Distribution Category:
Mathematics and Computers
(UC-32)

ANL-80-75

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

AUTOMATIC TRANSFORMATIONS IN THE INFERENCE PROCESS

by

Robert L. Veroff

Applied Mathematics Division

Based on a thesis prepared for
the degree Doctor of Philosophy
in Electrical Engineering and Computer Science
from Northwestern University

July 1980

TABLE OF CONTENTS

FIGURE

AUTOMATIC TRANSFORMATIONS IN THE INFERENCE PROCESS

by

Robert L. Veroff

ABSTRACT

A technique for incorporating automatic transformations into processes such as the application of inference rules, subsumption, 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 search 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 commutativity 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.

I. INTRODUCTION

Section A contains a review of the general problem area addressed by this thesis. Sections B and C discuss, respectively, the goals and relevance of this work to the field. Section D contains a brief survey of related work.

I.A.1. REVIEW OF RESOLUTION-BASED FIRST-ORDER THEOREM PROVING

This section provides a brief and informal review of resolution-based first-order theorem proving. In particular, it is oriented towards the Argonne National Laboratory Northern Illinois University (ANL-NIU) automated theorem proving system ([10], [17]). See [1], [10], [11], [12], [13], [14], [19], [20], [21], and [22] for a more thorough treatment of the material.

I.A.1.a. REPRESENTATION OF FACTS

Facts are represented in the ANL-NIU theorem proving system in a manner which is consistent with first-order predicate calculus.

DEFINITIONS

The basic symbols of the language of representation are:

1. a set of symbols called CONSTANTS,
2. a set of symbols called VARIABLES,

3. a set of symbols called FUNCTIONS,

4. a set of symbols called PREDICATES,

and 5. a single symbol called NEGATION.

The following definitions characterize the language of representation:

Definition 1.

A TERM is a constant, a variable, or any F(t1,t2,...tn), where F is an n-ary function symbol and the ti are terms.

Definition 2.

An ATOM is P(t1,t2,...tn), where P is an n-ary relation (predicate) symbol and the ti are terms. Note that atoms are the basic true/false items.

Definition 3.

A LITERAL is an atom or the negation of an atom.

Definition 4.

A CLAUSE is a disjunction of literals.

Definition 5.

A CLAUSE SPACE is a conjunction of clauses.

The following definition associates syntax and semantics.

Definition 6.

An INTERPRETATION consists of a non-empty domain D and an assignment of constant, function, and predicate symbols to fixed elements, functions, and relations on D. Variable symbols can be assigned arbitrary values from D.

To facilitate the implementation of paramodulation and demodulation (see Sections I.A.1.c and I.A.1.d of this chapter), the string of symbols 'EQUAL' has been reserved in the ANL-NIU theorem proving system. All predicate names beginning with this string represent some equality relation. Other than this one exception, the symbols used to represent facts have no inherent semantic meaning to the theorem proving system. In this sense the theorem prover is completely general. That is, for any set of clauses, any interpretation which is semantically consistent with the clauses is valid.

Note that existential quantifiers are replaced by Skolem functions in clause space representation [1].

Example: Replacing Existential Quantifiers with Skolem Functions

The expression (EXISTS x)(ALL y)(y+x=y) (existence of right additive identity in a group) can be represented in clause form as EQUAL(SUM(Y,E),Y) under an interpretation in which SUM denotes addition in a group, EQUAL denotes the equality predicate, E is a Skolem constant (replacing the existential quantifier) denoting the (right) additive identity, and the domain is the set of elements of a group.

Similarly, the expression (ALL x)(EXISTS y)(x+y=E) (existence of right additive inverse) can be represented in clause form as EQUAL(SUM(X,INV(X)),E) under an interpretation in which INV(X) denotes the (right) additive inverse of X and is a Skolem function of one argument. Note that the existential y is dependent on the choice of x.

By convention, all variables in a clause space are universally quantified and have scope restricted to the conjunct containing them. That is, while two variables with the same name occurring in the same clause must represent the same element, two variables with the same name occurring in different clauses are distinct. It follows that clause spaces are free of all explicit quantifiers.

Since clause form is clearly identical to (quantifier free) conjunctive normal form, it follows that a clause space is sufficient to represent an arbitrary expression in first-order predicate calculus [1].

In addition to the above definitions, the following naming conventions have been adopted in this paper for uniform representation of clause spaces: Names beginning with the letters A, B, C, D, and E are used to represent constants. Names beginning with the letters F, G, H, ..., T, and U are used to represent functions and predicates. Names beginning with the letters V, W, X, Y, and Z are used to represent variables. Blanks separate the literals of a clause, with the disjunction operator implied between them. And finally, clauses are written on separate lines, with the conjunction operator implied between them.

<u>Example:</u> Clause Space

The clause space,

```
LT(X,Y)  LT(Y,X)  EQUAL(X,Y)
¬LT(X,Y)  ¬LT(Y,X)
```

represents the expression, (ALL x,y)(x<y OR y<x OR x=y) AND (ALL x,y)(NOT x<y OR NOT y<x), under an interpretation in which LT represents the < relation and EQUAL represents the = relation.

I.A.1.b. FORMULATION OF PROBLEMS

The clause space representing a theorem to be proved by the ANL-NIU system may be considered to consist of the union of three sets of clauses: a set of axioms defining the field of study, the special hypothesis or the theorem to be proved, and the denial of the theorem.

This representation of the problem corresponds to finding a proof by contradiction. That is, the theorem to be proved is valid if and only if the conjunction of the axiom set, the special hypothesis, and the denial of the theorem forms an unsatisfiable clause space.

Example:

Consider the theorem: In a ring, if for all X, $X^3=X$, then the ring is commutative.

For this problem, the axioms would be the set of axioms that define a ring, the special hypothesis would be $X^3=X$, and the denial of the theorem would be AB ¬= BA (that is, there exist two elements, A and B, that do not commute).

The set of axioms that defines a particular area of study is not unique. In fact, the choice of an appropriate representation for the axioms can have a significant effect on the ability of the theorem proving system to find an inconsistency in the clause space. Note the two different representations of the ring axioms given in the examples below.

<u>Example:</u> Ring Axioms - Equality Formulation

The following clauses define a ring under an interpretation in which EQUAL(X,Y) denotes X=Y (equality predicate), SUM(X,Y) denotes X+Y (addition) in a ring, PROD(X,Y) denotes XY (multiplication) in a ring, INV(X) denotes -X (additive inverse) in a ring, and E denotes the ring identity:

```
EQUAL(SUM(X,Y),SUM(Y,X))
EQUAL(SUM(SUM(X,Y),Z),SUM(X,SUM(Y,Z)))
EQUAL(PROD(PROD(X,Y),Z),PROD(X,PROD(Y,Z)))
EQUAL(PROD(SUM(X,Y),Z),SUM(PROD(X,Z),PROD(Y,Z)))
EQUAL(PROD(X,SUM(Y,Z)),SUM(PROD(X,Y),PROD(X,Z)))
EQUAL(SUM(X,E),X)
EQUAL(SUM(E,X),X)
EQUAL(SUM(X,INV(X)),E)
EQUAL(SUM(INV(X),X),E)
EQUAL(X,X)
```

Note that the first nine clauses represent the axioms: commutativity of addition, associativity of addition and multiplication, right and left distributivity, right and left identity, and right and left inverse respectively. The clause, EQUAL(X,X), is added for closure and to define equality.

<u>Example:</u> Ring Axioms - Equality De-emphasized

The following clauses define a ring under an interpretation in which P(X,Y,Z) denotes XY=Z in a ring, S(X,Y,Z) denotes X+Y=Z in a ring, and the remaining symtols are defined as in the example above:

```
¬S(X,Y,Z)  S(Y,X,Z)
```

```
¬S(X,Y,V0) ¬S(Y,Z,V1) ¬S(X,V1,W)  S(V0,Z,W)

¬S(X,Y,V0) ¬S(Y,Z,V1) ¬S(V0,Z,W)  S(X,V1,W)

¬P(X,Y,V0) ¬P(Y,Z,V1) ¬P(X,V1,W)  P(V0,Z,W)

¬P(X,Y,V0) ¬P(Y,Z,V1) ¬P(V0,Z,W)  P(X,V1,W)

¬P(X,Y,V0) ¬P(X,Z,V1) ¬S(Y,Z,V2) ¬P(X,V2,W)  S(V0,V1,W)

¬P(X,Y,V0) ¬P(X,Z,V1) ¬S(Y,Z,V2) ¬S(V0,V1,W)  P(X,V2,W)

¬P(Y,X,V0) ¬P(Z,X,V1) ¬S(Y,Z,V2) ¬P(V2,X,W)  S(V0,V1,W)

¬P(Y,X,V0) ¬P(Z,X,V1) ¬S(Y,Z,V2) ¬S(V0,V1,W)  P(V2,X,W)

 S(X,E,X)

 S(E,X,X)

 S(X,INV(X),E)

 S(INV(X),X,E)

 S(X,Y,SUM(X,Y))

 P(X,Y,PROD(X,Y))
```

Note that these clauses represent the axioms: commutativity of addition, associativity of addition (two clauses), associativity of multiplication (two clauses), distributivity (four clauses), right and left identity, right and left inverse, and closure of addition and multiplication respectively.

