APPLICATION OP AUTOMATIC TRANSFORMATIONS TO PROGRAM VERIFICATION*

R. L. Veroff
Applied Mathematics Division
Argonne National Laboratory
Argonne, IL 60439

L. J. Henschen
Electrical Eng. and Computer Scl.
Northwestern University
Evanston, IL 60201

ABSTRACT

A technique for incorporating automatic transformations into processes such as the application of inference rules, subsumptlon, and demodulation provides a mechanism for improving search strategies for theorem proving problems arising from the field of program verification. The incorporation of automatic transformations into the inference process can alter the aearch space for a given problem and is particularly useful for problems having "broad" rather than "deep" proofs. The technique can also be used to permit the generation of inferences that might otherwise be blocked and to build some commutatlvlty or associativity into the unification process* Appropriate choice of transformations and new literal clashing and unification algorithms for applying them showed significant improvement on several real problems according to several distinct criteria.

INTRODUCTION

In this paper we provide a technique for improving the proof search strategy for a particular class of problems, namely, those in which the concept of transformation of a literal or term plays a large role. The technique involves incorporating automatic transformations Into the inference process by modifying the existing unification and clashing algorithms.

In addition to significantly reordering the proof search space for these problems, the incorporation of automatic transformations allows many desirable inferences that might otherwise be blocked (e.g., by demodulation or by ordering the arguments of equality literals in a canonical way).

The automatic transformation concept presented in this paper is applicable to any area that has "rewrite" .relations (e.g., commutatlvlty, associativity, or ordering relations). The new literal clashing and expanded unification algorithms are particularly effective in areas that tend to have "broad" rather than "deep" proofs. Program verification is one such area.

Various methods for building a theory into an automated theorem proving system have been considered in the literature. Most fall into one or more of three loosely defined approaches:

1. Unification in the presence of an equational theory ([1], [4], [5], [10], and [13])

   Simplification with complete set8 of reductions ([2], [3] and [4])

3. Special rules of inference ([6], [7], [8], [11] and [15])

The technique presented in this paper allows some theory to be built into the inference process but differs significantly from the above three approaches. Unlike the first approach, our technique does not attempt to find the full set of unifiers when used In the context of a unification algorithm. Unlike the second approach, the transformation concept can apply to predicates and literals as well as to terms and is not oriented towards complete sets of reductions (which, in fact, do not always exist). And unlike the third approach, the process is guided by the user's choice of eligible transformations.

We feel that it is important to reiterate our perspective and point of view. Complete methods (e.g., uniform strategies and complete sets of reductions) have many Interesting theoretical properties. Unfortunately, 15 years of experience has shown that such methods do not yield useful theorem provers by themselves. In order to actually prove theorems, it is necessary to include many heuristics that not only make the program Incomplete, but often make it too complex even to analyze. We feel that the technique presented in this paper Is important and useful in spite of the lack of completeness. Although we do prove some results about the operational characteristics of our proposals, we are not overly concerned about properties such as confluence, completeness, or minimality of the various expanded versions of our algorithms.

This pai er is divided into five sections. Section II discusses the motivation for Including automatic transformations in the Inference process. Section III presents the new literal clashing and unification algorithms. Section IV evaluates test results from the new algorithms. Finally, Section V summarises our ideas and suggests areas requiring further investigation.

It is assumed that the reader has some basic knowledge of logic, automated theorem proving and program verification. In particular, familiarity with the concepts of subsumptlon and demodulation and with the inference rules resolution and paramodulatlon is assumed.

* This work was supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W~31-iOt-Eng~38.

II. MOTIVATION

In this section we motivate our technique. Subsection A discusses the nature of program verification proofs, and Subsection B discusses built-in Incompleteness.

A. Nature of Program Verification Proofs

Program verification proofs often have a potentially useful property with respect to the graph that represents the search space. A proof often exists that is "broad" rather than "deep." That is, the last step is a hyper-resolvent of clauses CI through Cn, where each Ci is derivable from the Input clauses with a relatively low level deduction. Further, many of the CI differ from their parents only in the form or sign of a literal or term. For example, a literal LT(A,B) in an input clause may have to be transformed into ~LT(B,A) before it will clash with the denial of the theorem. This transformation is currently accomplished by resolving with one of the many clauses that give the properties of LT, EQUAL, GT, etc. Unfortunately, these axioms produce huge numbers of other resolvents at the same levels as the Ci. These clauses and their descendants often significantly delay progress to a proof.

