J. Symbolic Computation (1993) **15**, 705–744

On the Mechanical Derivation of Loop Invariants

RITU CHADHA[†] AND DAVID A. PLAISTED[‡]

[†] *Bell Communications Research, Morristown NJ, U.S.A.*
[‡] *Department of Computer Science, University of North Carolina at Chapel Hill, U.S.A.*

(Received 13 May 1992)

We describe an iterative algorithm for mechanically deriving loop invariants for the purpose of proving the partial correctness of programs. The algorithm is based on resolution and a novel unskolemization technique for deriving logical consequences of first-order formulas. Our method is complete in the sense that if a loop invariant exists for a loop in a given first-order language relative to a given finite set of first-order axioms, then the algorithm produces a loop invariant for that loop which can be used for proving the partial correctness of the program. Existing techniques in the literature are not complete.

1. Introduction

A loop invariant of a loop is a predicate that is true at the beginning of every iteration of the loop. To prove that a program with given input and output specifications is partially correct, loop invariants have to be supplied for every loop in the program. This enables us to partition the program into a finite number of execution paths whose extremities are annotated with predicates that are true at that point in the program. Verification conditions can then be written for each path (Manna, 1974). These verification conditions are formulated so that a proof of the validity of a verification condition shows that if the predicate at the beginning of the path is true, then after executing the statements along the path, the predicate at the end of the path will be true. If such verification conditions can be proved for all program paths, then, by a simple induction argument, the program is partially correct.

The objective of this paper is to describe a method for mechanically deriving loop invariants for a loop in a flowchart program for the purpose of proving the partial correctness of the program. Not all loop invariants will satisfy this purpose; e.g. the predicate "true" is a loop invariant for any loop, but will not always serve the purpose of proving partial correctness. Therefore, whenever the term "loop invariant" is used in this paper, it refers to a loop invariant that can be used to prove the partial correctness of the program.

No complete method can exist for automatically deriving loop invariants for all possible loops, since by Cook's completeness result (Cook, 1978), there exist loops for which no suitable loop invariants exist, unless the language being used is "expressive" in some sense (see (Loeckx and Sieber, 1987) for a detailed coverage of this topic). Any calculus based on attaching first-order formulas to arcs of flowchart programs may be incomplete

0747–7171/93/5–6705+40 $08.00/0 © 1993 Academic Press Limited

because the set of possible values on an arc of the flowchart may not be first-order definable (Wand, 1978). Most of the attempts made at developing methods for automatically generating loop invariants have been heuristic in nature, so none of these methods have been complete in any sense. In contrast, we make the following completeness claim about our method: given any loop, if a loop invariant exists for that loop in a given first-order language relative to a given finite set of first-order axioms, then our method produces a loop invariant for that loop which can be used for proving the partial correctness of the program. Of course, not all theories of interest can be expressed by a finite collection of first-order axioms.

In what follows, we first describe past work in the area of program verification. We then describe a method for deriving logical consequences of first-order formulas using resolution and unskolemization, analyze some of the properties of this method, and explain how to apply it for mechanically generating loop invariants. The method for deriving loop invariants is intimately tied in with the technique for derivation of logical consequences; thus for a complete presentation of the method, it is necessary to describe the technique for generation of logical consequences in some detail. This method is applicable when there exists an axiomatization of the model of the data structures and primitive operations of the language (see Section 5).

2. Past work

The inductive-assertion method for program verification, developed by Floyd in 1967 (Floyd, 1967), is now the basis for a large number of automated program verification systems. This method requires that the user annotate the loops of the program with *inductive assertions* (also called *loop invariants*) that are invariants of the loops. The automatic derivation of loop invariants is of great interest and of potentially great use.

Existing program verification systems can be divided, for our purposes, into two categories: those in which the user has to supply loop invariants and those in which the program verifier provides assistance in deriving loop invariants.

In his pioneer system, King (King, 1969) describes the program verifier of his Ph.D. dissertation. Loop invariants have to be provided by the programmer. The formal analysis of the program produces verification conditions that must be proved to be theorems over integers. These theorems are proved by formula simplification routines and specialized techniques for integer expressions. Deutsch describes the interactive program verifier PIVOT (Programmer's Interactive Verification and Organizational Tool) in (Deutsch, 1973). PIVOT is based on Floyd's method. Another program verification system in which the user provides loop invariants is described by Cooper (Cooper, 1971). Cooper states that iterating a loop a few times soon gives the programmer a good idea of what a suitable invariant might be, but he does not give a program to do this automatically. Good et al. (Good, London and Bledsoe, 1975) report the development of an interactive program verification system for verifying Pascal programs. In their system, the user is primarily responsible for correctness proofs for programs. All loop invariants are provided by the programmer. Good also describes the Gypsy verification environment, a large, interactive computer program that supports the construction of formal, mathematical proofs about the behavior of software systems (Good, 1985).

Some heuristic methods for mechanically generating loop invariants have been developed. German and Wegbreit (German and Wegbreit, 1975) describe a system that provides assistance to the user in synthesizing them. In (Wegbreit, 1973), Wegbreit describes

heuristic methods for mechanically deriving loop invariants from their boundary conditions and for mechanically completing partially specified loop invariants. The method uses the output predicate to derive suitable loop invariants by dragging the output predicate backwards through the program and modifying it suitably when passing through the statements of the program. Another alternative he gives is to take a programmer-supplied loop invariant, that contains the "essential idea" of a loop and mechanically fill in the details. Wegbreit describes some domain-dependent and some domain-independent heuristics for deriving loop invariants. He starts by using the weakest possible loop invariant for a particular loop that will satisfy one of the verification conditions and tries to strengthen it using a number of heuristics. The efforts of Katz and Manna (Katz and Manna, 1973) are also directed towards automatically deriving loop invariants. They describe two general approaches. The first is a top-down approach, similar to that of (Wegbreit, 1973). The second is a bottom-up approach, in which the loop invariant is generated directly from the statements in the loop, finding general expressions for the values of the program variables after n loop iterations and then eliminating n from these expressions. In (Seiichiro and Yamaguchi, 1989), a synthesizer of loop invariants is described that provides assistance to the user in discovering loop invariants. The system makes use of a difference-equation solver, symbolic execution, and an expression generalizer. Others have done research on methods for automatically deriving loop invariants for specific types of programs. Caplain (Caplain, 1975) describes a technique applicable to numerical programs, which is based on expressing the transformation of the n variables in a loop by an $n \times n$ matrix.

The iterative algorithm that we describe may appear similar to the abstract implementation approach (Cousot and Cousot, 1977), (Wegbreit, 1975); however, there are significant differences. Wegbreit (Wegbreit, 1975) gives a method for mechanically deriving certain classes of predicates for loops by symbolic execution of the program. Provided the class of predicates obeys certain constraints, symbolic execution is guaranteed to terminate. He deals with weak interpretations, i.e. models chosen to be appropriate for specific optimizations. This approach is valid but fundamentally incomplete. However, because the set of predicates useful to an optimizing compiler is generally fixed for that compiler, this incompleteness may be acceptable for the purpose of deriving predicates relevant for optimization purposes. Our approach is complete within the framework of first-order logic; a proof of this is provided in the appendix.

3. Derivation of logical consequences

3.1. RELATION TO PROGRAM VERIFICATION

The problem of deriving a loop invariant for a *while* loop is basically that of expressing an infinite disjunction of formulas in a closed form. To see this, suppose W is a loop invariant for a loop. Let A_i be the formula that holds before iteration i of the loop. Then we have:

$$(\forall i : 0 \leq i : A_i \Rightarrow W)$$

A loop invariant at this point would be

$$W = \bigvee_{i=1}^{\infty} A_i.$$

Thus we have to find a way of expressing $\bigvee_{i=1}^{\infty} A_i$ in a finite form. We know that $A_1 \Rightarrow W$, $A_2 \Rightarrow W$, $A_3 \Rightarrow W$, ..., and so on, i.e. W is a logical consequence of each A_i,

for all i, $1 \leq i$. Thus we need to develop a method for finding logical consequences of first-order formulas. We now describe the development of such a method. Suppose we want to find a certain, unknown, consequence W of a first-order formula H. It may not be possible to derive W from H by resolution (Robinson, 1965) without using tautologies and unskolemization, as will be shown in Section 3.2.1. Since the use of tautologies is undesirable (due to the enormous increase in search space that it creates), we will not attempt to derive W from H, but instead will try to derive a formula F with the property that

$$H \Rightarrow F \Rightarrow W.$$

However, if this is the only constraint on F, then why not take $F = H$? One obvious reason is that H may be infinite. Also, we want F to be as "close" as possible to W, in a certain sense. To define the concept of "closeness", we will define a relation "more general than" on first-order formulas and will require F to be "more general than" W. Relation "more general than" is defined in such a way that the number of formulas F that satisfy a given syntactic condition and are more general than a given formula W is finite up to variants; also, a formula F that is more general than W is structurally "similar" to W. Thus, we can only derive a finite number of formulas F satisfying both the following conditions:

(i) $H \Rightarrow F \Rightarrow W$
(ii) F is more general than W.

Of course, if H is more general than W, then we could have $F = H$. We will show that this method is complete, i.e. for any two formulas H and W, it is possible to derive F from H by our method such that (i) and (ii) above hold.

In the next section, we show why certain logical consequences of first-order formulas cannot be derived without using tautologies or unskolemization and describe an unskolemization algorithm. The algorithm is analyzed in Section 3.3, and some properties of formulas derived using this algorithm are given.

3.2. THE UNSKOLEMIZATION PROCESS

3.2.1. PRELIMINARIES

Unskolemization has been defined as the process of eliminating Skolem functions from a formula without quantifiers, replacing them with new existentially quantified variables, and transforming the resulting formula into a closed formula with quantifiers (for details about skolemization, see (Chang and Lee, 1973) or (Loveland, 1978)). McCune (McCune, 1988) presents an algorithm to solve the following problem: given a set S of clauses and a set F of constant and function symbols that occur in the clauses of S, obtain a fully quantified (closed) formula S' from S by replacing expressions starting with symbols in F with existentially quantified variables. S' is unsatisfiable if and only if S is unsatisfiable. McCune's algorithm is sound but not complete. Cox and Pietrzykowski (Cox and Pietrzykowski, 1984) present an algorithm for unskolemization, but their algorithm is applicable only to literals.

We expand the meaning of unskolemization slightly. In our definition, ordinary function applications can also be "unskolemized" by treating them as if they were Skolem functions. Thus, a function application may be replaced by an existentially quantified vari able during the unskolemization process. To illustrate, suppose we want to unskolemize the

formula $\forall x(P(f(x)) \vee Q(g(a), x))$ where f and a are (non-Skolem) function symbols, and suppose we want to treat f and a as if they were Skolem functions. The resulting formula would be $\exists z \forall x \exists y(P(y) \vee Q(g(z), x))$. Note that skolemizing $\exists z \forall x \exists y(P(y) \vee Q(g(z), x))$ yields the original formula (up to names of Skolem functions). In practice, the situation may be more complicated, since the formula being unskolemized may not be the skolemized form of any formula. Our algorithm shows how to cope with such situations. Also, unskolemization as presented here does not necessarily preserve unsatisfiability. For example, the formula $\exists x(succ(x) = 0)$ is false under the usual interpretation of "*succ*" as the successor function over natural numbers; however, if we unskolemize this function we get the formula $\exists y(y = 0)$ which is true. Henceforth, the term "function symbol" will denote function symbols and their applications.

We motivate the development of the unskolemization algorithm by the following example. Suppose we want to derive an unknown logical consequence B of A. Denote the Skolem form of a formula F by "Sk(F)". Since $A \Rightarrow B$, $A \wedge \neg B$ is unsatisfiable, so Sk($A \wedge \neg B$) is unsatisfiable (since skolemization preserves unsatisfiability), i.e. Sk(A)$\wedge$ Sk($\neg B$) is unsatisfiable. Therefore by the completeness of resolution, we can derive the empty clause from Sk(A)$\wedge$ Sk($\neg B$). Now, B is unknown, and we want to derive it from Sk(A). It may not be possible to derive B from Sk(A) without using tautologies or unskolemization, as is demonstrated by the following two examples:

(i) Suppose $A = P$ and $B = P \vee Q \vee R$. Clearly $A \Rightarrow B$. But the only way to derive B from A by resolution is by resolving A with the tautology $\neg P \vee P \vee Q \vee R$.

(ii) Suppose $A = P(a)$ and $B = \exists x P(x)$. Then B (or even Sk(B)) cannot be derived from A by resolution. Obtaining B from A requires unskolemizing A by replacing "a" by an existentially quantified variable. Unskolemizing $P(a)$ results in $\exists x P(x)$. In practice, there may be many function symbols in A, some of which may have to be replaced by existential quantifiers and some of which should not be thus replaced. This explains why our unskolemization algorithm will be nondeterministic.

In conclusion, B can be derived by resolution from Sk(A) (by the completeness of the resolution principle), but such a derivation can entail the use of tautologies and unskolemization. Using tautologies would increase the size of the search space tremendously, since there are an infinite number of tautologies; thus the use of tautologies is best avoided. Also, it is unclear how to handle unskolemization without a formal algorithm for doing so. This is best illustrated by an example: Suppose $A = \forall x \forall z P(x, f(a, x), z, g(z))$ and $B = \forall x \exists y \forall z \exists w P(x, y, z, w)$. Clearly $A \Rightarrow B$. Obtaining B from A requires replacing the terms $f(a, x)$ and $g(z)$ in A by existentially quantified variables, say x_1 and z_1. The question remains where to place the existential quantifiers $\exists x_1$ and $\exists z_1$ in the quantifier string for A. Since $f(a, x)$ was replaced by x_1, $\exists x_1$ should come after $\forall x$ (since a is a ground term, its presence as an argument of f is inconsequential); similarly, since $g(z)$ was replaced by z_1, $\exists z_1$ should come after $\forall z$. There are thus several choices for the unskolemized version of A, one of which is $\forall x \exists x_1 \forall z \exists z_1 P(x, x_1, z, z_1)$, which corresponds to B in this case.

In order to address the above issues formally, we present an unskolemization algorithm U with the following specifications:

INPUT: a first-order formula H

OUTPUT: set $\mathcal{L}$ of formulas such that for any logical consequence W of H, algorithm U can produce a formula $\mathbf{F}$ in $\mathcal{L}$ such that

(i) $\mathrm{H} \Rightarrow \mathbf{F} \Rightarrow \mathrm{W}$

(ii) F is more general than W

where relation "more general than" is defined later with the property that $\{F \mid F$ is more general than $W\}$ is finite up to variants under certain syntactic constraints.

The algorithm U unskolemizes a set of clauses $\mathcal{D}$ derived by resolution from Sk(H) to give a set of formulas $\mathcal{L}$. Briefly, the objective of unskolemizing $\mathcal{D}$ is to replace function symbols of $\mathcal{D}$ that do not occur in W by existentially quantified variables. That is, if for some literal L in $\mathcal{D}$, an argument d of L has a function symbol that does not appear in W, then that function symbol of d is unskolemized during the unskolemization of $\mathcal{D}$, yielding a set $\mathcal{L}$ of new formulas. Thus any $F \in \mathcal{L}$ will contain a new existentially quantified variable in place of d. Since W is unknown, this procedure will have to be carried out nondeterministically. This process will make the unskolemized formula "more general than" W (this term will be defined later).

NOTES. 1. The following algorithm makes use of the guarded command for conditional statements (Gries, 1981). Briefly, the general form of a conditional statement is

if $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
...
[] $B_n \rightarrow S_n$
fi

where $n \geq 0$ and each $B_i \rightarrow S_i$ is a guarded command. Each S_i can be any statement. The command is executed as follows. First, if any guard B_i is not well-defined in the state in which execution begins, abortion occurs. Second, if none of the guards is true, then abortion occurs; and finally, if at least one guard is true, then one guarded command $B_i \rightarrow S_i$ with true guard B_i is chosen and S_i executed. Note that if more than one guard is true, then one of the guarded commands $B_i \rightarrow S_i$ with true guard B_i is chosen arbitrarily and S_i is executed. Thus execution of such a statement can be nondeterministic. In steps 3 and 4 of the following algorithm, two of the guards are identical. This serves as a convenient way of representing nondeterminism: if the two identical guards are true in one of the steps, then one of the actions specified is performed and this action is picked arbitrarily from the two available actions.

