Stanford Artificial Intelligence Laboratory
Memo AIM-3 11

April 1978

Computer Science Department
Report No. STAN-CS-78-652

SIMPLIFICATION BY COOPERATING DECISION PROCEDURES

by

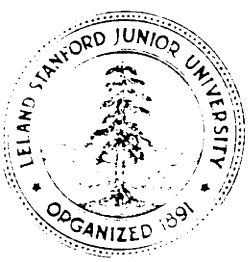
Greg Nelson and Derek C. Oppen

[Stanford Verification Group]

. Research sponsored by

National Science Foundation
and
Hertz Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University



Stanford Artificial Intelligence Laboratory
Memo AIM-3 11

April 1078

Computer Science Department
Report No. STAN-CS-78-662

SIMPLIFICATION BY COOPERATING DECISION PROCEDURES

by

Greg Nelson and **Derek C. Oppen**

[Stanford Verification Group]

We describe a simplifier for use in program manipulation and verification. The simplifier finds a normal form for any expression over the language consisting of individual variables, the usual boolean connectives, equality, the conditional function **cond** (denoting if-then-else), the numerals, the arithmetic functions and predicates +, - and <, the LISP constants, functions and predicates nil, car, cdr, cons and atom, the functions store and select for storing into and selecting from arrays, and uninterpreted function symbols. Individual variables range over the union of the reals, the set of arrays, LISP list structure and the **booleans** true and false.

The simplifier is complete; that is, it simplifies every valid formula to **true.** Thus it is also a decision procedure for the quantifier-free theory of reals, arrays and list structure under the above functions and predicates.

The organization of the simplifier is based on a method for combining decision procedures for several theories into a single decision procedure for a theory combining the original theories. More precisely, given a set S of functions and predicates over a fixed **domain,** a **satisfiability** program for S is a program which determines the satisfiability of conjunctions of **literals** (signed atomic formulas) whose predicate and function symbols are in S. We give a general procedure for combining satisfiability programs for sets S and T into a single satisfiability program for S u T, given certain conditions on S and T.

The simplifier described in this paper is currently used in the Stanford Pascal Verifier.

An earlier version of this paper appeared in the Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, 1978. This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, by the National Science Foundation under contract MCS 76-000327, and by the Fannie and John Hertz Foundation.

1. Introduction

In this section we give some examples of simplifications. We also specify the syntax and semantics of the language accepted by our simplifier. In section 2, we give a precise definition of a ***satisfiability program*** for a set S of functions, predicates, and constants. Essentially, such a program determines the satisfiability of conjunctions of **literals** (signed atomic formulas) whose predicate and function symbols are in S. The formal definition specifies the interpretations of the elements of S in such a way that it makes sense to "merge" satisfiability procedures for two sets S and T into one for S u T. We give a method for doing this, based on Craig's interpolation **lemma** ([Craig **1957]**). Section 3 shows how a satisfiability procedure can be used to implement a simplifier for general expressions.

1.1 Examples of the Use of the Simplifier

Here are some examples of simplifications.

```
2 + 3 * 5;
17;

P ⊃ ¬ P;
¬ P;

X = F(X) ⊃ F(F(F(X))) = X;
true;

cons(X, Y) = Z ⊃ car(Z) + cdr(Z) - X - Y = 0;
true;

X ≤ Y ∧ Y + D ≤ X ∧ 3 * D ≥ 2 * D ⊃ V[2 * X - Y] = V[X + D];
true;

A = store(store(A, I, A[J]), J, A[I]) ⊃ A[I] = A[J];
Crue
```

The last formula states the theorem that if the Ith and Jth elements of A are swapped, and if the resulting array equals the original one, then the Ith and Jth elements are equal.

1.2 The Theories $\mathcal{Z}$, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{E}$

All of the theories which we consider are formalized in classical first-order logic with equality, extended to include the three-argument function **cond**, where **cond(p, a, b) means** "if p then a else **b**". The *logical symbols* are **=,** A, v, **¬, ⊃, cond,** V and 3. A theory is determined by its non-logical symbols (that is, its constant, function, and predicate symbols) and its **axioms.**

The functions, predicates, and constants to which the simplifier currently gives an interpretation are those of the theory of reals under addition, the theory of list structure with car, cdr, cons, atom and nil, and the theory of arrays under storing (store) and selecting (select). We call these theories $\mathcal{Z}$, $\mathcal{L}$ and $\mathcal{A}$ respectively.

Given a quantifier-free expression F, the simplifier tries to find the simplest F' such that F = F' is entailed by the axioms for $\mathcal{Z}$, $\mathcal{A}$ and $\mathcal{L}$. In particular, if F is a formula entailed by the axioms, the simplifier returns the boolean constant true.