We propose to Incorporate certain relations (such as LT(X,Y) <—> ~LT(Y,X) and commutativity of a function) into the literal clashing and unification processes. The relations will behave like literal and term transformations in this context.

The new algorithms have two distinct effects:

1. More general clauses are generated sooner by allowing the resolution of more literals. The earlier generation of more general clauses can have a significant effect on a proof search by preventing the generation of many less general clauses and their corresponding descendants.

2. Certain clauses that previously were at a high level in the graph that represents the search space may now have a relatively low level. Because many of the clauses that are required in the proof may be generated at a lower level than before, strategies that are oriented towards breadth first search may be more effective than they were without the automatic transformation process.

Note that the algorithm will not prevent the generation of what would have been the intermediate clauses to inferences made with the new algorithm; it only reorders the search space, effectively delaying the generation of these clauses. The delay of these clauses until after a proof has been found is more likely when the proof is broad rather than deep. For this reason, the algorithm is particularly well suited for application to program verification problems.

B. Built-in Incompleteness

Building automatic transformations into a theorem proving system can help undo some of the Incompleteness caused by other very useful strategies such as demodulation (when the demodulator set is not a complete reduction set) and ordering of terms in equality literals.

Although our experience shows that demodulation is virtually necessary for theorem provers, it Is rarely practical to attempt to find a complete reduction system for every new problem. Therefore, demodulating clauses can block necessary deductions. For example, If EQUAL(G(A),G(B)) were a demodulator and P(F(A,G(A))) were generated, it would be changed to P(F(A,G(B))) and would not clash with the clause ~P(F(X,G(X))).

It has been found extremely useful in practice to keep every equality literal, EQUAL(T1,T}), in a single canonical form, either EQUAL(T1,T2) or EQUAL(T2,T1) but not both. (Literals of the form ~EQUAL(T1,T2) are handled the same way). This practice can significantly reduce the size of the clause space but can, In general, lead to incompleteness in a way similar to that caused by demodulation.

The new algorithms presented in this paper overcome both of these problems, at least to some extent.

III. AUTOMATIC TRANSFORMATIONS

A transformation is an operation performed on a literal or term. We study two types of transformations: literal transformations, which correspond to resolutions with clauses containing exactly two literals, and term transformations, which correspond to demodulation with equality units. The difference between the transformation process and the usual applications of the inference rules is that transformations are applied automatically during the clashing and unification processes when the blocking properties of other operations such as demodulation and equality ordering are not in effect. Because each automatic transformation corresponds to a valid logical operation, the new clashing and unification operations are also valid inferences.

We will present the new literal clashing and unification algorithms and comment on controlling their use. We assume that the reader is familiar with the standard terminology, Including ground term, composite term, major function symbol, set, bag, and WFF (literal or term).

Definition 1. A ground subterm of a term, T, is maximal in T If it is not the subterm of any ground term other than itself.

Definition 2. COM(WFF) • number of maximal ground subterms in WFF plus the number of composite subterms (including the term itself) that are not ground. Note that In this measure of the complexity of a WFF each maximal ground term counts as one item, namely, the single domain element that the term names.

Examples:

COM(F(X,J(Y,X))) - 0 + 2 - 2
C0M(F(X,J(Y,A))) - 1 ^ - 2 - 3
C0M(F(X,J(A,B))) - 1 + 1 - 2
C0M(F(A,J(B,C))) - 1 + 0 • 1

Note that the last example Is the least complex in the above sense because it names a single constant element of the relevant domain.

Definition 3. Two literals, LI and L2, pre-claah if they have the same major function symbol and are opposite in sign. Note that two literals clash (resolve) if they pre-clash and their atoms unify*

Definition 4. A list of clauses is fully clashed if every resolvent, $C3_t$ of clauses Cl and c2 on tne list is either already on the list or is subsumed by a clause on the list.

Definition5. A list of clauses is fully paramodulated if every paramodulant, C3, of clauses C1 and C2 on the list is either already on the list or Is subsumed by a clause on the list.

