search space for other heuristics. This independent processing of conjuncts applies both to ψ and to the results of G2. In the latter case, the new trial loop

predicate is the conjunction of the previous one and a set of new clauses, so this is a natural situation in which to try separation.

(G4) If the application of G3 results in verifying some but not all of the conjuncts, record those which do verify as consistent *loop predicates at that arc*. When subsequently searching for the proof of an inductive assertion at some arc A_k, the list of predicates previously verified at A_k may be a useful source of lemmas. Suppose, for example, that a trial predicate at A_k is $\{C_k^1 \wedge C_k^2\}$ and that C_k^1 is loop consistent while C_k^2 is not. If $\{C_k^1 \wedge C_k^2\}$ is not verifiable, and a new trial predicate is formed, verification of the new predicate may be aided by knowing that C_k^1 holds at A_k.

(G5) Each predicate which is found to hold at some arc A_k is propagated to all successors A_j under the transformation $\delta(k,j)$. This includes both the input predicate ϕ and the results of G4.

Because of G5, the first step of flowchart verification is to extend ϕ so far as possible, forming the assertion P_0 at one input to the first loop junction (c.f. Figures 1 and 2). Whenever, because of G4 or G5, it is known that some predicate P_1 holds at A_1, P_1 is *cojoined* as a clause on the left hand side of E2 and E3.

Integer-Specific Heuristics

Turning to domain-specific methods, the heuristics are richer and more powerful. The following heuristics apply to the integers (and to the reals, except as noted otherwise). They each replace a predicate by a weaker one; when used inside a negation (as is commonly the case) their application results in a stronger inductive assertion.

(I1) In a conjunction of inequalities, form *transitive closures* and drop the conjuncts used. For example, replace*

$\{A<B \wedge B\leq C \wedge \ldots\}$ by $\{A<C \wedge \ldots\}$,

$\{A\leq B \wedge B\leq C \wedge \ldots\}$ by $\{A\leq C \wedge \ldots\}$,

$\{A<B \wedge B<C \wedge \ldots\}$ by $\{A+1<C \wedge \ldots\}$.

(I2) In a conjunction of an equality and an expression ϵ equal to zero, a term in ϵ may be added to one side of the equality, e.g. replace

$\{X=\emptyset \wedge A=B\cdot C\}$ by $\{A=B\cdot C+K\cdot X\}$

where K is unknown. Less common, but possible is: multiply one side of the equality by K^{ϵ}, e.g. replace

$\{X=\emptyset \wedge A=B\cdot C\}$ by $\{A=B\cdot C\cdot(K+X)\}$ or

$\{(K+X)\cdot A=B\cdot C\}$.

(A similar case is the conjunction of an equality and an expression ϵ equal to 1: either side of the equality may be multipled by ϵ. In practice, this turns out to be useful less frequently than the zero case.)

*In the case of reals, the last of these is changed to: replace

$\{A<B \wedge B<C \wedge \ldots\}$ by $\{A<C \wedge \ldots\}$.

(I3) In a conjunction of an inequality and an existentially quantified conjunction, use transitivity to extend the range of the quantified variable, and drop the inequality conjunct. For example, replace

$\{N\leq L \wedge \exists K[K<N \wedge Q(K)] \wedge \ldots\}$ by
$\{\exists K[K<L \wedge Q(K)] \wedge \ldots\}$.

(I4) In a conjunction of an inequality and a universally quantified disjunction, use transitivity to extend the range of the quantified variable and drop the inequality conjunct. For example,

$\{I\leq N \wedge \forall K[K\geq N \vee Q(K)] \wedge \ldots\}$

may be replaced by the weaker form

$\{\forall K[K<I \rightarrow Q(K)] \wedge \ldots\}$.

Multiple Loops

The next level of loop complexity beyond the single loop schema is the class of simply-nested loop programs, in which the loop-body of Figure 3 contains inner loops. Consider, for example, the schema of Figure 4 where the inner-body and outer-body are loop free. To verify this, it suffices to find an inductive assertion P_4 such that:

(E4) $P_0 \wedge \delta(\emptyset,1,2,3,4) \rightarrow P_4'$

(E5) $P_4 \wedge \delta(4,5,6,4) \rightarrow P_4'$

(E6) $P_4 \wedge \delta(4,7,8,1,2,3,4) \rightarrow P_4'$

(E7) $P_4 \wedge \delta(4,7,8,1,9,10) \rightarrow \psi'$

The weakest possible choice for P_4 is $\tilde{P}_4=\{\delta(4,7,8,1,9,10) \rightarrow \psi'\}$.