The equality emphasized formulation is more function (term) oriented while the equality de-emphasized formulation is more predicate (literal) oriented. The question of the relative merits of the two forms of representation (function versus predicate orientation) is a source of controversy in the field of automated theorem proving.

I.A.1.c. INFERENCE RULES

Inference rules provide the mechanism for manipulating and expanding the clause space. That is, they provide a way to derive new facts (clauses) from the existing set of clauses.

Definition 7.

An inference rule is VALID if it cannot be used to generate an unsatisfiable set of clauses from a satisfiable set of clauses. That is, it cannot derive false statements from true statements.

We are only interested in inference rules that are valid. In particular, we are interested in the valid inference rules, resolution and paramodulation.

Definition 8.

An inference rule is REFUTATION COMPLETE if for every E-unsatisfiable set of clauses (unsatisfiable in extended first-order predicate calculus including equality), there exists a way to find a contradiction using only the one inference rule. An inference system (a set of inference rules that can be applied) is REFUTATION COMPLETE if for every E-unsatisfiable set of clauses, there exists a way to find a contradiction using only the inference rules in the system.

For the remainder of this paper, 'completeness' will be synonymous with refutation completeness, and 'inconsistency' will be synonymous with

E-unsatisfiability.

We are only interested in inference systems that are complete. When defined with factoring (see [20]), resolution alone, and paramodulation used in conjunction with resolution are complete inference systems.

A thorough introduction to these rules of inference is given in [1], [11] and [12].

For a brief review, consider the following:

Definition 9.

A SUBSTITUTION is an assignment of terms to a set of variables.

Definition 10.

The UNIFICATION of two terms is the process by which a substitution for the variables in the terms is found that makes the terms identical. That is, a common instance of the terms is found (if one exists) [12]. Note that two atoms are considered to be unified if their predicate symbols are identical and a substitution for the variables is found that makes the atoms identical.

Example: Resolution

Consider the clause space,

(1) $\neg P(X)$ $Q(A,X)$

(2) $P(B)$ $R(C)$

(3) $Q(A,B)$ $R(C)$

The first two clauses resolve to generate the third, because under the variable substitution, $X \leftarrow B$ in clause (1), the $\neg P(X)$ of (1) and the $P(B)$ of (2) clash (are the same atom with opposite sign).

Note that a proof to a theorem is found by deriving the EMPTY CLAUSE, the clause with no literals. That is, finding a contradiction in a clause space corresponds to resolving all of the literals of a clause without adding any literals. For example, the clause EQUAL(A,B) and the clause ¬EQUAL(A,B) form a contradiction (resolve to derive the empty clause).

Example: Paramodulation - Equality Substitution

Consider the clause space,

(1) Q(G(F(A,B)))

(2) EQUAL(F(X,Y),F(Y,X))

(3) Q(G(F(B,A)))

The second clause can be paramodulated into the term, F(A,B), of the first clause under the variable substitution, X <-- A and Y <-- B in clause (2), to generate the paramodulant clause (3). That is, an instance of the term, F(X,Y), is replaced by the corresponding instance of the term, F(Y,X).

An extension to resolution, hyper-resolution, has been developed [13]. One hyper-resolution step can combine several resolution steps to produce a clause with no negative literals. Hyper-resolution has been found to be an effective inference rule [8], and is commonly used in resolution-based theorem proving systems. A similar extension to paramodulation, hyper-paramodulation, is currently being studied [22].

I.A.1.d. DIFFICULTIES / SPECIAL FEATURES

There are general difficulties with resolution and paramodulation based theorem proving systems. In a simple combinatorial sense, it is clear that the larger the clause space, the more searching that might be done. In particular, the more 'useless' clauses, clauses that do not participate in a proof, that there are in the clause space, the more searching and expanding of the clause space that must be done to generate 'useful' clauses. It is clearly beneficial to keep the number of 'useless' clauses down to a minimum. In particular, redundant information should be eliminated.

The following examples review and illustrate various techniques that have been developed to improve the effectiveness of resolution and paramodulation-based theorem proving systems:

Example: Motivation for Subsumption

The clause, Q(F(X,Y)), is a more general form of all of the clauses: Q(F(A,B)), Q(F(G(X),Y)), Q(F(F(C),B),Y), ... etc.... Similarly, the clause, Q(A), is a more general form of all of the clauses: Q(A) Q(B), Q(A) Q(A), Q(A) Q(X) Q(C), ... etc... in that these clauses are all trivial consequences of the first. It would seem undesirable to keep the less general forms when all of the information is captured by the two clauses, Q(F(X,Y)) and Q(A).

An automatic process, SUBSUMPTION, has been added to the ANL-NIU theorem proving system. If at any time, clauses C1 and C2 are in the clause space and there exists a substitution S such that C1(S) is equal to

C2 or to any subset of the literals of C2, then clause C2 is deleted from the clause space. Note that the inclusion of subsumption in the theorem proving system does not effect the completeness of either resolution or paramodulation ([12] and [16]).

<u>Example:</u> Motivation for Demodulation as a Simplifier

If the fact X + 0 = X is known, it seems reasonable to replace the term, A + (B + 0) with the term, A + B. The process of automatically simplifying terms is known as DEMODULATION. If the equality unit, EQUAL(TERM1,TERM2) is marked as a demodulator, then any instance of TERM1 in the clause space will be replaced by the corresponding instance of TERM2 [21].

Note that while demodulation is similar to paramodulation in that it is essentially equality substitution, it differs from paramodulation in four fundamental ways. First, paramodulation is an inference rule that is directed by the proof search. Demodulation is an automatic process that occurs at all stages of the proof search. Second, paramodulation expands the clause space by deriving new clauses from old ones that remain in the clause space. Demodulation actually replaces existing terms in the clause space (effectively deleting the original clauses). Third, paramodulation allows substitution for the variables in both the equality literal as well as the clause that is being paramodulated into. Demodulation, as a simplifier, only allows substitution for the variables of the demodulator. And fourth, paramodulation allows the relevant equality literal to be in a clause with other literals. Demodulation, as a simplifier, requires that the relevant equality literal be a unit clause (no other literals).

Note that because a clause that is demodulated is deleted from the clause space, the inclusion of automated demodulation eliminates the completeness property. Demodulation can prevent inferences that are necessary for a proof (see Section II.B.2). Although the sacrifice of completeness is very important from a theoretical point of view, the primary concern in operational theorem proving systems is performance. Experimental evidence to date supports the inclusion of demodulation in a theorem proving system.

Example: Motivation for Demodulation as a Canonicalizer

It seems reasonable to keep all of the polynomials $A + B + C = 0$, $A + C + B = 0$, $C + A = -B$ ...etc... in the single canonical form, $A + B + C = 0$. This type of canonicalization can be simulated by giving the theorem proving system an appropriate set of demodulators [19].

I.A.1.e. STRATEGIES

All of the above features are completely mechanical in nature and are easily implemented with a computer program. The difficult part of automated theorem proving, however, is the algorithm that guides the proof search.

The search space can be thought of as a tree-like graph. The level 0 clauses (the clauses at the initial nodes of the graph) are the original set of input clauses, and in general, the level i+1 clauses are clauses whose ancestors (clauses that participated in the inference that derived the clause) come only from levels 0 through i, with at least one ancestor coming from level i.

Definition 11.

A search strategy is COMPLETE if when using a complete inference system and given enough time and memory, the strategy is guaranteed to find a proof if one exists.

One possible search strategy consists of deriving all level i clauses before generating any clauses in level i+1. This is called LEVEL SATURATION, and is a breadth first search of the graph that represents the search space. This strategy is complete, but is completely uninformed. That is, it makes no use of any acquired knowledge, and makes no evaluation of the existing state of the clause space. In general, this is a very poor search strategy.

At the other extreme is the depth first search. In this strategy only the most recently generated clauses are looked at. This means that the graph that represents the search space is expanded as deeply as possible along one path before considering an alternate path. This is clearly inefficient and incomplete unless the proof is already known and the correct path chosen from the start. .

Since the 'safest' search is the breadth first search, and the ideal search is the depth first search with the correct path chosen from the start, it is expected that the best search strategy will combine the positive aspects of both. That is, the search will be broad enough to maximize the likelihood of finding a proof eventually, and deep enough to find a proof as directly and efficiently as possible.

Various search strategies have been developed that attempt to achieve this goal. One particular search strategy that is used extensively in the ANL-NIU system is that of SET OF SUPPORT [20].

Recall that the three parts of the initial clause space are the axioms, the special hypothesis, and the denial of the theorem. The idea of set of support is to focus the proof search on the problem to be solved (the special hypothesis and the denial of theorem) rather than on the general field of study (the axioms). Basically, the rule says not to generate any clause that does not have as an ancestor a clause from either the special hypothesis or the denial of the theorem. It can be proved [20] that under this definition, set of support is a complete search strategy. That is, if the clause space corresponding to the theorem to be proved is unsatisfiable, and the inference system being used is complete, then there is a proof using the set of support strategy.