2. The following notation is used:

(i) L = SIGN(L) $P(a_1, a_2, ..., a_n)$ is a literal whose sign (negated or unnegated) is represented by "SIGN(L)"; e.g. for L=$Q(a)$, SIGN(L) is the null string, and for L= $\neg Q(a)$, SIGN(L) is $\neg$.

(ii) Let X be a term. FUNC(X) is defined to be the function symbol of X if X is not a variable, and is defined to be X otherwise. For example, FUNC($f(x, y)$) = f; FUNC(a) = a; FUNC(x) = x, where x is a variable.

(iii) In what follows, we refer to first-order formulas simply as formulas, when this does not cause any ambiguity; also, we assume that all the quantifiers of a formula appear at the beginning of the formula. This is not restrictive because there are well-known procedures for converting a formula with embedded quantifiers into one with all quantifiers at the beginning (Chang and Lee, 1973).

3.2.2. THE UNSKOLEMIZATION ALGORITHM

ALGORITHM U

Step 1. Skolemize input formula H. Let SK be the set of all Skolem symbols in Sk(H). Derive a set $\mathcal{D}$ of clauses by resolution from Sk(H).

Comment : This step is nondeterministic; there could be more than one such set $\mathcal{D}$. Also, $\mathcal{D}$ can contain resolvents of Sk(H) as well as clauses from Sk(H).

Step 2. Make i_k copies of every clause C_k of $\mathcal{D}$, where i_k is some integer (chosen nondeterministically). Call the resulting bag of clauses M_CLAUSES.

Comment : We may need multiple copies of clauses because multiple instances of a clause may be needed to derive the empty clause from Sk($H \wedge \neg W$). It is possible to bound i_k by the number of resolutions performed when deriving the empty clause from Sk($H \wedge \neg W$). The reason for this is demonstrated in the proof of Theorem 3.1. In practice, for each k, we can try setting i_k to 1, then 2, then 3, and so on, and eventually i_k will become large enough.

Step 3. For every literal L in every clause of M_CLAUSES, process the arguments of L as follows. Suppose L=SIGN(L) $P(d_1, d_2, ..., d_s)$. For each i, $1 \le i \le s$, perform the following:

if FUNC(d_i) $\in$ SK $\rightarrow$
  replace d_i by X $\leftarrow d_i$, for some fresh variable X
[] FUNC(d_i) $\notin$ SK $\wedge$ d_i is not a variable $\rightarrow$
  replace d_i by X $\leftarrow d_i$, for some fresh variable X
[] FUNC(d_i) $\notin$ SK $\wedge$ d_i is not a variable $\rightarrow$ skip
[] d_i is a variable $\rightarrow$ skip
fi

Call the resulting set of processed clauses MARK.

Comment : If FUNC(d_i) $\notin$ SK and d_i is not a variable, then d_i is either replaced by X $\leftarrow d_i$ or left unchanged. This choice is made nondeterministically. The replacement of d_i by X $\leftarrow d_i$ is just a way of marking d_i with a variable name. This will be changed later. Any argument of the form "X $\leftarrow d_i$" is called a *marked* argument.

Step 4. For every pair of marked arguments "X $\leftarrow \alpha$", "Y $\leftarrow \beta$" in MARK do
if α, β are unifiable $\rightarrow$ unify all occurrences of X and Y
[] *true* $\rightarrow$ skip
fi

Comment: In the next step, C is the set of constraints on the ordering of new existential quantifiers relative to universal quantifiers that will be introduced in Step 6. The presence of an ordered pair (y, z) in C signifies that "$\exists z$" must come *after* "$\forall y$" in the quantifier string of the unskolemized formula.

Step 5. Let set C be initially empty, and let Q be an initially empty quantifier string. Let FREE be the set of all free variables in MARK (this does not include marked arguments). For every marked argument "$x \leftarrow \alpha$" do

{Collect all marked arguments with the same variable on the left-hand side of the "$\leftarrow$" sign. Suppose these are

$$x \leftarrow \alpha_1, x \leftarrow \alpha_2, ..., x \leftarrow \alpha_n.$$

Let $\{y_1, y_2, ..., y_r\}$ be the set of all variables occurring in $\alpha_1, \alpha_2, ..., \alpha_n$. Then

replace "$x \leftarrow \alpha_i$", for $1 \leq i \leq n$, everywhere by a new variable z (say) and add the r ordered pairs (y_i, z) to C. If $r = 0$, place "$\exists z$" at the head of the partially completed quantifier string Q.
}

Step 6. (i) For every y in FREE, define DEP(y) = $\{z|(y, z) \in \mathrm{C}\}$. This is the set of all variables z such that "$\exists z$" must come after "$\forall y$".

Define partial order PO on set FREE by:

$$(x, y) \in \mathrm{PO} \text{ iff } \mathrm{DEP}(x) \supseteq \mathrm{DEP}(y)$$

(ii) Extend partial order PO to a linear order on FREE in all possible ways, yielding a set LIN of linear orders.

(iii) Let QUANT be an initially empty set of quantifier strings. For every linear order O in LIN do

{P := Q;
add universal quantifiers for every variable in FREE to P in the order prescribed by O (i.e. if $x < y$ in O, then $\forall x$ precedes $\forall y$ in P). These quantifiers come *after* any existential quantifiers already present in Q;
QUANT := QUANT ∪ {P}
}

This yields a set QUANT of quantifier strings.

(iv) Let set $\mathcal{L}$ be initially empty. For every Q in QUANT do

{Insert an existential quantifier for every z such that $(y, z) \in \mathrm{C}$ (for some y) as far forward in Q as possible, subject to the constraint that "$\exists z$" comes after "$\forall y$" for every y such that $(y, z) \in \mathrm{C}$;
Rewrite MARK in conjunctive normal form by rewriting the set of clauses MARK as a conjunction of all its clauses and a clause as the disjunction of its literals;
Add the formula "Q MARK" to $\mathcal{L}$.
} □

EXAMPLE 1. Let
$H = \forall x \forall y \forall z \forall w \forall t((Q(y) \vee L(b, y, t)) \wedge \neg Q(g(t)) \wedge L(g(t), a, t) \wedge (R(x, g(t)) \vee \neg P(x, g(t))) \wedge (\neg R(w, z) \vee \neg D(w, z)))$,
$W = \forall s \exists u \forall v(L(b, u, s) \wedge L(u, a, s) \wedge (\neg P(v, u) \vee \neg D(v, u) \vee M(a)))$.

It is easy to see that $H \Rightarrow W$. We show how algorithm U derives a formula F from H such that $H \Rightarrow F \Rightarrow W$. We show later that F is more general than W. Let us reiterate that formula W is not normally available when performing the unskolemization. The choices made here based on properties of W are made nondeterministically by the algorithm. We are using W to show that there exist choices that will result in the derivation of a formula F with the desired properties. Now,

$\neg W = \exists s \forall u \exists v((\neg L(b, u, s) \vee \neg L(u, a, s) \vee P(v, u)) \wedge (\neg L(b, u, s) \vee \neg L(u, a, s) \vee D(v, u)) \wedge (\neg L(b, u, s) \vee \neg L(u, a, s) \vee \neg M(a)))$

$\mathrm{Sk}(\neg W) = \{\{\neg L(b, u, c), \neg L(u, a, c), P(f(u), u)\}, \{\neg L(b, u, c), \neg L(u, a, c), D(f(u), u)\}, \{\neg L(b, u, c), \neg L(u, a, c), \neg M(a)\}\}$.

In Sk($\neg W$), f and c are Skolem functions that replace the existentially quantified variables v and s of $\neg W$, respectively. The five clauses of Sk(H) and the three clauses of Sk($\neg W$) are listed below, in that order. Variables in Sk($\neg W$) have been renamed so that no two clauses in Sk($\neg W$) share the same variables.

1. $\{Q(y), L(b, y, t)\}$

2. $\{\neg Q(g(t))\}$
3. $\{L(g(t), a, t)\}$
4. $\{R(x, g(t)), \neg P(x, g(t))\}$
5. $\{\neg R(w, z), \neg D(w, z)\}$
6. $\{\neg L(b, u, c), \neg L(u, a, c), P(f(u), u)\}$
7. $\{\neg L(b, w, c), \neg L(w, a, c), D(f(w), w)\}$
8. $\{\neg L(b, z, c), \neg L(z, a, c), \neg M(a)\}$

A resolution proof of the empty clause from Sk(H)$\wedge$ Sk($\neg W$) is given below:

9. $\{L(b, g(t), t)\}$	from clauses 1 and 2
10. $\{\neg P(x, g(t)), \neg D(x, g(t))\}$	from clauses 4 and 5
11. $\{\neg L(g(c), a, c), D(f(g(c)), g(c))\}$	from clauses 9 and 7
12. $\{\neg L(g(c), a, c), P(f(g(c)), g(c))\}$	from clauses 9 and 6
13. $\{D(f(g(c)), g(c))\}$	from clauses 3 and 11
14. $\{P(f(g(c)), g(c))\}$	from clauses 3 and 12
15. $\{\neg P(f(g(c)), g(c))\}$	from clauses 10 and 13
16. $\{\}$	from clauses 14 and 15.

This sequence of resolutions is depicted pictorially in Figure 1. The clauses in the figure are numbered as above. Resolutions among clauses of Sk(H) were performed first and then some of these clauses were used during the remainder of the resolution process. We define set $\mathcal{D}\subset$Sk(H) to consist of the three clauses $\{\{L(b, g(t), t)\}, \{L(g(t), a, t)\}, \{\neg P(x, g(t)), \neg D(x, g(t))\}\}$, which were obtained from Sk(H) and were used to derive the empty clause from Sk(H) $\wedge$ Sk($\neg W$). These clauses are enclosed in boxes in Figure 1. Note that $L(g(t), a, t)$ belongs to Sk(H), and the other two clauses were derived by resolution from Sk(H).

We now perform algorithm U.

INPUT : Formula H given above.

Step 1: We define set $\mathcal{D}$ to consist of the three clauses below, as explained above.

1. $\{L(b, g(t), t)\}$
2. $\{L(g(t), a, t)\}$
3. $\{\neg P(x, g(t)), \neg D(x, g(t))\}$.

Step 2: Since the clauses $\{L(b, g(t), t)\}$ and $\{L(g(t), a, t)\}$ are both used twice during the resolution in Figure 1, we make two copies each of these clauses, and we make one copy of the clause $\{\neg P(x, g(t)), \neg D(x, g(t))\}$. Thus, bag M_CLAUSES consists of the following five clauses:

1. $\{L(b, g(t), t)\}$
2. $\{L(b, g(t), t)\}$
3. $\{L(g(t), a, t)\}$
4. $\{L(g(t), a, t)\}$
5. $\{\neg P(x, g(t)), \neg D(x, g(t))\}$.

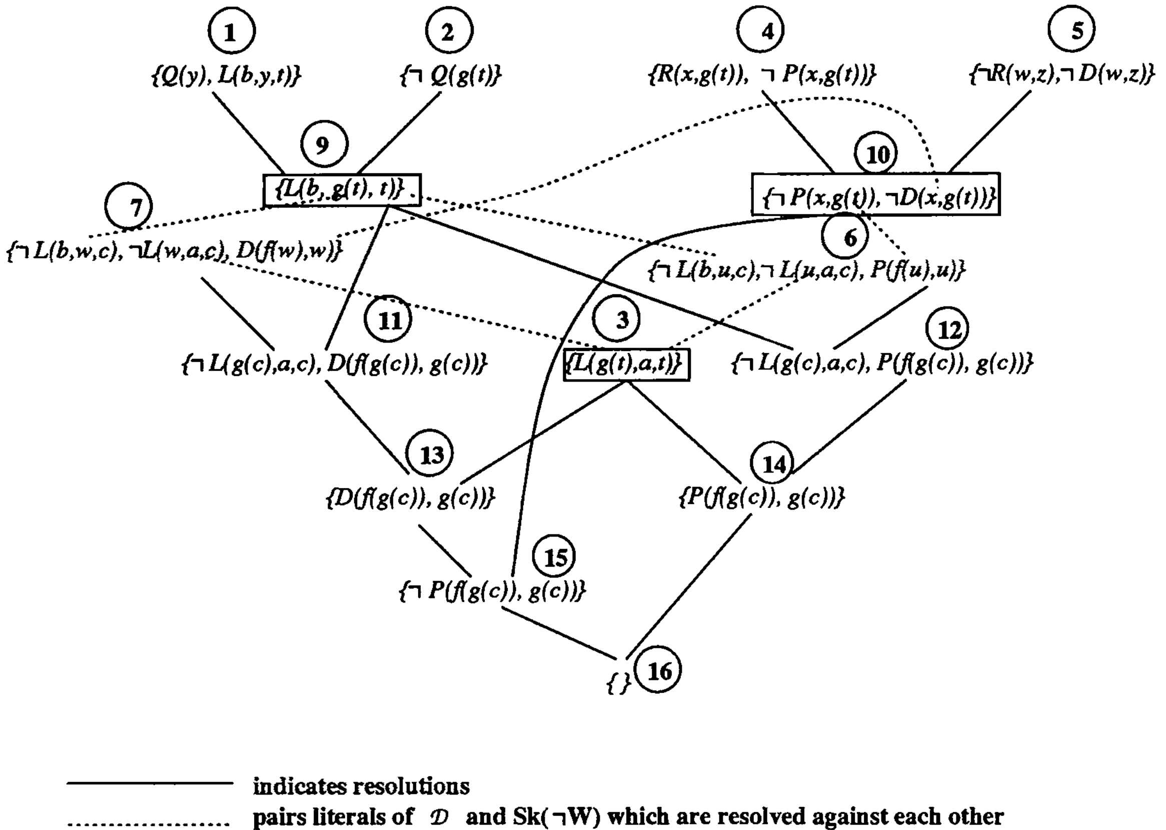

Figure 1. Derivation of the empty clause from Sk(H) and Sk($\neg W$) by resolution for Example 1

Step 3: Now "mark" arguments of elements of M_CLAUSES as follows. Look at which literal of Sk($\neg W$) each of the above literals resolves against in Figure 1. Pairs of literals of $\mathcal{D}$ and Sk($\neg W$) that resolve against each other are linked by dotted lines in the figure. We see that

$L(b, g(t), t)$ resolves against $\neg L(b, w, c)$
$L(b, g(t), t)$ resolves against $\neg L(b, u, c)$
$L(g(t), a, t)$ resolves against $\neg L(w, a, c)$
$L(g(t), a, t)$ resolves against $\neg L(u, a, c)$
$\neg P(x, g(t))$ resolves against $P(f(u), u)$
$\neg D(x, g(t))$ resolves against $D(f(w), w)$.

Note: By looking at (for instance) the resolution between clauses 3 and 11, which yields clause 13, it appears that $L(g(t), a, t)$ (from clause 3) resolves against $\neg L(g(c), a, c)$ (from clause 11). However, the literal $\neg L(g(c), a, c)$ in clause 11 is an instance of the literal $\neg L(w, a, c)$ in clause 7. Clause 7 belongs to Sk($\neg W$) (and clause 11 doesn't). Thus the literal in Sk($\neg W$) that $L(g(t), a, t)$ resolves against is $\neg L(w, a, c)$ in this case.