The non-logical symbols of $\mathcal{Z}$ are +, -, <, >, ≤, ≥ and the numerals. Its axioms are:

$$x + 0 = x$$
$$x + -x = 0$$
$$(x + y) + z = x + (y + z)$$
$$x + y = y + x$$
$$x \leq x$$
$$x \leq y \vee y \leq x$$
$$x \leq y \wedge y \leq x \supset x = y$$
$$x \leq y \wedge y \leq z \supset x \leq z$$
$$x \leq y \supset x + z \leq y + z$$
$$0 \neq 1$$
$$0 \leq 1$$

The numerals 2, 3, . . . and <, >, and ≥ are defined in terms of 0, 1, t, - and ≤ in the usual way. We also allow multiplication by integer constants; for instance, 2 * x abbreviates x t x.

The integers, rationals and reals are all models for these axioms. Any formula which is unsatisfiable over the rationals or reals can be shown unsatisfiable as a consequence of these axioms. Thus our simplifier is complete for the rationals or reals. It is not complete if the variables range over the integers, since there are unsatisfiable formulas, such as x t x = 5, which cannot be shown unsatisfiable as a consequence of the above axioms. In this respect, our simplifier does not differ from most theorem provers. The reason for the incompleteness is that determining the unsatisfiability of a conjunction of integer linear inequalities -- the integer linear programming problem -- is much harder in practice than determining the satisfiability of a conjunction of rational linear inequalities. This incompleteness is not as bad as it seems, since most formulas that arise in program verification and program manipulation do not depend on subtle properties of the integers. Further, there are some easily-implemented heuristics (such as converting x < y into x t 1 ≤ y) for integer variables which work well in practice.

The theory of arrays, $\mathcal{A}$, has the non-logical symbols store and select, and the axioms:

$$\mathbf{select}(\mathbf{store}(v, i, e), j) = \mathbf{cond}(i = j, e, \mathbf{select}(v, j))$$
$$\mathbf{store}(v, i, \mathbf{select}(v, i)) = v$$
$$\mathbf{store}(\mathbf{store}(v, i, e), i, f) = \mathbf{store}(v, i, f)$$
$$i \neq j \supset \mathbf{store}(\mathbf{store}(v, i, e), j, f) = \mathbf{store}(\mathbf{store}(v, j, f), i, e)$$

select(v, i) is the ith component of the one-dimensional array v. We may write **v[i]** for **select**(v, i). **store**(v, i, e) is the vector whose ith component is e and whose jth component, for $j \neq i$, is the jth component of v. Thus, if the program variable A has the value A_0 before the assignment $A[i] \leftarrow e$, then afterwards A will have the value **store**(A_0, i, e). A two-dimensional array can be treated as a vector of vectors, so **A[i, j]** is shorthand for **A[i][j]**. Had the assignment above been $A[i, j] \leftarrow e$, the value of A after the assignment would be **store**(A_0, **i, store**(A_0[**i**], j, e)).

The last three axioms are only needed if equalities between array terms are allowed.

The theory of list structure, $\mathcal{L}$, has the non-logical symbols car, cdr, cons, atom and nil, and the axioms:

$$\begin{aligned} &\text{car}(\text{cons}(x, y)) = x \\ &\textbf{cdr}(\textbf{cons}(x, y)) = y \\ &\neg\, \text{atom}(x) \supset \textbf{cons}(\textbf{car}(x), \textbf{cdr}(x)) = x \\ &\neg\, \text{atom}(\text{cons}(x, y)) \\ &\textbf{atom}(\textbf{nil}) \end{aligned}$$

Notice that acyclicity is not assumed; for instance, car(x) = x is regarded as satisfiable.

Finally, it is technically convenient to define the theory $\mathcal{E}$ whose non-logical symbols are all uninterpreted function, constant, and predicate symbols and which has no axioms. The theorems of $\mathcal{E}$ follow from the properties of equality; hence its name.

2. Merging Sat isfiability Programs

In this section, we define *satisfiability programs.* We then show how to "merge" satisfiability programs for two theories which have no common non-logical symbols.

2.1 Satisfiability Programs

If $\mathcal{S}$ is a theory, then a term is an *S-term* if each non-logical symbol occurring in the term is a non-logical symbol of $\mathcal{S}$. We define *S-literal* and *S-formula* **analogously.** For example, x = y and $x \leq y + 3$ are $\mathcal{Z}$**-literals** but $x \leq \text{car}(y)$ is not. Notice that a term is an $\mathcal{E}$**-term** if it contains only uninterpreted function symbols.

If $\mathcal{S}$ is a theory, a *satisfiability program* for $\mathcal{S}$ is a program which determines whether a conjunction $L_1 \wedge \ldots \wedge L_k$ of $\mathcal{S}$**-literals** is satisfiable in $\mathcal{S}$. A satisfiability program is therefore a **decision** procedure for satisfiability for con junctions of literals.