I.A.2. PROGRAM VERIFICATION - GENERAL ESCRIPTION

Any problem that can be stated as a theorem in any area that can be axiomatized in first-order predicate calculus can theoretically be solved with the theorem prover. Because of this generality, there are several interesting and useful application areas including mathematics, database information retrieval, and program verification.

The verification of a program consists of two distinct steps. Given an assertion that represents the properties of the program variables on input to a program, and an assertion that represents the desired properties of the program variables at each 'halt' statement, proof of CORRECTNESS consists of proving for each 'halt' that if the input assertion is satisfied and the 'halt' is reached, then the corresponding 'halt' assertion is satisfied. Proof of TERMINATION consists of proving that at least one of the 'halt' statements will be reached under all possible valid inputs. Clearly, a program that has been proved to satisfy both correctness and termination has been verified.

Both correctness and termination are proved by breaking a program into a set of small program segments that are each easily verified. To insure that all possible paths through the program are accounted for, it is important that none of the segments has any loops or branches. Also, since each segment is to be verified separately, an additional set of input and 'halt' assertions must be assigned to each point where the program is broken such that the 'halt' assertion of the first of two consecutive segments is the same as the input assertion of the segment that follows it.

Example: Proof of Correctness

Consider the flowchart in FIGURE 1 that represents a program to compute the largest integer, Z, such that $Z^2 <= X$, for any natural number X ([7] page 178). The input assertion (point A) is that X >= 0 (the domain of integers is assumed). The 'halt' assertion (point C) is that $Z^2 <= X < (Z+1)^2$. Assigning the assertion, $Y1^2 <= X$ and $Y2 = (Y1+1)^2$ and Y3 = 2*Y1+1, to point B, the proof of correctness consists of verifying the three program segments A to B, B to B, and B to C. Note that since the point B is inside the loop, every possible path through the program must consist only of the paths A to B, B to B, and B to C. It follows then that these are the only program segments that need to be verified.

For each program segment, the theorem, "If the input assertion is correct and the program segment is executed then the 'halt' assertion is correct.", is attempted by the automated theorem proving system. The execution of the program segment is reflected by the values of the program variables in the assertions.

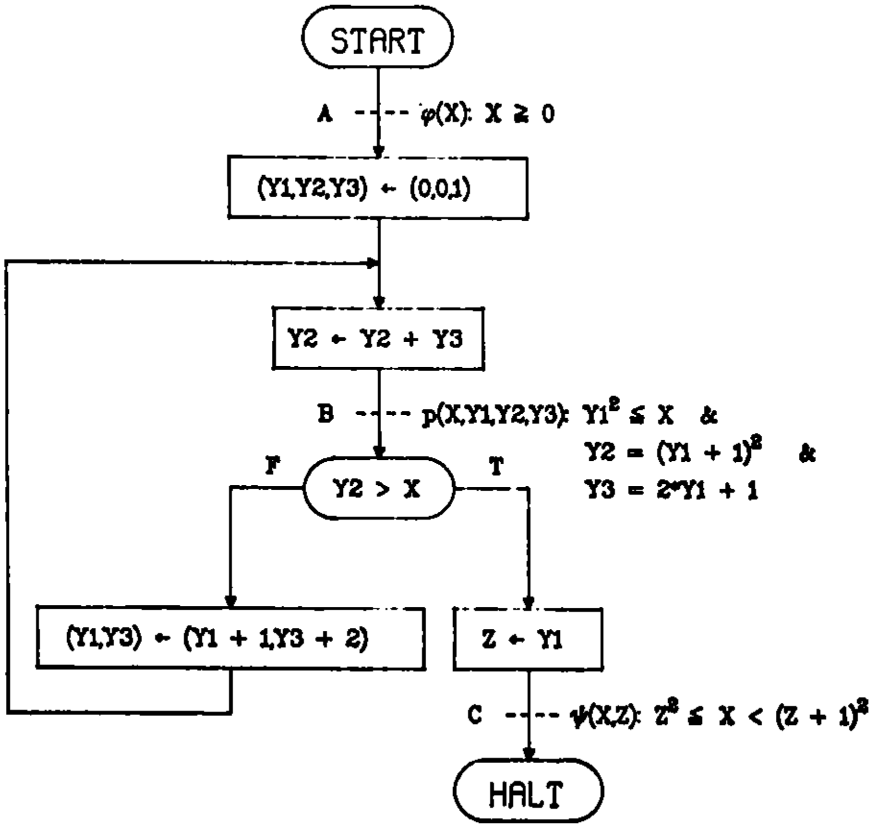

FIGURE 1

I.B. GOALS

Since the most important and most difficult part of automated theorem proving is the proof search strategy, the general goal of this thesis is to provide a mechanism for improving the strategy for problems arising from the field of program verification.

The proof search process consists, in part, of a set of algorithmic processes such as the application of inference rules, subsumption, and demodulation. The specific goal of this thesis is to present a method for increasing the deductive power of these individual processes. In particular, a method for incorporating automatic transformations into the inference process will be motivated and described.

Literal clashing and unification are essential and fundamental processes in any resolution and paramodulation theorem proving system. The automatic transformation concept will be illustrated with new algorithms for these two processes.

It will be shown that in addition to reordering the proof search space for a given problem in a significant way, the incorporation of automatic transformations permits the generation of inferences that might otherwise be blockrd (e.g. by demodulation or by ordering the arguments of equality literals in a canonical way - see Section II.A.2).

I.C. RELEVANCE

Any resolution and paramodulation theorem proving system is dependent on the deductive power of its inference processes. The automatic transformation concept presented in this paper is applicable to any area that has 'rewrite' relations (e.g. commutativity, associativity, ordering relations, ...). The new literal clashing and expanded unification algorithms presented in this paper are particularly effective in areas that tend to have broad rather than deep proofs (see Chapter II). Program verification is one such area.

I.D. SURVEY OF OTHER WORK

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 categories: unification, simplification, and inference rules.

The concept of defining unification in the context of an equational theory is not new ([2], [5], [6], [9], [15], [18], and [21]). Since some sort of matching (unification) is of fundamental importance to any resolution-based theorem proving system, much attention has been paid to the theoretical aspects of building more powerful matching systems. In general, methods for finding all possible matches in a given environment have been of primary interest.

As an illustration, consider Fay [2]. This paper discusses the incorporation of rules into the unification process to get complete sets of unifiers. For example, the terms, F(X,B) and F(A,Y), have the common instance F(A,B) as well as the common instance F(A,F(Z,B)) in the presence of the associative axiom, EQUAL(F(X,F(Y,Z)),F(F(X,Y),Z)).

A second way to build a theory into an automated theorem proving system is to define rules for the construction and application of sets of simplifiers (complete sets of reductions) ([3], [4], and [5]). For example, Gloess and Laurent [3] propose as an alternative to the Knuth-Bendix algorithm [4] a dynamic algorithm for applying simplifiers to terms.

As an alternative to modifying the unification process, a third way to build a theory into an automated theorem proving system is to define new rules of inference that reflect that theory [16]. Slagle in [16] discusses a technique for designing new inference rules based on the axioms of a theory being studied. For example, the transitivity axiom for partially ordered sets can be replaced by an appropriate inference rule.

The methods presented in this thesis provide a technique for building some theory into the inference process. Since this technique has been designed for performance in an applied environment (program verification in particular), it does not attempt to find complete sets of unifiers when used in the context of a unification algorithm. Although the new technique can be used in the context of simplification, it differs from existing techniques in two important ways. First, the transformation concept can apply to predicates and literals as well as to terms. And second, the technique is not oriented towards complete sets of reductions which in fact, do not always exist. Finally, although the transformation concept does increase the deductive power of the inference rules when applied to literal clashing or unification algorithms, the rules themselves remain unchanged.

II. AUTOMATIC TRANSFORMATIONS

The automatic transformation concept is introduced in this chapter. The concept is motivated in Section A. New algorithms are given for literal clashing and unification in Section B to illustrate the concept. These algorithms are discussed further in Section C.

II.A.1. NATURE OF PROGRAM VERIFICATION PROOFS

Recall that the proof search can be thought of as the expansion of a tree-like graph. Proofs in many areas are deep by nature:

Input clauses -> C1 -> C2 -> ... -> empty clause (proof)

where the Ci are generated clauses.

That is, the subgraph that corresponds to the proof has a relatively large number of levels. Program verification proofs, however, tend to be very broad:

Input clauses -> C1

Input clauses -> C2

.

.

.

Input clauses -> Cn

where C1, C2, ... Cn -> empty clause in one hyper-resolution step.

That is, the subgraph that corresponds to the proof has only a few levels with a relatively large number of nodes.

A problem will be considered to have a broad proof it there exists a proof which is broad, and will be considered to have a deep proof if every proof is deep.