For any function symbol F in a literal of M_CLAUSES that resolves against a variable X in Sk($\neg W$), mark it by replacing F by "X ← F". This yields the following set of marked clauses MARK:

1'. $\{L(b, w \leftarrow g(t), t)\}$
2'. $\{L(b, u \leftarrow g(t), t)\}$
3'. $\{L(w \leftarrow g(t), a, t)\}$
4'. $\{L(u \leftarrow g(t), a, t)\}$
5'. $\{\neg P(x, u \leftarrow g(t)), \neg D(x, w \leftarrow g(t))\}$.

Step 4: Unify some of the variables on the left hand side of the "$\leftarrow$" in marked arguments. To decide which variables will be unified, we look at variables in unmarked arguments of MARK. There are two such variables: x and t. These variables were unified with $\{f(u), f(w)\}$ and $\{c\}$ respectively (see the analysis in the previous step). Since x was unified with both $f(u)$ and $f(w)$, we unify $f(u)$ with $f(w)$; thus u and w get unified. MARK now consists of:

1'. $\{L(b, u \leftarrow g(t), t)\}$
2'. $\{L(b, u \leftarrow g(t), t)\}$
3'. $\{L(u \leftarrow g(t), a, t)\}$
4'. $\{L(u \leftarrow g(t), a, t)\}$
5'. $\{\neg P(x, u \leftarrow g(t)), \neg D(x, u \leftarrow g(t))\}$.

Since MARK is actually a set, we can eliminate two duplicates; MARK consists of:

1'. $\{L(b, u \leftarrow g(t), t)\}$
3'. $\{L(u \leftarrow g(t), a, t)\}$
5'. $\{\neg P(x, u \leftarrow g(t)), \neg D(x, u \leftarrow g(t))\}$.

Step 5: Here FREE $= \{t, x\}$ The marked arguments in MARK are "$u \leftarrow g(t)$". We replace these arguments by a fresh variable Z and add the pair (t, Z) to **C**. This yields $\mathbf{C} = \{(t, Z)\}$ and MARK consists of the clauses

1'. $\{L(b, Z, t)\}$
3'. $\{L(Z, a, t)\}$
5'. $\{\neg P(x, Z), \neg D(x, Z)\}$.

Step 6: (i) Here DEP$(t) = \{Z\}$, DEP$(x) = \{\ \}$. Since DEP$(t) \supseteq$ DEP(x), PO $= \{(t, x)\}$.
(ii) Partial order PO is a linear order on FREE, so LIN $=$ {PO}.
(iii) QUANT $= \{\forall t \forall x\}$
(iv) The existential quantifier for Z must be placed after $\forall t$, as far forward as possible; thus

$$\text{QUANT} = \{\forall t \exists Z \forall x\}$$

and the resulting set of formulas is

$$\mathcal{L} = \{\forall t \exists Z \forall x (L(b, Z, t) \wedge L(Z, a, t) \wedge (\neg P(x, Z) \vee \neg D(x, Z)))\}.$$

Thus $\mathcal{L}$ contains only one formula for this example; call it **F**. It can easily be verified that H $\Rightarrow$ **F** $\Rightarrow$ W. □

3.3. ANALYSIS OF UNSKOLEMIZATION ALGORITHM U

This section defines relation "more general than" and lists some theorems that prove that algorithm U satisfies its specification. Proofs for the theorems in this section can be found in the appendix.

THEOREM 3.1. *Let H, W be formulas such that $H \Rightarrow W$. If suitable non-deterministic choices are made, given H as input, algorithm U produces a set $\mathcal{L}$ of formulas such that for any formula F in $\mathcal{L}$, for any literal $L = SIGN(L)\ P(d_1, d_2, ..., d_s)$ of $Sk(F)$, there exists a literal M of W such that $M = SIGN(M)\ P(b_1, b_2, ..., b_s)$, $SIGN(M) = SIGN(L)$, and such that for all $i, 1 \leq i \leq s$,*

(i) If d_i is a Skolem function, then b_i is existentially quantified in W.

(ii) If d_i is a non-Skolem function, then one of the following holds:

(a) b_i is the same function symbol with the same number of arguments, and (i) and (ii) here hold recursively for each corresponding argument of d_i and b_i.

(b) b_i is existentially quantified and the function symbol of d_i (with the same arity as d_i) appears in W. □

Intuitively, we are trying to say that Skolem symbols in $\mathcal{D}$ (where $\mathcal{D}$ is as specified in Step 1 of algorithm U) are replaced by existentially quantified variables in W ((i) in the theorem statement), and non-Skolem function symbols in $\mathcal{D}$ *which do not appear* in W are also replaced by existentially quantified variables. Thus any non-Skolem function symbol that remains in an unskolemized formula F must appear somewhere in W ((ii) in the theorem statement). This is crucial because it allows us to define a relation "more general than" (based on the theorem statement) such that the number of formulas more general than a given formula is finite under certain elementary syntactic constraints. Also, the fact that every literal L in Sk(F) has a corresponding literal M in W with the same predicate, arity, and sign shows that formula F is similar to W in the predicates that it contains. The following example illustrates the theorem.

EXAMPLE 2. In Example 1, we had
$H = \forall x \forall y \forall z \forall w \forall t((Q(y) \vee L(b,y,t)) \wedge \neg Q(g(t)) \wedge L(g(t),a,t) \wedge (R(x,g(t)) \vee \neg P(x,g(t))) \wedge (\neg R(w,z) \vee \neg D(w,z))$,
$W = \forall s \exists u \forall v(L(b,u,s) \wedge L(u,a,s) \wedge (\neg P(v,u) \vee \neg D(v,u) \vee M(a)))$
and algorithm U yielded

$F = \forall t \exists Z \forall x(L(b,Z,t) \wedge L(Z,a,t) \wedge (\neg P(x,Z) \vee \neg D(x,Z)))\}$.

Here $\text{Sk}(F) = \{\{L(b,f(t),t)\}, \{L(f(t),a,t)\}, \{\neg P(x,f(t)), \neg D(x,f(t))\}\}$,
where f is a Skolem function replacing the existentially quantified variable Z. For every literal in Sk(F), we have a corresponding literal in W such that the conditions (i) and (ii) of Theorem 3.1 hold. These correspondences are:

Literal in Sk(F)	Corresponding literal in W
1. $L(b,f(t),t)$	$L(b,u,s)$
2. $L(f(t),a,t)$	$L(u,a,s)$
3. $\neg P(x,f(t))$	$\neg P(v,u)$
4. $\neg D(x,f(t))$	$\neg D(v,u)$

For the first pair of literals, (ii) (a) holds for the first pair of arguments ("b" and "b"); (i) holds for the second pair of arguments ("$f(t)$" and "u"); neither (i) nor (ii) applies to the third pair of arguments ("t" and "s").

For the second pair of literals, (i) holds for the first pair of arguments ("$f(t)$" and "u"); (ii) (a) holds for the second pair of arguments ("a" and "a"); neither (i) nor (ii) applies to the third pair of arguments ("t" and "s").

For the third pair of literals, neither (i) nor (ii) applies to the first pair of arguments ("x" and "v"); (i) holds for the second pair of arguments ("$f(t)$" and "u").

For the fourth pair of literals, neither (i) nor (ii) applies to the first pair of arguments ("x" and "v"); (i) holds for the second pair of arguments ("$f(t)$" and "u").

Note that the literal $M(a)$ in W has no corresponding literal in Sk(F). □

Motivated by Theorem 3.1, we introduce the following definition.

DEFINITION. A formula F is *more general than* a formula W if for every literal L of F, there exists a literal M of W such that if L = SIGN(L)$P(a_1, a_2, ..., a_s)$, then M = SIGN(M)$P(b_1, b_2, ..., b_s)$, where SIGN(L) = SIGN(M), and for all i such that $1 \leq i \leq s$,

(i) If a_i is an existentially quantified variable, then so is b_i.

(ii) If a_i is a function symbol followed by u arguments $e_1, e_2, ..., e_u$, then either

(a) b_i is the same function symbol followed by the same number of arguments, say $f_1, f_2, ..., f_u$, and conditions (i) and (ii) hold for every pair of arguments e_k and $f_k, 1 \leq k \leq u$, or

(b) b_i is an existentially quantified variable and a_i has a function symbol that occurs in W. □

Note the similarity between the statement of Theorem 3.1 and the above definition. As we shall see in the next theorem, the number of formulas more general than a given formula is finite under certain elementary syntactic constraints. The reason we want this to be true is the following. Recall that the problem being solved is that we are given a formula H that implies some (unknown) formula W, and we are trying to derive a logical consequence F of H such that

$$H \Rightarrow F \Rightarrow W.$$

Now, some additional constraint must be placed on F, since otherwise we could simply take $F = H$, if H is finite. We want F to be "close" to W, in some sense. One way to ensure this is to define some constraint on F so that only a finite number of formulas satisfy this constraint, and so that these formulas are "similar" to W in some sense. Thus only a finite number of formulas F satisfying this constraint and such that $H \Rightarrow F \Rightarrow W$ can be derived. This finiteness will insure the termination of our algorithm for deriving loop invariants, as will be seen in the proof of its completeness. This is the reason for defining the "more general then" term above. Note that the definition of "more general than" given above does not allow function symbols that do not appear in W to appear in F if F is more general than W.

COROLLARY TO THEOREM 3.1 For every $F \in \mathcal{L}$, F is more general than W.

PROOF. Follows directly from Theorem 3.1. □

DEFINITION. Let F, W be two first-order formulas. We say that $F \preceq W$ if and only if

(i) F is more general than W
(ii) $F \Rightarrow W$. □

EXAMPLE 3. The following examples illustrate the meaning of the relation "more general than".

(i) $F = \forall x \exists y \forall z (P(x,y) \wedge Q(y,z))$
$W = \forall u \exists v P(u,v)$
F is not more general than W because for the literal $Q(y,z)$ in F, there is no literal in W with the specified properties, since W does not even have a literal with predicate symbol Q.

(ii) $F = \forall x \exists y \forall z (P(x,y) \wedge Q(y,z))$
$W = \forall u \exists v (P(u,v) \wedge Q(v,v))$
Here F is more general than W, because for $P(x,y)$ in F, there is a corresponding literal $P(u,v)$ in W with the specified properties; similarly, for $Q(y,z)$ in F, there is a corresponding literal $Q(v,v)$ in W with the specified properties.
Also, $F \Rightarrow W$; therefore $F \preceq W$.

(iii) $F = \forall x \exists y \forall z (P(x,y) \vee Q(y,z))$
$W = \forall u \exists v (P(u,v) \wedge Q(v,v)$
As in (ii) above, F is more general than W. However, $F \not\Rightarrow W$; therefore $F \not\preceq W$.

(iv) In Example 1, we had
$F = \forall t \exists Z \forall x (L(b,Z,t) \wedge L(Z,a,t) \wedge (\neg P(x,Z) \vee \neg D(x,Z)))$,
$W = \forall s \exists u \forall v (L(b,u,s) \wedge L(u,a,s) \wedge (\neg P(v,u) \vee \neg D(v,u) \vee M(a)))$.
In Example 2, we saw the correspondences between literals of Sk(F) and W. The correspondences are the same for literals of F and W. Thus F is more general than W. Also, $F \Rightarrow W$; therefore $F \preceq W$. □

THEOREM 3.2. *$\{F \mid F \preceq W\}$ is finite up to variants, assuming that if F is written in conjunctive normal form, then no two disjunctions of F are identical, and no disjunction of F contains more than one occurrence of the same literal.*

PROOF. We show that the set $\{F \mid F$ is more general than $W\}$ is finite up to variants subject to the above condition. Suppose a formula W is given, and suppose F is a formula that is more general than W. By definition, for every literal L = SIGN(L) $P(a_1, a_2, ..., a_s)$ of F, there exists a literal M = SIGN(M) $P(b_1, b_2, ..., b_s)$ of W such that SIGN(L) = SIGN(M) and such that conditions (i), (ii)(a) and (ii)(b) in the definition of "more general than" hold. From these conditions, the following statements are true for every k, $1 \leq k \leq s$:

(i) If b_k is an existentially quantified variable, then a_k is either an existentially quantified variable, a universally quantified variable, or a function symbol that occurs in W.
(ii) If b_k is a universally quantified variable, then so is a_k.
(iii) If b_k is a function symbol with u arguments $e_1, e_2, ..., e_u$, then a_k is either:
(a) the same function symbol with the same number of arguments, say $f_1, f_2, ..., f_u$, and conditions (i), (ii) and (iii) hold for every pair of arguments e_i and $f_i, 1 \leq i \leq u$, or
(b) a_k is a universally quantified variable.

From the above analysis, it can be seen that only a finite number of distinct literals L (up to variants) can be constructed that satisfy these conditions. But then there exist only

a finite number of formulas F made up of conjunctions of disjunctions of such literals, provided no two such disjunctions are identical, and no disjunction of F contains more than one occurrence of the same literal.

Hence, the number of formulas that are more general than W is finite up to variants, subject to the conditions in the statement of the theorem; this means that $\{F \mid F \preceq W\}$ is also finite up to variants subject to the same conditions. □

THEOREM 3.3. *For every $F \in \mathcal{L}$, $H \Rightarrow F$, where $\mathcal{L}$ is the set of formulas obtained by unskolemizing a formula H according to algorithm U.* □

THEOREM 3.4. *Given formulas H, W such that $H \Rightarrow W$, there exists $\mathbf{F} \in \mathcal{L}$ such that $\mathbf{F} \Rightarrow W$, where H and $\mathcal{L}$ are the input and output of algorithm U respectively.* □

COROLLARY TO THEOREM 3.4 There exists $\mathbf{F} \in \mathcal{L}$ such that $\mathbf{F} \preceq \mathrm{W}$.

PROOF. From the Corollary to Theorem 3.1, Theorem 3.4, and the definition of $\preceq$. □

THEOREM 3.5. *If F_1, F_2 and W are three formulas such that*

$$F_1 \preceq W, F_2 \preceq W,$$

then

$$(F_1 \wedge F_2) \preceq W, \ (F_1 \vee F_2) \preceq W. \quad \square$$

4. Overview of the method

In this section we give an overview of an iteration method to derive loop invariants. Perform the following steps for a given program:

1. Draw a flowchart for the program, cut the loops, and attach loop invariants (these are unknown) and input and output assertions where appropriate. We are assuming that loop invariants exist for all loops; if they do not, this method is not applicable. A symbol representing an unknown loop invariant is attached at every loop cutpoint.
2. Generate verification conditions for the program as explained in (Manna, 1974).
3. Apply the iteration method to the verification conditions to obtain the loop invariants.

Step 3 needs to be described in detail. We give below a brief overview of our method. The detailed algorithm is given in Section 8.

A "known" formula is one that does not contain any loop invariant. In the following, W, W_1 and W_2 denote loop invariants and H, H_1 and H_2 denote known formulas. Any verification condition involving a loop invariant is of one of the following three forms:

(i) $H \Rightarrow W$
(ii) $H \wedge W_1 \Rightarrow W_2$
(iii) $H_1 \wedge W \Rightarrow H_2$.

To see that this is true, recall that there is one cutpoint for every loop in the program, one cutpoint at the entry of the program, and one cutpoint at every exit of the program. Therefore, a path in the program could be of one of the following four types:

(1) A path from the entry cutpoint to a loop cutpoint
(2) A path from a loop cutpoint to a loop cutpoint
(3) A path from a loop cutpoint to an exit cutpoint
(4) A path from the entry cutpoint to an exit cutpoint

Of these four types, a verification condition for a path of type (4) does not involve any loop invariants and will therefore not be considered here. A verification condition for a path of type (1) will be of the form $H \Rightarrow W$ (where H is a known formula representing the conditions that hold at the beginning of the path and during the path traversal, and W is the loop invariant at the cutpoint at the end of the path); a verification condition for a path of type (2) will be of the form $H \wedge W_1 \Rightarrow W_2$ (where H is a known formula representing the conditions that hold during the path traversal and W_1, W_2 are the loop invariants of the cutpoints at the beginning and end of the path respectively); and a verification condition for a path of type (3) will be of the form $H_1 \wedge W \Rightarrow H_2$ (where H_1, H_2 are known formulas representing the conditions that hold during the path traversal and the output condition that holds at the end of the path respectively, and W is the loop invariant of the cutpoint at the beginning of the path).