We use the name of a theory to denote both its satisfiability program and the conjunction of its axioms; for example, we may say that $\mathcal{Z} \wedge 0 > 1$ is unsatisfiable, or that the size of $\mathcal{Z}$ is **3.5K.**

There are efficient satisfiability programs for $\mathcal{Z}$, $\mathcal{E}$ and $\mathcal{L}$. For $\mathcal{Z}$, the simplex algorithm is very fast in practice ([Nelson 1978]). [Nelson and Oppen 19781 describe satisfiability programs for $\mathcal{L}$ and $\mathcal{E}$ which determine the satisfiability of conjunctions of length n in time $O(n^2)$. [Johnson and Tarjan1977] have improved the underlying algorithm to $O(n \log^2 n)$. [Oppen1978] describes a satisfiability program for $\mathcal{L}$ which runs in linear time if list structure is assumed to be acyclic. The satisfiability problem for conjunctions of $\mathcal{A}$-literals is NP-complete ([Downey and Sethi1976]).

2.2 Example of the Joint Satisfiability Procedure

We illustrate how $\mathcal{Z}$, $\mathcal{L}$ and $\mathcal{E}$ together detect the unsatisfiability of the following conjunction F:

$$X \le Y \wedge Y \le X + car(cons(0, X)) \wedge P(F(X)-F(Y)) \wedge \neg P(0)$$

We call a formula *homogeneous* if all its non-logical symbols are from the same theory. The first step we take is to make each atomic formula homogeneous, by introducing new variables to replace terms of the wrong "type" and adding equalities defining these new variables. For instance, the second conjunct would be a $\mathcal{Z}$-literal except that it contains the term car(cons(0, X)), which is not a $\mathcal{Z}$-term. We therefore replace car(cons(0, X)) by a new variable, say G 1, and add to the conjunction the equality G I = car(cons(0, X)) defining Cl. By continuing in this fashion we eventually obtain a formula F' which is satisfiable if and only if F is, with each literal of F' homogeneous. In our example, F' is

$$X \le Y \wedge Y \le X + G1 \wedge P(G2) \wedge \neg P(G5)$$
$$\wedge G1 = car(cons(G5, X)) \wedge G2 = G3 - G4$$
$$\wedge G3 = F(X) \wedge G4 = F(Y) \wedge G5 = 0$$

We next divide F' up into three conjunctions F_E, F_Z and F_L. F_E contains all the &-literals, F_Z all the $\mathcal{Z}$-literals and F_L all the $\mathcal{L}$-literals. Here is F' divided up into homogeneous parts:

F_Z	F_E	F_L
$X \le Y$	P(G2) = true	G 1 = car(cons(G5, X))
$Y \le X + G1$	P(G5) = false	
G2 = G3 - G4	G3 = F(X)	
G5 = 0	G4 = F(Y)	

These three conjunctions are given to the three satisfiability programs $\mathcal{Z}$, $\mathcal{E}$, and $\mathcal{L}$. Since each conjunction is satisfiable by itself, there must be interaction between the programs for the unsatisfiability to be detected. The interaction takes a particular, restricted form. We require that each satisfiability program deduce and *propagate* to the other satisfiability programs all equalities between variables entailed by the conjunction it is considering. For example, if $X \le Y$ and $Y \le X$ are asserted to $\mathcal{Z}$, it must deduce and propagate to the other satisfiability programs the fact that X = Y. The other satisfiability programs add X = Y to their conjunctions and the process continues.

In our example, neither F_Z nor F_E entail any equalities between variables, but F_L entails G1 = G5. $\mathcal{L}$ propagates this equality. $\mathcal{Z}$ uses this equality to deduce and propagate X = Y. $\mathcal{E}$ then propagates G3 = G4. $\mathcal{Z}$ then propagates G2 = G5. Now $\mathcal{E}$ has an inconsistent conjunction, and signals unsatisfiable. The following shows the literals received by the satisfiability programs, and the propagated equalities, listed in the order in which they were propagated.

$\mathcal{Z}$	$\mathcal{E}$	$\mathcal{L}$
X ≤ Y	P(G2) = true	G1 = car(cons(G5, X))
Y ≤ X + G1	P(G5) = false	
G2 = G3 - G4	G3 = F(X)	
G5 = 0	G4 = F(Y)	
		G1 = G5
X = Y		
	G3 = G4	
G2 = G5		
	unsatisfiable	

If one of the conjunctions F_Z, F_E, and F_L becomes unsatisfiable as a result of these propagations, the original conjunction must be unsatisfiable. For $\mathcal{Z}$, $\mathcal{E}$, and $\mathcal{L}$, the converse holds as well; that is, if the original con junction is unsatisfiable, then one of the conjunctions F_Z, F_E, and F_L will become unsatisfiable as a result of propagations of equalities between variables. For some other theories, such as $\mathcal{A}$, the converse does not hold. For these theories, a final "case-splitting" step, described in the next section, is required.

It is important to realize that it is never necessary to propagate disequalities, nor equalities other than those between variables. For instance, after receiving G 1 = G5, there was no need for $\mathcal{Z}$ to propagate that Y ≤ X or that X = Y + G5, even though these were deducible facts. None of the other satisfiability programs could make use of this information -- none of them knows anything about ≤ or +. Further, no disequality need be propagated, even though every theory shares = and ¬. A disequality x ≠ y is needed to prove inconsistency only if x = y is deduced. If some program deduces x = y, it will propagate this fact to the other programs, and the one that has deduced x ≠ y will detect the inconsistency.