Definition 6. A literal (or term), T, is transformable by a list of transformations if there exists at least one transformation, Tr, on the list such that Tr(T) = T.

A. Literal Clashing Algorithm

The basic step of resolution is the clashing of literals. The usual notion is that two literals clash (resolve) if they are opposite in sign and have a common instance. The new notion is that two literals claah if there are transformed versions of the literals that clash in the usual sense.

The spirit of the new concept is to automate the "obvious** transformations by incorporating them into the literal clashing process. The "obvious** transformations would include relatione that are in some sense rewrite rules such as LT(X,Y) —> ~LT(Y,X), where LT represents the "less than" relation. The allowable transformations have a very restricted form which we describe below. These restrictions are heuristic in nature and result in a more efficient and effective theorem prover than would otherwise be possible. We distinguish these restricted transformations from transformations that have more deductive power, such as LT(X,Y) —> LT(X,S(Y)), where S(X) stands for the successor of X. The distinction is informal and clearly subject to interpretation.

A literal transforming clause (transformation) is a clause with exactly two literals, LI and L2. The mechanism for applying these transformations is resolution. Note that each such clause in fact represents two transformations, -LI —> L2 and H-2 —> LI.

To be of any value, a transformation clause clearly must be deduclble from the clause space representing the problem to be solved. We assume that this property is satisfied in all further discussions of transformations.

For ease and efficiency we use two distinct lists of tranaformations:

LCLASHI -	those that change sign and/or major function symbol
LCLASH2 -	those that permute arguments

Restricting the set of transformations that can first be applied to those that change sign and/or predicate symbol of a literal provides an efficient sieve for literals that are not clashable. That Is, no attempt will ever be made to unify the atoms of two literals unless they have transformed versions that pre-clash. Although having a single fully clashed list of transformations would lend Itself to a very simple algorithm for applying the transformations, we feel that the trade-off between the computational efficiency of having two Hats and the simple organisation of the algorithm justifies having the two lists.

We also impose several restrictions as follows:

1. In applying transformation LI L2 to a literal L, no substitutions may be made to L Itself.

2. LI and L2 contain exactly the same set of variables.

3. The major function symbols of LI and L2 have exactly the same number of arguments.

4. COM(LI) - C0M(L2)

5. LCLASH2 is fully clashed. Also, the union of LCLASHI and LCLASH2 is fully clashed subject to the qualifications:

   a. Tautologies generated by resolving clauses in LCLASHI need not be added.

   b. After the union of lists LCLASHI and LCLASH2 is fully clashed, consider the set of all transformations, LI L2, on LCLASHI such that exactly one of the literals, LI or L2, Is transformable by LCLASH2. If there are two transformations in this subset that differ only by a single application of a transformation in LCLASH2 (that is, the clauses are permutation variants of each other where the permutation is an eligible transformation), then only one of the transformations need be kept on LCLASHI. Keeping all such permutation variants will not affect the results of the algorithm but might cause some unnecessary duplication of work.

Although omitting the qualification to restriction 5 would result In lists with nice theoretical properties (see Lemma 1 below), removal of redundant and Ineffective transformations is consistent with the goal of keeping the set of applicable transformations small.

Note, too, that restriction 5 is not prohibitive if the number of clauses Involved is small. In particular, note that the resulting set will be finite because of the restrictions placed on the complexity of the transformations (for example, -P(X) P(F(X)) is not allowed).

A transformation (set of transformations) is not eligible If it indirectly violates the above restrictions, even if the transformation clause Itself Is eligible. For example, the pair of transformation clauses --Q(A) R(A) and -*(A) Q(B) would not be eligible because the fully clashed property would require the transformation clause -*Q(A) Q(B) to be present. This transformation clause Is not eligible because It does not correspond to a transformation that changes sign and/or predicate symbol (for LCLASHI) or to a transformation that permutes arguments of a literal (for LCLASH2).