We will obtain successively more accurate approximations to the loop invariants. For this purpose, we will define function GET-APPROX in Section 9 to be a binary function that takes as arguments a formula H and a symbol W and returns a formula F that is an approximation for W, where $H \Rightarrow F$ and $F \Rightarrow W$. Note that H must be a known formula; W is the name of an unknown loop invariant.

Initially, only the input and output assertions are given. We initially approximate all the unknown loop invariants by setting them to $false$. We will represent approximation i to W by W_i; the initial approximation to W is W_0. Informally, the method we will use is the following : suppose W is some (unknown) loop invariant in the program. Consider all the verification conditions in which W appears on the right-hand side of the implication sign. We replace all occurrences of loop invariants in these verification conditions with their current approximations. Suppose that the last approximation calculated for W was W_i. Suppose the resulting verification conditions are $H_1 \Rightarrow W_i$, $H_2 \Rightarrow W_i$, $H_3 \Rightarrow W_i$, ..., $H_n \Rightarrow W_i$ (where i gives the number of the current iteration). Note that loop invariants may occur in the formulas $H_1, H_2, ..., H_n$ above; all such occurrences are replaced by the current approximations for these loop invariants. For all the H_j's, $1 \leq j \leq n$, check whether $H_j \Rightarrow W_i$ is true or not. Let *Tconditions* be the set of all H_j's such that $H_j \Rightarrow W_i$ is not true. If *Tconditions* is empty, then set $W_{i+1} = W_i$; if *Tconditions* is not empty, then set $W_{i+1} =$ GET-APPROX$(W_i \vee R, W)$, where R $= \bigvee\{H_j | H_j \in Tconditions\}$. Note that for all j such that $1 \leq j \leq n$, $H_j \Rightarrow W_{i+1}$ and $W_i \Rightarrow W_{i+1}$. This is because the new formula W_{i+1} generated by the function GET-APPROX is a logical consequence of the disjunction of W_i and all the formulas in the set *Tconditions*; thus it is a logical consequence of each of these formulas.

We then look for another loop invariant and find the next approximation to it exactly as described for W_i (this time using W_{i+1} as an approximation for W), and so on. Recall that all the verification conditions in which W appears on the right-hand side of the implication sign are $H_1 \Rightarrow W$, $H_2 \Rightarrow W$, $H_3 \Rightarrow W$, ..., $H_n \Rightarrow W$. If we have $W_{i+1} = W_i$ (this happens when the set *Tconditions* is empty), then since $H_j \Rightarrow W_{i+1}$ for all j such that $1 \leq j \leq n$, and since $W_{i+1} = W_i$, we have $H_j \Rightarrow W_i$ for all j such that $1 \leq j \leq n$. When this happens for all the loop invariants, the procedure terminates.

5. Some observations about the programming language model

A program is partially correct if all the verification conditions derived from the program after assigning appropriate loop invariants are valid in a model $\mathcal{M}$ of the data structures and primitive operations of the language. For instance, most programming languages

contain the arithmetic operators + and −, and a model $\mathcal{M}$ of such a programming language would reflect the semantics of these operations. In other words, we would like to prove $\mathcal{M} \models vc$ for all the verification conditions vc of the program being verified.

Now the question arises whether there exists an axiomatization of model $\mathcal{M}$. (A theory T is said to be *axiomatizable* if there exists a decidable set W ⊂ T such that T is exactly the set of all formulas derivable from W in the predicate calculus.) Peano arithmetic (addition and multiplication along with the predicate "<" over the natural numbers) is not axiomatizable; however, the well-known Peano axioms along with the principle of induction over the natural numbers characterize all properties of the natural numbers, including those of the Peano arithmetic, i.e. those that may be expressed as formulas of first-order logic. This is not in contradiction to the fact that Peano arithmetic is not axiomatizable, because the principle of induction cannot be expressed in first-order logic, since it involves quantification over predicates.

Assuming that there exists an axiomatization A of a model $\mathcal{M}$ (A is a set of axioms such that A is decidable and such that the set of all formulas true under $\mathcal{M}$ is exactly the set of all formulas that are derivable from A in the predicate calculus), the verification problem reduces to a proof of the form $A \vdash vc$ for all verification conditions vc of the program being verified. Now, any verification condition is of the form $L \Rightarrow M$ (see Section 4), for first-order formulas L and M. Therefore the above can be written as $A \vdash (L \Rightarrow M)$, which is equivalent to $\vdash ((A \wedge L) \Rightarrow M)$.

Henceforth, we assume that the models of the programming languages under consideration are axiomatizable and that an axiomatization A of the language is provided when the verification conditions are being proved; in other words, the formulas in A are taken to be axioms and can be used for any proof. This may seem like a restrictive assumption, since we know that even Peano arithmetic is not axiomatizable; however, in many cases, we circumvent this problem by providing suitable instances of the principle of induction as required, or by providing the system with enough facts to be able to derive the desired formulas from these facts. For example, it is necessary to use induction to deduce from the Peano axioms that addition is commutative; we circumvent this problem by adding commutativity of addition as an axiom to the system. The reader should nevertheless be aware of this restriction on the power of our system.

6. A clarification

Subsequently, we sometimes need to know whether a formula of the form $A \Rightarrow B$ is valid. However, since first-order logic is semi-decidable, we are only guaranteed to get an answer to this question if $A \Rightarrow B$ is valid. If $A \Rightarrow B$ is not valid, the procedure for determining the validity of $A \Rightarrow B$ may or may not halt. Some theorem provers can show, for some sets of clauses, that a given set of clauses is not unsatisfiable by providing a model for the set of clauses, that is, a truth assignment that makes all the clauses in the set true (see for example (Lee, 1990)).

However, the above facts do not invalidate the completeness of our iteration algorithm. In any place in the algorithm where the validity of $A \Rightarrow B$ has to be proved or disproved, we can try to prove that $A \Rightarrow B$ is valid; if this attempt fails after some finite amount of time t, the attempt can be abandoned and $A \Rightarrow B$ can be assumed to be invalid. We can do this because even if $A \Rightarrow B$ is assumed to be invalid when it is actually valid, our algorithm still remains sound; it will just run longer than necessary. If no loop invariant is derived using the current value of t, t can be incremented. At some point, t will become

large enough for all valid formulas A $\Rightarrow$ B to be proved within time t, since there are only a finite number of such formulas that need to be proved. Another strategy for dealing with the above problem could be to run the procedure for finding loop invariants in parallel for different values of t. Henceforth, we will assume that such a strategy is used to deal with the above situation.

7. Description of algorithm for generating loop invariants

The notation described in Section 4 is used throughout this algorithm. We first briefly describe the algorithm. Assume that W^1, W^2, ..., W^n are the loop invariants of the program. As mentioned previously, let the initial approximations of all verification conditions be *false* and denote the initial approximation of each W^i by W^i_0. Approximation j for W^i is denoted by W^i_j. If the last approximation that has been calculated for a loop invariant W^i is approximation k, then $index(W^i)$ is set to k, i.e. $index(W^i_k) = k$. Initially, $index(W^i) = 0$ for every i, $1 \leq i \leq n$. These initializations are performed in step 1.

Step 2 constructs a list *list_of_loop_invariants* which contains the names of all the loop invariants (i.e. W^1, W^2, ..., W^n) in a particular order (this order is immaterial). The list built in step 2 is used in step 3 to provide the order in which the iteration will proceed. Starting with the first loop invariant in this list, and repeating the same process for each element of the list in order, the following is done. Initialize set *Tconditions* to the empty set. Suppose that the first element in *list_of_loop_invariants* is loop invariant W. For every verification condition $J \Rightarrow W$ (say) that has W on the right-hand side of its implication sign, the following is done. J could either be of the form $J = H$, for some known formula H, or of the form $J = H \wedge W^j$, for some loop invariant W^j. Note that W^j could equal W. If J is of the latter form, then W^j is replaced by $W^j_{index(W^j)}$ in J. Call the transformed formula J'. If $J' \Rightarrow W_{index(W)}$ is false, J' is added to set *Tconditions*. This process is repeated for all verification conditions. The next approximation for W is obtained as follows. First, $index(W)$ is incremented by 1. If *Tconditions* is empty, then the current approximation for W is retained and $flag(W)$ is set to *true* to mark this fact. If *Tconditions* is not empty, then the next approximation $W_{index(W)}$ for W is obtained by calling function GET-APPROX to return a formula $W_{index(W)}$ such that

$$W_{index(W)-1} \vee R \Rightarrow W_{index(W)},$$

where R is the disjunction of all the elements of *Tconditions*. The fact that *Tconditions* was non-empty is marked by setting $flag(W)$ to *false*.

This whole process is repeated until all the flags for all the loop invariants are *true* at the same time. This indicates that the current approximations for the loop invariants satisfy all the verification conditions and can therefore be used as loop invariants. Step 4 sets W^i_{approx} to be the last approximation obtained for W^i, which is a loop invariant for every i such that $1 \leq i \leq n$.

The algorithm is given below in Pascal-like pseudo-code.

8. The iteration algorithm

{COMMENT : Let V be the set of verification conditions for the program; suppose $V = \{vc_1, vc_2, ..., vc_m\}$. Let all the (unknown) loop invariants be $W^1, W^2, ..., W^n$. We denote the formula on the left-hand side of the implication sign in a verification condition vc by

lhs(vc), and similarly we denote the formula on the right-hand side of the implication sign in a verification condition vc by $rhs(vc)$. Also, to make the algorithm easier to read, the number n of verification conditions is written as *number_of_loop_invariants* and the number m of verification conditions is written as *number_of_verification_conditions*.}

1. For $i := 1$ to *number_of_loop_invariants* do
    $\{index(W^i) := 0;$
    $W_0^i := false$
    }
2. *list_of_loop_invariants* := empty;
which_loop_invariant := 1;
for $i := 1$ to *number_of_verification_conditions* do
    if (vc_i is of the form $H \Rightarrow W$) and
      ($W \notin$ *list_of_loop_invariants*) then
        append(*list_of_loop_invariants*, W);
while length(*list_of_loop_invariants*) < *number_of_loop_invariants* do
    {*current_loop_invariant* := *which_loop_invariant*th element in
    list_of_loop_invariants;
    for every vc in V such that
        ($lhs(vc)$ contains *current_loop_invariant*) and
        ($rhs(vc) = W^i$ for some i) and
        ($W_i \notin$ *list_of_loop_invariants*) do
            append(*list_of_loop_invariants*, W^i);
    which_loop_invariant := *which_loop_invariant* + 1
    }
3. repeat
    for $i := 1$ to *number_of_loop_invariants* do
        $\{W := i^{th}$ element in *list_of_loop_invariants*;
        Tconditions := empty;
        for $j := 1$ to *number_of_verification_conditions* do
            {if $rhs(vc_j) = W$ then
                if $\neg\ (lhs(vc_j) \Rightarrow W_{index(W)})$ then
                    $Tconditions := Tconditions \cup \{lhs(vc_j)\}$
            };
        $index(W) := index(W) + 1;$
        if $Tconditions \neq$ empty then
            $\{flag(W) := false;$
            $W_{index(W)} :=$ GET-APPROX$(W_{index(W)-1} \vee$
            $(\bigvee_j \{H | H \in Tconditions\}), W)$ (see Note after algorithm)
            }
        else
            $\{flag(W) := true;$
            $W_{index(W)} := W_{index(W)-1}$
            }
        }
until $\bigwedge_{i=1}^{n} flag(W^i)$;
4. for $i := 1$ to *number_of_loop_invariants* do

$$W^i_{approx} := W^i_{index(W)}.$$

NOTE. Each time GET-APPROX is called, any occurrence of an unknown loop invariant W^j in the first argument of GET-APPROX is replaced by its current approximation, which is $W^j_{index(W^j)}$.

9. Function GET-APPROX

Function GET-APPROX takes two arguments H and W, where H is a known formula and W is the name of an unknown loop invariant for which GET-APPROX will return an approximation S (say). W is a logical consequence of H, and an approximation S for W can be derived from H by resolution and unskolemization. We show that this process can be guided by the verification conditions of the program and is therefore much more efficient than a pure generate-and-test approach. The approach is explained below and illustrated in Example 4.

The derivation of S can be made more efficient by noting that the problem at hand is simpler than just deriving logical consequences of one formula. H is the following disjunction

$$H = W_i \vee H_1 \vee H_2 \vee ... \vee H_k, \qquad (1)$$

where W_i is the previous approximation obtained for W and each H_j $(1 \leq j \leq k)$ is the left-hand side of a verification condition for which the right-hand side is W, and which is not valid with the current approximations for loop invariants (see the iteration algorithm). Note that k could be zero here. Our goal is to generate a formula S that is a logical consequence of H, i.e. such that

$$W_i \vee H_1 \vee H_2 \vee ... \vee H_k \Rightarrow S.$$

First approach

The above implication is equivalent to the $k+1$ implications $W_i \Rightarrow S, H_1 \Rightarrow S, ..., H_k \Rightarrow S$. We therefore need to generate a formula S which is a logical consequence of each of $W_i, H_1, ..., H_k$. We can generate a logical consequence for each of $W_i, H_1, ..., H_k$ by the resolution and unskolemization method described in Section 3. Suppose a logical consequence F of one of these formulas has been thus generated. We then check if F is implied by all of the formulas $W_i, H_1, ..., H_k$ (it is obviously implied by at least one of them, since F is a logical consequence of one of these formulas). Such a formula F is obtained for each of the formulas $W_i, H_1, ..., H_k$. We then collect together the F's that are implied by all of the formulas $W_i, H_1, ..., H_k$ (i.e. $W_i \Rightarrow F, H_1 \Rightarrow F, H_2 \Rightarrow F, ..., H_k \Rightarrow F$) and let S be their conjunction. S is then returned by GET-APPROX as approximation $i+1$ for W. Clearly, $H \Rightarrow S$, since each F in the conjunction S was implied by all of the formulas $W_i, H_1, ..., H_k$.

If after a number "b" of trials, we are not able to obtain any F that is derived from one of $W_i, H_1, ..., H_k$ and is implied by all of them, we take S to be the disjunction of all the $k+1$ formulas each of which was a logical consequence of one of $W_i, H_1, ..., H_k$. Here too, $H \Rightarrow S$. In the algorithm, "b" is a bound input by the user.

Second approach

The approach is slightly different when an approximation is being generated for a loop invariant W for which there exists at least one verification condition of the form $H' \wedge W \Rightarrow H''$, where H' and H'' are known formulas (i.e. W appears on the left-hand side of a verification condition whose right-hand side is a known formula). In this case, we adopt an approach that guides the search for a loop invariant more effectively than that described above.

We first check whether $H' \wedge H \Rightarrow H''$ is valid for all verification conditions of the form $H' \wedge W \Rightarrow H''$ (if this is not the case, we backtrack). If so, then

$$H' \wedge (W_i \vee H_1 \vee \ldots \vee H_k) \Rightarrow H''$$

is valid for all such verification conditions (since $H = W_i \vee H_1 \vee H_2 \vee \ldots \vee H_k$, from (1)). Therefore the formulas

$$H' \wedge W_i \Rightarrow H'',\ H' \wedge H_1 \Rightarrow H'',\ \ldots,\ H' \wedge H_k \Rightarrow H'' \tag{2}$$

are all valid. In the first approach, we generated logical consequences of each of W_i, H_1, H_2, ..., H_k in our search for an approximation for W. In this case, however, we have one more piece of information about W, namely:

$$H' \wedge W \Rightarrow H''.$$

Therefore, an approximation S for W must satisfy the above formula when substituted for W, i.e. we must have

$$H' \wedge S \Rightarrow H''.$$

Let B_j be any one of $W_i, H_1, \ldots, H_k$. Since

$$H' \wedge B_j \Rightarrow H''$$

is valid (from (2)), $H' \wedge B_j \wedge \neg H''$ is unsatisfiable in the model of the programming language being used. Therefore there exists a resolution proof of the unsatisfiability of $H' \wedge B_j \wedge \neg H''$ in this model (by the completeness of resolution).