Notice that the only satisfiability programs that can make use of a propagated equality between two variables are those whose conjunctions contain occurrences of both variables. For instance, when $\mathcal{L}$ propagated G1 = G5, only $\mathcal{Z}$ ever made use of this equality. When equalities are propagated, the only satisfiability programs that need to receive the equality are those which already "know" about both variables in the equality.

2.3 Joint Satisfiability Procedure

In this section we present the *joint satisfiability procedure* illustrated in the previous section. We assume that we have two theories $\mathcal{S}$ and $\mathcal{T}$ with no common non-logical symbols. The case for more than two theories follows easily.

Given a conjunction F of literals whose non-logical symbols are among those of $\mathcal{S}$ and $\mathcal{T}$, the joint satisfiability procedure determines whether F is satisfiable in the theory axiomatized by $\mathcal{S} \wedge \mathcal{T}$. $\mathbf{F_S}$ and $\mathbf{F_T}$ are program variables containing conjunctions of literals.

1. [Make F homogeneous.1 Assign con junctions to $\mathbf{F_S}$ and $\mathbf{F_T}$ by the method described in section 2.2 so that $\mathbf{F_S}$ contains a conjunction of $\mathcal{S}$-literals, $\mathbf{F_T}$ a conjunction of $\mathcal{T}$-literals, and $\mathbf{F_S} \wedge \mathbf{F_T}$ is satisfiable if and only if F is.

2. [&satisfiable?] If either $\mathbf{F_S}$ or $\mathbf{F_T}$ are unsatisfiable, return unsatisfiable.

3. [Propagate equalities.1 If either $\mathbf{F_S}$ or $\mathbf{F_T}$ entail some equality between variables not entailed by the other, then add the equality as a new conjunct to the one that does not entail it. Go to step 2.

4. [Case split necessary?] If either $\mathbf{F_S}$ or $\mathbf{F_T}$ entail a disjunction $\mathbf{u_1 = v_1} \vee \ldots \vee \mathbf{u_k = v_k}$ of equalities between variables, without entailing any of the equalities alone, then apply the procedure recursively to the k formulas $\mathbf{F_S} \wedge \mathbf{F_T} \wedge \mathbf{u_1 = v_1}, \ldots, \mathbf{F_S} \wedge \mathbf{F_T} \wedge \mathbf{u_k = v_k}$. If any of these formulas are satisfiable, return satisfiable. Otherwise return unsatisfiable.

5. Return satisfiable.

If the procedure returns unsatisfiable, it is clear that F is unsatisfiable. We will prove in the next section that the procedure is also correct if it returns satisfiable. The procedure always halts, since each repetition of step 3 or recursive call in step 4 conjoins an equality to one of the con junctions $\mathbf{F_S}$ or $\mathbf{F_T}$ not previously entailed by the conjunction. This can happen at most $\mathbf{n-1}$ times, where n is the number of variables appearing after step 2, since there can be no more than $\mathbf{n-1}$ non-redundant equalities between n variables.

We have not implemented the joint satisfiability procedure. **It is** subsumed **by** the simplification algorithm described in section 3.

[Kaplan 19681 proves that the quantifier-free theory of arrays with constant indices is decidable. [Shostak **1978]** proves that quantifier-free Presburger arithmetic with uninterpreted function symbols is decidable. [Suzuki and Jefferson **1977]** prove that quantifier-free Presburger arithmetic with arrays is decidable. The joint satisfiability procedure provides practical decision procedures for each of these theories.

2.4 Convexity and Case Splitting

In this section, we characterize the theories which require case splitting.

A formula F is non-convex if there exist 2n variables $x_1, y_1, \ldots, x_n, y_n$, $n \geq 2$, such that $F \supset x_1 = y_1 \vee \ldots \vee x_n = y_n$ but for no i between I and n does $F \supset x_i = y_i$. Otherwise, F is convex.

A theory $\mathcal{S}$ is ***convex*** if every conjunction of $\mathcal{S}$**-literals** is convex. If the satisfiability programs merged by the joint satisfiability procedure are satisfiability programs for convex theories, case splitting never occurs. Case splitting may occur if one or more of the theories are non-convex.

$\mathcal{Z}$ is convex, since the solution set of a conjunction of $\mathcal{Z}$**-literals** is the intersection of a convex set with a finite number of complements of hyperplanes. Such a set cannot be **a** subset of a union of finitely many hyperplanes unless it is a subset of one of them.