This class of flowcharts is of interest for two reasons. (1) It and its generalizations to depth n occur frequently. (2) Obtaining a solution for P_4 requires no new techniques, since E5 and E6 each have the same form as E2 - the set of implications they generate cover the set of possible paths from arc A_4 to itself - and the nesting is otherwise irrelevant. These considerations extend to simply-nested flowcharts of arbitrary depth.

In general, the situation is less clean. Consider, for example, the schema of Figure 5. Here, there is no single innermost arc: A_3 and A_7 are each innermost. The verification conditions are

(E8) $P_0 \wedge \delta(\emptyset,1,2,3) \rightarrow P_3'$

(E9) $P_3 \wedge \delta(3,4,5,3) \rightarrow P_3'$

(E10) $P_3 \wedge \delta(3,4,6,7) \rightarrow P_7'$

(E11) $P_7 \wedge \delta(7,8,9,7) \rightarrow P_7'$

(E12) $P_7 \wedge \delta(7,8,10,1,2,3) \rightarrow P_3'$

(E13) $P_7 \wedge \delta(7,8,10,1,11) \rightarrow \psi'$

Finding P_3 and P_7 is complicated by the coupling due to E10 and E12. Loops such as these can sometimes be decoupled by the following heuristic.

(G6) Approximate the first loop by a finite expansion to some depth i -- one disjunct per time around the loop. Use this to find a valid inductive assertion on the second loop. Then use that assertion to obtain the assertion for the first loop. For example, the loop <3,4,5,3> may be approximated to depth $i=1$ by

$\Delta_{i=1}=(B_2 \wedge \sim D_2) \vee (B_2 \wedge D_2 \wedge B_2' \wedge \sim D_2')$

P_7 must validate

(E1′) $P_o \wedge \sim D_1 \wedge \Delta_i \rightarrow P_7'$

(E2$_a$′) $P_7 \wedge \delta(7,8,9,7) \rightarrow P_7'$

(E2$_b$′) $P_7 \wedge \delta(7,8,10,1,2,3) \wedge \Delta_i \rightarrow P_7'$

(E3′) $P_7 \wedge \delta(7,8,10,1,11) \rightarrow \psi'$

which is in standard form. With P_7 obtained, P_3 is determined by E8, E9, E10, and E12.

More Complex Examples

The flowchart of Figure 6 (taken from [4]) sets the flag variable J to 0 or 1 as A is or is not prime. P_o={A≥2 ∧ I=2}, ψ={[J=0 → ∀K(2≤K<A → A MOD K≠0)] ∧ [J=1 → A MOD I=0]}. Using E3, the weakest possible inductive assertion at arc A_1 is

$\tilde{P}_1$={[(I≥A ∧ J′=0) ∨ (I<A ∧ A MOD I=0 ∧ J′=1)] → ψ'}.

Simplifying this, and converting to disjunctive form

$\tilde{P}_1$={I<A ∨ ∀K(2≤K<A → A MOD K≠0)}.

(The disjunct J′≠0 has been dropped, since it involves primed variables.) Since $P_o \rightarrow \tilde{P}_1$, E1 is validated. $\tilde{P}_1 \wedge \delta(1,2,3,4,1) \rightarrow \tilde{P}_1'$ is invalid but satisfiable. To obtain a stronger inductive assertion, I3 is used to replace ~{I≥A ∧ ∃K(2≤K<A ∧ A MOD K=0} by the stronger predicate ~{∃K(2≤K<I ∧ A MOD K=0}. That is, the next trial P_1 is

{∀K (2≤K<I → A MOD K≠0)}.

This validates E1 and E2, so the flowchart is verified.

The flowchart of Figure 7 (taken from [2]) computes* the fractional quotient P/Q to within tolerance E. ϕ={0≤P<Q ∧ 0<E} so that P_o={0≤P<Q ∧ 0<E ∧ A=0 ∧ B=Q/2 ∧ D=1 ∧ Y=0}; ψ={P/Q-E<Y≤P/Q}. Using E3, the weakest possible loop predicate at A_6 is

$\tilde{P}_6$ = {D<E → ψ'}
= ~{(D<E ∧ P/Q-E≥Y) ∨ (D<E ∧ Y>P/Q)}.

This validates implication E1, since $P_o \wedge \delta(0,1,2,3,5,6) \rightarrow \tilde{P}_6'$ and $P_o \wedge \delta(0,1,4,5,6) \rightarrow \tilde{P}_6'$, E2 generates two implications

(7.1) $P_6 \wedge \delta(6,7,1,2,3,5,6) \rightarrow P_6'$

(7.2) $P_6 \wedge \delta(6,7,1,4,5,6) \rightarrow P_6'$

With $\tilde{P}_6$, neither of these is valid but both are satisfiable, hence $\tilde{P}_6$ must be replaced by a stronger predicate.