Recall that from the discussion in Section 5, a set AXIOMS of axioms that characterizes the programming language model is to be used in this resolution proof. Consider some derivation of the empty clause from Sk(AXIOMS $\wedge H' \wedge B_j \wedge \neg H''$). We can perform as many of the resolutions in this derivation as possible in Sk($B_j \wedge$AXIOMS) first, and then perform resolutions with the resulting clauses and Sk($H' \wedge \neg H''$). Consider the set of clauses (called PROOFS, say) thus derived from Sk($B_j \wedge$AXIOMS). From the above, PROOFS $\wedge$ Sk($H' \wedge \neg H''$) is unsatisfiable. We therefore unskolemize some subset of PROOFS to obtain a formula F. After such an F is obtained for each of $W_i, H_1, \ldots, H_k$, we proceed as explained in the first approach and obtain a formula S as before. Note that S is a logical consequence of each of $W_i, H_1, H_2, \ldots, H_k$, and hence of H. We then check whether $H' \wedge S \Rightarrow H''$ is valid for all verification conditions of the form $H' \wedge W \Rightarrow H''$ (if this is not the case, we backtrack). As noted earlier, this is a necessary condition for S to be a valid approximation for W.

The method described above restricts the search for an approximation S for W to all possible sets "PROOFS" generated as explained above, rather than the set of all possible logical consequences of H. This method of generating S helps to restrict the search for a loop invariant, since the known formulas H'' on the right-hand sides of verification conditions of the form $H' \wedge W \Rightarrow H''$ help to direct the search for S, and thus reduce the search space considerably.

NOTE. W_i may itself be a disjunction of formulas, i.e. we may have

$$W_i = A_1 \vee A_2 \vee \ldots \vee A_m.$$

In this case, $H = A_1 \vee \ldots \vee A_m \vee H_1 \vee \ldots \vee H_k$; thus we will have $k + m$ formulas for which logical consequences have to be generated (unlike the above-described situation where we had $k + 1$ such formulas).

The algorithm for function GET-APPROX and two procedures that it calls are given below. We denote the set of all clauses that can be derived by resolution from a formula F by $Res(F)$.

```
function GET-APPROX(H, W);
begin
  input(b);
  if there is one or more verification condition of the form H1 ∧ W ⇒ H2 then
      S := DIRECTED_SEARCH(H, W)
  else S := CONSEQUENCE(H, W);
  return(S)
end.

function CONSEQUENCE(H, W);
begin
  S := true;
  num_iterations := 0;
  Suppose H = B1 ∨ B2 ∨ ... ∨ Br;
  while (S = true) and (num_iterations < b) do
        {num_iterations := num_iterations + 1;
        for i := 1 to r do
                generate a formula Si by unskolemizing Res(Bi∧ AXIOMS) using algo-
                rithm U;
        for i := 1 to r do
                if Bk ⇒ Si for all k such that 1 ≤ k ≤ r then
                        S := S ∧ Si }
  if (S = true) then let S = S1 ∨ S2 ∨ ... ∨ Sr;
  return(S)
end.

function DIRECTED_SEARCH(H, W);
begin
  check if H1 ∧ H ⇒ H2 for all verification conditions of the form H1 ∧ W ⇒ H2; if not,
  BACKTRACK;
  num_iterations := 0;
  S := true;
  Suppose H = B1 ∨ B2 ∨ ... ∨ Br;
  while (S = true) and (num_iterations < b) do
        {num_iterations := num_iterations + 1;
        for i := 1 to r do
                { PROOFS := set of all clauses generated from Bi∧ AXIOMS by reso-
                lution during some proof of H1 ∧ Bi ⇒ H2; (see Note below)
```

generate a formula S_i by unskolemizing SUB using algorithm U, for some subset SUB of PROOFS}
for $i := 1$ to r do
if $B_k \Rightarrow S_i$ for all k such that $1 \leq k \leq r$ then
$S := S \wedge S_i$ }
if (S=*true*) then let $S = S_1 \vee S_2 \vee ... \vee S_r$;
check if $H_1 \wedge S \Rightarrow H_2$ for all verification conditions of the form $H_1 \wedge W \Rightarrow H_2$; if not, BACKTRACK;
return(S)
end.

NOTE. Since $H_1 \wedge H \Rightarrow H_2$ is valid for all verification conditions of the form $H_1 \wedge W \Rightarrow H_2$, and since $H = B_1 \vee ... \vee B_r$, clearly for every i such that $1 \leq i \leq r$, $H_1 \wedge B_i \Rightarrow H_2$. Consider a proof of unsatisfiability of Sk(AXIOMS $\wedge H_1 \wedge B_i)\wedge$ Sk($\neg H_2$) by resolution, for one such verification condition $H_1 \wedge W \Rightarrow H_2$. These resolutions can be rearranged so that any resolutions among clauses of B_i and clauses of AXIOMS are performed first. Let the set of clauses from B_i and AXIOMS and the clauses generated by these resolutions be the set PROOFS.

Proofs of the completeness and soundness of the iteration algorithm for deriving loop invariants are given in the appendix and Section 11 respectively. A refinement to the algorithm appears in (Chadha, 1991); briefly, this refinement arises from the fact that assertions that do not mention the program variables need not be included in loop invariants, since they can be generated from the axioms of the programming language operations. This knowledge makes the task of generating loop invariants more efficient, since any formula that is generated by the function GET-APPROX and that does not contain program variables can be immediately discarded. This greatly reduces the search space for a loop invariant.

10. Completeness of the iteration algorithm

The proof of completeness is based on the following five facts, which are proved in the appendix:

(i) The first time that GET-APPROX(H, W) is called, $H \Rightarrow W$ is valid.

(ii) If $H \Rightarrow W$, then GET-APPROX(H, W) can return S such that $S \preceq W$.

(iii) If GET-APPROX(H, W) has returned S such that $S \preceq W$ all the n times it has been called, then when it is called for the $n + 1^{th}$ time, H will imply W.

(iv) GET-APPROX(H, W) can always return S such that $S \preceq W$.

(v) If GET-APPROX always returns a formula such that $S \preceq W$, where W is its second argument, then the algorithm terminates and returns a loop invariant.

This claim of completeness does not say that no matter which formulas GET-APPROX generates, the algorithm will terminate with the correct answer. Since the loop invariants W are unknown, there is no way of verifying that an approximation S generated by GET-APPROX indeed satisfies $S \preceq W$; and a potentially infinite number of formulas can be generated by GET-APPROX, only a finite number of which satisfy this condition. However, it does say that if a loop invariant exists, there is a way of deriving it using this algorithm. In the same way, the completeness of resolution does not guarantee that no

matter which clauses are chosen to be used in resolution steps, the proof will terminate; rather, it says that there is a way of obtaining a proof, if one exists, if the proper clauses are chosen for resolution.

It may seem that the completeness result is obvious, due to the assumption of the existence of an axiomatization and the way in which GET-APPROX has been implemented. However, the results in Section 3 are crucial here, because not all methods of deriving logical consequences would yield the same completeness result. In particular, we mentioned in Section 3.2.1 that certain logical consequences cannot be derived without using unskolemization. Thus the completeness of the iteration algorithm is dependent on the properties of the unskolemization algorithm of Section 3.

11. Soundness of the iteration algorithm

To show that this algorithm is sound, all we need to do is to show that the final approximations for the loop invariants that are generated are loop invariants that make the verification conditions valid. This can be done by showing that all the verification conditions still hold when the generated loop invariants are substituted for the actual loop invariants. When the algorithm terminates, all the flags (for every loop invariant) are set to true. This means that every verification condition with a loop invariant on the right-hand side of the implication sign is true if W^i_{approx} is substituted for the loop invariant. Therefore the only verification conditions that need to be checked are those that do not have a loop invariant on the right-hand side of the implication sign, i.e. those of type (iii):

$$H_1 \wedge W^i \Rightarrow H_2.$$

However, for any loop invariant W^i for which a verification condition of the above form exists, the function DIRECTED_SEARCH returns an approximation S for the loop invariant W^i such that $H_1 \wedge S \Rightarrow H_2$, since this condition is specifically checked for at the end of the function. Thus function GET-APPROX also returns an approximation S for the loop invariant W^i such that $H_1 \wedge S \Rightarrow H_2$. This means that every approximation W^i_j for W^i satisfies the formula

$$H_1 \wedge W^i_j \Rightarrow H_2.$$

Therefore in particular,

$$H_1 \wedge W^i_{approx} \Rightarrow H_2$$

and the soundness of the algorithm is proved. □

EXAMPLE 4. The program shown in Figure 2 computes $z = gcd(x_1, x_2)$ for a pair of positive integers x_1 and x_2; that is, z is the greatest common divisor of x_1 and x_2. The computation method is based on the fact that

If $y_1 > y_2$, then $gcd(y_1, y_2) = gcd(y_1 - y_2, y_2)$
If $y_1 < y_2$, then $gcd(y_1, y_2) = gcd(y_1, y_2 - y_1)$
If $y_1 = y_2$, then $gcd(y_1, y_2) = y_1 = y_2$.

The program is to be proved partially correct with respect to the input predicate $\phi(\overline{x})$: $x_1 > 0 \wedge x_2 > 0$ and the output predicate $\psi(\overline{x}, z) : z = gcd(x_1, x_2)$. The two loops of the program have been cut at point B and an unknown loop invariant W^1 attached to this point. We will use the iteration algorithm to derive this invariant.

Since the domain of the given program is the set of integers, and the operations of the

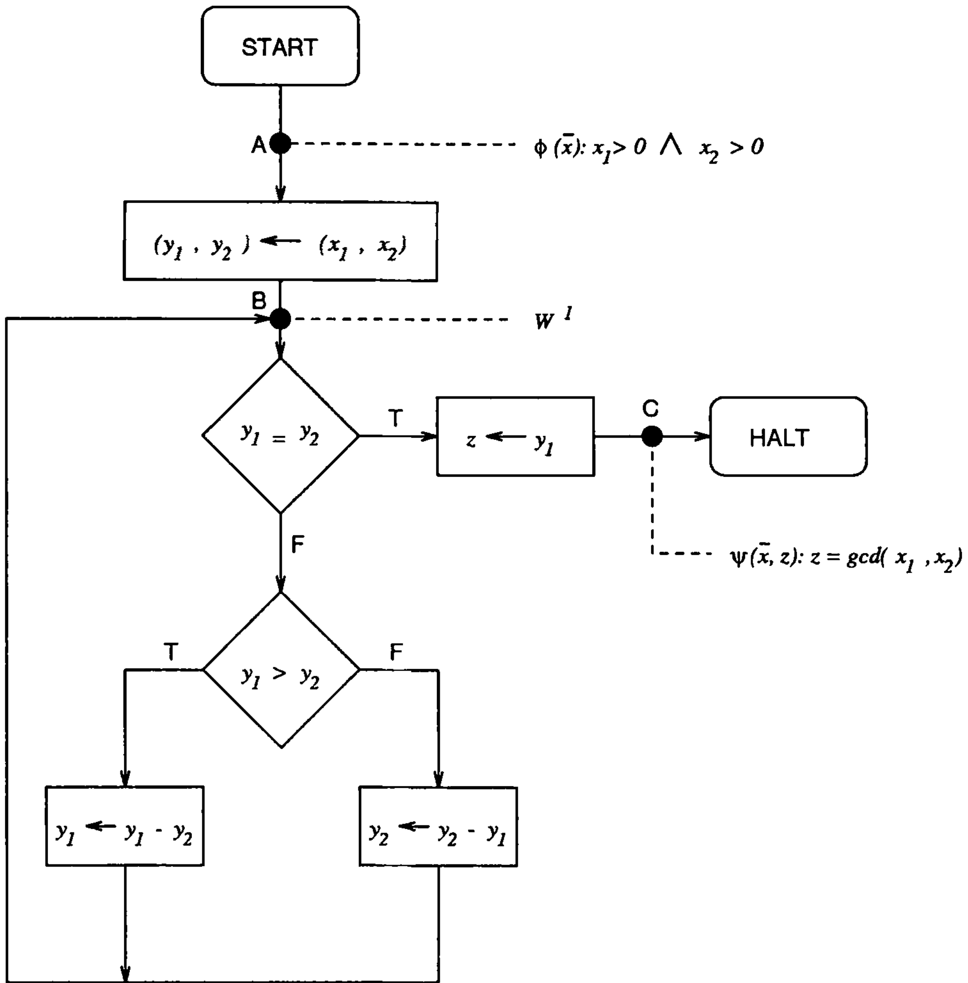

Figure 2. Calculating the g.c.d. of two numbers

language include arithmetic operations, comparison and equality, we must include the necessary axioms for arithmetic operations, comparison and equality when performing resolutions. Also, we must provide the definition of the *gcd* function, since the function is mentioned in the output assertion ψ. The axioms listed above are used to define the *gcd* function. Let the set of all these axioms be AXIOMS.

We perform the iteration algorithm step by step. There is only one loop invariant here, so *number_of_loop_invariants*=1. There are four paths leading from one cutpoint to another, since two different paths exist in the loop, depending on which branch is taken after the test $y_1 > y_2$. We denote old values for the variables y_1 and y_2 by y_1' and y_2' respectively.

There are four verification conditions for the program, which are

$vc_1 \equiv (x_1 > 0) \wedge (x_2 > 0) \wedge (x_1 = y_1) \wedge (x_2 = y_2) \Rightarrow W^1(\bar{x}, y_1, y_2)$

$vc_2 \equiv \exists y_1'(W^1(\bar{x}, y_1', y_2) \wedge (y_1 = y_1' - y_2) \wedge (y_1' \neq y_2) \wedge (y_1' > y_2) \Rightarrow W^1(\bar{x}, y_1, y_2))$

$vc_3 \equiv \exists y_2'(W^1(\bar{x}, y_1, y_2') \wedge (y_2 = y_2' - y_1) \wedge (y_1 \neq y_2') \wedge (y_1 \leq y_2') \Rightarrow W^1(\bar{x}, y_1, y_2))$

$vc_4 \equiv W^1(\overline{x}, y_1, y_2) \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.

Step 1. $index(W^1) = 0$, $W_0^1 = false$

Step 2. $list_of_loop_invariants = [W^1]$.

Step 3. First iteration:

$W = W^1$

$Tconditions = \{\text{lhs}(vc_1)\}$

$index(W^1) = 1$

$flag(W^1) = false$

$W_1^1 = \text{GET-APPROX}(W_0^1 \vee \text{lhs}(vc_1), W^1)$

$= \text{GET-APPROX}(false \vee \text{lhs}(vc_1), W^1)$

$= \text{GET-APPROX}(\text{lhs}(vc_1), W^1)$

$= \text{GET-APPROX}(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2, W^1)$.

Call GET-APPROX(H, W^1), where

$H = x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2$

input b to be some large number

S := DIRECTED_SEARCH(H, W^1)

Call DIRECTED_SEARCH(H, W^1)

check if AXIOMS $\wedge H \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is valid; since it is, continue;

$S := true$;

$H = B_1$;

WHILE LOOP :

(r=1)

$num_iterations := 1$

First FOR loop :

PROOFS := set of all clauses generated from $B_1 \wedge$ AXIOMS by resolution during some proof of AXIOMS $\wedge H \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of AXIOMS $\wedge H \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is given in the appendix; thus we have

PROOFS := $\{\{x_1 > 0\}, \{x_2 > 0\}, \{x_1 = y_1\}, \{x_2 = y_2\}, \{y_1 \neq y_2, y_1 = gcd(y_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, y_2)\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, x_2)\}\}$;

From this set, choose S_1 = all clauses in PROOFS, leaving out axioms $= x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$;

Second FOR loop :

Since $B_1 \Rightarrow S_1$, therefore $S := (true \wedge S_1) =$
$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$.