$\mathcal{E}$ and $\mathcal{L}$ are convex; this follows from the characterization in [Nelson and Oppen **1978]** of the set of equalities entailed by a conjunction of $\mathcal{E}$- or $\mathcal{L}$**-literals.**

$\mathcal{A}$ is not convex, as shown by the following example. Suppose that the theories merged by the joint satisfiability procedure are $\mathcal{A}$ and $\mathcal{Z}$, and that after step 1 the two formulas are:

F_A: store(v, i, **e)[j] = x** A **v[j] =** y

F_Z: **x > e** ∧ **x >** y

Each formula is satisfiable, the whole conjunction is unsatisfiable, but there are no equalities to propagate in step 3. In step 4, $\mathcal{A}$ propagates the disjunction x = e v x = y; each case leads to a contradiction in $\mathcal{Z}$.

We plan to extend $\mathcal{Z}$ to be complete for the integers. It will then no longer be convex, since for example x = **1** A y = 2 A **1** ≤ **z** A z ≤ 2 entails the disjunction x = z v y = **z** without entailing either disjunct alone. However, since we need only propagate equalities between variables, not between variables and constants, literals such as 1 ≤ z ≤ 100 will not cause splits (unless there are 100 variables equal to **1,** 2, . . . , 100 respectively!).

The theory of sets, which we intend to add to the simplifier, is another example of a non-convex theory; for example, x ∈ **{y,** z} causes the case split x = y v x = z.

Non-convexity complicates simplification. If a case split occurs for which some, but not all, cases are satisfiable, a good simplifier must determine which of the cases are satisfiable. To see this, consider the problem of simplifying x ∈ (4, -6) A **x >** 0 to x = 4. This conjunction of literals is satisfiable, as the joint satisfiability procedure determines by doing the case split x = 4 v x = -6. The simplifier must discover that the satisfiable branch of the split is the one in which x = 4.

2.5 Correctness of the Joint Satisfiability Procedure

The proof of correctness requires several lemmas. Our first goal is to define the residue of a formula. Essentially the residue is the strongest boolean combination of equalities between variables which the formula entails. For example the residue of the formula $x = f(a) \wedge y = f(b)$ is $a = b \supset x = y$, and the residue of $x \leq y \wedge y \leq x$ is $x = y$.

We make the following assumptions about the underlying formal system: (1) Individual variables are distinguishable from function variables. (2) There is no quantification over functions or predicates. (3) There are no propositional variables. The third restriction is not essential, but it simplifies the statement of the proof.

A ***parameter*** of a formula is any non-logical atomic symbol which occurs free in the formula. Thus the parameters of $a = b \vee \forall x\, P(x, f(x)) = c$ are a, b, P, f, and c.

We define a simple formula to be one whose only parameters are individual variables. For instance, $x \neq y \vee z = y$ and $\forall x\, x \neq y$ are simple, but $x < y$ and $f(x) = y$ are not. Thus an unquantified simple formula is a propositional formula whose atomic formulas are equalities between individual variables. The next lemma characterizes quantified simple formulas.

Lemma 1: Every quantified simple formula F is equivalent to some unquantified simple formula G. G can be chosen so that its variables are all free variables of F.

Proof: Suppose F is of the form $\exists x\, \Psi(x)$. Let Ψ_0 be the formula resulting from Ψ by first replacing any occurrences of $x = x$ and $x \neq x$ by true and false respectively, and then replacing any remaining equality involving x by false. Then, if $v_1, \ldots, v_k$ are the parameters of Ψ, F is equivalent to $\Psi_0 \vee \Psi(v_1) \vee \ldots \vee \Psi(v_k)$, since, in any interpretation, x either equals one of the v_i or else differs from all of them. By repeatedly eliminating quantifiers in this manner, we eventually obtain an equivalent quantifier-free formula whose only variables are free variables of F.

Any interpretation ψ for a formula F determines an equivalence relation $\sim$ on the free variables of F by the rule $u \sim v$ if and only if $\psi(u) = \psi(v)$. It follows from lemma 1 that if F is simple, $\sim$ completely determines whether ψ satisfies F.

Lemma 2: (Craig's interpolation lemma) If F entails G, then there exists a formula H such that F entails H and H entails G, and each parameter of H is a parameter of both F and G.

Proof: see [Craig, 19571.

Lemma 3: If F is any formula, then there exists a simple formula Res(F), the ***residue*** of F, which is the strongest simple formula that F entails; that is, if H is any simple formula entailed by F, then Res(F) entails H. Res(F) can be written so that its only variables are free variables of F.

Proof: Let $\{G_\lambda\}$ be the set of all simple formulas which F entails. For each G_λ, choose H_λ so that $F \supset H_\lambda \supset G_\lambda$, the only parameters of H_λ are parameters of both F and G_λ, and H_λ is

unquantified. The existence of H_λ is guaranteed by lemmas I and 2. Now, each H_λ is a propositional formula whose atomic formulas are equalities between individual parameters of F. It is easy to show that an infinite conjunction of propositional formulas over a finite set of atomic formulas is equivalent to some finite propositional formula over these atomic formulas. Therefore the conjunction of the H_λ is equivalent to some finite subconjunction H. Any simple formula G_λ entailed by F is entailed by some H_λ, and so by H. The only parameters of H are free individual parameters of F. Thus H is the residue of F.