Most of the restrictions above put limits on the set of transformations and the way they can apply. A few however, like the fully clashed properties, make the operation of the algorithm simple and efficient in that at most two transformation steps will be required on any literal, one from LCLASHI and one from LCLASH2. Although these restrictions may seem to make the set of transformations on lists LCLASH1 and LCLASH2 very complicated, most of the restrictions are necessary only to cover special cases that will not commonly arise in practice. On one set of real problems that was tested (see Section IV), the entire set of transformations consisted of the following:

~LT(X,Y) ~LT(Y,X)

-LT(X,Y) -EQUAL(X.Y)

~LT(X,NUM1) -IB(CC.X)

HLT(CN.X) ~IB(CC,X)

HEQUAL(X,Y) EQUAL(Y,X)

HBQUALARR(X,Y) EQUALARR(Y,X)

These clauses were formulated under an interpretation in which IB(X,Y) represents the fact that index Y is in bounds for array X; CC is an array; NUM1 and CN are Integer constants; and EQUALARR represents equality between arrays.

The following lemmas and theorems help motivate an algorithm that effectively makes use of the lists of transformation clauses defined above. The two theorems illustrate the trade-off between techniques that can be shown to have nice theoretical properties and those that are useful in practice. Theorem 1 characterizes the literal clashing properties of the lists iXLASHI and LCLASH2 when transformations can be applied without a substitution restriction (e.g., restriction 1 above). Theorem 2 characterizes the transformation properties of the lists when the substitution restriction is In effect.

Note that the identity transformation (represented by tautologies) is implicitly (but not explicitly) in every set of transformations (clauses). That is, while the reference to the existence of a transformation with certain properties includes the possibility of the identity transformation, the reference to literals that are transformable by a certain set does not.

Notation:

C I— c if clause c is deducible from clause space C with ordinary resolution (without a substitution restriction).

a |—> b with respect to a set of clauses C if unit clause b is deducible from unit clause a with a single ordinary resolution step.

Lemma 1. Let a and b be literals treated as unit clauses. Let C be a set of transformation clauses (exactly two literals) that is fully clashed. If the conjunction of b and C is satlsflable, but the conjunction of a, b, and C is unsatlsfiable, then a I—> ~b' with respect to C, where b and ~b' clash.

Proof. Because a must clearly participate in the derivation of the empty clause, the fully clashed property implies that if a and C I——b', where b and -b' clash, then a I—> M>'. The lemma then follows from Corollary 3 on page 539 of [9].

Let C above be partitioned into two sets, CI and C2, such that CI consists of those clauses which (when thought of as transformations) change sign and/or predicate symbol, and C2 those clauses that remain. Now replace CI with CI' which 'is constructed from CI as follows: For each clause in CI, add the clause to CI' unless it is the resolvent of a clause In C2 and a clause already in CI^1 and exactly one of its literals is transformable by C2. In other words, the omitted clauses are clauses that can be derived by applying a transformation from C2 to one of the literals of a transformation clause In CI'.

Note that CI' may not be uniquely determined by CI. The order that the clauses are inspected may determine which clauses in CI are omitted from CI'. This has no bearing on the lemmas and theorems that follow.

Lemma 2. Let al, a2, ... an be unit clauses such TTial ST* |—> a2 |—> ... |— > an with respect to CI' and C2. At least one of the following must hold:

(1) There exists a unit clause, b, such that al I—> b with respect to CI' and b |—> an with respect to C2.

(2) There exists a unit clause, b, such that al I—> b with respect to C2 and b I—> an with respect to CI'.

Proof. Because C is fully clashed, al I—> an with respect to C. Let c be the clause in C that resolves with al to produce an. If c is in either C2 or CI', then there is nothing to show because the identity transformation Is implicitly in CI' (b - al) and C2 (b - an). If c is not in C2 or CI', then it must be the resolvent of a clause in CI' with a clause in C2, and the lemma follows.

Lemma 3. Let al, a2, ... an be as in Lemma 2. If "ail Is transformable by C2, then outcome (1) of Lemma 2 holds.

Proof. Because C is fully clashed and CI and C2 partition C, it follows that al |—> an with respect to either CI or C2. If al and an have the same sign and predicate symbol, then al I—> an with respect to C2. In this case both (1) and (2) of Lemma 2 hold because the relevant transformation from CI' is the identity transformation.

If al and an differ in sign and/or predicate symbol, then al I—> an with respect to CI. If both al and an can clash against clauses in C2, then al I—> an with respect to CI' because CI' contains all clauses from CI In which both literals are transformable by C2. In this case, both (1) and (2) hold as the relevant clause from C2 is the identity transformation. If al cannot clash against any clauses in C2, then (2) cannot hold unless the relevant clause from C2 is the identity transformation. The fact that outcome (1) must hold then follows from Lemma 2.