Clearly AXIOMS $\wedge S \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is valid, therefore DIRECTED_SEARCH and GET-APPROX return $(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$.

Hence we obtained, after calling function GET-APPROX,

$W_1^1 \equiv (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$.

Second iteration of Step 3:

$W = W^1$

$Tconditions = \{\text{lhs}(vc_2), \text{lhs}(vc_3)\}$
$index(W^1) = 2$
$flag(W^1) = false$
$W_2^1 = \text{GET-APPROX}(W_1^1 \vee \text{lhs}(vc_2) \vee \text{lhs}(vc_3), W^1)$
Call GET-APPROX(H, W^1), where
$H = (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)) \vee$
$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1' \wedge x_2 = y_2 \wedge (y_1 = y_1' - y_2) \wedge (y_1' \neq y_2) \wedge y_1' > y_2) \vee$
$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2' \wedge (y_2 = y_2' - y_1) \wedge (y_2' \neq y_1) \wedge y_1 \leq y_2')$
$= B_1 \vee B_2 \vee B_3$.
input b to be some large number
$S :=$ DIRECTED_SEARCH(H, W^1)
Call DIRECTED_SEARCH(H, W^1)
check if AXIOMS $\wedge H \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is valid; since it is, continue;
$S := true$;
$H = B_1 \vee B_2 \vee B_3$
First iteration of WHILE loop :
(r=3)
$num_iterations := 1$
First iteration of first FOR loop : $i = 1$:
PROOFS := set of all clauses generated from $B_1 \wedge$ AXIOMS by resolution during some proof of AXIOMS $\wedge B_1 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.
A proof of AXIOMS $\wedge B_1 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is given in the appendix; thus we have
PROOFS := $\{\{x_1 > 0\}, \{x_2 > 0\}, \{x_1 = y_1\}, \{x_2 = y_2\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{y_1 \neq y_2, y_1 = gcd(y_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, x_2)\}\}$;
From this set, choose S_1 = all clauses in PROOFS, leaving out axioms
$= x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$;
Second iteration of first FOR loop : i = 2 :
PROOFS := set of all clauses generated from $B_2 \wedge$ AXIOMS by resolution during some proof of AXIOMS $\wedge B_2 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.
A proof of AXIOMS $\wedge B_2 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is given in the appendix; thus we have
PROOFS := $\{\{(Y > Z), gcd(Y, Z) = gcd(Y - Z, Z)\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{x_1 > 0\}, \{x_2 > 0\}, \{y_1' > y_2\}, \{x_1 = y_1'\}, \{x_2 = y_2\}, \{y_1 = y_1' - y_2\}, \{x_1 > y_2\}, \{x_1 > x_2\}, \{gcd(x_1, x_2) = gcd(x_1 - x_2, x_2)\}, \{y_1 = x_1 - y_2\}, \{y_1 = x_1 - x_2\}, \{gcd(x_1, x_2) = gcd(y_1, x_2)\}, \{gcd(x_1, x_2) = gcd(y_1, y_2)\}\}$;
From this set, choose S_2 = all clauses generated from $B_2 \wedge$ AXIOMS in PROOFS, leaving out axioms
$= (x_1 > 0 \wedge x_2 > 0 \wedge y_1' > y_2 \wedge x_1 = y_1' \wedge x_2 = y_2 \wedge y_1 = y_1' - y_2 \wedge x_1 > y_2 \wedge x_1 > x_2 \wedge (gcd(x_1, x_2) = gcd(x_1 - x_2, x_2)) \wedge y_1 = x_1 - y_2 \wedge y_1 = x_1 - x_2 \wedge (gcd(x_1, x_2) = gcd(y_1, x_2)) \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$.
Third iteration of first FOR loop : i = 3 :
PROOFS := set of all clauses generated from $B_3 \wedge$ AXIOMS by resolution during some proof of AXIOMS $\wedge B_3 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of AXIOMS $\wedge B_3 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$ is given in the appendix; thus we have
PROOFS := $\{\{(Y < Z), gcd(Y,Z) = gcd(Y, Z - Y)\}, \{Y \neq Z, Y = gcd(Y,Z)\}, \{x_1 > 0\}, \{x_2 > 0\}, \{y_1 < y_2', y_1 = y_2'\}, \{x_2 = y_2'\}, \{x_1 = y_1\}, \{y_2 = y_2' - y_1\}, \{y_2' \neq y_1\}, \{y_1 < y_2'\}, \{y_1 < x_2\}, \{x_1 < x_2\}, \{gcd(x_1,x_2) = gcd(x_1, x_2 - x_1)\}, \{y_2 = x_2 - y_1\}, \{y_2 = x_2 - x_1\}, \{gcd(x_1,x_2) = gcd(x_1,y_2)\}, \{gcd(x_1,x_2) = gcd(y_1,y_2)\}\}$;
From this set, choose S_3 = all clauses generated from $B_3 \wedge$ AXIOMS in PROOFS, leaving out axioms
$= (x_1 > 0 \wedge x_2 > 0 \wedge (y_1 < y_2' \vee y_1 = y_2') \wedge x_2 = y_2' \wedge x_1 = y_1 \wedge y_2 = y_2' - y_1 \wedge y_2' \neq y_1 \wedge y_1 < y_2' \wedge y_1 < x_2 \wedge x_1 < x_2 \wedge (gcd(x_1,x_2) = gcd(x_1, x_2 - x_1)) \wedge y_2 = x_2 - y_1 \wedge y_2 = x_2 - x_1 \wedge (gcd(x_1,x_2) = gcd(x_1,y_2)) \wedge gcd(x_1,x_2) = gcd(y_1,y_2))$.

Second FOR loop :

$i = 1$:
We find that
$B_1 \Rightarrow S_1$ is valid;
$B_2 \Rightarrow S_1$ is not valid;
$B_3 \Rightarrow S_1$ is not valid.

$i = 2$:
We find that
$B_1 \Rightarrow S_2$ is not valid;
$B_2 \Rightarrow S_2$ is valid;
$B_3 \Rightarrow S_2$ is not valid.

$i = 3$:
We find that
$B_1 \Rightarrow S_3$ is not valid;
$B_2 \Rightarrow S_3$ is not valid;
$B_3 \Rightarrow S_3$ is valid.

Since S=*true*, another iteration of the WHILE loop must be performed. We use the following heuristic to guide the choice of an appropriate "S". We first look for clauses that appear in all of S_1, S_2, S_3. There are only two such clauses, namely $\{x_1 > 0\}$ and $\{x_2 > 0\}$. We then look for clauses that appear in any two of the formulas S_1, S_2 and S_3, and check if these clauses are implied by the third. We find that the only such clause is $\{gcd(x_1,x_2) = gcd(y_1,y_2)\}$. This clause appears in S_2 and S_3 and is implied by S_1 (since $x_1 = y_1$ and $x_2 = y_2$ in S_1). Thus we choose

$$S_2 = x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1,x_2) = gcd(y_1,y_2),$$

which satisfies

$$B_1 \Rightarrow S_2$$
$$B_2 \Rightarrow S_2$$
$$B_3 \Rightarrow S_2.$$

Thus S is set to be
$S = x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1,x_2) = gcd(y_1,y_2)$.

Clearly AXIOMS $\wedge S \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1,x_2))$ valid, therefore DIRECTED_SEARCH and GET-APPROX return $(x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1,x_2) = gcd(y_1,y_2))$

Hence we obtained, after calling function GET-APPROX,

$W_2^1 = (x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$

Third iteration of Step 3:

$W = W^1$

$Tconditions = \emptyset$ (it can be verified that all the verification conditions are valid with W_2^1 substituted for W^1 everywhere)

$index(W^1) = 3$

$flag(W^1) = true$

$W_3^1 = W_2^1$

Since $flag(W^1)$ is true, Step 3 terminates.

Step 4. $W_{approx}^1 = (x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$.

The loop invariant derived is

$W^1 = (x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$. □

NOTE. It may appear that the "closeness" of the axioms to the derived invariant facilitates the search for the invariant. However, if the axioms for this example had been given in a different form, the derived invariant may have been different too.

12. Conclusions

We have developed a method for automatically deriving loop invariants for loops. The methods described in this paper have not been implemented, but have been manually applied to many examples, including all nine examples from King's thesis (King, 1969). Loop invariants were successfully derived for all of these programs. In all these programs, the guidance provided in the search for loop invariants by function GET-APPROX (as explained in Section 9) greatly reduced the search space. The heuristics mentioned in Example 4 for choosing the clauses for forming a candidate loop invariant proved effective for these examples. The method seems to perform better on programs without nested loops. More examples can be found in (Chadha, 1991). This method might help to automate the verification of very large programs, where current techniques are impractical. Many people have voiced the opinion that the goal of automating the derivation of loop invariants is unattainable (see for example (Dijkstra, 1985)). Of course, they can be proved wrong only if the method we have developed can be made "acceptably" efficient by the use of suitable strategies. Basically, function GET-APPROX needs to be implemented with the use of strategies that will include rewriting terms to some normal form to improve the efficiency of the resolution procedure, detecting structural similarities among terms, and so on. The function, as it stands now, provides some guidance to the process of deriving the invariants, as was demonstrated in Example 4. It is a great deal more efficient than a pure generate-and-test approach, as explained in Section 9. Guidance is provided in the search for loop invariant by making use of the verification conditions of a program. The efficiency can probably be further improved with the use of some good heuristics. Owing to the existence of a large number of such heuristics in the literature, this aspect has not been explored in much detail here, other than the heuristics mentioned in Example 4. However, even though heuristics will be able to improve the performance of our algorithm, the algorithm still stands out from the previous purely heuristic methods in the literature. This is because in our method, heuristics can be embedded within the framework of a complete and sound algorithm. Thus, even if all heuristics fail, our algorithm can still derive a loop invariant. This is in direct contrast to previously developed methods, which have not been complete in any sense.

13. Appendix

PROOF. (Theorem 3.1, Section 3.3)

We prove this theorem by showing that by making some of the nondeterministic choices in algorithm U judiciously, a set of formulas $\mathcal{L}$ can be produced by algorithm U such that for every $F \in \mathcal{L}$, the statement of the theorem is true.

Since $H \Rightarrow W$, $H \wedge \neg W$ is unsatisfiable, so Sk(H)$\wedge$ Sk($\neg W$) is unsatisfiable. In a derivation of the empty clause from Sk(H)$\wedge$ Sk($\neg W$), the order of resolutions performed can be arranged so that resolutions performed among clauses of Sk(H) are performed first, yielding a set of clauses $\mathcal{D}$, and resolutions between clauses of Sk(H) and Sk($\neg W$) or among clauses of Sk($\neg W$) are performed later. This means that there exists a set $\mathcal{D}$ of clauses of resolvents of Sk(H) such that $\mathcal{D} \wedge$ Sk($\neg W$) is unsatisfiable and such that every clause of $\mathcal{D}$ is used at least once in the derivation of the empty clause by resolution from $\mathcal{D} \wedge$ Sk($\neg W$); also, since no more resolutions are performed among clauses of $\mathcal{D}$ after $\mathcal{D}$ has been derived, each literal L of $\mathcal{D}$ must be resolved against some literal M of Sk($\neg W$). This means that L and $\neg M$ are unifiable, i.e. there exists a substitution θ such that $L\theta = \neg M\theta$. Let σ be a substitution such that

$$\neg W\sigma = \text{Sk}(\neg W),$$

i.e. σ is a substitution that replaces existentially quantified variables of $\neg W$ by Skolem functions. Then there exists some literal M' of $\neg W$ such that

$$M'\sigma = M.$$

$$\text{Therefore } L\theta = \neg M\theta = \neg M'\sigma\theta.$$

Also, since M' is a literal of $\neg W$, $\neg M'$ is a literal of W. Hence, writing $\neg M'$ as N, we see that

$$L\theta = N\sigma\theta$$

i.e. for every literal L of $\mathcal{D}$ there exists a literal N of W such that $L\theta = N\sigma\theta$.

We now show that there is a way to make the nondeterministic choices in Steps 2, 3, and 4 of algorithm U such that the properties described in the theorem will hold. For any clause C in $\mathcal{D}$, multiple instances of C could be used during the derivation of the empty clause from $\mathcal{D} \wedge$ Sk($\neg W$). If k instances of C are used, we make k copies of the clause C in Step 2. It is clear that if the number of resolutions performed to derive the empty clause from Sk($H \wedge \neg W$) is r, then no more than r copies of each clause is required.

Now suppose L = SIGN(L) $P(d_1, d_2, ..., d_s)$, N = SIGN(N) $P(b_1, b_2, ..., b_s)$, where SIGN(L) = SIGN(N). For every i such that $1 \leq i \leq s$, consider d_i and b_i. If either d_i is a Skolem symbol, or if d_i is a non-Skolem function symbol not occurring in W and b_i is a variable, then in Step 3 of the algorithm, we replace the argument d_i by the argument $b_i \leftarrow d_i$ and we say that this argument of L has been **marked**. If d_i is a variable, then we replace the argument d_i by the argument "$b_i \hookrightarrow d_i$" and call this argument **marked** too. (Note that this symbol "$\hookrightarrow$" has nothing to do with the implication sign "$\Rightarrow$".) The marking symbol "$\hookrightarrow$" has been introduced here to guide Step 4 of the algorithm.

We now describe how Step 4 is performed. Consider all marked arguments of the form "$\alpha_i \hookrightarrow x$" in groups, each group containing all such marked arguments with the same variable x on the right-hand side of the "$\hookrightarrow$" sign. Suppose $\alpha_1 \hookrightarrow x, \alpha_2 \hookrightarrow x, ..., \alpha_n \hookrightarrow X$ are all the marked arguments with x on the right-hand side of the "$\hookrightarrow$" sign. Consider the set B $=\{\alpha_1, \alpha_2, ..., \alpha_n\}$. This set contains terms that, during the resolution process, unify with x or with whatever x has been instantiated to so far, and any two terms in this set can be unified with each other. B can contain variables and function/Skolem symbols.

Let B = VAR ∪ FUNCT, where VAR is the set of variables in B, and FUNCT is B − VAR. First we choose one element of VAR, say y_1 (if VAR is non-empty), and replace all the other variables of VAR by y_1 everywhere in the formula MARK. Now consider FUNCT. From our remarks above, since any two elements of FUNCT are unifiable, every element of FUNCT must be the same function or Skolem symbol with the same number, say k, of arguments, for some $k \geq 0$. Let

$$ARG_i = \{i^{th} \text{ argument of } z \mid z \in \text{FUNCT}\}, \text{ for } 1 \leq i \leq k.$$

Repeat the above process (which was performed for the set B) for each of the k sets $ARG_1, ARG_2, ..., ARG_k$. Note that this is not really unification, since variables are not being replaced by the terms with which they unify. We are just unifying all the variables by replacing them by the same variable name. After this has been done for all the marked arguments of this form, drop the "$\hookrightarrow$" signs from the modified set of clauses MARK as well as the elements on the left-hand side of the "$\hookrightarrow$" signs. If any two clauses of MARK are now identical, one of them can be dropped. This shows how we can choose which variables to unify in marked arguments in Step 4 of the algorithm.

Now perform Steps 5 and 6 of the algorithm, and let the set of formulas obtained be $\mathcal{L}$. For every F belonging to the set of formulas $\mathcal{L}$, consider the set of clauses Sk(F). Sk(F) is a set of clauses that is the same as $\mathcal{D}$, except that:

(1) Some arguments of literals of clauses of $\mathcal{D}$ have been replaced by Skolem functions (this is true if and only if the corresponding argument in $\mathcal{D}$ was a "marked" function symbol during step 3), and

(2) There may be more than one copy of certain clauses of $\mathcal{D}$ (since multiple copies of some clauses of $\mathcal{D}$ were made during Step 2), each of which is possibly altered as mentioned in (1) above.