The following example illustrates a sample program verification proof:

<u>Example:</u>

Consider the following set of input clauses:

(1) LT(A,B) EQUAL(C,D) LT(E,F(X,A))

(2) LT(B,A)

(3) LT(C,D)

(4) LT(F(A,B),E)

(5) ¬LT(X,Y) ¬LT(Y,X)

(6) ¬LT(X,Y) ¬EQUAL(X,Y)

(7) EQUAL(F(X,Y),F(Y,X)),

and the following set of generated clauses:

(8) ¬LT(A,B)	(resolve (2) and (5))
(9) ¬EQUAL(C,D)	(resolve (3) and (6))
(10) ¬LT(E,F(A,B))	(resolve (4) and (5))
(11) ¬LT(E,F(B,A))	(paramodulate (7) into (10)).

None of the literals in clauses (1) through (4) will resolve, but all of the literals of clause (1) can be resolved with clauses (8), (9), and (11), which are transformed versions of clauses (2), (3), and (4).

The existing theorem proving system must go through the following sequence to find this proof:

1. Clauses (2), (3), and (4) must be chosen by the proof search to generate clauses (8), (9), and (10).
2. Clause (10) must be chosen by the proof search to

generate clause (11).

3. Clauses (8), (9), and (11) must be chosen by the proof search to resolve against clause (1).

This procedure may, in general, have the side effect of generating many clauses that do not participate in the proof, and may prevent the search from finding the right clauses as a result (because of the size of the clause space).

The key to the new literal clashing algorithm presented in the next section is the incorporation of certain 'rewrite' transformations (e.g. LT(X,Y) --> ¬LT(Y,X), LT(X,Y) --> ¬EQUAL(X,Y), and F(X,Y) --> F(Y,X)) into the literal clashing process. In the example above, clauses (2), (3), and (4) will now clash against clause (1), effectively finding a proof without generating any new clauses at all.

The new algorithm has two distinct effects:

1. The new literal clashing algorithm has the effect of generating more general clauses sooner by allowing the resolution of more literals. More general clauses imply a smaller clause space (because of subsumption), which in turn implies a more efficient proof search. The limiting case is the empty clause which subsumes all other clauses and signifies that a proof has been found. It follows then that 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. The new literal clashing algorithm causes the graph that represents the search space to be broader and less deep. That is, clauses

that were originally generated at level i may now be generated at level j where $j < i$. This implies that a search strategy that has elements of breadth first search in it will be more effective than in the original search space.

In general, the broader (and less deep) a proof is, the more effective the new literal clashing algorithm will be. 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 algorithm is more likely to delay the generation of these intermediate clauses until after a proof has been found (in effect not generating them at all) when the proof is broad rather than deep. For this reason, the algorithm is particularly well suited for application to program verification problems.

II.A.2. BUILT-IN INCOMPLETENESS

Some processes have been built into the ANL-NIU theorem proving system (and others) that eliminate the completeness property. These processes are often added because the general gain in effectiveness of the theorem proving system is felt to outweigh the loss of generality caused by incompleteness. The automatic transformation process can eliminate some of the blocks to completeness caused by these processes.

<u>II.A.2.a. DEMODULATION</u>

Although demodulation can cause a significant simplification of the clause space, there are some negative side effects. One of the most serious side effects is the blocking of certain unifications (and literal clashes).

<u>Examples:</u> Unification Blocked by Demodulation

Consider the demodulator EQUAL(G(A),G(B)) and the clause ¬EQUAL(F(X,G(X)),E). If the clause EQUAL(F(A,G(A)),E) were generated, it could clash against the inequality, and a proof would have been found. With the demodulator present, however, the clause EQUAL(F(A,G(A)),E) will be demodulated to EQUAL(F(A,G(B)),E), which does not clash against the inequality, before it is added to the clause space.

Consider a second example. The terms, F(X,A) and F(B,Y), which are normally unifiable, cannot be unified in the presence of the demodulator, EQUAL(F(X,Y),F(Y,X)) (used as a canonicalizer - see [19]), because the terms will be demodulated to their corresponding canonical forms, F(X,A) and F(Y,B), which are not unifiable.

Note that it is this type of blocking that causes theorem proving systems using demodulation to be incomplete.

One of the features of the expanded unification algorithm presented in the next section is to allow the clash of the literal ¬EQUAL(F(X,G(X)),E) against the literal EQUAL(F(A,G(B)),E) in the presence of the demodulator EQUAL(G(A),G(B)), and to allow the unification of the term F(X,A) with the term F(Y,B) in the presence of the demodulator EQUAL(F(X,Y),F(Y,X)).

II.A.2.b. EQUALITY ORDERING

Some automated theorem proving systems have built in equality ordering. That is, every equality literal, EQUAL(T1,T2), is kept 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 can significantly reduce the size of the clause space but can, in general, lead to incompleteness.

Example: Resolution Blocked by Equality Ordering

Consider the clause, ¬EQUAL(F(A,B),F(C,D)). If the clause EQUAL(F(X,B),F(C,D)) were generated, it might (depending on the equality ordering rule) be reordered to EQUAL(F(C,D),F(X,B)), which does not clash against the first clause, before it is added to the clause space.

One of the features of the new literal clashing algorithm presented in the next section is to allow the clashing of EQUAL(F(A,B),F(C,D)) with EQUAL(F(C,D),F(X,B)) by incorporating the transformations, EQUAL(X,Y) --> EQUAL(Y,X) (and ¬EQUAL(X,Y) --> ¬EQUAL(Y,X)), into the literal clashing process.

II.A.3. COMMUTATIVE AND ASSOCIATIVE UNIFICATION

The benefit c˜ ⌣n expanded unification algorithm that would include commutativity for commutative functions and associativity for associative functions has been discussed in the literature ([2], [5], [6], [9], [15], and [18]). The expanded unification portion of the new literal clashing algorithm will allow unification subject to any of a number of special axioms supplied by the user.

Example: Expanded Unification

Consider the terms, F(J(X,B),A) and EVAL(A,Y). These two terms will unify in the presence of the clauses EQUAL(EVAL(X,Y),F(X,Y)) and EQUAL(F(X,Y),F(Y,X)) with the expanded algorithm.

II.B. LITERAL CLASHING AND EXPANDED UNIFICATION

The new concept is to incorporate automatic transformations of literals and terms into the inference process. That is, a single 'new' inference may in fact represent a sequence of inferences. The concept has been motivated by two areas of interest, literal clashing and unification. Algorithms will be given to illustrate the implementation of the new concept in these areas. These algorithms have two aspects that require special discussion, the choice of transformations eligible to apply and rules for applying them. Three factors must be considered in the choice of transformations, validity of the resulting inferences, complexity properties of the algorithm, and effectiveness of the theorem prover with the algorithm included.

Every transformation will have a corresponding 'transformation clause'. That is, for every transformation, Tr, there will exist a clause, C, such that the application of Tr to a literal or term is equivalent to making a single resolution or paramodulation step with C. Since all transformation clauses will be required to be immediate consequences of the clause space representing the problem in question, all transformations and resulting inferences will be valid. In the description of the algorithms given below, transformations will in fact be defined by giving the corresponding transformation clause. That is, all transformations will be given either as a clause with exactly two literals, L1 and L2, or as an equality unit clause, EQUAL(T1,T2).

To be useful, the transformation process must be direct and efficient. One requirement is to keep the set of transformations that can apply at any point reasonably small. In addition, it can be shown that if the set of applicable transformations satisfies certain properties (see below) then the process will in fact be efficient in the sense that the number of dead end paths pursued will be reasonably small.

The increased effectiveness of the theorem prover due to the inclusion of such an algorithm must justify the cost (e.g. time and/or memory) of the algorithm. The inclusion of the two algorithms given below can effect the search space in two significant ways: first, by generating inferences that might otherwise be blocked (e.g. by demodulation or equality ordering), and second, by reordering the search space. Reordering the search space can cause shortcuts in a particular proof (shorter paths to key inferences). Some transformations, however, 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, $\neg Q(X,Y) \quad Q(Y,X)$, which might apply at many unnecessary places. It is important to limit the set of available transformations to those which can have a significant effect (hopefully positive) on the search space.

The following definitions facilitate the formal description of the transformations:

Definition 12.

A term is GROUND if it contains no variables. Note that a ground term names a constant element of the relevant domain.

Definition 13.

A term is COMPOSITE if it is not a constant and not a variable.

Definition 14.

A BAG is a collection of items in which duplications are allowed. Note that a set is no more than a bag with no duplications.

Example: Bags and Sets

Consider the list (1,2,1,3,3,4).

The set of elements in the list is (1,2,3,4), but the bag of elements in the list is (1,2,1,3,3,4) (or (1,1,2,3,3,4) since a bag is an unordered list).

Definition 15.

A ground subterm of a term, T, is MAXIMAL in T if it is not the subterm of any ground term other than itself.

Examples: Maximal Ground Subterms

Consider the term, F(G(J(A,B)),J(X,C)). The subterms, G(J(A,B)) and C are maximal, but the subterms A, B, and J(A,B) are not.

