Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers

D. SARKAR AND S. C. DE SARKAR, MEMBER, IEEE

Abstract—**At the heart of any program verifier lies a theorem prover which proves theorems over the domain of the program. For any meaningful program, the theorems encountered are quite complex. The problem, which is equivalent to the validity problem of second-order logic, reduces to that of first-order logic when the assertions of the program are available [9]. In both the cases, the problem remains undecidable and human intervention at some stage or other becomes essential [6]. Resolution-based theorem provers proposed for first-order logic are very popular because they allow easy human intervention. However, the theorems encountered in proving programs do not follow the exact syntax of predicate calculus; rather, they are obtained in more popular algebraic notation. Thus, the inference rules available in first-order logic are not directly applicable to the verification conditions of the paths of the program.**

In the present paper significant modifications of the first-order rules have been developed so that they apply directly to the algebraic expressions. The importance and implication of normalization of formulas in any theorem prover have been discussed. It has also been shown how the properties of the domain of discourse have been taken care of either by the normalizer or by the inference rules proposed. Through a nontrivial example the following capabilities of the verifier, which would use these inference rules, have been highlighted: 1) closeness of the proof construction process to human thought process and 2) efficient handling of user provided axioms; such capabilities make the interfacing with human element easy.

Index Terms—**Decision procedure, mechanical theorem proving, program verification, solvability, soundness and completeness of inference rules.**

I. Introduction

ASSUMING that a program has been annotated with input assertions, output assertions, and invariant assertions at the cutpoints in its loops, the steps involved in proving its correctness are as follows [8], [9]:

1) Decompose the program into a finite set of finite paths each having an assertion at the beginning and one at the end.

2) Obtain a verification condition (abbreviated to vc) corresponding to each path.

3) Prove the vcs to be always "true," that is, valid formulas or theorems.

While steps 1) and 2) can be automated very easily, the most difficult step is the last one. Since both linear and nonlinear inequalities over integer are involved, the problem is essentially unsolvable [7] and scope for human intervention needs to be accommodated in any useful decision procedure [6]. Several decision procedures for such a system or its subsystems have been proposed [7], [8], [14], which take into account the properties of the integer domain and those of the functions and relations over this domain. In these methods all the premises and the consequent clauses, constituting the formula whose validity is to be proved, are together subjected to pass through a hierarchy of processing steps. Each phase transforms the whole of the input formula. Consequently, it becomes difficult to identify exactly which constituents participated in a particular phase. In contrast, the methods used in rule-based theorem proving employ inference rules to derive inferences from the given premises. Because of this step-by-step inferencing capability, such a theorem prover works in a fashion similar to a human being and is expected to allow easier human intervention.

In rule-based theorem proving, resolution was found to be a very strong inference rule and justifiably created great enthusiasm among the researchers in the field after it was first proposed by Robinson [15]. As a result, much progress was made to have efficient theorem provers by incorporating various refinements of resolution such as semantic resolution [18], PI-clash, hyperresolution, set-of-support strategy [22], locking [2], etc. The essence of the strength of resolution lies in the fact that the rule, along with unification algorithm [16], offers a simple syntactic proof procedure which is also complete. Subsequently, when another inference rule, called paramodulation, was proposed by Wos and Robinson [17] to take care of equality-relation, the strength of resolution-approach was further highlighted by the fact that all the refinements proposed for resolution was found to be easily extendable to accommodate paramodulation. Introduction of paramodulation not only helped in increasing the efficiency of resolution theorem provers, but also suggested the possibility of having a unified approach to accommodate other theories. Slagle and Norton [19], [20] have fully exploited this aspect and described a general method called nuclearization, which consists of devising inference rules to avoid explicit use of axioms of the theory. They applied this method successfully to arrive at a theorem prover for partial and total ordering and set theory [19]. Although various limitations have been pointed out since

Manuscript received May 1, 1986; revised May 18, 1987.

D. Sarkar is with the Department of Electronics and Electrical Communication Engineering, Indian Institute of Technology, Kharagpur 721 302, India.

S. C. De Sarkar is with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721 302, India.

IEEE Log Number 8824608.

then by Bledsoe [1], Boyer and Moore [3], and Brown [4], effort is still being continued to improve upon resolution theorem provers. Murray [11] describes a method to apply it on formulas which are not in clause form. Plaisted [12], [13] identified some limitations of resolution provers and proposed a method called abstraction incorporating the idea of analogy in problem solving. The method of m-abstraction suggested in [13] results in complete search strategies. Moreover, one of the most interesting aspects of his method is that the search space gets more and more restricted as proof construction proceeds to a deeper level. Manna and Waldinger [10] introduced the notion of polarity of a subexpression with respect to any two binary relations; the notion of polarity helps in handling transitivity property through restricted substitution.

There remains, however, a major hurdle in using the inference rules and the control strategies proposed in the above mentioned works in a program verifier. The inference rules are designed to work for theorems coded in predicate calculus. However, for program verification, it is more natural that user will provide assertions not in predicate calculus but in more widely used algebraic notation. Consequently, the verification conditions resulting from such assertions will not follow the syntax of predicate calculus. Of course, it is always possible to convert the vc into a formula of first-order logic but the process is very cumbersome and is not pursued. Moreover, to allow easier human intervention it is desirable that the proof construction should proceed using the algebraic notations themselves and not their coded versions.

In the present paper we propose several inference rules which can be applied directly to the formulas of integer arithmetic. These inference rules have been framed taking into consideration the normal form chosen and the properties of the domain under consideration. The verifier which will employ these inference rules will have an overall structure similar to a resolution-based theorem prover [5].

Introducing a normalizer for the formulas as a preprocessor helps both the design and the function of a theorem prover. In Section II some properties of the integer domain and those of the associated functions and relations have been identified which can be accounted for in the normalization phase itself.