First, assume that the transformation process uses ordinary resolution. That is, substitution is not restricted to the transformation clauses.

Theorem 1. Consider the conjunction of the clauses Tn——LCLASH1, the clauses in LCLASH2, and two literals, LITERALA and LITERALB as unit clauses. If the resulting clause space Is unsatlsflable but is satlsflable without either of the two literals, then there exists a transformation, Trl, from LCLASH1 and a transformation, Tr2, from LCLASH2 such that either Tr2(Trl(LITERALA)) clashes with LITERALB or Tr2(Trl(LITERALB)) clashes with LITERALA. In particular, the following hold true:

(1) If LITERAL* is transformable by LCLASH2, then there exist Trl and Tr2 such that Tr2(Trl(LITERALS)) clashes with LITERALA.

(2) If LITERALB is transformable by LCLASH2, then there exist Trl and Tr2 such that Tr2(Trl(LITERALA)) clashes with LITERALB.

(3) If neither LITERALA nor LITERALB is transformable by LCLASH2, then there exist Trl and Tr2 such that Tr2(Trl(LITERALB)) clashes with LITERALA.

Proof. Recall that LCLASHI consists only of transformations that change sign and/or predicate symbol and that LCLASH2 consists only of transformations that permute arguments of a literal. Recall also that LCLASH2 is fully clashed, and that the conjunction of the two lists is fully clashed up to deletion of tautologies and clauses that can be derived by the resolution of a clause on LCLASHI with a clause on LCLASH2.

Because the clause space is unsatlsflable, there must exist a sequence of unit (single literal) clauses aO, al, ... an such that LITERALB - aO |— > al |— > a2 |— > ... |— > an - M-ITERALA* where -LITERALA' and LITERALA clash (see [9]). Similarly, there must exist a sequence of unit clauses bO, bl$_t$ ... bm such that LITERALA - bO |--> bl I— > b2 I— > ... I—> bm - -LITERALB' where -LITERALB' and LITERALB clash.

The first two parts of the theorem follow immediately from Lemma 2 and Lemma 3. The third part follows from Lemma 2 and the observation that outcome (1) of Lemma 2 must hold when al of Lemma 2 is not transformable by C2.

Now, consider the transformation process as defined originally. That is, substitution is now allowed only into the transformation clauses themselves.

Notation:

Let a and b be literals.

a —> b if there exists a transformation Tr on either LCLASHI or LCLASH2 such that Tr(a) - b.

a -l-> b (a -2-> b) if there exists a transformation Tr on LCLASHI (LCLASH2) such that Tr(a) - b.

a -(k)-> b if there exists al, a2, ... ak such that a —> al —> «2 —> ... —> ak - b. (If k - 0 then a - b.)

a -(*)-> b if a -(k)-> b for some k >_ 0.

Lemma 4. a -l-> b if and only if -b ▪l-> (similarly for LCLASH2).

Proof, a -l-> b if and only if there exists a transformation clause, LI L2, and a substitution, S, such that L1(S) - -a and L2(S) - b (because substitutions are allowed only into the transformation clauses).

Theorem 2. a -(*)-> b implies that there exists a' "sucE ETiaT either a -l-> a' -2-> b or -b -l-> a' -2-> -a.

Proof. This proof follows from Lemma 2 and Lemma 4 where al and an in Lemma 2 are thought of as ground literals (so no substitutions are possible).

Theorem 1 and Theorem 2 imply that although it is sufficient to transform only one literal when attempting to clash two literals with transformations, it may be necessary to choose the appropriate literal to transform. Theorem 1, in effect, says that without the substitution restriction the choice is based on a simple inspection of the transformations on LCLASH2. Theorem 2, which refers only to the transformation process Itself and not to the underlying goal of clashing literals, gives no information about making the choice.

This difficulty can be overcome in a program by attempting transformations in both directions. Alternatively, a program might attempt to analyse the two literals and the set of transformations. In our program, however, we have found it adequate to choose, by simple inspection as above, one literal to transform.

algorithm

The algorithm used in our program is as follows: Let LITERAL1 and LITERAL2 be the literals to clash.