For another example, consider the atom, Q(J(A,B),J(C,D)). Since an atom is not a term, both J(A,B) and J(C,D) are maximal.

Definition 16.

A WFF (well formed formula) is a literal, an atom, or a term.

The following functions are defined by their action on an arbitrary WFF:

Definition 17.

VARBAG(WFF) = bag of all variable names in WFF.

Definition 18.

VARSET(WFF) = set of all variable names in WFF.

Definition 19.

MFS(WFF) = major function symbol of WFF.

Definition 20.

NARGS(WFF) = number of arguments of MFS(WFF).

Examples:

VARBAG(F(X,J(Y,X))) = (X,X,Y)

VARSET(F(X,J(Y,X))) = (X,Y)

NARGS(F(X,J(Y,X))) = 2 (X and J(X,Y))

MFS(F(X,J(Y,X))) = F

Definition 21.

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 this is one measure of the complexity of a WFF because it counts each maximal ground term as one item, namely, the single constant element that the term names (as discussed in Definition 12).

Examples:

COM(F(X,J(Y,X))) = 0 + 2 = 2

COM(F(X,J(Y,A))) = 1 + 2 = 3

COM(F(X,J(A,B))) = 1 + 1 = 2

COM(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 22.

SGN(literal) = '+' or '-' (sign of atom)

Definition 23.

Two literals, L1 and L2, PRE-CLASH if MFS(L1) = MFS(L2) and SGN(L1) ¬= SGN(L2). Note that two literals clash (resolve) if they pre-clash and their atoms unify.

Definition 24.

A list of clauses (or a clause space) is FULLY CLASHED if no new resolvents can be found. That is, if clauses C1 and C2 are on the list and can be resolved to generate clause C3, then either C3 or a clause that subsumes C3 must be on the list.

Definition 25.

A list of clauses (or a clause space) is FULLY PARAMODULATED if no new paramodulants can be found. That is, if clauses C1 and C2 are on the list and can be paramodulated to generate clause C3, then either C3 or a clause that subsumes C3 must be on the list.

Definition 26.

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.

Literal clashing and unification are fundamental steps in the inference process of any resolution and paramodulation theorem proving system. The incorporation of automatic transformations into these processes can have a dramatic effect on the clause space corresponding to a problem by permitting more inferences to be generated from a given set of clauses. The algorithms presented in the following subsections illustrate the incorporation of automatic transformations into these two processes.

II.B.1. 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. That is, there exists a substitution to the variables such that the resulting atoms are identical. The new notion is that two literals clash if there are transformed versions of the literals that clash in the usual sense.

The new literal clashing algorithm presented here makes use of two distinct lists of applicable transformations, LCLASH1 and LCLASH2. LCLASH1 contains transformations that change the sign and/or predicate symbol of a literal. LCLASH2 contains transformations that permute the arguments of a literal.

Both LCLASH1 and LCLASH2 consist of clauses with exactly two literals, L1 and L2. The mechanism for applying the transformations is ordinary resolution. Note that every clause in fact represents two transformations, ¬L1 --> L2 and ¬L2 --> L1.

The clauses on LCLASH1 must satisfy the following properties:

1. The clause, L1 L2, is in (or known to be deducible from) the clause space representing the problem in question.

   This is to maintain the validity of all inferences. The transformations on LCLASH2 also must satisfy this property.

2. Either SGN(L1) = SGN(L2) or MFS(L1) ¬= MFS(L2).

   The transformations of LCLASH1 correspond to changes in sign and/or predicate symbol.

One additional required property of LCLASH1 will be given after the description of LCLASH2 below.

The spirit of the new concept is to automate the 'obvious' transformations, that is, to automate transformations that are in some sense 'rewrite' rules (for literals as well as terms) such as, ¬LT(X,Y) ¬LT(Y,X), where LT is the 'less than' relation. The restrictions that follow are an attempt to distinguish these kinds of 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.

It was desirable in this first study to greatly restrict the lists of eligible transformations and the rules for applying them. Promising results with the restricted lists of transformations indicate that future studies with some or all of the restrictions relaxed might be worthwhile.

The additional restrictions to LCLASH1 that are currently implemented are:

1. VARSET(L1) = VARSET(L2).

   Transformations do not introduce new variables to or eliminate variables from a literal.

2. NARGS(L1) = NARGS(L2)

   Transformations do not introduce new arguments to or eliminate arguments from a literal.

3. COM(L1) = COM(L2)

   Transformations do not change the complexity of a literal in the sense of Definition 21.

4. No substitutions for the variables in the literals being transformed can be made. That is, only the transformation clauses themselves can be instantiated. The application of the transformations on LCLASH2 is also restricted in this way.

Example: Restricted Substitution

The clause, ¬Q(X,X) R(X,X), is an eligible transformation clause but cannot be applied to the literal, Q(X,Y), because of the restriction on substitution.

In addition to the properties for LCLASH1 and LCLASH2 already mentioned, the clauses on LCLASH2 must satisfy the following properties:

1. MFS(L1) = MFS(L2)
2. SGN(L1) ¬= SGN(L2)
3. The major subarguments of L2 are a permutation of the major subarguments of L1. That is, there exists some permutation of arguments, PERM, such that PERM(L1) and L2 differ only in sign.
4. LCLASH2 must be fully clashed.

The final required property of LCLASH1 is that the union of the lists LCLASH1 and LCLASH2 must be fully clashed subject to the two qualifications that follow. Although omitting the qualifications would result in lists with nice theoretical properties (see Lemma 1 below), it is consistent with the goal of keeping the set of applicable transformations small to remove transformations that are redundant and not useful.

1. Tautologies generated by resolving clauses in LCLASH1 need not be included. For example, the clauses ¬P(X) Q(X) and P(X) ¬Q(X) resolve to generate the clauses ¬P(X) P(X) and ¬Q(X) Q(X), which do not satisfy property 2 above for LCLASH1 and would not be added to the list.
2. After the union of lists LCLASH1 and LCLASH2 is fully clashed, consider the set of all transformations, L1 L2, on LCLASH1 such that exactly one of the literals, L1 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 LCLASH1.

Keeping all such permutation variants will not effect the results of the algorithm, but might cause some unnecessary duplication of work.

Example: Permutation Variants

If the clause, ¬LT(X,Y) ¬EQUAL(X,Y) is on LCLASH1, and the clause, ¬EQUAL(X,Y) EQUAL(Y,X), is on LCLASH2, then it is not necessary to include the clause, ¬LT(X,Y) ¬EQUAL(Y,X), on LCLASH1.

Note that the fully clashed requirement 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, although the clause, ¬P(X) P(F(X)), itself can generate a countably infinite set of distinct clauses, transformations of this kind are not allowed.

A transformation (set of transformations) is not eligible if it indirectly violates the above requirements, even if the transformation clause itself is eligible. For example, the pair of transformation clauses, ¬Q(A) R(A) and ¬R(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 LCLASH1) or to a transformation that permutes arguments of a literal (for LCLASH2).

Most of the requirements above are relevant to restricting the set of transformations and how they can apply. A few, however, like the fully clashed properties, are relevant to permitting a simple organization to the new literal clashing algorithm. Although these requirements may seem to make the set of transformations on lists LCLASH1 and LCLASH2 very complicated, most of the requirements are only necessary to cover special cases that will not commonly arise in practice. On one set of real problems that was tested (see Chapter III), the entire set of transformations consisted of the following transformation clauses:

¬LT(X,Y) ¬LT(Y,X)

¬LT(X,Y) ¬EQUAL(X,Y)

¬LT(X,NUM1) ¬IB(CC,X)

¬LT(CN,X) ¬IB(CC,X)

¬EQUAL(X,Y) EQUAL(Y,X)

and ¬EQUALARR(X,Y) EQUALARR(Y,X)

The set of eligible transformations has been divided into the two lists, LCLASH1 and LCLASH2, for reasons of efficiency. 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, it is felt that the trade-off between the computational efficiency of having two lists against the simple organization of the algorithm justifies having the two lists.

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 LCLASH1 and LCLASH2 when transformations can be applied without a substitution restriction. 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 |-- 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. That is, there exists a clause, c, in C such that b is a resolvent of a and c.

<u>Lemma 1.</u> 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 satisfiable, but the conjunction of a, b, and C is unsatisfiable, then a |--> ¬b' with respect to C, where b and ¬b' clash.

<u>Proof.</u> Since a must clearly participate in the derivation of the empty clause, the fully clashed property implies that if a and C |-- ¬b', where b and ¬b' clash, then a |--> ¬b'. The lemma then follows from Corollary 3

on page 539 of [14]. □

Note that the case in which the conjunction of b and C is unsatisfiable is of no interest because of the fully clashed property of C.

Let C above be partitioned into two sets, C1 and C2, such that C1 consists of those clauses which (when thought of as transformations) change sign and/or predicate symbol, and C2 those clauses that remain. Now replace C1 with C1' which is constructed from C1 as follows: For each clause in C1, add the clause to C1' unless it is the resolvent of a clause in C2 and a clause already in C1' 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 C1'.