The assertions annotated to programs over integers have occurrences of binary predicate constants such as $>=$, $<=$, $=$, and $\neq$ and function constants such as addition, multiplication, etc. The properties of these entities can be explicitly stated by axioms which may be added to the premises. This approach, however, is inefficient [19], [20] because axioms may be large in number. For example, to depict the substitutivity property of "$=$," one must include axioms corresponding to each predicate and each function in the formula. A more efficient approach would be to frame inference rules to replace the axioms. Taking the lead from the existing rules in first-order logic, several inference rules have been proposed in Section III to accommodate the properties of the above-mentioned predicates. If all the properties of each of these predicates are accounted for, one can claim that the prover will not need any axioms for these predicates or equivalently, the rule-set is complete. Based on this criterion the completeness issues have also been discussed.

The above set of inference rules has been used successfully to hand-verify many examples of flowchart program verification available in the literature [5], [7], [19]. In Section IV an illustrative example has been considered to indicate the effectivity of the rule-set proposed.

The paper concludes with a discussion on the scope of a theorem prover based on such an approach.

II. Utility of Normalization in Theorem Proving

The method of proving theorems can be viewed as transforming the valid formulas to "true." For a system having canonical form, all theorems have the canonical form "true." The method which converts any formula to its canonical form is itself a theorem prover.

Unfortunately, the system we are concerned with does not have a canonical form [7]. For such systems normal forms are used. In the case of canonical forms, computationally equivalent formulas are converted to a single formula, whereas in the case of normal forms every equivalence class of formulas is mapped to a subset. A theorem prover has to deal with this subset to establish the equivalence among the members. Normalization not only reduces the dimensionality of the domain over which the theorem prover works but also forces all the formulas to follow a uniform structure. These two facts obviously make the design of the theorem prover simpler.

As has been mentioned above, the task of the theorem prover is to establish the functional equivalence. In some simple situations, however, the equivalence can be easily shown. That is, some of the properties of the entities involved can be taken care of along with the normalization process. We refer to this process as simplification of formulas, which essentially reduces the burden of the theorem prover. The latter, as a result, can be made to concentrate on more complex situations.

We now briefly discuss the normal form chosen and the simplification carried out at various levels of a normalized formula. The properties that are taken care of during simplification are also identified.

Every boolean expression is converted into conjunctive normal form (cnf) and is conveniently expressed as a set of clauses. A clause is a disjunction of literals. A literal is an atomic formula or its negation. Some definitions are now in order.

Definition 2.1—Atomic Formula:

1) $S\ \{R\}\ 0$ is an atomic formula, where S is a normalized sum (defined below) and R is one of $>=$, $<=$, $=$, or $\neq$,
2) $p(s_1, \cdots, s_n)$ is an atomic formula, where p is an n-adic predicate constant and $s_1, \cdots, s_n$ are normalized sums.

The remaining syntactic entities used in a normal form are defined by means of productions of the following grammar.

Definition 2.2:

$$S \rightarrow S + T \mid c_s,$$

where c_s is any integer

$$T \rightarrow T * P \mid c_t,$$

where c_t is any integer

$$P \rightarrow S \uparrow C_e \mid \text{abs}(S) \mid (S) \bmod (S) \mid S \div c_d \mid c_m,$$

where c_m is a symbolic constant

$$C_e \rightarrow S \uparrow C_e \mid S$$

$$C_d \rightarrow S \div C_d \mid S.$$

Thus, exponentiation and (integer) division are depicted by infix notation and all functions have arguments in the form of normalized sums.

For convenience, we refer to T as (normalized) term or as combination (product) of primaries, S as combination (sum) of (normalized) terms, P as (normalized) primary, c_s as constant term, and c_t as constant primary.

Example: $(x + 3y + 7 >= 0 \wedge 4x^2 + 3yz + 2 \neq 0) \vee (x \uparrow y > 0)$ will have the normal form

$$[1 * x + 3 * y + 7 >= 0 \vee (1 * x + 0)$$
$$\uparrow (1 * y + 0) + 1 <= 0]$$
$$\wedge [4 * x \ * x + 3 * y * z + 2 \neq 0 \vee (1 * x + 0)$$
$$\uparrow (1 * y + 0) + 1 <= 0].$$

It should be noted that the normal form chosen above is the same as that suggested by King [7]. In addition to the above structure, any arithmetic expression is arranged using a lexicographic ordering of its constituent subexpressions from the bottom-most level. All relational expressions involve relations from the set $\{>=, =, <=, \neq\}$. This set functionally spans the set of all arithmetic relations. In the integer domain, the conversion of any relation to one in the above set be achieved without introducing any new variable. For example, $s + c > 0 \equiv s + (c - 1) >= 0$, etc. (It should be noted that for the real domain, new variables are to be introduced. Thus, $s + c > 0$ is to be replaced by $\exists t(s + c - t >= 0)$.)

Various simplifications that are carried out at the normalization phase itself are briefly listed below.

1) The common subexpressions in an arithmetic expression are collected. For example, $x^2 + 3x + 4z + 7x$ is mapped to $x^2 + 10x + 4z$.

2) The sum in a relational literal is reduced by a common constant factor, if any, and the literal is accordingly simplified [7]. For example, $3x^2 + 9xy + 6z + 7 >= 0$ is mapped to $x^2 + 3xy + 2z + 2 >= 0$, where $\lfloor 7 \div 3 \rfloor = 2$.

3) A clause $C = l_1 \vee l_2 \vee \cdots \vee l_n$ is first expressed as $\sim(\sim l_1 \wedge \sim l_2 \cdots \wedge \sim l_n)$ and then literals are deleted from C by the rule "if$(l_1 \rightarrow l_2)$then$(l_1 \wedge \sim l_2 \equiv l_1)$." C is reduced to "true," if it is detected that l_1 implies $(\sim l_2)$.