STEP 1: Choose which literal to transform.

If LITERAL2 is transformable by LCLASH2, then transform LITERAL1, else transform LITERAL2.

Let LITERALB be the literal chosen to be transformed, and let LITERALA be the literal that remains unchanged.

STEP 2: Find a transformation from LCLASHI to apply to LITERALB to make it pre-clash with LITERALA.

STEP 3: For each pre-clashable pair, find a transformation on LCLASH2 that makes the literals unifiable.

In our program, we halt when the first clash is found rather than finding all possible clashes. It follows that the completeness property is sacrificed unless transformations are available for normal inference. Because the automatic transformation concept was designed for performance in an applied environment, however, completeness is not of large concern. Also, because many clauses that participate in program verification proofs are ground (and so can clash in at most one way) [12], the first clash is very often the only clash.

It is important to note that the new literal clashing algorithm is not a "pre*theorem prover." That is, it is not the case that the algorithm corresponds to using the theorem prover (or theorem prover search strategies) to find a proof that two literals are inconsistent. The process is a direct and finite search through the two lists, LCLASHI and LCLASH2.

Let m be the number of transformations on LCLASHI that can apply to LITERALB, and let n be the number of transformations on LCLASH2 that can apply to the major function symbol of LITERALA. It follows that at most mn transformations can be applied in the algorithm to test the clash of LITERAL1 and LITERAL2. In general, m and n will be small. This fact is important because each application of a transformation requires a unification test, which can significantly add to the cost of the algorithm.

It is important to choose an effective transformation set. Some transformations can cause unnecessary redundancies and inefficiencies. For example, it might be better to have the unit clauses Q(A,B) and Q(B,A) both in the clause space than to have the single unit clause Q(A,B) and the transformation clause -Q(X,Y) Q(Y,X), which might apply at many unnecessary places.

B" Expanded Unification Algorithm

A similar transformation process has been developed for unification of terms using equality literals EQUAL(T1,T2) in place of transformation clauses and paramodulation in place of resolution (again with a substitution restriction). Although ve consider the concept of literal transformations to be the most useful and important proposal In this paper, we present the following for two reasons. First, there are some similarities but also some Important differences between the transformation of literals and terms. And second, the heuristics cited below may help offset some of the impractical aspects of various complete unification systems that build in siroplifiers.

We again partition the transformations into two lists, UNIFY1 and UNIFY2, that change the major function symbol of a term and permute arguments, respectively. We have the same restrictions on the complexity of terms and variable substitutions that we had In the literal clashing algorithm. In addition, we have the following restrictions:

6. The bag8 of variables in Tl and T2 are identical.

7. Tl (T2) is not a proper subterm of T2 (Tl).

8. The set of transformations (ideally) should be fully paramodulated.

Practical considerations limit the application of last property. For example, the pair of clauses EQUAL(F(X,Y),F(Y,X)) and EQUAL(F(X,F(Y,Z)),F(F(X,Y),Z)) can generate an infinite set of eligible transformations. Although the elimination of any such transformations can cause blocks in the expanded unification algorithm given below, it is reasonable to restrict the list to a small set of the most simple and roost general transformations.

Note that the fully clashed property implies that all instances of application of transitivity of equality will be present. That is, if EQUAL(T1,T2) and EQUAL(T2,T3) are eligible transformations, then EQUAL(T1,T3) must be an eligible transformation.

Note also that the requirement COM(Tl) - C0M(T2) and restrictions 6 and 7 prevent infinite sequences of expanding transformations such as A —> F(F(A)) —> ....

The functional reflexivity axioms (instances of EQUAL(X.X)), which act as identity transformations, will be assumed to be implicitly on all lists, but need not be explicitly present. This assumption corresponds to the assumption about tautologies in the discussion of the literal clashing algorithm.

Notation:

Let r and s be terms*

r —> s if there exists a transformation Tr on either UNIPYl or UNIFY2 such that Tr(r) - s.

Theorem 3. r —> 8 if and only if s —> r.

Proof, r —> s if and only if there exists a transformation clause, EQUAL(T1,T2), and a substitution, S, such that T1(S) - r and T2(S) ■ s (because substitutions are allowed only into the transformation clauses).