Recall that for literals L, N of $\mathcal{D}$ and W respectively, $L\theta = N\sigma\theta$, where σ is a substitution that replaces existentially quantified variables of $\neg W$ by Skolem functions; in other words, σ replaces universally quantified variables of W by Skolem functions. This means that all the variables of $N\sigma$ are existentially quantified in W. Using the same notation as before, suppose L = SIGN(L) $P(d_1, d_2, ..., d_s)$, N = SIGN(N) $P(b_1, b_2, ..., b_s)$, and $N\sigma$ = SIGN(N) $P(c_1, c_2, ..., c_s)$, where SIGN(L) = SIGN(N). For any i such that $1 \leq i \leq s$, if d_i is a variable, then b_i could be anything. If d_i is not a variable, then either of the following could hold:

Case (i) : If d_i is a Skolem function symbol, then (since b_i cannot contain the same Skolem symbol) b_i must be a variable. If this variable were universally quantified in W, then c_i would be a new Skolem symbol and therefore could not unify with d_i; hence b_i must be an existentially quantified variable in W.

Case (ii) : If d_i is a non-Skolem function symbol, then one of the following are possible:

(a) b_i is the same function symbol (with the same arity as d_i)

(b) b_i is a variable and the function symbol of d_i (with the same arity as d_i) appears in W. By the same argument as in (i) above, b_i must be existentially quantified in W.

(c) b_i is a variable and the function symbol of d_i (with the same arity as d_i) does not appear anywhere in W. By the same argument as in (i) above, b_i must be existentially quantified in W.

For Case (ii) (a), if the function symbol that is common to d_i and b_i has more than zero arguments, repeat the above analysis recursively for all these arguments (this analysis

must eventually terminate since L and N are of finite length). But note that during the marking process described for step 3, we marked for unskolemization all arguments that fall under category (ii)(c) above. Therefore in F, all such arguments became existentially quantified variables; and therefore in Sk(F), these variables became new Skolem functions, which fall under category (i) above. Hence in Sk(F), no argument d_i can belong to category (ii)(c), since all arguments of $\mathcal{D}$ falling in category (ii)(c) were unskolemized. Hence all arguments of Sk(F) belong to categories (i), (ii)(a) or (ii)(b), and our theorem is proved. □

PROOF. (Theorem 3.3, Section 3.3)

Let H be a given formula and let $\mathcal{D}$ be the set of clauses derived by resolution from Sk(H) in Step 1 of algorithm U. Let M be a model for $\mathcal{D}$ with domain DOM (regarding free variables as universally quantified in $\mathcal{D}$), and let $F \in \mathcal{L}$. We show that M is also a model for F.

$\mathcal{D}$ and F differ in that all Skolem functions that are arguments of predicates in $\mathcal{D}$ are replaced by existentially quantified variables in F, and in that some functions that are arguments of predicates in $\mathcal{D}$ and that are marked during the marking process of Step 3 are replaced by existentially quantified variables in F. Also, F may contain several copies of some clauses of $\mathcal{D}$. Suppose $f(\nu_1, \nu_2, ..., \nu_m)$ is a function in $\mathcal{D}$ that is marked as "$z \leftarrow f(\nu_1, \nu_2, ..., \nu_m)$" in Step 3 of the algorithm and is replaced by the existentially quantified variable z in F, and suppose $x_1, x_2, ..., x_n$ are all the (distinct) variables that occur in $\nu_1, \nu_2, ..., \nu_m$. Then, by the unskolemization process we used, "$\exists z$" comes after "$\forall x_1$", "$\forall x_2$", ..., "$\forall x_n$" in the quantifier string of F.

The model M assigns an element d of the domain DOM of M to the function $f(\nu_1, \nu_2, ..., \nu_m)$. This element d depends on the mapping assigned to f in M, and on the values of the arguments $\nu_1, \nu_2, ..., \nu_m$, which in turn depend on the variables $x_1, x_2, ..., x_n$ and on the constant and function symbols occurring in $\nu_1, \nu_2, ..., \nu_m$. Thus, given the values for variables $x_1, x_2, ..., x_n$, there exists an element d of the domain DOM such that when d is used in place of $f(\nu_1, \nu_2, ..., \nu_m)$ in the formula $\mathcal{D}$, the formula $\mathcal{D}$ is true. If we do the above for all such functions in $\mathcal{D}$ that are replaced by existentially quantified variables in F, these elements "d" can be used in place of the corresponding existentially quantified variables "z" in F and will result in the formula F being true under interpretation M (since each such existentially quantified variable z depends on the universally quantified variables $x_1, x_2, ..., x_n$ in F, and possibly some others). Hence M is also a model for F, and therefore $\mathcal{D} \Rightarrow F$. But since $\mathcal{D}$ is a set of clauses derived by resolution from Sk(H), therefore

Sk(H) $\Rightarrow \mathcal{D}$ (where free variables are regarded as universally quantified), hence
Sk(H) $\Rightarrow F$ (since $\mathcal{D} \Rightarrow F$ from the above), hence
$H \Rightarrow F$ (since Skolem functions in Sk(H) do not appear in F).
This is true for any $F \in \mathcal{L}$, and therefore the theorem is proved. □

PROOF. (Theorem 3.4, Section 3.3)

Let $\mathcal{D}$ be the set of clauses derived from Sk(H) in step 1 of algorithm U. We know that $\mathcal{D} \wedge$ Sk($\neg W$) is unsatisfiable, and in the proof of Theorem 3.1 we looked at every literal L in $\mathcal{D}$ and found the corresponding literal N in Sk($\neg W$) against which L was resolved during the derivation of the empty clause from $\mathcal{D} \wedge$ Sk($\neg W$). Then for every argument that was a function symbol in the literal L and did not occur in W, and that was unified with a variable in the literal N, we "marked" this function argument, and unskolemized

it so that every formula F in $\mathcal{L}$ (the set of unskolemized formulas resulting from $\mathcal{D}$) had an existentially quantified variable in that position (see proof of Theorem 3.1).

For any $F \in \mathcal{L}$, $\mathcal{D}$ and F are formulas that are identical in structure; the only difference is that some functions and all Skolem functions of $\mathcal{D}$ are replaced by existentially quantified variables in F, and that F may contain several copies of some clauses of $\mathcal{D}$. Two functions were replaced by the same existentially quantified variable if and only if the two functions unified with the same variable during the derivation of the empty clause from $\mathcal{D} \wedge \mathrm{Sk}(\neg W)$, or if the functions unified with two variables that unified with each other during the course of the derivation of the empty clause from $\mathcal{D} \wedge \mathrm{Sk}(\neg W)$.

Therefore the only difference between $\mathcal{D}$ and $\mathrm{Sk}(F)$ is that all Skolem functions of $\mathcal{D}$ are replaced by Skolem functions in $\mathrm{Sk}(F)$ with possibly different arguments; and those functions of $\mathcal{D}$ that are resolved against variables in literals of $\mathrm{Sk}(\neg W)$ and that do not occur in W are replaced by Skolem functions in $\mathrm{Sk}(F)$. Since any n functions in $\mathcal{D}$ that resolved against the same variable in $\mathrm{Sk}(\neg W)$, or that resolved against some variables in $\mathrm{Sk}(\neg W)$ that unified with each other during the course of the resolution, were replaced by the same existentially quantified variable in F, the Skolem function replacing that variable in $\mathrm{Sk}(F)$ will be the same for all n of these argument positions, and therefore they can all still be resolved against the same variables in $\mathrm{Sk}(\neg W)$ against which they were resolved during the course of the derivation of the empty clause from $\mathcal{D} \wedge \mathrm{Sk}(\neg W)$.

So all we need to do here is to show that there exists some $F \in \mathcal{L}$ such that the empty clause can be derived from $\mathrm{Sk}(F) \wedge \mathrm{Sk}(\neg W)$ by using exactly the same sequence of resolutions that was used to derive the empty clause from $\mathcal{D} \wedge \mathrm{Sk}(\neg W)$. We do this by showing that there is a certain linear order in the set LIN derived in step 6 (ii) of algorithm U such that $\mathrm{Sk}(F) \wedge \mathrm{Sk}(\neg W)$ is unsatisfiable for the formula F produced using the quantifier string derived using this linear order.

In the proof of Theorem 3.1, for every literal L of $\mathcal{D}$ we found a literal N of W such that $\mathrm{L}\theta = \mathrm{N}\sigma\theta$. Let the corresponding literal in $\mathrm{Sk}(F)$ be A. Let $\mathrm{L} = P(d_1, d_2, ..., d_s)$, $\mathrm{N} = P(b_1, b_2, ..., b_s)$, $\mathrm{A} = P(a_1, a_2, ..., a_s)$ (without loss of generality we have assumed that all three literals here are not negated; the same result can easily be seen to hold if all three literals are negated).

Consider any constraint (y, z) in C, where z is a variable in F that was replaced by a Skolem function, say a_i, in A. We will show that this constraint must also hold in W for the arguments with which y and z unify in $\mathrm{Sk}(\neg W)$, if these arguments are universally and existentially quantified respectively in W; in other words, the existential quantifier for the variable in W unifying with z must come after the universal quantifier for the variable in W unifying with y. a_i is a function containing y as an argument, say $a_i = \mathrm{g}(y, \text{other arguments})$. Either y appears elsewhere in A, or it doesn't. If it doesn't, then we don't need to worry about the constraint (y, z) since it is not relevant for this particular literal. If it does, then suppose a_j contains y. Now consider b_i and b_j. Since a_i is a Skolem function, b_i must be an existentially quantified variable, say $b_i = v$, in W. Since a_j contains y, b_j contains a term, say u, which unifies with y; u could either be a universally quantified variable, an existentially quantified variable, or a function symbol. If one of the latter two is true, we need not worry about it; if the first of these is true, i.e. if u is a universally quantified variable, then we must show that the constraint that $\forall u$ must precede $\exists v$ in the quantifier string of W holds for W.

Suppose it doesn't. Then in $\neg W$, "$\exists u$" comes after "$\forall v$" in the quantifier string for $\neg W$. Therefore σ assigns a Skolem function, say β, to u which contains v as an argument

(recall that $\neg W\sigma = \text{Sk}(\neg W)$); say the assignment is : $u \leftarrow \beta(v$, other arguments). The substitution σ leaves v unchanged, since v is universally quantified in $\neg W$.

We have, $\text{L}\theta = \text{N}\sigma\theta$.

Therefore $y\theta = \beta(v$, other arguments) θ

and g(y, other arguments) $\theta = v\theta$.

But this means that y unifies with $\beta(v$, other arguments) and v unifies with g(y, other arguments). From this we see that y gets unified with a term containing y, which is a contradiction since such a unification cannot succeed due to occur check. Hence our assumption must be wrong, i.e. the quantifier "$\forall u$" must precede "$\exists v$" in the quantifier string for W.

We have shown that any constraint (y, z) in **C** must hold for the corresponding arguments in W, if the arguments u and v (say) corresponding to y and z are universally and existentially quantified respectively. Thus the relation DEP and the partial order PO, whose definitions are based on the contents of **C**, will be the same for these arguments in Sk(F) and W. Thus there exists a linear order in LIN that will order the universal quantifiers in the same order as in W, and existential quantifiers can be inserted into this string in the same order as in W. Name the formula constructed in this way "**F**"; then **F**$\Rightarrow W$. □

PROOF. (Theorem 3.5, Section 3.3)

Since $F_1 \preceq$ W, $F_2 \preceq$ W, therefore we know that

$$F_1 \Rightarrow W, F_2 \Rightarrow W$$

and therefore

$$(F_1 \wedge F_2) \Rightarrow W,\ (F_1 \vee F_2) \Rightarrow W$$

Also, since each of F_1 and F_2 are more general than W, from the definition of "more general than" it can be seen that both $F_1 \wedge F_2$ and $F_1 \vee F_2$ are more general than W. Hence

$$(F_1 \wedge F_2) \preceq W,\ (F_1 \vee F_2) \preceq W,$$

by definition. □

PROOF. (Completeness of the iteration algorithm, Section 10)

PROOF OF (i). Suppose GET-APPROX has not yet been called. Then the approximations for all loop invariants are currently set to *false*. Consider the first argument H of GET-APPROX. If the second argument of GET-APPROX is W, then H is a disjunction of W_0 and the left-hand sides of verification conditions that have W on their right-hand sides and that are not valid with W set to *false*. However, it can be seen that none of these left-hand sides can contain an occurrence of any loop invariant; the reason for this is that since all the current approximations for all loop invariants are *false*, any left-hand side containing an occurrence of a loop invariant would have the value *false* (since *false* $\wedge H' \equiv$ *false* for all H'). And since *false* $\Rightarrow$ X is valid no matter what X is, a verification condition containing a loop invariant in its left-hand side would be valid. Thus all the left-hand sides of verification conditions that are included as disjunctions in H must be known formulas without any occurrences of loop invariants, and the right-hand sides of these verification conditions are all W. Hence each of these left-hand sides must imply W. Since $W_0 \Rightarrow W$ (because $W_0 =$ *false*), clearly here $H \Rightarrow W$. □

PROOF OF (ii). Suppose GET-APPROX(H, W) is called, and suppose $H \Rightarrow W$. GET-APPROX in turn calls either CONSEQUENCE or DIRECTED_SEARCH.

Case 1. Suppose GET-APPROX calls CONSEQUENCE(H, W). We have $H = B_1 \vee B_2 \vee \ldots \vee B_r$, where some of the B_i's constitute the previous approximation for W, and the remaining B_j's are left-hand sides of verification conditions whose right-hand sides are W. Since $H \Rightarrow W$, we have

$$B_1 \vee B_2 \vee \ldots \vee B_r \Rightarrow W$$

Therefore for any i,

$$B_i \Rightarrow (B_1 \vee B_2 \vee \ldots \vee B_r) \Rightarrow W$$

i.e. $(B_i \Rightarrow W)$ is valid for all i such that $1 \leq i \leq r$.

Since for each i, an S_i can be found by unskolemizing Res($B_i \wedge$ AXIOMS) with the property that $S_i \preceq W$ (because $B_i \Rightarrow W$; thus we can use the results from Section 3), it is possible to obtain the S_i's above so that $S_i \preceq W$ for all i such that $1 \leq i \leq r$. It may happen that S_i is implied by some of the other B_k's; if it is implied by all the B_k's, then S_i is added as a conjunct in formula S. After this is done for each i such that $1 \leq i \leq r$, S will be a conjunction of S_i's such that $S_i \preceq W$ for all i. But then by Theorem 3.5, $S \preceq W$.

If not even one formula S_i can be derived from B_i such that S_i is implied by all the other B_j's (for any i such that $1 \leq i \leq r$) in a number b of trials, then S is taken to be the disjunction of the last set of S_i's that were obtained in the WHILE loop; since each of these S_i's had the property that $S_i \preceq W$, by Theorem 3.5 we have $S \preceq W$.

Hence we see that it is possible for the function CONSEQUENCE to return a formula S which has the property that $S \preceq W$. Since GET-APPROX also returns S, this case is proved.

Case 2 : Suppose DIRECTED_SEARCH(H, W) gets called by GET-APPROX. We have $H = B_1 \vee B_2 \vee \ldots \vee B_r$, where some of the B_i's constitute the previous approximation for W, and the remaining B_j's are left-hand sides of verification conditions whose right-hand sides are W. As in Case 1, since $H \Rightarrow W$, we get

$$B_i \Rightarrow W \text{ for all } i \text{ such that } 1 \leq i \leq r.$$

Now, since DIRECTED_SEARCH has been called, there exist verification conditions of the form $H_1 \wedge W \Rightarrow H_2$, where W is the second argument of DIRECTED_SEARCH and where H_1, H_2 are known formulas. Note that since $H \Rightarrow W$, we have $H_1 \wedge H \Rightarrow H_1 \wedge W \Rightarrow H_2$, i.e. $H_1 \wedge H \Rightarrow H_2$ is valid and therefore the condition to be checked at the entry to the function holds. Let $H_1 \wedge W \Rightarrow H_2$ be one such verification condition.