It can be easily seen that some of the properties of $>=$, $<=$, $=$, and $\neq$ predicates are accommodated in course of normalization and simplification of formulas. For example, the symmetry property of $\{=, \neq\}$ is taken care of by choosing the normal form of relational literals as $S\{R\}0$ and by imposing ordering on the constituent subexpressions at all levels. Thus, both $x = y$ and $y = x$ will be expressed either as $x - y = 0$ or as $y - x = 0$, consistently. Similar is the case for $\neq$ - literals. Again, the reflexivity of $\{>=, =, <=\}$ and the irreflexivitiy of $\{\neq\}$ are accommodated by collecting the common subexpressions in a sum. For example, $x >= x$ changes to $x - x >= 0$, whereupon the left-hand side reduces to "0" by collecting the common subexpression x; accordingly, $x >= x$ reduces to "true" by normalization.

III. Inference Rules

The verification condition is essentially an implication relation with both sides in prenex cnf. The verifier proves the vc by splitting it into several cases. For example, let the vc be $\{a_1, a_2, \cdots, a_n\} \rightarrow \{C_1, C_2, \cdots, C_p\}$. The vc is proved by p cases, in each case, $\{a_1, a_2, \cdots, a_n\} \rightarrow \{C_i\}$, for $i = 1, \cdots, p$, is shown to be valid. Since the proof is by refutation, the set $S = \{a_1, \cdots, a_n, \sim C_i\}$ is shown to be unsatisfiable [19].

S is considered to be consisting of two subsets, $S_1 = \{a_1, \cdots, a_n\}$ and $S_2 = \{\sim C_i\}$. That is, the clauses from the left-hand side of the vc are put in S_1, *the antecedent set*, and those from the right hand side are put in S_2, *the consequent set.* Clauses of S_2 will be unit (one-literal) clauses, each containing a negated literal of the clause C_i.

The symbolism, $\{C_1, C_2, \cdots, C_n\} \vdash_r C$ is used to denote that the clauses C_1 through C_n infer the clause C by the inference rule r.

A. Resolution

Resolution is the most effective inference rule known for combining two clauses to obtain this logical consequences. It has been treated in detail in the literature [5], [9], [19]. Resolution for the first-order predicate calculus is defined below.

Definition 3.1: Let C_1 and C_2 be two clauses with no variable in common. Let L_1 and L_2 be two literals in C_1 and C_2, respectively. If L_1 and $\sim L_2$ have a most general unifier σ, then the clause $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$ is called a binary resolvent of C_1 and C_2.

In program verification one frequently encounters free variables. In particular, all the program variables, which are used in the program to carry out the computation, do always occur free in the assertions as well as in the vcs. Literals involving only free variables are essentially ground literals and they cannot participate in unification. Thus, resolution, as defined above, is applicable to equivalent ground literals only and will find a very limited application. If L_1 implies $\sim L_2$, the logical consequence of the clauses C_1 and C_2 can be obtained in an identical fashion. This motivates us to frame the following inference rule which we call implication-resolution (r_{ir}).

Definition 3.2: Let C_1 and C_2 be two clauses with no common variables. Let L_1 and L_2 be two literals in C_1 and C_2, respectively. If there exists a substitution θ such that

$L_1\theta \rightarrow \sim L_2\theta$, then $\{C_1, C_2\} \vdash_{r_{ir}} C$, where $C = (C_1\theta - L_1\theta) \cup (C_2\theta - L_2\theta)$.

Theorem 3.1: Implication-resolution is a valid inference rule.

Proof: r_{ir} is actually a two-step resolution process. Since $L_1\theta \rightarrow \sim L_2\theta$, $(\sim L_1\theta \vee \sim L_2\theta)$ can be included as an axiom in S, the set of clauses to be proved unsatisfiable. Therefore,

$$\{C_1\theta, (\sim L_1\theta \vee \sim L_2\theta)\} \underset{r_r}{\vdash} ((C_1\theta - L_1\theta) \cup \sim L_2\theta)$$

and

$$\{((C_1\theta - L_1\theta) \cup \sim L_2\theta), C_2\} \underset{r_r}{\vdash} (C_1\theta - L_1\theta) \cup (C_2\theta - L_2\theta)$$

(r_r is the resolution inference rule). □

Application of r_{ir} assumes a built-in capability of the prover to detect implications in the associated domain. Note that in the above proof we have mentioned that $L_1\theta \rightarrow \sim L_2\theta$ may be included as an axiom. This is to highlight the fact that the verifier is expected to detect such implication without much overhead. Subsequently, we shall list the various cases where implication is imminent. As mentioned earlier, one of the major motivations for incorporating inference rules such as paramodulation [5] and rules of partial ordering and set theory [19] in a decision procedure is to do away with axioms from the search space. Implication-resolution is a simple extension of the resolution rule to achieve the same goal.

The above inference rule covers resolution as well, because if θ is the most general unifier, then $L_1\theta = \sim L_2\theta$ and r_{ir} can be applied. For ground literals, $\theta = \{\epsilon\}$, the empty substitution.

In the integer domain the implication can be easily detected in the following cases.

Let $L_1 = Q_1 + d_1\{R_1\}\ 0$ and $L_2 = \sim L_2' = \sim(Q_2 + d_2\{R_2\}\ 0)$, where Q_1 and Q_2 are the nonconstant parts of the arithmetic expressions involving both symbolic constants and variables, R_1 and $R_2 \in \{=, >=, \neq, <=\}$, and d_1 and d_2 are integer constants. If θ is the most general unifier for Q_1 and Q_2, that is, $Q_1\theta = Q_2\theta$, then Table I gives the sufficient conditions on d_1 and d_2 depending on R_1 and R_2, for which $L_1\theta \rightarrow \sim L_2\theta$.