Here are some examples of residues.

Formula	Residue
x = f(a) ∧ y = f(b)	a = b ⊃ x = y
x + y - a - b > 0	¬ (x = a ∧ y = b) ∧ ¬ (x = b ∧ y = a)
x = store(v, i, e)[j]	i = j ⊃ x = e
x = store(v, i, e)[j] ∧ y = v[j]	cond(i = j, x = e, x = y).

Notice in the last two formulas how the addition of an individual variable as a "label" affects the residue.

As a final example to relate the notion of residue to that of joint satifiability, here are the residues of the formulas which appeared in the example of section 2.2:

$\mathcal{Z}$	$\mathcal{E}$	$\mathcal{L}$
X ≤ Y	P(G2)	G I = car(cons(G5, X))
Y ≤ X + G I	¬ P(G5)	
G2 = G3 - G4	G3 = F(X)	
G5 = 0	G4 = F(Y)	

G5=G I = X=Y ∧ G3=G4 = G2=G5

G2 ≠ G5 ∧ X = Y ⊃ G3 = G4

G I = G5

As we found in section 2.2, the residues are inconsistent. An essential fact needed for proving the correctness of the joint satisfiability procedure is that these residues are always inconsistent if the original formula is. This fact is a consequence of the following lemma.

Lemma 4: If A and B are formulas whose only common parameters are individual variables, then Res(A ∧ B) = Res(A) ∧ Res(B).

Proof: Obviously the left side of the equivalence entails the right side, so we need only show the converse.

From A ∧ B ⊃ Res(A ∧ B) we get A ⊃ (B ⊃ Res(A ∧ B)) and so, by Craig's interpolation lemma, there is a formula H entailed by A which entails B ⊃ Res(A ∧ B) and whose only parameters are parameters of A and B. But these must be individual variables, so H is simple and therefore Res(A) ⊃ (B ⊃ Res(A ∧ B)). Writing this as B ⊃ (Res(A) ⊃ Res(A ∧ B)), and observing that the right hand side is simple, we have Res(B) ⊃ (Res(A) ⊃ Res(A ∧ B)), or, equivalently, Res(A) ∧ Res(B) ⊃ Res(A ∧ B), which proves the lemma.

Lemma 5: Let $F_1, F_2, \ldots, F_n$ be simple, convex formulas and V be the set of all variables appearing in any F_i. Suppose that for all x, y in V and for all i, j from 1 to n, either both F_i and F. entail x = y, or neither do. Then $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ is satisfiable if and only if each F_i is satisfiable!

Proof: The "only if" part is obvious. To prove the "if,, part, assume that each F_i is satisfiable. Let S be the set of equalities between variables in V entailed by some (hence all) of the F_i and T be the set of all other equalities between variables of V. We claim that any interpretation which makes every equality in S true and every equality in T false satisfies each F_i. If it does not satisfy F_i, then F_i entails the disjunction of all equalities in T. Now we consider three cases. If T is empty, F_i is unsatisfiable. If T contains only one equality, it is entailed by F_i and so it is in S. If T contains more than one equality, F_i is non-convex. Each case contradicts our assumptions.

We can now complete the proof of correctness of the joint satisfiability procedure by showing that if it returns satisfiable, F is satisfiable. To show that F is satisfiable, it suffices to show that $Res(\mathcal{S} \wedge F_S \wedge \mathcal{T} \wedge F_T)$ is not the constant fake. But by lemma 4, this residue is equivalent to $Res(\mathcal{S} \wedge F_S) \wedge Res(\mathcal{T} \wedge F_T)$. If step 5 of the procedure is reached, each of these residues must be convex, since step 4 did not cause a case split. Furthermore, the residues entail the same set of equalities and are each satisfiable, since steps 2 and 3 were passed. By lemma 5, the conjunction of the residues is satisfiable. Thus F is satisfiable if the algorithm returns from step 5. It follows, by induction on the depth of recursion, that F is satisfiable whenever step 4 returns satisfiable.

3. Simplification Based on Satisfiability Programs

In section 3. I we describe *cond normal form* for boolean expressions. In section 3.2 we give a simplification algorithm for formulas in **cond** normal form. In section 3.3 we discuss some aspects of the efficiency of our simplification algorithm, and in section 3.4 discuss some of its deficiencies.

3.1 Cond Normal Form

For convenience we use LISP list notation in this section. That is, the term f(a, b) is denoted (f a b).

Our simplifier first puts expressions into *cond normal form*. This is similar to the **cond** normal form in [McCarthy 1963]. An expression is in **cond** normal form if:

(1) The expression does not contain any boolean connectives other than **cond**. Thus A ∧ B is replaced by the equivalent (**cond** A B false), and -A by (**cond** A false true).

(2) No first argument to a **cond** is a **cond**. Thus (**cond** (**cond** P A B) C D) is replaced by (**cond** P (**cond** A C D) (**cond** B C D)).