Note that C1' may not be uniquely determined by C1. The order that the clauses are inspected may determine which clauses in C1 are omitted from C1'. This has no bearing on the lemmas and theorems that follow.

<u>Lemma 2.</u> Let a1, a2, ... an be unit clauses such that a1 |--> a2 |--> ... |--> an with respect to C1' and C2. If a1 and an differ in sign and/or predicate symbol, then at least one of the following must hold:

(1) There exists a unit clause, b, such that a1 |--> b with respect to C1' and b |--> an with respect to C2.

(2) There exists a unit clause, b, such that a1 |--> b with respect to C2 and b |--> an with respect to C1'.

Proof. Since C is fully clashed, a1 |--> an with respect to C. Since a1 and an differ in sign and/or predicate symbol, it follows that a1 |--> an with respect to C1. Let c1 be the clause in C1 that resolves with a1 to produce an. If c1 is in C1', then there is nothing to show since the identity transformation is implicitly in C2 (b = an). If c1 is not in C1', then it must be the resolvent of a clause in C1' with a clause in C2, and the lemma follows. □

Lemma 3. Let a1, a2, ... an, be as in Lemma 2. If an is transformable by C2, then outcome (1) of Lemma 2 holds.

Proof. Note that a1 |--> an with respect to the original C1 as in the proof of Lemma 2. If both a1 and an can clash against clauses in C2, then a1 |--> an with respect to C1' since C1' contains all clauses from C1 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 a1 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 themselves:

Theorem 1. Consider the conjunction of the clauses in LCLASH1, the clauses in LCLASH2, and two literals, LITERALA and LITERALB as unit clauses. If the resulting clause space is unsatisfiable but is satisfiable without either of the two literals, then there exists a transformation, Tr1, from

LCLASH1 and a transformation, Tr2, from LCLASH2 such that either Tr2(Tr1(LITERALA)) clashes with LITERALB or Tr2(Tr1(LITERALB)) clashes with LITERALA. In particular, the following hold true:

(1) If LITERALA is transformable by LCLASH2, then there exist Tr1 and Tr2 such that Tr2(Tr1(LITERALB)) clashes with LITERALA.

(2) If LITERALB is transformable by LCLASH2, then there exist Tr1 and Tr2 such that Tr2(Tr1(LITERALA)) clashes with LITERALB.

(3) If neither LITERALA nor LITERALB is transformable by LCLASH2, then there exists Tr1 such that Tr1(LITERALB) clashes with LITERALA.

Proof. Recall that LCLASH1 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 LCLASH1 with a clause on LCLASH2 (see Section II.B.1).

Case 1: LITERALA is transformable by LCLASH2.

Since the clause space is unsatisfiable, there must exist a sequence of unit (single literal) clauses a0, a1, ... an such that LITERALB = a0 |--> a1 |--> a2 |--> ... |--> an = ¬LITERALA' where ¬LITERALA' and LITERALA clash (see [14]). Let k <= n be such that ak, ak+1 ... an are of the same sign and predicate symbol and such that ak-1 and an differ in sign and/or predicate symbol. Since LCLASH2 is fully clashed, there exists (by Lemma 1) a transformation Tr2'' from LCLASH2 such that Tr2''(ak) = an.

It remains to show that there exists a transformation Tr1 from LCLASH1 and a transformation Tr2' from LCLASH2 such that Tr2'(Tr1(a0)) = ak. This suffices because Lemma 1 and the fact that LCLASH2 is fully clashed would then imply that there exists a transformation Tr2 from LCLASH2 such that Tr2(Tr1(a0)) = Tr2''(Tr2'(Tr1(a0))) = Tr2''(ak) = an.

If a0 and ak are of the same sign and predicate symbol, then Tr1 is the identity transformation and Tr2' is guaranteed by Lemma 1 (using LCLASH2 for C) since LCLASH2 is fully clashed.

If a0 and ak differ in sign and/or predicate symbol, then the result follows from Lemma 3.

Case 2: LITERALB is transformable by LCLASH2.

This proof is completely symmetric to the proof of Case 1.

Case 3: Neither LITERALA nor LITERALB is transformable by LCLASH2.

Since the two possible outcomes in Lemma 2 are identical when the candidate from C2 is the identity transformation, this case follows from Lemma 2. □

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

Notation:

Let a and b be literals.

a --> b if there exists a transformation Tr on either LCLASH1 or LCLASH2 such that Tr(a) = b.

a -1-> b (a -2-> b) if there exists a transformation Tr on LCLASH1 (LCLASH2) such that Tr(a) = b.

a -(k)-> b if there exists a1, a2, ... ak such that a --> a1 --> a2 --> ... --> ak = b. (If k = 0 then a = b.)

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

<u>Lemma 4.</u> a -1-> b if and only if ¬b -1-> ¬a (Similarly for LCLASH2).

<u>Proof.</u> a -1-> b if and only if there exists a transformation clause, L1 L2, and a substitution, S, such that L1(S) = ¬a and L2(S) = b (since substitutions are only allowed into the transformation clauses). □

<u>Theorem 2.</u> a -(*)-> b implies that there exists a' such that either a -1-> a' -2-> ı or ¬b -1-> a' -2-> ¬a.

<u>Proof.</u> This follows from Theorem 1 where LITERALA and LITERALB in Theorem 1 are thought of as ground literals (so no substitutions are possible). □

Theorem 1 does not characterize the transformation process of the algorithm below because it does not account for the substitution restriction to the application of transformations. Although Theorem 2 does characterize the transformation process of the algorithm, it is considerably weaker than Theorem 1 in that it only refers to the transformation process itself and not to the underlying goal of clashing literals. This difference reflects the trade-off made between nice theoretical results and practical algorithms, and is illustrated in the following example.

<u>Example:</u> Weakness of Theorem 2

Consider the two literals, Q(A) and R(X), and the transformation clause, ¬Q(A) ¬R(B). Q(A) can be transformed to ¬R(B) which clashes with R(X), but R(X) cannot be transformed to ¬Q(A) to clash with Q(A) (because of the substitution restriction). Although it is necessary to choose the correct literal to transform for this clash to succeed, the algorithm does not make this distinction and so the success is left to chance (which order the literals are passed to the clashing algorithm).

Since the motivation for the substitution restriction is to preserve the 'renaming' spirit of the transformation concept, it is reasonable to allow Q(A) to be transformed to ¬R(B) without allowing R(X) to be transformed to ¬Q(A). It is felt that this anomaly in the selection process (choosing which literal to transform), does not warrant the price of a different selection algorithm, which would be significantly more complex and costly.

<u>Example:</u> Inherent Complexity of a Better Selection Algorithm

Consider the two literals, Q(A,X) and R(Y,B), and the transformation clauses, ¬Q(A,X) ¬R(B,X) and ¬Q(X,A) ¬R(X,B). If only the first transformation clause is present, then it is necessary to transform the literal, Q(A,X), in order to clash the two literals (because of the substitution restriction). Similarly, if only the second transformation clause is present, then it is necessary to transform the literal, R(Y,B). This distinction is dependent on the complete structure of the transformation clauses as well as the literals to be transformed. In other words, for a more effective selection algorithm, detailed information about the eligible transformations and the literals to be clashed must be taken into account.

ALGORITHM

Let LITERAL1 and LITERAL2 be two literals that are to be clashed with the new literal clashing algorithm.

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.
       Go to STEP 2 to clash LITERALA and LITERALB.
STEP 2: Apply transformations of LCLASH1.
          Let LITERALB'' = LITERALB
          Do while LITERALA and LITERALB'' do not clash.
             Let LITERALB' = LITERALB
             Do while LITERALA and LITERALB' do not pre-clash.
                Choose a transformation from LCLASH1 that has not
                been applied to LITERALB yet.
                Apply this transformation to generate a new
                LITERALB'.
                If none apply then STOP (with failure).
             End
             Go to STEP 3 to unify the corresponding atoms of
             LITERALA and LITERALB'.
          End
          STOP (with success)
STEP 3: Apply transformations of LCLASH2.
          Let LITERALB'' = LITERALB'
          Do while LITERALA and LITERALB'' do not clash.
             Choose a transformation from LCLASH2 that has not
             been applied to LITERALB' yet.
             Apply this transformation to LITERALB'
             to generate a new LITERALB''.
```