TABLE I
CONDITIONS UNDER WHICH A LITERAL L_1 IMPLIES NEGATE OF ANOTHER LITERAL L_2

$R_1 \backslash R_2$	$=$	$\geqq$	$\neq$	$\leqq$
$=$	$d_1 = d_2$	$d_2 \geqq d_1$	$d_1 \neq d_2$	$d_2 \leqq d_1$
$\geqq$		$d_2 \ngeqq d_1$	$d_2 > d_1$	
$\neq$			$d_1 = d_2$	
$\leqq$			$d_2 < d_1$	$d_2 \nleqq d_1$

B. Paramodulation (r_p)

Paramodulation [5], [19], [20] is an inference rule which applies when the given formula contains equality-literals. Substitutivity is the most important property of equality that is to be accounted for by paramodulation. Keeping this in view we frame the paramodulation inference rule in the following way:

Definition 3.3: Let C_1 and C_2 be two clauses.

$$\text{Let } C_1\text{: } l(s) \vee C_1'$$
$$C_2\text{: } t = 0 \vee C_2'$$

where $l(s)$ is a literal with occurrences of subexpression s, C_1' and C_2' are disjunction of literals. When the literal l involves the relations $>=$, $<=$, $=$, or $\neq$, the subexpression s is actually the whole of the normalized sum in l. Let θ_1, θ_2 be any two substitutions, s' be any arithmetic expression, and m, n be any two positive integers; then

$$\{C_1, C_2\} \underset{r_p}{\vdash} l(m * s\theta_1 \pm n * s' * t\theta_2) \cup C_1'\theta_1 \cup C_2'\theta_2.$$

Theorem 3.2: Paramodulation (r_p) is a valid inference rule.

Proof: Since $t = 0$ the proof follows.

1) Completeness Issues: It has already been pointed out in the previous section how simplification during normalization takes care of the reflexivity and symmetry properties of "=." A relation which is substitutive is also transitive. Hence, if we can take care of substitutivity of equality in all its entirety, we would automatically take care of its transivity property.

Although a formal proof as to whether substitutivity is accommodated in its entirety by this rule is lacking, we notice that the rule allows substitution of any entity, say a symbol, a single term, or any subexpression, in any expression. A detailed treatment of all these cases follows.

Case 1—Substitution for a symbolic constant x:

1.1) If the paramodulating literal contains a linear term involving the symbol x with coefficient $+1$ or -1, then all the occurrences of the symbolic constant x can be replaced in a single step. Although carried out in a single step, this essentially embeds repetitive application of paramodulation.

1.2) If the paramodulating literal contains a linear term involving x with coefficient not equal to ± 1 but an integer, then each occurrence of x will have to be replaced one by one (even in a single literal). Stepwise replacement can be carried out for occurrences in any term, sum of terms, exponent subexpression, and argument subexpressions of a function. The process involves equalizing the integer coefficients of x in the paramodulating literal and the literal to be paramodulated.

Example:

$$C_1\text{: } 2xy + 3xz + 7 >= 0,$$

$$C_2\text{: } 4x + 3y + 3 = 0,$$

$$C_1\text{: } 4xy + 6xz + 14 >= 0 \quad \text{(multiplying by 2)}$$

$$\{C_1, C_2\} \underset{r_p}{\vdash} (-3y - 3)y + 6xz + 14 >= 0$$
$$= -6y(y + 1) + 12xz + 28 >= 0$$
$$\text{(multiplying by 2)}$$
$$= C_3$$

$$\{C_3, C_2\} \underset{r_p}{\vdash} -6y(y + 1) - 9z(y + 1) + 28 >= 0.$$

Case 2—Substitution for a nonlinear term t: Some examples of t are $x * y$, $x \uparrow s$, $f(x)$, etc.

2.1) If the paramodulating literal contains a nonlinear term t with coefficient ± 1, then all the occurrences of t can be removed in a single step.

2.2) If the paramodulating literal contains a nonlinear term t with coefficient not equal to ± 1 but an integer, then the occurrences of t will have to be removed in steps.

Case 3—Substitution for any subexpression s, where s is in normalized sum form:

$$\text{Given } C_1\colon l(s) \vee C_1' \quad \text{and}$$
$$C_2\colon t = 0 \vee C_2',$$

where $t = s + t_2$, one should be able to conclude

$$l(-t_2) \vee C_1' \vee C_2'.$$

This is automatically taken care of either by case 1 or by case 2.

Example:

$$C_1\colon x + 4y - z - u - v >= 0$$
$$C_2\colon x + 4y + 7z - u - v = 0$$
$$\{C_1, C_2\} \underset{r_p}{\vdash} -8z >= 0 \quad \text{or} \quad z <= 0.$$

Here

$$s = x + 4y - z - u - v$$
$$t = x + 4y + 7z - u - v$$
$$= s + 8z.$$

It should be noted that the above inference is reached by solving for x using the method suggested in Case 1.1.

C. *Transitivity Rule (r_t)*

This rule has been devised to account for the transitivity property of $>=$ and $<=$ relations.

Definition 3.4: Let C_1 and C_2 be two clauses which are of the form

$$C_1\colon s_1 + t_1 + c_1\{R\}0 \vee C_1'$$
$$C_2\colon s_2 - t_2 + c_2\{R\}0 \vee C_2'$$

where s_1, s_2, t_1, t_2 are arithmetic subexpressions, R is either $>=$ or $<=$, c_1, c_2 are numeric constants, and C_1' and C_2' are disjunction of literals.

Let θ be the most general unifier for t_1 and t_2. Then,

$$\{C_1, C_2\} \underset{r_t}{\vdash} s_1\theta + s_2\theta + c_1 + c_2\{R\}0 \cup C_1'\theta \cup C_2'\theta.$$

Theorem 3.3: Transitivity rule is a valid inference rule.