The following theorem helps justify the recursive orientation of the expanded unification algorithm. In general, it is possible that the transformation of a term, T, might be blocked unless some transformation is first applied to a proper subterm of T. The theorem shows that this problem does not arise within the context of the expanded unification algorithm.

Theorem 4. Let T be a term with proper subterm R, "IH3 let Trl and Tr2 be two transformations represented by transformation clauses, EQUAL(T1,T2) and EQUAL(T3,T4), respectively. Assume that Trl and Tr2 are on a list that is fully paramodulated. If T* is the term that is generated by substituting R with Trl(R) in T, and T'• - Tr2(T^f), then there exists a transformation, Tr3, such that Tr3(T) subsumes T^{1}'.

Proof. The fact that Trl is applicable to R implies that there exists a substitution, Sl, such that either T1(S1) - R or T2(S1) - R. Without loss of generality, assume that Ti(Si) - R. Then T' has subterm T2(S1). The fact that Tr2 is applicable to T* implies that there exists a substitution, S2, such that either T3(S2) - T^f or T4(S2) - T*. Without loss of generality, assume that T3(S2) - T'. Now, because T3(S2) has subterm, T2(S1), it follows that EQUAL(T3*,T4) is a paramodulant of some instances of EQUAL(T1,T2) and EQUAL(T3,T4), where T3' is the result of replacing the subterm, T2(S1), in T3(S2) with T1(S1). Because the list of transformations is fully paramodulated (and the functional reflexivity axioms are implicitly present), Tr3 is the transformation that Is represented by either clause, EQUAL(T3*,T4), or by a clause that subsumes EQUAL(T3*,T4).

Because the fully paramodulated property accounts for applications of transitivity of equality, at any point in the expanded unification algorithm, it suffices to apply at most one transformation to a term.

algorithm

The new unification algorithm can be described as a simple recursive process. At the outer level it is similar to the literal clashing algorithm in that it first uses the transformations on UNIFY1 to match major function symbols and then uses the the transformations on UN1FY2 to get unlfiable sequences of arguments. The recursion comes to play, of course, when unification is to be applied to each pair of (permuted) arguments.

Term transformations can be Incorporated into any process that uses unification including clashing, demodulation and subsumption. Our experience is that these transformations will have less impact in certain applications, such as program verification, than literal transformations.

XV. TESTING AMD EVALUATION

Two sets of problems were used to test the new literal clashing algorithm with the expanded unification algorithm included as the unification step:

1. Eleven real problems currently being tested by B. T. Smith on the environmental theorem proving system [12]

2. Slight variations of the eleven probles

The second set of problems was designed to help characterise the conditions in which the new literal clashing algorithm has the most favorable effects on the proof search space.

The eleven real problems were tested with various search strategies, including those most commonly used by B. T. Smith [12J. The problems run with transformations and those run without transformations will be referred to as the "trans" and "notrans" versions, respectively. The following observations have been made:

Finding proofs; In no case did the notrans version find a proof when the trans version did not. In one case the trans version found a proof when the notrans version did not.

Total number of clauses In search space (in cases In which a proof was found): The notrans version tended to have fewer clauses than the trans version. This result is reasonable because the transformation process has the effect of producing more clauses at earlier levels In the graph that represents the search space. Because all of the search strategies used have some element of breadth first search in them, the general effect of the transformation process is a net increase in the total number of clauses added to the clause space. There were isolated cases in which the trans version actually had up to 40 percent fewer clauses than the notrans version, and a few cases in which the trans version had as many as 40 percent more clauses than the notrans version, but on the average, the trans version produced approximately 18 percent more clauses than the notrans version.

Number of clauses that participate In proof: The value for the trans version never exceeded the value for the notrans version. The differences ranged as high as six clauses (10 percent).

Number of clauses selected by the search strategy: The value for the trans version exceeded the value for the notrans version in only one case. The single Increased value was from 13 to 14 clauses (less than 8 percent). The decreased values ranged up to over 50 percent (9 to 4 clauses).

Number of selected clauses that participate in the proof: The value for the trans version never exceeded the value for the notrans version, Reductions ranged up to 40 percent.