```
        If none apply then RETURN to STEP 2.
    End
    RETURN to STEP 2.
```

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 to find a proof that two literals are inconsistent. Let m be the number of transformations on LCLASH1 that can apply to LITERALB, and let n be the number of transformations on LCLASH2 that can apply to MFS(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 is important because each application of a transformation requires a unification test, which can significantly add to the cost of the algorithm.

The expanded unification algorithm presented in the next subsection can be used as the unification step in the new literal clashing algorithm (or independently as the unification step for paramodulation).

II.B.2. EXPANDED UNIFICATION ALGORITHM

Unification is a fundamental part of every resolution and paramodulation step. The usual notion of unification is that two terms (or atoms) unify if they have a common instance. That is, there exists a substitution to the variables such that the resulting terms (atoms) are identical. In expanded unification, two terms (atoms) unify if there are transformed versions that unify in the usual sense.

The transformation process in the expanded unification algorithm presented here is fundamentally different than the transformation process in the new literal clashing algorithm presented in the last subsection. The application of transformations in the new literal clashing algorithm corresponds to making resolutions with appropriate transformation clauses. The application of transformations in the expanded unification algorithm, however, corresponds to making paramodulations (equality substitutions) with appropriate transformation clauses.

The expanded unification algorithm presented here makes use of two distinct lists of applicable transformations, UNIFY1 and UNIFY2. Both UNIFY1 and UNIFY2 consist of equality unit clauses, EQUAL(T1,T2), where T1 and T2 are terms. UNIFY1 contains transformations that change the major function symbol of a term (MFS(T1) ¬= MFS(T2)). UNIFY2 contains transformations that change the subterms of a term (MFS(T1) = MFS(T2)).

The mechanism for applying the transformations to a term, T, is a restricted form of paramodulation (equality substitution). The difference between ordinary paramodulation and the application of transformations is that transformations must substitute for the entire term, T, that is being transformed. This is an organizational restriction (for an algorithm) that helps minimize duplication of effort.

Example: Restricted Paramodulation

The transformation, EQUAL(F(X,Y),F(Y,X)), can apply to the term, F(A,F(B,C)), to produce F(F(B,C),A), but not to produce F(A,(F(C,B)).

Note that in some automated theorem proving systems paramodulation from the right side of an equality is prevented. That is, the equality unit clause, EQUAL(T1,T2), cannot be used to substitute instances of T1 for instances of T2. The application of the transformation, EQUAL(T1,T2), is not restricted in this way.

It is important to maintain the 'rewrite' spirit of the transformation concept. In particular, it is important to prevent transformations that expand terms from being included as eligible transformations.

Examples: Expanding Transformations

If the clause, EQUAL(F(A,A),A), was an eligible transformation, then the following sequence would be possible: A --> F(A,A) --> F(A,F(A,A)) --> F(A,F(A,F(A,A))) --> .....

Similarly, if the clause, EQUAL(F(X,F(X,Y)),F(Y,X)), was an eligible transformation, then the following sequence would be possible: F(X,Y) --> F(Y,F(Y,X)) --> F(Y,F(X,F(X,Y))) --> .....

Such sequences must clearly be avoided.

Each clause, EQUAL(T1,T2), on UNIFY1 and UNIFY2 must satisfy the following properties:

1. EQUAL(T1,T2) is in (or known to be deducible from) the clause space representing the problem in question.
2. VARBAG(T1) = VARBAG(T2)
3. COM(T1) = COM(T2)
4. T1 (T2) is not a proper subterm of T2 (T1).

The following general restrictions also apply:

1. No substitutions can be made for the variables in the term

that is being transformed. Only the transformation itself can be instantiated.

2. The set of transformations ideally should be fully paramodulated. For example, if the clauses EQUAL(A,B) and EQUAL(F(C,A),F(D,A)) are present, then the clauses EQUAL(F(C,B),F(D,A)), EQUAL(F(C,A),F(D,B)), and EQUAL(F(C,B),F(D,B) should be present.

   Practical considerations limit the application of this rule. 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 most general transformations.

Note that this 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.

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 corresponds to the assumption about tautologies in the discussion of the literal clashing algorithm.

The clauses that satisfy the above properties are partitioned into two sets. UNIFY1 consists of those in which MFS(T1) ¬= MFS(T2), and UNIFY2 consists of those in which MFS(T1) = MFS(T2).

As in the case of the literal clashing algorithm, the set of eligible transformations has been divided into the two lists, UNIFY1 and UNIFY2, for reasons of efficiency. Restricting the set of transformations that can first be applied to those that change the major function symbol of a term provides an efficient sieve for terms that are not unifiable. That is, no attempt will ever be made to unify terms unless they have transformed versions that have the same major function symbol.

The following helps motivate the organization of the algorithm given below:

Notation:

Let r and s be terms.

r --> s if there exists a transformation Tr on either UNIFY1 or UNIFY2 such that Tr(r) = s.

Theorem 3. r --> s 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 (since substitutions are only allowed 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, and let Tr1 and Tr2 be two transformations represented by transformation clauses, EQUAL(T1,T2) and EQUAL(T3,T4), respectively. Assume that Tr1 and Tr2 are on a list that is fully paramodulated. If T' is the term that is generated by substituting R with Tr1(R) in T, and T'' = Tr2(T'), then there exists a transformation, Tr3, such that Tr3(T) subsumes T''.

Proof. The fact that Tr1 is applicable to R implies that there exists a substitution, S1, such that either T1(S1) = R or T2(S1) = R. Without loss of generality, assume that T1(S1) = 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' or T4(S2) = T'. Without loss of generality, assume that T3(S2) = T'. Now, since 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). Since 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). □

Since 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

Let TERM1 and TERM2 be two terms that are to be unified with the expanded unification algorithm. The algorithm applies transformations to TERM2 until it can be unified with TERM1 with ordinary unification. TERM1 remains unchanged and is only used to test for unification with the transformed versions of TERM2.

The expanded unification algorithm is presented as a recursive algorithm. That is, to unify two terms, first match or unify their major function symbols and then unify each of their subarguments left to right.

```
STEP 1:  Apply transformations of UNIFY1.
            Let TERM2' = TERM2
            Do while MFS(TERM1) ¬= MFS(TERM2')
               Choose a transformation from UNIFY1 that has not
               been applied to TERM2 yet.
               Apply this transformation to generate a new TERM2'
               If none apply then STOP (with failure).
            End
            Go to STEP 2 to unify the major subarguments of TERM1
            and TERM2'.
            STOP (with success)
STEP 2:  Apply transformations of UNIFY2.
            Let TERM2'' = TERM2'
            Do while TERM1 and TERM2'' do not unify.
               Let SUBTERM1 be the first argument of TERM1.
               Let SUBTERM2 be the first argument of TERM2''.
               Do while SUBTERM1 and SUBTERM2 unify with expanded
               unification.
                  If last argument then STOP (with success).
                  Let SUBTERM1 and SUBTERM2 be the next arguments of
```