Proof: Since addition of two nonnegative (nonpositive) expressions results in a nonnegative (nonpositive) expression, the proof follows.

1) Completeness Issue: Simplification during normalization takes care of the reflexivity property of $\{>=, <=\}$. The fact that transitivity is taken care of by the above rule is clear from the following example.

Example: Let C_1: $x - y >= 0$ and C_2: $y - x >= 0$. Then,

$$\{C_1, C_2\} \underset{r_t}{\vdash} x - z >= 0; \quad \text{here} \quad \theta = \{\epsilon\}, s_1 = x,$$
$$t_1 = t_2 = -y, \quad s_2 = -z \quad \text{and} \quad c_1 = c_2 = 0.$$

It is worthwhile to point out here that the advantage of replacing the relations $>$ ($<$) in favor of $>=$ ($<=$) in the normal form of integer arithmetic is that the transitivity can be used to derive the strongest consequence. For example, let C_1: $x - y > 0$ and C_2: $y - z > 0$; when presented in normal form C_1 will be $x - y - 1 >= 0$, C_2 will be $y - z - 1 >= 0$ and $\{C_1, C_2\} \vdash_{r_t} x - z - 2 >= 0$, which is stronger than $x - z > 0$.

To take care of the antisymmetry property of $\{>=, <=\}$ the above rule is extended by the following definition. We call this antisymmetry rule and denote it by r_{as}.

Definition 3.5: If $s_1\theta + s_2\theta + c_1 + c_2$ in Definition 3.4 evaluates to "0," then

$$\{C_1, C_2\} \underset{r_{as}}{\vdash} s_1\theta + t_1\theta + c_1 = 0 \cup C_1'\theta \cup C_2'\theta.$$

Note that r_t can also apply to literal-pairs $\{>=, <=\}$ by changing any one of the relation from $<=$ ($>=$) to $>=$ ($<=$). The following example illustrates how r_{as} applies.

Example: Let

$$C_1\colon s + 4 >= 0 \vee C_1'$$
$$C_2\colon s + 4 <= 0 \vee C_2'.$$

Since relations are different we can change any one of them. Let C_2 be changed to $(-(s + 4)) >= 0 \vee C_2'$. Now,

$$\{C_1, C_2\} \underset{r_{as}}{\vdash} s + 4 = 0 \vee C_1' \vee C_2'.$$

IV. An Illustrative Example

For ease of application of inference rules, clauses of S_2 are maintained in negated form. In the sequel, clauses of S_1 are referred to as nuclei, represented by N_1, N_2, etc., and those of S_2 are called electrons, represented by E_1, E_2, etc.

The program (Fig. 1) computes Z as $A \uparrow B$ by considering the binary representation of B [7]. The annotated assertions are given in arbitrary form but vcs and formulas obtained during proof construction are maintained in normal form.

Path (A : B):

$$VC1\colon \{x - A = 0, y - B = 0, B >= 0\}$$
$$\rightarrow \{y >= 0, x \uparrow y - A \uparrow B = 0\}$$

Case 1:

$$S_1\colon \{x - A = 0, y - B = 0, B >= 0\}$$
$$S_2\colon \{\sim y >= 0\}$$
$$\{N_2, N_3\} \underset{r_p}{\vdash} y >= 0 = N_3'$$
$$\{N_3', E_1\} \underset{r_{ir}}{\vdash} .$$

□

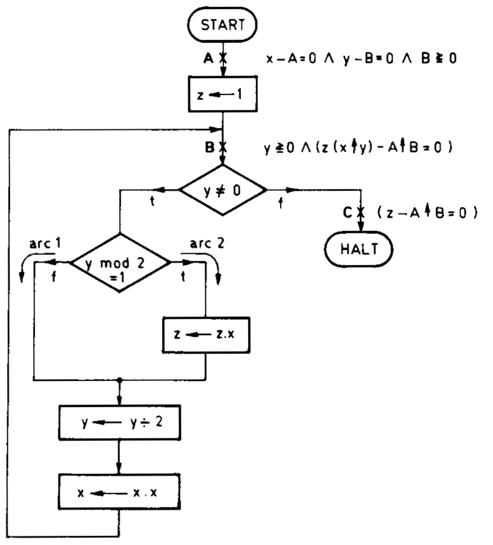


Fig. 1. An illustrative example.

Case 2:

$$S_1\text{: } \{x - A = 0, y - B = 0, B >= 0\}$$

$$S_2\text{: } \{\sim (x \uparrow y - A \uparrow B = 0)\}$$

$$\{N_1, E_1\} \underset{rp}{\vdash} \sim (A \uparrow y - A \uparrow B = 0) = E_1'$$

$$\{N_2, E_1'\} \underset{rp}{\vdash} \sim (A \uparrow B - A \uparrow B = 0)$$

$$= \text{false, by simplification.} \qquad \square$$

Path (B:B, via arc 1):

$$\begin{aligned} VC2\text{: } &\{y >= 0, y \neq 0, y \bmod 2 - 1 \neq 0, \\ &\quad z \cdot x \uparrow y - A \uparrow B = 0\} \\ &\rightarrow \{y \div 2 \ >= 0, z(x \uparrow 2 \cdot (y \div 2)) \\ &\quad - A \uparrow B = 0\} \\ &\equiv \{y - 1 >= 0, y \bmod 2 - 1 \neq 0, \\ &\quad z \cdot x \uparrow y - A \uparrow B = 0\} \\ &\rightarrow \{y \div 2 \ >= 0, z(x \uparrow 2 \cdot (y \div 2)) \\ &\quad - A \uparrow B = 0\}. \end{aligned}$$

Since

$$(y >= 0 \wedge y \neq 0 \equiv y > 0 \equiv y - 1 >= 0).$$

Case 1:

$$S_1\text{: } \{y - 1 >= 0, y \bmod 2 - 1 \neq 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$

$$S_2\text{: } \{\sim (y \div 2 >= 0)\}.$$

The following axiom defining the special function "÷" is used to replace its occurrence [7].

$$\begin{aligned} &(y >= 0 \wedge x - q \cdot y >= 0 \wedge y - (x - q \cdot y) \\ &\quad > 0 \wedge x \div y = q) \\ &\quad \vee (y < 0 \wedge x - q \cdot y >= 0 \wedge (-y - (x - q \cdot y) \\ &\quad > 0) \wedge x \div y = q), \end{aligned}$$

where q is a newly introduced variable.