Depth of empty clause: The value for the trans version was less than or equal to the value for the notrans version in all but one of the problems tested, with reductions of up to 50 percent. In the one exception, the trans version Increased the depth of the notrans version from 2 to 3. This behavior is possible because there can be more than one proof to a problem, and because the search strategies used are not exactly breadth first searches. In the problem In which the depth was higher in the trans version than in the notrans version, a longer path to the empty clause (deeper proof) was found before the shorter path (less deep proof) was discovered.

In a second set of experiments, noise (extraneous literals and complications of terms) was added to the real problems. The negative effect of adding noise was consistently and significantly worse in the problems run without transformations than in the problems run with transformations.

SUMMARY

The modified real problems tested above indicate that the automatic transformation concept does have the potential to become a powerful extension to a resolution-based automated theorem proving system. Except for isolated cases program verification problems run with automatic transformations did no worse than problems run without transformations, and sometimes, the performance was significantly better.

The results, however, were not as promising as expected. This suggests that although the concept may be quite useful, better search strategies putting more emphasis on breadth first search might be developed to capitalise on the power of the transformation process. At present the user must pick a subset of transformation clauses to become automatic transformations. Work needs to be done to provide general rules for making these choices.

Various modifications to the new literal clashing and expanded unification algorithms might also be investigated, including relaxation of some of the rules for eligibility of transformations and r<les for applying transformations. In particular, relaxation of the rule about restricted substitution into the terms and/or literals being transformed could significantly affect the resulting clause space. Finally, the incorporation of the automatic transformation concept into other areas such *as* demodulation or subsumptlon might be investigated.

REFERENCES

[1] Fay, M., "First order unification in an equatlonal theory," 1979 Deduction Workshop Proceedings, Univ. of Texas, Austin, Texas.

[2] Gloess, P. and Laurent, J., "Adding dynamic paramodilation to rewrite algorithms, 5th Conference on Automated Deduction, Vol. 87,

Lecture Notes in Computer Science, ed. W. Bibel an3 FT. kawalskl, Springer-Verlag, Berlin, 1980, pp. 195-207.

[3] Knuth, D.E. and Bendix, P.G., "Simple word problems in universal algebras", J. Leech (ed.), COMPUTATIONAL PROBLEMS IN ABSTRACT ALGEBRAS, Pergamon Press, New York, 1970, pp. 263-297.

[4] Livesay, M., Seikmann, J., Szabo, P., and Unvericht, E., "Unification problems for combinations of associativity, commutativity, distrlbutlvity and idempotence axioms," unpublished report.

[5] Plotkin, G.D., "Building-in equational theories," Machine Intelligence 7, B. Meltzer and D. Michie, Editors, 1972, pp. 73-90.

(6] Overbeek, R.A., McCharen, J., and Wos, L., "Complexity and related enhancements for automated theorem-proving programs," Computers and Mathematics with Applications, Vol. 2, 1976, pp. 1-16.

[7] Robinson, G.A. and Wos, L., "Paramodulatlon and theorem-proving in first-order theories with equality," Machine Intelligence, Vol. IV, 1969, pp. 135-150.

[8] Robinson, J.A., "Automatic deduction with hyper-resolution," International Journal of Comput. Math., Vol. 1, 1965, pp. 227-234.

[9] Slagle, J.R., "Interpolation theorems for resolution in lower predicate calculus," JACM, Vol. 17, 1970, pp. 535-542.

[10] Slagle, J.R., "Automated theorem-proving for theories with simplifiers, commutativity, and associativity," JACM, Vol. 21, 1974, pp. 622-642.

(11) Slagle, J.R., "Automatic theorem proving with built-in theories including equality, partial ordering, and sets," JACM, Vol. 19, 1972, pp. 120-135.

(12) Smith, B.T., "Reference manual for the environmental theorem prover", to be published as an Argonne National Laboratory technical report.

(13) Stickel, M.E., "A complete unification algorithm for associative-commutative functions," Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, 1975, pp. 71-76.

[14] Veroff, R., "Automatic transformations in the inference process," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Northwestern University, June 1980.

[15] Wos, L., Overbeek, R.A., and Henschen, L.J., "Hyper-paramodulation: a refinement of paramodulatlon," 5th Conference on Automated Deduction, Vol. ¥77 Lecture dotes in Computer Science, ed. W. BilJel *and* IT ka wax ski, Springer-Verlag, Berlin, 1980, pp. 208-219.