In determining the correct P_6, it is helpful to use heuristic G3: break ψ into two conjuncts and verify each separately. Consider first

$\tilde{P}_6^1$=~{D<E ∧ P/Q-E≥Y}.

With this choice of P_6, both 7.1 and 7.2 are invalid but satisfiable. In replacing $\tilde{P}_6^1$ by a stronger predicate, heuristic I1 is used to get ~(P/Q-D>Y), i.e. {P≤QY+QD}. With this choice of P_6, E1 and 7.1 are each validated, while 7.2 is invalid but satisfiable. Since 7.2 is {P≤QY+QD ∧ D≥E ∧ P<A+B → P≤QY+QD/2}, heuristic G2 suggests the next trial choice for P_6 is

{[P≤QY+QD] ∧ [D≥E ∧ P<A+B → P≤QY+QD/2]},

while heuristic G3 suggests verifying the two conjuncts separately. When G2 and G3 are applied together, it suffices to verify the new conjuncts (here, second conjunct). With P_6={D<E ∨ P≥A+B ∨ P≤QY+QD/2}, E1 is validated while 7.1 and 7.2 are each invalid but satisfiable. This suggests using a heuristic to get a stronger predicate; I1 and G1 applied to ~{D≥E ∧ P<A+B ∧ P>QY+QD/2} give ~{A+B>QY+QD/2}.
Using this for P_6, E1 is validated while 7.1 and 7.2 are each invalid but satisfiable. Using G2, I1, and then G3, 7.1 and 7.2 respectively give rise to the new conjuncts {B≤QD/2} and {B≥QD/2}.
Combining these into the single conjunct {B=QD/2} and using this for P_6, E1, 7.1 and 7.2 are each validated. Hence the output conjunct ψ^1={P/Q-E<Y} is verified with the loop predicate P_6^1={P≤QY+QD ∧ A+B≤QY+QD/2 ∧ B=QD/2}.

The second output conjunct ψ^2={Y≤P/Q} is simpler. Successive conjuncts generated for P_6^2 are {Y≤P/Q}, then {A+B≥QY+QD/2}, and finally {B=QD/2}. As in the case of P_6^1, the primary choice at each step is between using G2 to add another conjunct and some other heuristic to simplify by strengthening a conjunct. Since ψ^i is verified by P_6^i for i=1,2, the flowchart is verified. Combining P_6^1 and P_6^2 gives P_6={P/Q-D≤Y≤P/Q ∧ B=QD/2 ∧ A=QY}.

Figure 8 (taken from [4]) gives a simple exchange sort.
P_o={}; ψ={∀M 2≤M≤N → A[M-1]≤A[M]}.
As A_3 is the unique overlap arc of the two loops, the inductive assertion is formed there (c.f. Section 3.3). From E3,

$\tilde{P}_3$={I>N ∧ J=0 → ψ'} = ~{I>N ∧ J=0 ∧ ∃M(2≤M≤N ∧ A[M-1]>A[M])}.

This validates E1, trivially. E2 generates three implications:

(8.1) $P_3 \wedge \delta(3,10,11,1,2,3) \rightarrow P_3'$

(8.2) $P_3 \wedge \delta(3,4,5,6,8,9,3) \rightarrow P_3'$

(8.3) $P_3 \wedge \delta(3,4,7,8,9,3) \rightarrow P_3'$

With $\tilde{P}_3$, these are invalid but satisfiable. Hence, it is necessary to replace $\tilde{P}_3$ with a stronger assertion.

Using heuristic I3,

P_3= ~{J=0 ∧ ∃M(2≤M<I ∧ A[M-1]>A[M])}
= {J≠0 ∨ ∀M(2≤M<I → A[M-1]≤A[M])}.

This choice of P_3 validates E1 and the three implications of E2. The most difficult proof is for 8.3:

{J≠0 ∨ ∀M(2≤M<I → A[M-1]≤A[M])}
∧ I≤N ∧ A[I-1]≤A[I] ∧ I′=I+1 →
{J≠0 ∨ ∀M(2≤M<I′ → A[M-1]≤A[M])}.

which requires a case analysis on the quantified variable M.

*In this example, the domain is the reals, rather than the integers. The techniques carry over with only minor changes in the treatment of transitivity.

Conclusion

Having shown that it is possible in some number of cases to derive the inductive assertions by heuristics, we turn to the application of this in a mechanical verification system.

Implementation

We have not yet implemented this technique and the above examples are the result of hand simulations. It appears that programming this is reasonable, although by no means trivial.