Since the verifier replaces the occurrence of $y \div 2$ using the above clauses, it substitutes y for x and 2 for y. Let us assume that the verifier introduces new variables $\$_1$, $\$_2$, etc., to avoid clash with variables used in the program. With the above substitution, we have

$$\begin{aligned} &(2 >= 0 \ \wedge y - 2 \cdot \$_1 >= 0 \wedge 2 - y + 2 \\ &\quad \cdot \$_1 > 0 \wedge y \div 2 = \$_1) \\ &\quad \vee (2 < 0 \wedge y - 2 \cdot \$_1 >= 0 \wedge -2 - y + 2 \\ &\quad \cdot \$_1 > 0 \wedge y \div 2 = \$_1) \\ &\quad \equiv y - 2 \cdot \$_1 >= 0 \wedge 2 - y + 2 \cdot \$_1 \\ &\quad > 0 \wedge y \div 2 - \$_1 = 0. \end{aligned}$$

Let

$$A_1 = y - 2 \cdot \$_1 >= 0,$$

$$A_2 = 2 - y + 2 \cdot \$_1 > 0,$$

and

$$A_3 = (y \div 2 - \$_1 = 0) . \ \{A_3, E_1\} \underset{rp}{\vdash} \sim (\$_1 \geqq 0)\, E_1'.$$

A_1 and A_2 (normalized) are absorbed in S_1, whereas E_1' is maintained in S_2.

Thus,

$$\begin{aligned} S_1\text{: } &\{y - 1 >= 0, y - 2 \cdot \$_1 >= 0, y - 2 \\ &\quad \cdot \$_1 - 1 <= 0, \\ &\quad y \bmod 2 - 1 \neq 0, \quad z \cdot (x \uparrow y) - A \uparrow B = 0\} \end{aligned}$$

$$S_2\text{: } \{\sim (\$_1 >= 0)\}$$

$$\{N_1, N_3\} \underset{rt}{\vdash} 2 \cdot \$_1 >= 0 \equiv (\$_1 >= 0) = N_6$$

$$\{N_6, E_1'\} \underset{rir}{\vdash} . \qquad \square$$

It should be noted that in the above deductions the only clause from the original antecedent set that has been used is $N_1 = y - 1 >= 0$. Thus, the verifier is capable of deducing $y - 1 >= 0 \rightarrow (y \div 2 >= 0)$.

Case 2:

$$S_1 = \{y - 1 >= 0, y \bmod 2 - 1 \neq 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$

$$S_2 = [\sim (z \cdot (x \uparrow 2(y \div 2)) - A \uparrow B = 0)\}.$$

The occurrence of "$\div$" is replaced in an identical fashion as before.

$$S_1\colon \{y - 1 >= 0, y - 2 \cdot \$_1 >= 0, \\ y - 2 \cdot \$_1 - 1 <= 0, \\ y \bmod 2 - 1 \neq 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$

$$S_2\colon \{\sim (z \cdot x \uparrow (2 \cdot \$_1) - A \uparrow B = 0).$$

For replacement of the "mod" function the following axiom is provided [7].

$$(y >= 0 \wedge x - y \cdot \$_2 >= 0 \wedge y - (x - y \cdot \$_2) \\ > 0 \wedge x \bmod y = x - y \cdot \$_2) \\ \vee (y < 0 \wedge x - y \cdot \$_2 >= 0 \wedge -y \\ - (x - y \cdot \$_2) > 0 \wedge x \bmod y = x - y \cdot \$_2).$$

The substitution in this case are y for x and 2 for y. After simplification we have

$$A_1 = y - 2 \cdot \$_2 >= 0, \quad A_2 = 2 - y + 2 \cdot \$_2 > 0$$

and

$$A_3 = (y \bmod 2 = y - 2 \cdot \$_2).$$

A_1 and A_2 are absorbed in S_1, whereas A_3 paramodulates upon N_4 (that is, $y \bmod 2 - 1 \neq 0$) for replacement.

Therefore,

$$S_1\colon \{y - 1 >= 0, y - 2 \cdot \$_1 >= 0, \\ y - 2 \cdot \$_1 - 1 <= \underline{0, y - 2 \cdot \$_2 >= 0}, \\ \underline{y - 2 \cdot \$_2 - 1 <= 0}, \underline{y - 2 \cdot \$_2 - 1 \neq 0}, \\ z \cdot (x \uparrow y) - A \uparrow B = 0\}.$$

S_2 remains unchanged. $\{N_1, N_3\} \vdash_{r_t} \$_1 >= 0 = N_8$.