(3) No expression of the form (**cond** P A B) is the argument to any function other than **cond**; thus (F (**cond** P A B)) is replaced by (**cond** P (F A) (F B)).

(4) Every boolean subexpression, other than constant subexpressions true and false, is the first argument to a **cond**. For instance, a single atomic formula P which is not the first argument to a **cond** is replaced by (**cond** P true false). F(X=Y) is successively replaced by (F (**cond** (= X Y) true false)) and (**cond** (= X Y) (F true) (F false)).

(In practice, the transformation required by (4) is not carried out if the subexpression is a second or third argument to **cond**, since this would waste space. If A and B are boolean, the **cond** normal form of (**cond** P A B) is (**cond** P (**cond** A true false) (**cond** B true false)) but we store it as (**cond** P A B).)

Cond normal form is not a canonical form, since two syntactically different expressions, each in **cond** normal form, may be logically equivalent.

An expression in **cond** normal form corresponds naturally to a binary tree whose nodes are labelled with atomic formulas. We call this tree the ***cond tree*** for the expression. To the expression (**cond** P A B) corresponds the tree whose root is labelled with P, whose left son is the tree for the expression A, and whose right son is the tree for the expression B. The tree for any non-cond expression E is a node with no sons labelled with E. Thus every node in a **cond** tree is either an internal node with two sons and a boolean expression as label, or a leaf node whose label is either non-boolean or one of the constants true or false.

The maximum number of nodes in the **cond** tree for an expression of length n is exponential in n. But, by sharing structure, the tree can be represented as a directed graph; the amount of storage required is linear in **n**.

Let N be a node of the **cond** tree for some expression. Then $\langle N_1 N_2 \ldots N_k \rangle$ is the ***branch*** to N if N_1 is the root of the tree, $N_k = N$, and, for each $I \le i < k$, N_{i+1} is a son of N_i. The context ***at*** N is the conjunction $L_1 \wedge \ldots \wedge L_{k-1}$, where each L_i is the label of N_i if N_{i+1} is the left son of N_i, and the negation of the label of N_i otherwise.

The context of a node is exactly the condition that must hold for an evaluator to reach the node during evaluation of the expression. That is, if the conditional expression is regarded as a program fragment, the context of a node is the strongest "invariant assertion" on the arc leading to the node. For example, consider the following expression in **cond** normal form: (**cond** P (**cond** QA B) (**cond** R C D)). The context of the node for B, that is, $P \wedge \neg Q$, is the condition that B would be evaluated if the whole expression were evaluated.

It follows that the disjunctive normal form of a formula is the disjunction of the contexts of the leaves labelled with true in the **cond** tree for the formula. Cond normal form is more compact than traditional disjunctive normal form because, in **cond** normal form, **disjuncts** are represented as branches in a tree (or paths in a directed graph) and thus may share structure.

3.2 The Simplification Algorithm

To simplify an expression, the simplifier traverses its **cond** tree, maintaining as it does so a representation of the context of the node it is visiting. When a node is reached with an inconsistent context, the node and the **subtree** below it are ignored. Thus the simplifier "prunes" away all inconsistent branches in the tree. The simplifier also collapses together branches to leaves with equivalent labels by replacing expressions of the form **(cond** p x x) by x. If the expression is a valid formula, every leaf which is reached will be labelled true; all these branches will be collapsed, and true will be returned. Similarly an unsatisfiable formula simplifies to **false.**

If the context of a node N is non-convex, the simplifier traverses the **subtree** rooted at N once for each branch of the case split.

SIMPLIFY takes two arguments: F, an expression in **cond** normal form, and CONTEXT, a conjunction of **literals**. It returns the simplest F' such that CONTEXT $\supset$ F = F'. If CONTEXT is unsatisfiable, it returns the atomic symbol omega. We assume that **omega** does not appear in F. The algorithm uses the auxiliary function SIMPATOM; if T is a term, **SIMPATOM(T,** CONTEXT) returns the simplest term T' such that CONTEXT $\supset$ T = T'.

. SIMPLIFY(F, CONTEXT):

1. If CONTEXT is unsatisfiable, return **omega.**

2. If F is not of the form **(cond** P A B), return SIMPATOM(F, CONTEXT).

3. If CONTEXT is not convex, let $E_1 \vee \ldots \vee E_k$ be a disjunction of equalities entailed by CONTEXT no disjunct of which is entailed by CONTEXT.

Set F $\leftarrow$ **(cond** E_1 F **(cond** E_2 F . . . **(cond** E_k F omega). . .)).

4. F is of the form **(cond** P A B). Set A $\leftarrow$ SIMPLIFY(A, P $\wedge$ CONTEXT), B $\leftarrow$ SIMPLIFY(B, $\neg$ P $\wedge$ CONTEXT). If A = **omega,** return B. If B = **omega,** return A. If A = B, return A. Otherwise, let P = SIMPATOM(P, CONTEXT). If A = true and B = **false,** return P. Otherwise return the expression **(cond** P A B).

The proofs of termination, correctness, and completeness of this procedure are straightforward.