Modifying the trial loop predicates and searching the space of possible modifications is the heart of this method. A problem-solving language (e.g. [7], [8] or [9]) with facilities for pattern-matching, backtracking, and multiple environments is the preferred implementation tool. The need for pattern-matching to invoke heuristics, and backtracking to make retractible trials should be clear. Experience with hand simulation suggests that additionally a breadth-first search capability (i.e. multiple environments) is required, since an incorrect choice of heuristic often leads not to a recognizable failure but rather to a non-convergent series of loop predicates - each inconsistent.

The theorem-proving required is within the range of contemporary domain-specific theorem provers (e.g. [10]). This can also be programmed in the problem-solving language.

The interface between heuristic predicate generation and theorem proving can be organized in two ways. A disjoint organization would separate the two and use the theorem prover as an oracle returning one of four replies for any formula submitted to it: valid, unsatisfiable, invalid-but-satisfiable, or can't-tell (escape clause for undecidable domains). A better organization is based on the observation that the heuristics used in trial predicate generation and in theorem proving are often the same and that this can be exploited by combining the two more closely. Further, in seeking to validate an E2 implication with some trial predicate, the theorem prover may generate a number of clauses which are invalid but satisfiable; it is often useful to take these as new goals and try them as additional conjuncts in the next trial predicate (i.e. a variation on G2).

Application

We do not believe that the proposed techniques will be capable of generating all inductive assertions on a large, complex program in a reasonable time scale. Rather, we assume that a program submitted to a verifier will have its loops tagged with assertions of varying degrees of completeness: some complete, some partial, and some untagged. In general, interaction will be required in developing correct assertions and proving very difficult lemmas. It is desirable, however, to do mechanically as much as practical. The proposed technique allows completing certain loop predicates and generating certain inner predicates from outer ones. This should suffice to fill in a significant portion of the inductive assertions mechanically and, therefore, should expedite the task of proving programs correct.

ACKNOWLEDGEMENTS

The author would like to thank L. Peter Deutsch, carl Hewitt, Ralph London and Jay Spitzen for discussions concerning various aspects of mechanical program verification. Ralph London is the source for the observation of predicate incompleteness in the example discussed in Section 1.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract No. DAHC-71-C-0088.

REFERENCES

1. R.w. Floyd, "Assigning Meanings to Programs," in Proc. of a Symposium in Applied Mathematics, Vol. 19, ed. by J.T. Schwartz, AMS, pp. 19-32 (1967).
2. B. Elspas, K.N. Levitt, R.J. Waldinger and A. Waksman, "An Assessment of Techniques for Proving Programs Correct," Computing Surveys, Vol. 4, No. 2, pp. 97-147 (June 1972).
3. R.L. London, "The Current State of Proving Programs Correct," Proc. ACM 25th Annual Conference, pp. 39-46 (1972).
4. J. King, A Program Verifier, Doctoral Dissertation, Computer Science Dept., Carnegie-Mellon university, Pittsburgh, Pennsylvania (1969).
5. B. Elspas, M.W. Green, K.N. Levitt and R.J. Waldinger, "Research in Interactive Program-Proving Techniques," SRI, Menlo Park, California (May 1972).
6. D.C. Cooper, "Programs for Mechanical Program Verification," Machine Intelligence £, American Elsevier, pp. 43-59 (1971).
7. J.F. Bulifson, J.A. Derksen and R.J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," SRI, Technical Note 73 (November 1972).
8. D.V. McDermott and G.J. Sussman, "The CONNIVER Reference Manual," MIT, A.I. Lab., Memo No. 259 (May 1972).
9. C. Hewitt, "PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot," Ph.D. Thesis, MIT, Department of Mathematics (January 1971).
10. L.P. Deutsch, "An Interactive Program Verifier," Ph.D. Thesis, Department of Computer Science, University of California Berkeley (forthcoming, June 1973).