By simplification among conjuncts [7], N_5 and N_6 reduce to $y - 2 \cdot \$_2 <= 0 = N_9$.

$$\{N_9, N_4\} \underset{r_{as}}{\vdash} (y - 2 \cdot \$_2 = 0)\colon N_{10}.$$

Paramodulation by N_{10} to solve for y in S_1 and S_2 results in

$$S_1\colon \{2 \cdot \$_2 - 1 >= 0, \$_2 - \$_1 >= 0, \\ \underline{2\$_2 - 2\$_1 - 1 <= 0}, \$_1 >= 0, \\ z \cdot (x \uparrow (2 \cdot \$_2)) - A \uparrow B = 0\}.$$

N_3 (underlined) is normalized to $\$_2 - \$_1 <= 0$ [7], by factoring out and on the assumption that all expressions have integer values. This is combined with N_2 by r_{as} yielding $\$_2 - \$_1 = 0$, which is used in paramodulation to solve for $\$_2$. Hence, we have

$$S_1 = \{\$_1 >= 0, z \cdot (x \uparrow (2 \cdot \$_2)) - A \uparrow B = 0\}$$

Now,

$$\{N_2, E_1\} \underset{r_{ir}}{\vdash} . \qquad \square$$

Path (B:B, via arc 2):

$$VC3\colon \{y >= 0, y \neq 0, y \bmod 2 - 1 = 0, \\ z \cdot (x \uparrow y) - A \uparrow B = 0\} \\ \rightarrow \{y \div 2 >= 0, z \cdot x \uparrow (2 \cdot (y \div 2) + 1) \\ - A \uparrow B = 0\}.$$

Case 1:

$$S_1\colon \{y - 1 >= 0, y \bmod 2 - 1 = 0, \\ z \cdot (x \uparrow y) - A \uparrow B = 0\}$$

$$S_2\colon \{\sim (y \div 2 >= 0)\}.$$

The proof for this case is similar to that in Case 1 of VC2.

Case 2:

$$S_1\colon \{y - 1 >= 0, y \bmod 2 - 1 = 0, \\ z \cdot (x \uparrow y) - A \uparrow B = 0\}$$

$$S_2\colon \{\sim (z \cdot x \uparrow (2 \cdot (y \div 2) + 1) - A \uparrow B = 0)\}.$$

After replcement of "$\div$" and applying transitivity rule (as in Case 1 of VC2) S_1 and S_2 get modified to

$$S_1\colon \{y - 1 >= 0, y - 2 \cdot \$_1 >= 0, \\ y - 2 \cdot \$_1 - 1 <= 0, \$_1 >= 0, \\ y \bmod 2 - 1 = 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$

$$S_2\colon \{\sim (z \cdot x \uparrow (2 \cdot \$_1 + 1) - A \uparrow B = 0)\}.$$

Now, the function "mod" is replaced (as in Case 2, VC2)

$$S_1\colon \{y - 1 >= 0, y - 2 \cdot \$_1 >= 0, \\ y - 2 \cdot \$_1 - 1 <= 0, \$_1 >= 0, \\ \underline{y - 2 \cdot \$_2 >= 0}, \underline{y - 2 \cdot \$_2 - 1 <= 0}, \\ \underline{y - 2 \cdot \$_2 - 1 = 0}, z \cdot x \uparrow y - A \uparrow B = 0\}.$$

The underlined clauses get simplified to a single clause $y - 2 \cdot \$_2 - 1 = 0$, which is used to eliminate the occurrence of y from S_1.

$$S_1\colon \{2 \cdot \$_2 >= 0 \ (\equiv \$_2 >= 0), \\ 2 \cdot \$_2 - 2 \cdot \$_1 + 1 >= 0 \\ (\equiv \$_2 - \$_1 >= 0), \\ 2 \cdot \$_2 - 2 \cdot \$_1 <= 0 \\ (\equiv \$_2 - \$_1 <= 0), \$_1 >= 0, \\ z \cdot x \uparrow (2 \cdot \$_2 + 1) - A \uparrow B = 0\}.$$

The subsequent simplifications after substitution are shown *by the side of each clause.*

N_2, N_3 combine by r_{as} to $\$_2 - \$_1 = 0$ which is used to replace $\$_2$ from S_1 (since S_2 is free of $\$_2$).

$$S_1\colon \{\$_1 >= 0, z \cdot (x \uparrow (2 \cdot \$_1 + 1) - A \uparrow B = 0)\}$$

$$\{N_2, E_1\} \underset{r_{ir}}{\vdash} . \qquad \square$$

Path (B : C):

$$VC4: \{y >= 0, y = 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$
$$\rightarrow \{z - A \uparrow B = 0\}$$
$$S_1: \{y >= 0, y = 0, z \cdot x \uparrow y - A \uparrow B = 0\}$$
$$S_2: \{\sim (z - A \uparrow B = 0)\}.$$

N_1, N_2 combine with $y = 0$ [7], which is used to eliminate y from S_1, $S_1 = \{z - A \uparrow B = 0\}$.

$$\{E_1, N_1\} \underset{rir}{\vdash}. \qquad \square$$

All the vcs of this program are proved.

V. Conclusion

Available rules for automatic proving of theorems in different theories, namely, equality and partial ordering theory, have been significantly modified for direct application to 1) the domain of discourse, namely, the integer domain, 2) the formulas involving integer arithmetic, and 3) the normal form of the formulas chosen. The problem is decidable for linear systems in integer domain [14], but when nonlinear systems are considered the problem becomes undecidable [7]. It is not known whether the proposed verifier preserves completeness for linear systems. Instead of trying to preserve completeness for cases of linear systems, which are rarely encountered in practice, emphasis has been laid so that the verifier is capable of handling efficiently some cases of nonlinear systems as well, which occur in real programs.

The example treated in Section IV clearly indicates how proof construction proceeds in a way similar to human thought. In deducing a consequence only the relevant clauses are used. For example, in proving Case 1 of vc1 only one clause of S_1, namely, $(y - 1 >= 0)$, has been used. The other clauses of S_1 were used only in proving Case 2. Similar is the situation for all other proofs. This immediately points to the scope of easier human intervention during verification. Considering the indispensability of human element in a verification process due to the undecidability of the problem, this is an important criterion every verifier should meet.