```
        TERM1 and TERM2'' respectively.
      End
      Choose a transformation from UNIFY2 that has not been
      applied to the current TERM2'.
      Apply this transformation to generate a new TERM2''.
      If none apply then STOP (with failure).
    End
```

As with the new literal clashing algorithm, the expanded unification algorithm is not a 'pre-theorem prover'. That is, it is not the case that the algorithm corresponds to using the theorem prover to find a proof that two terms (or atoms) have instances that are equal. Let k be the number of function symbols in TERM2, m be the maximum number of transformations on UNIFY1 that potentially apply to a given term (by comparing major function symbols), and let n be the maximum number of transformations on UNIFY2 that potentially apply to a given term. It follows that at most k(m+n) transformations can be applied in the algorithm to test for the unification of TERM1 and TERM2. In general, m and n will be small.

The transformation concept can easily be incorporated into the subsumption and demodulation processes with slight variations of the expanded unification algorithm. For example, consider the clause, Q(F(A,B)). This clause could be subsumed by Q(F(B,X)) or demodulated by EQUAL(F(B,X),C) by applying the transformation, EQUAL(F(X,Y),F(Y,X)). Because of the apparent relative importance of the processes, it is expected that these applications of the transformation concept will have little impact on the effectiveness of a theorem proving system compared to the impact of the new literal clashing and expanded unification algorithms.

II.C. EFFECTIVE USE OF THE TRANSFORMATION CONCEPT

The cost (in computer resources) of the new literal clashing and expanded unification algorithms is directly related to the number of applicable transformations. In general, the user of the theorem proving system must decide (from the semantics of the problem under consideration) whether the potential effect of an eligible transformation on the proof search space warrants its inclusion on the relevant lists. That is, it may be better to allow a clause to apply only in the normal inference process of the theorem prover than to include it as an automatic transformation. In general, it seems desirable to keep the set of eligible transformations restricted to a small set that are very general in nature.

Example:

The transformation clause, ¬P(X,Y) Q(X,Y), is generally more useful as an automatic transformation than the transformation clause, ¬R(X,A) S(B,X). It might be better (in terms of the general effectiveness and efficiency of the theorem prover) to include the second less general transformation as a clause in the clause space, but not as an automatic transformation.

Note that the literal clashing algorithm finds at most one way to clash two literals, even if more that one way exists (in the sense that different variable substitutions are necessary). For example, since the literals Q(X,A) and ¬Q(A,Y) already clash, (X <-- A and Y <-- A), the algorithm will not find the clash in which (X <-- Y) in the presence of the transformation clause, ¬Q(X,Y) Q(Y,X).

Although it might be desirable to find all possible clashes from a theoretical standpoint, it seems less desirable in practice. It is clearly more efficient to find just one clash. Also, there are two reasons why the effect on the clause space of finding only one clash will not be great in general. First, one natural way to prove problems in program verification is to emphasize ground terms [17]. That is, aside from the initial axioms, most clauses that participate in a proof will have no variables in them. This implies that there will not be many literals in the clause space that can clash in more than one way. Second, if transformation clauses corresponding to applicable transformations are included as axioms in the clause space, then some of the clashes not found by the literal clashing algorithm will eventually be found by the normal search process.

There is a completely parallel discussion for the expanded unification algorithm.

III. IMPLEMENTATION OF THE ALGORITHMS

The algorithms described in Chapter II have been implemented in PL/1 and integrated into the ANL-NIU theorem proving system. Because the algorithms consist of straightforward applications of processes that already commonly occur, it is believed that the algorithms are easily integrated into most existing resolution-based theorem proving systems.

III.A. TESTING AND EVALUATION

The new literal clashing algorithm was tested with the extended unification algorithm included as the unification step. Three sets of problems were tested:

1. A set of artificial problems was designed to illustrate the power of the automatic transformation concept. One of the purposes of this set of experiments was to uncover any possibly unforeseen side effects of the automatic transformation process.
2. A set of real problems currently being tested by B. T. Smith on the environmental theorem proving system [17] was tested to show that the theorem proving system with the new literal clashing algorithm included is no worse than, and is sometimes better than, the system with the usual algorithm.
3. The real problems supplied by B. T. Smith were varied slightly to help characterize the conditions in which the new literal clashing algorithm has the most favorable effects on the proof search space.

It is a fundamental property of automated theorem proving that in general, it is impossible to know when progress is being made on a given search path. For this reason, it is often difficult to draw concrete conclusions from comparisons of different algorithms and strategies. It is not clear what the definition of 'good' or 'better' should be. The most obvious criterion for evaluation and comparison is the finding (or not finding) of proofs. This alone, however, is not sufficient. Questions of efficiency must also be considered. For example, the breadth first search of the clause space is guaranteed to succeed (with unlimited computer resources), but this is clearly not a reasonable strategy to build an effective automated system on. Some other reasonable criteria are: computer resources used (time and/or memory), total number of clauses in the clause space, number of clauses that participate in the proof, number of clauses actually selected by the search strategy (not counting automatic processes), number of clauses selected by the search strategy that actually participate in the proof, and the depth of the empty clause in the graph representing the search space.

The literal clashing and expanded unification algorithms were implemented with development (flexibility and testing) rather than computational efficiency in mind. The program was written as an independent unit and then patched into the existing theorem proving system as a last step. Although the system ran significantly slower with the algorithms integrated than without, it is believed that the running time of the algorithms themselves can be decreased by approximately 80 percent by rewriting the programs to use the data structures and routines that already exist within the current theorem proving system.

The algorithms commonly used by B. T. Smith for program verification problems consist of several automatic process that do not involve the selection of clauses by a search strategy (e.g. demodulation, case splitting ...[17]). For this reason, a distinction has been made between 1) clauses that are generated by automatic processes and 2) clauses that are generated by the application of an inference rule to clauses that have been chosen from the clause space by the proof search strategy. In particular, the depth of a clause in the graph representing the search space will be considered with respect to this distinction. That is, the depth of a clause is equal to the maximum depth of its ancestors if it is generated by an automatic process, and is one greater than the maximum depth of its ancestors if it is generated as an inference from a clause selected by the search strategy.

III.B. RESULTS

Fourteen artificial problems were tested with and without transformations. These consisted of at least one essential clause (a clause necessary for the proof) with many literals, and several unit clauses that could clash against these literals with the application of one or more transformations.

In every case, the problems with the transformations found proofs in exactly one hyper-resolution or UR-resolution [10] step. The problems without the transformations either found a proof in several steps or failed due to an incompleteness (see II.A.2) or by exhausting reasonable computer resource limits.

Eleven real problems were tested with various search strategies, including those most commonly used by B. T. Smith. 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: There was no case in which the notrans version found a proof and the trans version did not. There was exactly one case in which the trans version found a proof and 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 is reasonable because the transformation process has the effect of producing more clauses at earlier levels in the graph that represents the search space. Since 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 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 6 clauses (ten percent).

Number of clauses selected by the search strategy: The value for the trans version never exceeded the value for the notrans version in all cases but one. 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, with reductions of 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 versions in all tests with all but one of the eleven problems tested, with reductions of up to 50 percent. In the one exception, the trans versions all increased the depth of the notrans version from 2 to 3. This 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. That is, sometimes level i clauses can be selected by the search strategy before level j clauses, where i is greater than j. In the problems 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 third 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.

IV. SUMMARY

Do the benefits or potential benefits of the automatic transformation concept justify the incorporation of transformations into the literal clashing and unification processes for some problems? What further investigations might be made to better characterize and take advantage of the power of the automatic transformation concept? In what other areas might it apply?

IV.A. AUTOMATIC TRANSFORMATIONS

The artificial problems and 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, the tests with the real problems indicate that program verification problems run with automatic transformations incorporated into the literal clashing and unification processes did no worse than and sometimes did significantly better than the same problems run without transformations. The results, however, were not as promising as expected. This suggests that although the concept may be quite useful, better search strategies not currently available to the theorem proving system might be developed to capitalize on the power of the transformation process. In particular, for program verification problems, a strategy that is more oriented to breadth first search (but is not an exact breadth first search) might prove very useful.

Further study in this area might prove fruitful.

IV.B. IDEAS FOR FURTHER STUDY

Various modifications to the new literal clashing and expanded unification algorithms including relaxation of some of the rules for eligibility of transformations and rules for applying transformations might be investigated. In particular, relaxation of the rule about restricted substitution into the terms and/or literals being transformed could have a significant effect on the resulting clause space.

As mentioned in the last subsection, results of tests made indicate that further testing of the transformation concept with new search strategies not currently available might be fruitful.

Finally, the incorporation of the automatic transformation concept into other areas, such as demodulation or subsumption, might be investigated.

ACKNOWLEDGMENTS

I would like to take this opportunity to acknowledge some of the many individuals who have aided my progress through graduate school. My few successes here are in large part attributable to their efforts.

First of all, I wish to thank Professor Lawrence J. Henschen. As my advisor and chairman of my doctoral committee he gave unselfish support and guidance to all aspects of my education, research, and the production of this dissertation. I also wish to thank the other members of my doctoral committee: Professors Albert A. Grau, Gilbert K. Krulee, and Benjamin Mittman.

In addition, I would like to thank Dr. Lawrence T. Wos, who supplied the initial motivation and education for my study of automated theorem proving, and Dr. Ross A. Overbeek, who integrated my programs into the existing theorem proving system at Argonne National Laboratory. A special note of thanks goes to Dr. Brian T. Smith for stimulating my interest in program verification and for many discussions and helpful suggestions. All of these people have been friends as well as colleagues and teachers.

Finally, I wish to thank the Applied Mathematics Division at Argonne National Laboratory for its financial support and for the use of its facilities.

REFERENCES

[1]-Chang, C.L. and Lee, R.C.T, SYMBOLIC LOGIC AND MECHANICAL THEOREM PROVING, Academic Press, New York, 1973.

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

[3]-Gloess, P. and Laurent, J., "Adding dynamic paramodulation to rewrite algorithms," to appear in the proceedings of the Fifth Conference on Automatic Deduction, Les Arcs, Savoie, France, 1980.

[4]-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, pp. 263-297, 1970.

[5]-Lankford, D.S. and Ballantyne, A.M., "Decision procedures for simple equational theories with commutative-associative axioms: complete sets of commutative-associative reductions," Technical Report, Mathematics Dept., University of Texas at Austin, August 1977.

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

[7]-Manna, Z., MATHEMATICAL THEORY OF COMPUTATION, McGraw-Hill Inc., New York, 1974.

[8]-McCharen, J., Overbeek, R.A., and Wos, L., "Problems and experiments for and with automated theorem-proving programs," IEEE Transactions on Computers, vol. C-25, pp. 773-781, 1976.

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

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

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

[12]-Robinson, J.A., "A machine-oriented logic based on the resolution principle," JACM, vol. 12, pp. 23-41, 1965.

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

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

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

[16]-Slagle, J.R., "Automatic theorem proving with built-in theories including equality, partial ordering, and sets," JACM, vol. 19, pp. 120-135, 1972.

[17]-Smith, B.T., "The environmental theorem prover", to be published as an Argonne National Laboratory technical report.

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

[19]-Veroff, R., "Canonicalization and demodulation," to be published as an Argonne National Laboratory technical report.

[20]-Wos, L., Robinson, G.A., and Carson, D.F., "Efficiency and completeness of the set of support strategy in theorem proving," JACM, vol. 12, pp. 536-541, 1965.

[21]-Wos, L., Robinson, G.A., Carson, D.F., and Shalla, L., "The concept of demodulation in theorem proving," JACM, vol. 14, pp. 698-709, 1967.

[22]-Wos, L., Overbeek, R.A., and Henschen, L.J., "Hyper-paramodulation: a refinement of paramodulation," to be published.