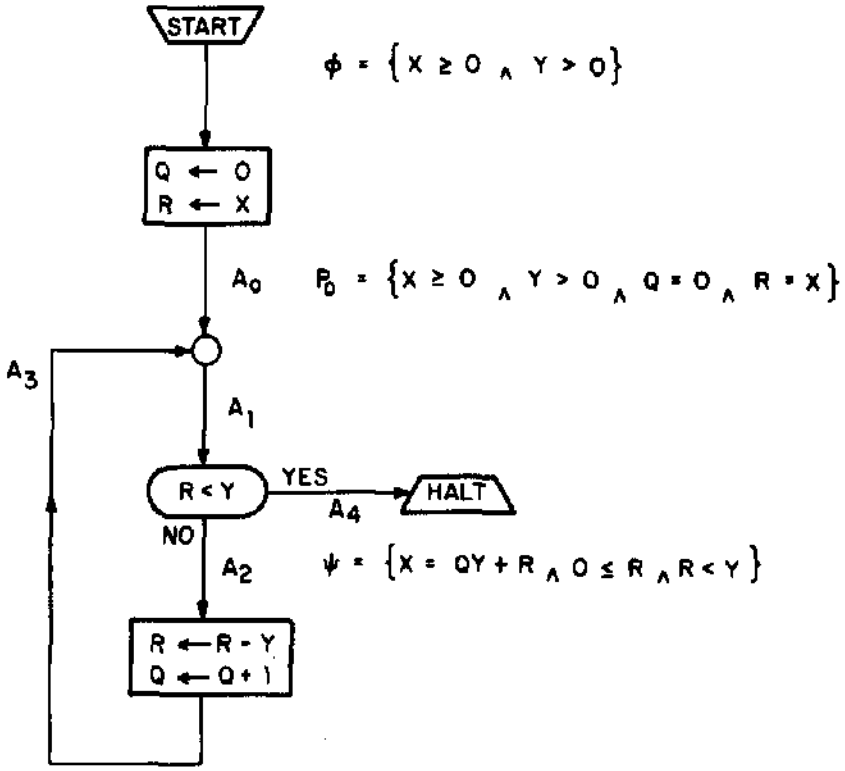


Figure I. Division by Repeated Subtraction

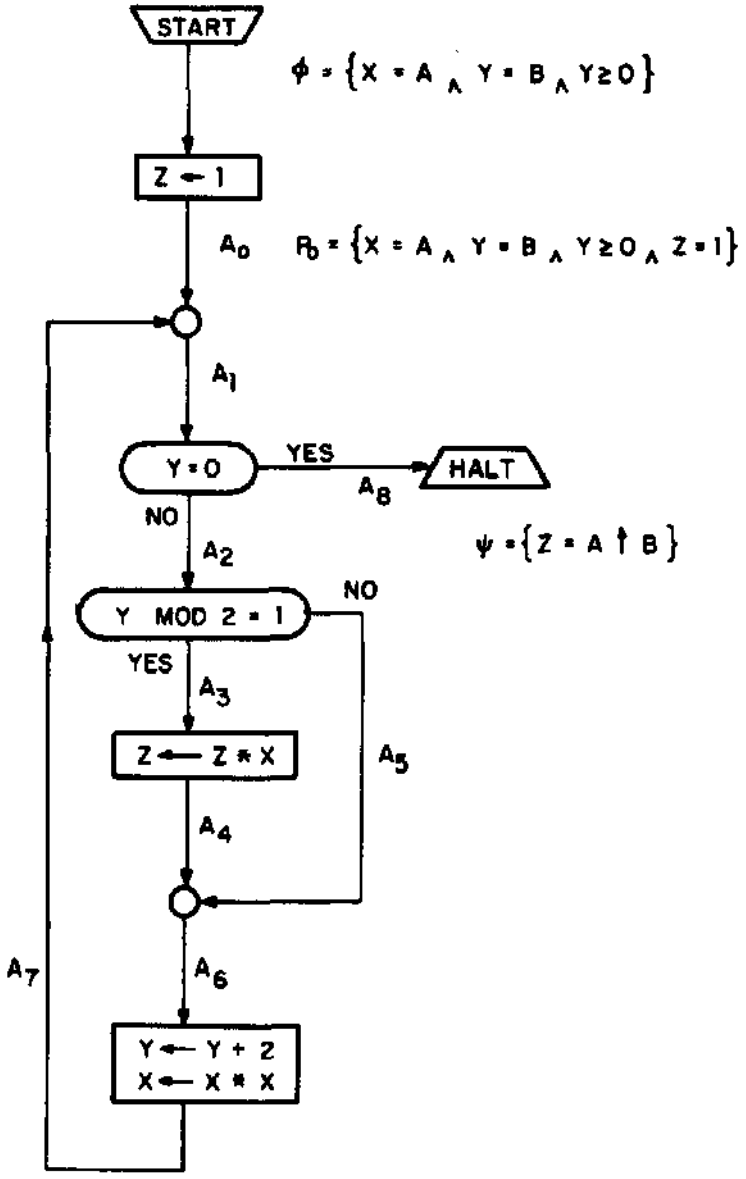


Figure 2. Exponentiation by Binary Decomposition