The capability of a verifier to support terms involving "mod" and "÷," suggests that user defined functions or predicates can be incorporated in assertions, provided the functions or predicates are defined by means of axioms. Such a situation becomes unavoidable in some cases. For example, in verifying a program that computes factorial, assertions involving "factorial (x)" need to be given. Similarly, a program for finding whether x is prime or not would use "prime (x)" in the output assertion. Again, such predicates will be defined by user through axioms. Axioms, however, will give an exhaustive definition of the entity in question; only a subset of the clauses resulting from the axiom will be required for proving a theorem having some specific instantiation(s) of the entity. In the example it has been shown how the verifier can evoke proper clauses pertaining to a particular instantiation. This capability of absorbing axioms makes the verifier function at a level closer to the human element, resulting in a straightforward human interface.

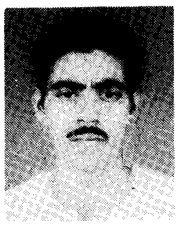
In the present work we have concentrated on the inference rules, their completeness issue, and their effectivity in verifying programs. To realize a complete theorem prover which would use these inference rules, it will be necessary to devise efficient proof-search strategies. Several heuristics have been proposed in the literature [5], [13], [21] and a proper one may be adopted depending upon its suitability on the normal form chosen.

Acknowledgment

We would like to thank the anonymous reviewers whose valuable suggestions have helped in improving the presentation.

References

[1] W. W. Bledsoe, "Non-resolution theorem proving," *Artificial Intell.*, vol. 9, no. 1, pp. 1–35, 1977.
[2] R. S. Boyer, "Locking: A restriction of resolution," Ph.D. dissertation, Univ. Texas at Austin, 1971.
[3] R. S. Boyer and J. S. Moore, *A Computational Logic*. New York: Academic, 1979.
[4] F. M. Brown, "An investigation into the goals of research in automatic theorem proving as related to mathematical reasoning," *Artificial Intell.*, vol. 14, no. 3, pp. 221–242, 1980.
[5] C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic, 1973.
[6] D. I. Good, "Toward a man-machine system for proving program correctness," Ph.D. dissertation, Univ. Wisconsin, 1970.
[7] J. C. King, "A program verifier," Ph.D. dissertation, Carnegie-Mellon Univ., 1970.
[8] —, "Proving programs to be correct," *IEEE Trans. Comput.*, vol. C-20, pp. 1331–1336, Nov. 1971.
[9] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.
[10] Z. Manna and R. Waldinger, "Deduction with relation matching," in *Proc. Fifth Conf. Foundations of Software Technology and Theoretical Computer Science*, New Delhi, (Lecture Notes in Computer Science 206), G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1985, pp. 212–224.
[11] N. V. Murray, "Completely non-clausal theorem proving," *Artificial Intell.*, vol. 18, pp. 67–85, 1982.
[12] D. A. Plaisted, "Abstraction mappings in mechanical theorem proving," in *Proc. Fifth Conf. Automated Deduction*, (Lecture Notes in Computer Science 87). New York: Springer-Verlag, 1980, pp. 264–280.
[13] —, "Theorem proving with abstraction," *Artificial Intell.*, vol. 16, no. 1, pp. 47–108, 1981.
[14] C. R. Reedy, "Automated theorem proving: New results on goal trees and Presburger arithmetic," Ph.D. dissertation, Duke Univ., 1978.
[15] J. A. Robinson, "A machine oriented logic based on resolution principle," *J. ACM*, vol. 19, pp. 23–41, 1965.
[16] —, "Generalized resolution principle," in *Machine Intelligence*, vol. 3, D. Michie, Ed. New York: American Elsevier, 1968, pp. 77–94.
[17] J. A. Robinson and L. Wos, "Paramodulation and theorem proving in first-order theories with equality," in *Machine Intelligence*, vol. 4, B. Meltzer and D. Michie, Eds. New York: American Elsevier, 1969, pp. 135–150.
[18] J. R. Slagle—"Automated theorem proving with renamable and semantic resolution," *J. ACM*, vol. 14, pp. 687–697, 1967.
[19] —, Automated theorem proving with built-in theories," *J. ACM*, vol. 19, pp. 120–135, Jan. 1972.
[20] J. R. Slagle and L. M. Norton, "Automated theorem proving for the theories of partial ordering," *Comput. J.*, vol. 18, no. 1, pp. 49–54, Feb. 1975.
[21] —, "Experiments with an automated theorem prover having partial ordering inference rules," *Commun. ACM*, vol. 16, pp. 682–688, 1973.
[22] L. Wos, J. A. Robinson, and D. F. Carson, "Efficiency and completeness of the set-of-support strategy in theorem proving," *J. ACM*, vol. 12, pp. 536–541, 1965.



D. Sarkar received the B.Tech. degree in electronics and electrical communication engineering and the M.Tech. degree in computer engineering from the Indian Institute of Technology, Kharagpur, India, in 1974 and 1977, respectively.

He is at present a Lecturer in the Department of Electronics and Electrical Communication Engineering of the same Institute, which he joined in 1981. His areas of interest are program verification, program synthesis, software reliability, and computability theory.

S. C. De Sarkar (M'80) received the M.Tech. degree in advanced electronics and the Ph.D. degree in computer science from the University of Calcutta, Calcutta, India.

From 1971 to 1977 he was with the Department of Radiophysics and Electronics, University of Calcutta, as a Lecturer. In 1977 he joined the Indian Institute of Technology, Kharagpur, India, as an Assistant Professor in the Department of Electronics and Electrical Communication Engineering. He is presently a Professor in the Department of Computer Science and Engineering of the same Institute. His current research interests include data driven architecture: computer aided instructions, software reliability, artificial intelligence, and knowledge based systems.