Since $B_i \Rightarrow W$ for all i such that $1 \leq i \leq r$, we have $H_1 \wedge B_i \Rightarrow H_1 \wedge W \Rightarrow H_2$, i.e. $H_1 \wedge B_i \Rightarrow H_2$. For any i, consider the set of clauses Sk($B_i \wedge$ AXIOMS $\wedge H_1$)$\wedge$ Sk($\neg H_2$) (where $1 \leq i \leq r$). This set of clauses is unsatisfiable and therefore there exist derivations of the empty clause from these clauses. It is possible to derive a set of clauses $\mathcal{D}_i$ from Sk($B_i \wedge$ AXIOMS) and to unskolemize $\mathcal{D}_i$ to give a formula S_i such that

$B_i \Rightarrow S_i \Rightarrow W$ and $S_i \preceq W$ (from the theorems in Section 3).

Then, since $S_i \Rightarrow W$, therefore $H_1 \wedge S_i \Rightarrow H_2$ is valid; hence since by Theorem 3.3, ($B_i \wedge$AXIOMS) $\Rightarrow S_i$ is valid, therefore $H_1 \wedge$ ($B_i \wedge$AXIOMS) $\Rightarrow H_1 \wedge S_i \Rightarrow H_2$, i.e. $H_1 \wedge B_i \wedge$AXIOMS $\Rightarrow H_2$ is valid; therefore there exists a derivation of the empty clause from Sk($H_1 \wedge B_i \wedge$AXIOMS)$\wedge$ Sk($\neg H_2$).

Thus there exists a derivation of the empty clause from Sk($B_i \wedge$ AXIOMS) $\wedge$ Sk(H_1)$\wedge$ Sk($\neg H_2$) such that a set $\mathcal{D}_i$ of clauses is produced by resolution from Sk($B_i \wedge$ AXIOMS)

during this derivation of the empty clause such that $\mathcal{D}_i$ has the above-mentioned properties (namely that $\mathcal{D}_i$ can be unskolemized to give S_i such that $S_i \preceq W$).

Therefore if we let PROOFS be defined as in the note at the end of the function DIRECTED_SEARCH, for such a derivation of the empty clause, it is possible to pick a set of clauses $\mathcal{D}_i$ from PROOFS such that for some S_i obtained by unskolemizing $\mathcal{D}_i$, $S_i \preceq W$. As noted in Section 9, this method directs the search for W. This was demonstrated in Example 4. Thus for each i, it is possible to choose the S_i's above so that $S_i \preceq W$ for all i such that $1 \leq i \leq k$. It may happen that S_i is implied by some of the other B_k's; if it is implied by all the B_k's, then S_i is added as a conjunct in formula S. After this is done for all i such that $1 \leq i \leq r$, S will be a conjunction of S_i's such that $S_i \preceq W$ for all i. But then by Theorem 3.5, $S \preceq W$.

If no such formulas S_i such that S_i is implied by all the B_k's can be derived from the B_i's in a number b of trials, then S is taken to be the disjunction of the last set of S_i's that were obtained in the WHILE loop; since each of these S_i's had the property that $S_i \preceq W$, by Theorem 3.5, $S \preceq W$.

Hence we see that it is possible for the function DIRECTED_SEARCH to return a formula S that has the property that $S \preceq W$.

Finally, since $S \preceq W$, therefore by definition $S \Rightarrow W$, and hence

$$(H_1 \wedge S) \Rightarrow (H_1 \wedge W) \Rightarrow H_2,$$
$$\text{i.e. } H_1 \wedge S \Rightarrow H_2$$

and thus the last condition for exit from the function is satisfied; therefore GET-APPROX can return S such that $S \preceq W$, and the proof is complete.□

PROOF OF (iii). Suppose GET-APPROX(H, W) has returned S such that $S \preceq W$ all the n times it has been called, and suppose that it is called for the $n+1^{th}$ time, with first and second arguments H and W respectively. We know that

$$H = B_1 \vee B_2 \vee \ldots \vee B_r,$$

where some of the B_i's constitute the previous approximation for W, and the remaining B_j's are left-hand sides of verification conditions whose right-hand sides are W. For the B_i's that constitute the previous approximation for W, we know that each of these B_i's imply W (since GET-APPROX returned S such that $S \preceq W$ all the n times it has been called so far, therefore any approximation S for W returned by GET-APPROX had the property that $S \Rightarrow W$). For the B_j's which are left-hand sides of verification conditions with W on their right-hand sides, B_j can either be written as H_1 or as $H_1 \wedge W'_k$, for some known formula H_1 and some approximation W'_k to a loop invariant W' (W' could be equal to W). If B_j can be written as H_1, then $H_1 \Rightarrow W$ is valid (since it is a verification condition); if $B_j = H_1 \wedge W'_k$, then since $H_1 \wedge W' \Rightarrow W$ is a verification condition, and since GET-APPROX returned S such that $S \preceq W$ all the n times it has been called so far, therefore the approximation W'_k for W' implies W', therefore we know that

$$H_1 \wedge W'_k \Rightarrow H_1 \wedge W' \Rightarrow W$$

i.e. $H_1 \wedge W'_k \Rightarrow W$ is also valid. Hence each B_j implies W. Therefore $H \Rightarrow W$ (since $H = B_1 \vee \ldots \vee B_r$ and since $B_k \Rightarrow W$ for all k such that $1 \leq k \leq r$). □

PROOF OF (iv). We prove that GET-APPROX can always return a formula S such that $S \preceq W$ by induction on the number of times GET-APPROX has been called.

Base case : If GET-APPROX is being called for the first time with first and second arguments H and W respectively, then by (i), $H \Rightarrow W$. Therefore by (ii), GET-APPROX can return S such that $S \preceq W$.

Inductive hypothesis : Suppose that for all the n times that GET-APPROX has been called, it has returned formulas S such that $S \preceq W$, where W was the second argument of GET-APPROX in that call.
Inductive step : Suppose GET-APPROX(H, W) is called, this being the $n+1^{th}$ call of GET-APPROX. By the inductive hypothesis, GET-APPROX returned formulas S such that $S \preceq W$ all the n times that GET-APPROX was called (W being the second argument of GET-APPROX in each case). Therefore by (iii), $H \Rightarrow W$; and therefore by (ii), GET-APPROX can return S such that $S \preceq W$ in this $n+1^{th}$ call of GET-APPROX too, and the proof is complete by induction. □

PROOF OF (v). We saw from (iv) that for each loop invariant W^i, it is possible to derive approximations W^i_j (by calling GET-APPROX) such that each W^i_j derived has the property that

$$W^i_j \preceq W^i \text{ for all } k.$$

And since $\{F \mid F \preceq W^i\}$ is finite up to variants for any W^i (provided that in the conjunctive normal form of F, no two disjunctions of F are identical, and no disjunction contains more than one occurrence of any literal) from Theorem 3.2, only a finite number of distinct W^i_j's exist. But then this means that at some point during the execution of the algorithm, the W^i_j's derived will start repeating themselves.

Thus there exists some integer λ such that

$j \geq \lambda \Rightarrow (\exists k(k < \lambda \wedge W^i_j = W^i_k))$ for every i, $1 \leq i \leq n$.

Recall that $W^i_j \Rightarrow W^i_{j+1}$ for every $j \geq 0$. We have to show that there will be a time when $\bigwedge_{i=1}^n flag(W^i)$ will be true. We first show that $W^i_\lambda = W^i_{\lambda-1}$ for all i, $1 \leq i \leq n$.

We know from the above that there exists $k < \lambda$ such that

$$W^i_\lambda = W^i_k.$$

Therefore $W^i_\lambda = W^i_k \Rightarrow W^i_{k+1} \Rightarrow W^i_{k+2} \Rightarrow \dots \Rightarrow W^i_{\lambda-1} \Rightarrow W^i_\lambda$
i.e. $W^i_\lambda \Rightarrow W^i_{\lambda-1}$, $W^i_{\lambda-1} \Rightarrow W^i_\lambda$.
Therefore $W^i_{\lambda-1} \equiv W^i_\lambda$.

This is true for every i, $1 \leq i \leq n$. Therefore $flag(W^i)$ will be set to *true* after W^i_λ has been calculated. Since this is true for every i, $1 \leq i \leq n$, $\bigwedge_{i=1}^n flag(W^i)$ will be true after W^i_λ has been calculated for every i, $1 \leq i \leq n$, and then the algorithm halts. □

PROOFS USED IN EXAMPLE 4, SECTION 11.

• Proof of AXIOMS $\wedge x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$. (The same proof can be used to show that AXIOMS $\wedge x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)) \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2)))$

1. $x_1 > 0$	Given
2. $x_2 > 0$	Given
3. $x_1 = y_1$	Given
4. $x_2 = y_2$	Given
5. $Y \neq Z \vee Y = gcd(Y, Z)$	Axiom
6. $(y_1 = y_2)$	Given
7. $(y_1 \neq gcd(x_1, x_2))$	Given

8. $y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)$	Instance of 5
9. $y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)$	Paramodulate 3,8
10. $y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)$	Paramodulate 4,9
11. $y_1 = gcd(x_1, x_2)$	Resolve 6,10
12. empty clause	Resolve 7,11.

- Proof of AXIOMS $\wedge(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1' \wedge x_2 = y_2 \wedge (y_1 = y_1' - y_2) \wedge (y_1' \neq y_2) \wedge y_1' > y_2 \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.

1. $(Y > Z) \vee gcd(Y, Z) = gcd(Y - Z, Z)$	Axiom
2. $Y \neq Z \vee Y = gcd(Y, Z)$	Axiom
3. $y_1' > y_2$	Given
4. $x_1 = y_1'$	Given
5. $x_2 = y_2$	Given
6. $y_1 = y_1' - y_2$	Given
7. $y_1 = y_2$	Given
8. $y_1 \neq gcd(x_1, x_2)$	Given
9. $x_1 > y_2$	Paramodulate 3,4
10. $x_1 > x_2$	Paramodulate 5,9
11. $\text{gcd}(x_1, x_2) = gcd(x_1 - x_2, x_2)$	Resolve 1,10
12. $y_1 = x_1 - y_2$	Paramodulate 6,4
13. $y_1 = x_1 - x_2$	Paramodulate 5,12
14. $\text{gcd}(x_1, x_2) = gcd(y_1, x_2)$	Paramodulate 11,13
15. $\text{gcd}(x_1, x_2) = gcd(y_1, y_2)$	Paramodulate 5,14
16. $y_1 = gcd(y_1, y_2)$	Resolve 2,7
17. $\text{gcd}(x_1, x_2) = y_1$	Paramodulate 15,16
18. empty clause	Resolve 8,17

- Proof of AXIOMS $\wedge(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2' \wedge (y_2 = y_2' - y_1) \wedge (y_2' \neq y_1) \wedge y_1 \leq y_2' \wedge (y_1 = y_2) \Rightarrow (y_1 = gcd(x_1, x_2))$.

1. $(Y < Z) \vee gcd(Y, Z) = gcd(Y, Z - Y)$	Axiom
2. $Y \neq Z \vee Y = gcd(Y, Z)$	Axiom
3. $y_1 < y_2' \vee y_1 = y_2'$	Given
4. $x_2 = y_2'$	Given
5. $x_1 = y_1$	Given
6. $y_2 = y_2' - y_1$	Given
7. $y_2' \neq y_1$	Given
8. $y_1 = y_2$	Given
9. $y_1 \neq gcd(x_1, x_2)$	Given
10. $y_1 < y_2'$	Resolve 3,7
11. $y_1 < x_2$	Paramodulate 4,10
12. $x_1 < x_2$	Paramodulate 5,11
13. $\text{gcd}(x_1, x_2) = gcd(x_1, x_2 - x_1)$	Resolve 1,12

14. $y_2 = x_2 - y_1$	Paramodulate 6,4
15. $y_2 = x_2 - x_1$	Paramodulate 5,14
16. $gcd(x_1, x_2) = gcd(x_1, y_2)$	Paramodulate 13,15
17. $gcd(x_1, x_2) = gcd(y_1, y_2)$	Paramodulate 5,16
18. $y_1 = gcd(y_1, y_2)$	Resolve 2,8
19. $\text{gcd}(x_1, x_2) = y_1$	Paramodulate 17,18
20. empty clause	Resolve 9,19 □

14. Acknowledgments

We would like to thank Professor Alan W. Biermann and Professor David Gries for their suggestions on improving the presentation of this material. We also thank the anonymous referees for their helpful comments and suggestions.

References

Caplain, M. (1975). Finding invariant assertions for proving programs. *Proc. Intl. Conf. on Reliable Software*, 165–171.

Chadha, R. (1991). *Applications of Unskolemization.* TR91-027, Ph.D. dissertation, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill NC 27599-3175.

Chang, C.-L., Lee, R. (1973). *Symbolic Logic and Mechanical Theorem Proving.* New York: Academic Press.

Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM J. on Computing* **7**(1), 70–90.

Cooper, D. C. (1971). Programs for Mechanical Program Verification. *Machine Intelligence* **6**, 43–59.

Cousot, P., Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Fourth ACM Symposium on Principles of Programming*, 238–252.

Cox, P. T., Pietrzykowski, T. (1984). A complete, nonredundant algorithm for reversed skolemization. *Theoretical Computer Science* **28**, 239–261.

Deutsch, L. P. (1973). *An Interactive Program Verifier.* Ph.D. dissertation, Univ. of California at Berkeley.

Dijkstra, E. W. (1985). Invariance and non-determinacy. *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, 157–165.

Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of the Symposium on Applied Mathematics*, American Mathematical Society **19**, 19–32.

German, S. M., Wegbreit, B. (1975). A Synthesizer of Inductive Assertions. *IEEE Transactions on Software Engg.*, Vol. SE-1 (1), 68–75.

Good, D. I., London, R. L., Bledsoe, W. W. (1975). An Interactive Program Verification System. *IEEE Transactions on Software Engg.*, Vol. SE-1 (1).

Good, D. I. (1985)., Mechanical proofs about computer programs. *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, 55–75.

Gries, D. (1981). *The Science of Programming.* Springer-Verlag.

Katz, S. M., Manna, Z. (1973). A heuristic approach to program verification. *Third Intl. Joint Conf. on Artificial Intelligence*, 500–512.

Manna, Z. (1974). *Mathematical Theory of Computation*, New York: McGraw-Hill.

King, J. C. (1969). *A Program Verifier.* Ph.D. dissertation, Carnegie-Mellon University.

Lee, S.-J. (1990). *CLIN : An Automated Reasoning System Using Clause Linking*, Ph.D. dissertation, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill.

Loeckx, J., Sieber, K. (1987). *The Foundations of Program Verification*, John Wiley and Sons, Ltd..

Loveland, D. (1978). *Automated Theorem Proving, A Logical Basis.* North-Holland Publishing Co..

McCune, W. W. (1988). Un-Skolemizing clause sets. *Information Processing Letters*, 257–263.

Robinson, J. A. (1965). Machine-oriented Logic based on the Resolution Principle. *Journal of the ACM* **12** (1), 23–41.

Seiichiro, D., Yamaguchi, T. (1989). Program Verification System with Synthesizer of Invariant Assertions. *Systems and Computers in Japan* **20** (1), 1–13.

Wand, M. (1978). A new incompleteness result for Hoare's system. *Journal of the ACM* **25**, 168–175.

Wegbreit, B. (1973). Heuristic Methods for Mechanically Deriving Inductive Assertions. *Third Intl. Joint Conf. on Artificial Intelligence.*

Wegbreit, B. (1975). Property Extraction in Well-Founded Property Sets. *IEEE Transactions on Software Engineering*, Vol. SE-1 (3), 270–285.