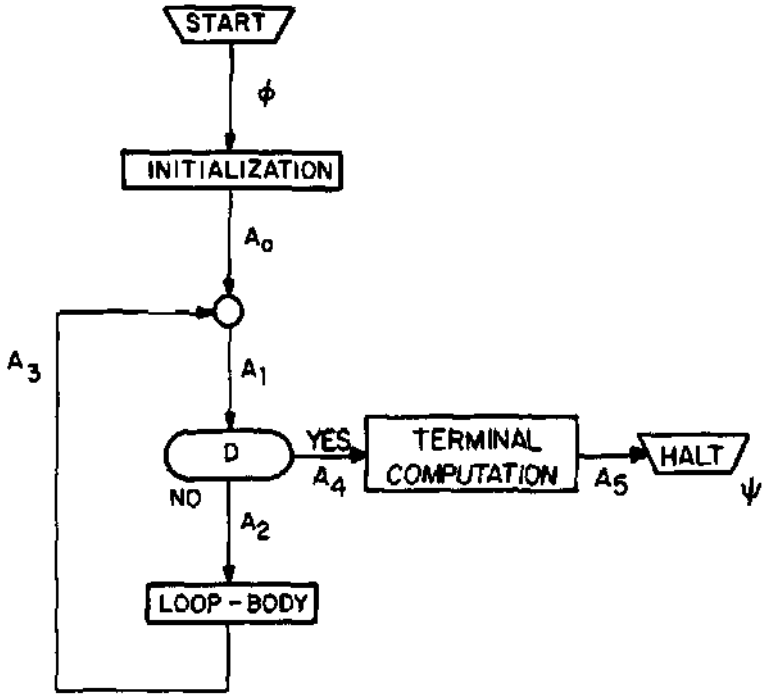


Figure 3. Single Loop Schema

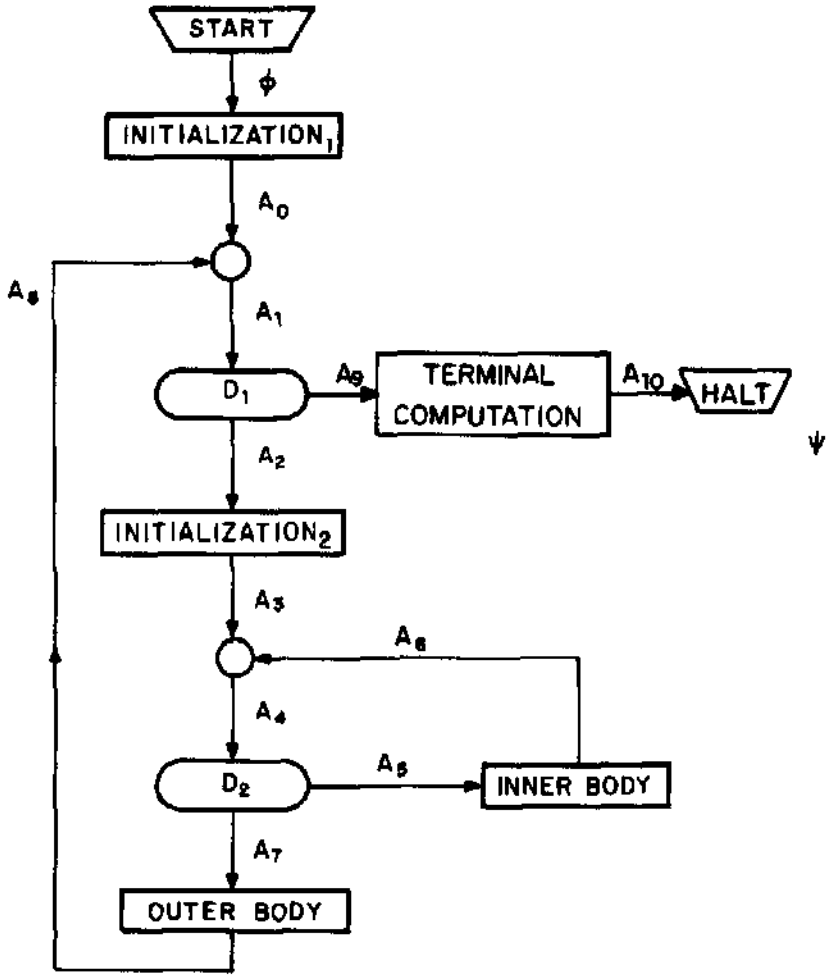


Figure 4. Simply Nested Loop Schema

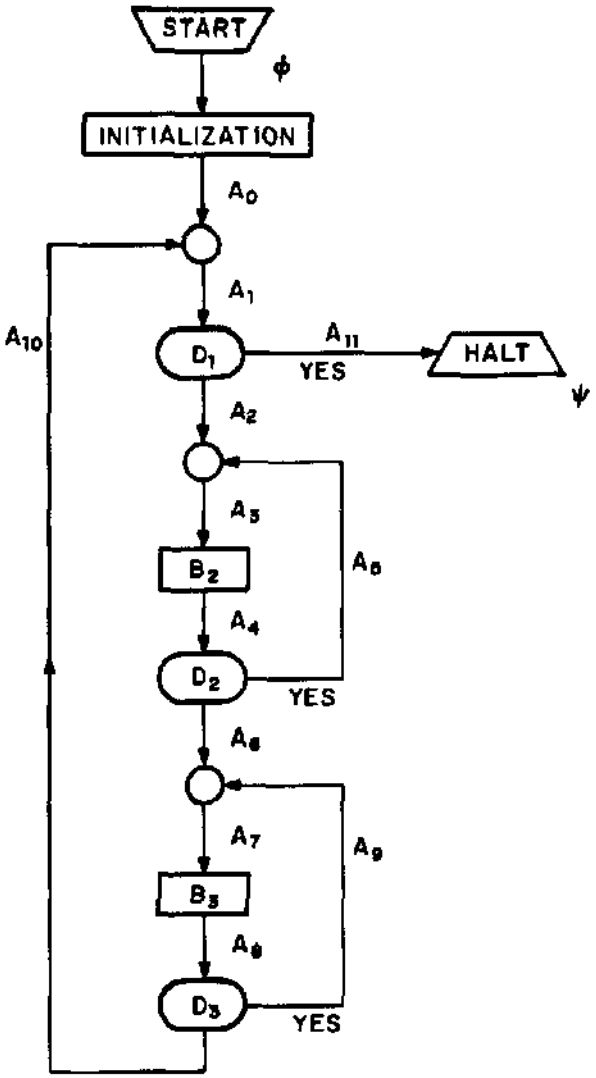
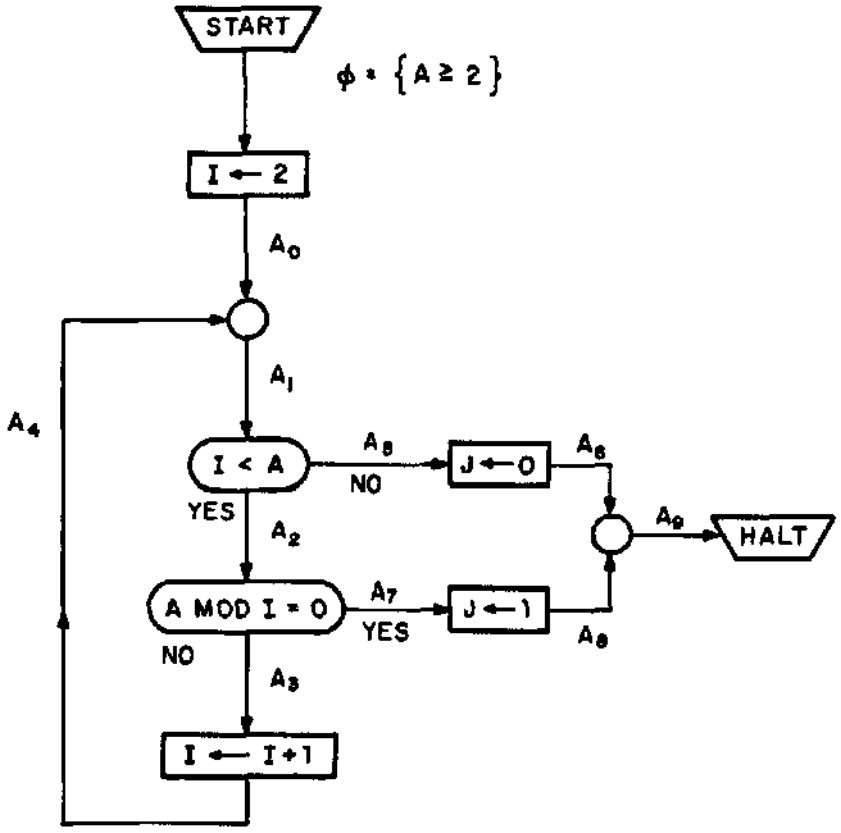


Figure 5. Multiple Loop Flowchart



$$\psi = \{[J = 0 \Rightarrow \forall K\ (2 \leq K < A \Rightarrow A \bmod K \neq 0)]_A$$
$$[J = 1 \Rightarrow A \bmod I = 0]\}$$

Figure 6. Testing for Prime

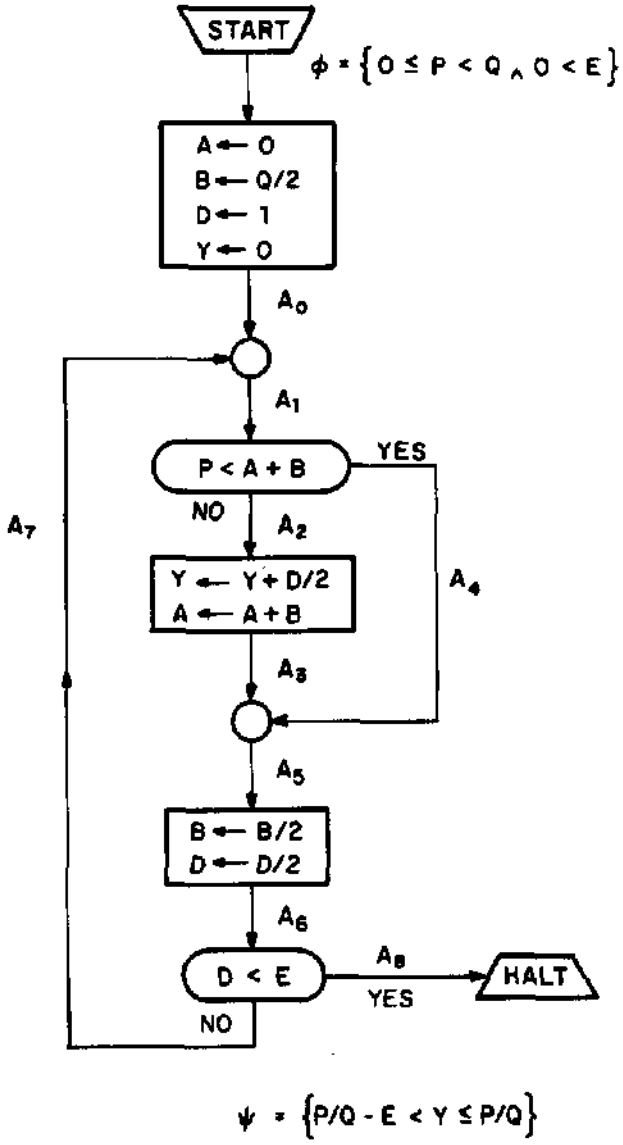
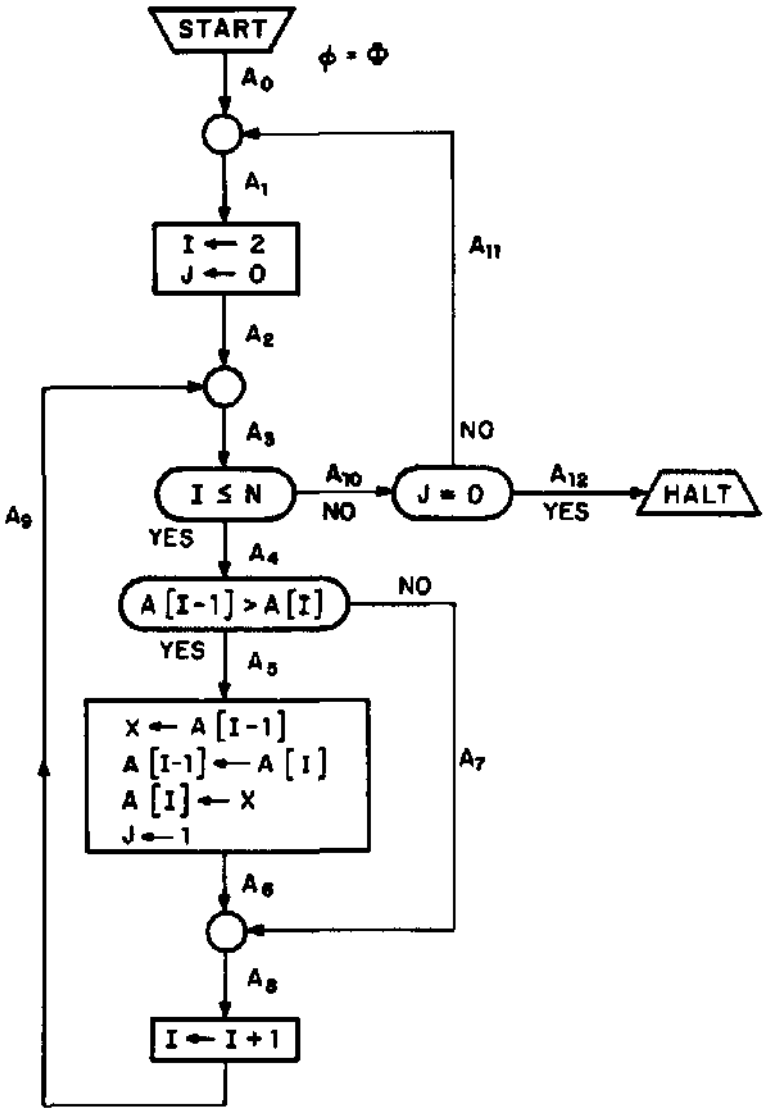


Figure 7. Wensley's Quotient Algorithm



$$\psi = \{ \forall M \; 2 \leq M \leq N \Rightarrow A[M-1] \leq A[M] \}$$

Figure 8. Simple Exchange Sort