In the remainder of this section, we describe our implementation of this simplification algorithm.

The implementation uses a function ASSERT, which conjoins an arbitrary literal to a global context representing a conjunction of iiterais. The value returned by ASSERT indicates either that the resulting conjunction is convex and satisfiable, or that the conjunction is unsatisfiable, or that the conjunction has become non-convex. In the last case, ASSERT also specifies the case split to be done.

In order to implement ASSERT efficiently, we require that the individual satisfiabiiity programs have certain properties. An ***incremental*** satisfiabiiity program is one which accepts iiterais one by one and which can determine at any time whether their conjunction is satisfiable. If in addition it can mark its state, accept more literais, and later return to the marked state by "undoing" the iiterals asserted after the mark, it is called ***wettable.*** To be used in our simplifier, a satisfiabiiity program must be resettable and must propagate the equalities and disjunctions of equalities which are entailed by the conjunction is has received.

More precisely, a satisfiabiiity program for a theory $\mathcal{S}$ consists of a global data structure, CONTEXTS, for representing conjunctions of $\mathcal{S}$-literals, together with the following functions for manipulating it.

$\mathbf{ASSERT_S(P)}$ where P is a literal, changes CONTEXTS to represent $Q \wedge P$, where Q is the con junction currently represented by CONTEXTS. If $Q \wedge P$ is unsatisfiable, ASSERTS(P) returns false. Otherwise, if there are any equalities between variables which are entailed by $Q \wedge P$ but not by Q, then $\mathbf{ASSERT_S(P)}$ returns the conjunction of ail such equalities. Otherwise, if $Q \wedge P$ is non-convex, $\mathbf{ASSERT_S(P)}$ returns a disjunction of equalities between variables entailed by $Q \wedge P$ no disjunct of which is entailed by $Q \wedge P$. Otherwise, ASSERTS(P) returns the constant true.

$\mathbf{PUSH_S()}$ saves the current state of CONTEXTS.

$\mathbf{POP_S()}$ restores CONTEXTS to the state it was in just before the last call to $\mathbf{PUSH_S()}$.

$\mathbf{SIMPATOM_S(F)}$, where F is an S-term or S-literal, returns an expression F' equivalent to F in $CONTEXT_S$. F' is the normal form for F in this context. For example, $\mathbf{SIMPATOM_Z(x + 0)}$ returns x and $\mathbf{SIMPATOM_Z(x - y)}$ returns 0 if $x = y$ is entailed by CONTEXTS. (SIMPATOMS will only be called when CONTEXTS is consistent).

Now we are ready to define ASSERT. ASSERT accepts an arbitrary literal, splits it into homogeneous pieces, and calls the appropriate assertion functions of the individual satisfibiiity programs. We define it for the case where there are two theories $\mathcal{S}$ and $\mathcal{T}$. The case where there are more than two theories is analogous.

In this program, PS, P_T, Q_S, and Q_T are variables containing formulas.

ASSERT(Q):

1. Divide Qinto homogeneous pieces Q_S and Q_T as described in section 2.

2. Set $P_S \leftarrow \mathbf{ASSERT}_S(Q_S)$; $P_T \leftarrow \mathbf{ASSERT}_T(Q_T)$.

3. If either P_S or P_T are fake, return fake.

4. If either P_S or P_T are disjunctions, return one of these disjunctions.

5. If both P_S and P_T are true, return true.

6. (One or both of the formulas is a conjunction of equalities. This step propagates the equalities.) Set each of the variables Q_S and Q_T to be the formula $P_S \wedge P_T$, and go to step 2.

ASSERT propagates equalities between the satisfiabiiity programs until one of them propagates false, or one of them splits (by returning a disjunction), or both of them stabilize.

Notice that a term t in an inhomogeneous literal which has been replaced by a new variable v in step 1 of some call to ASSERT may in a subsequent call be replaced by another new variable w. This is ail right, since both $t = v$ and $t = w$ are sent to the same satisfiability program, which will propagate $v = w$.

It is not necessary to send ail the equalities to all the satisfiability programs in step 6. As mentioned in section 2.2, an equality need only be sent to a satisfiability program if both variables in the equality are parameters of the conjunction represented in the program.

There is one feature of our the simplification algorithm described above which makes it unsuitable for implementation. If CONTEXT is represented by CONTEXT_S a.nd $\mathbf{CONTEXT}_T$, then the tests in steps 1 and 3 require a case split if either CONTEXTS or CONTEXT,. is, non-convex. If two or more of the cases are satisfiable, the simplifier will repeat the case split. A better approach is to return omega from step 1 if CONTEXTS or $\mathbf{CONTEXT}_T$ is unsatisfiable, and to split in step 3 if CONTEXT_S or CONTEXT,. is non-convex. Using this approach, the tests can be made without a case split, since a case split is not necessary to determine if an individual context is unsatisfiable or non-convex. ASSERT avoids redundant case **splits** by returning immediately if one of the satisfiabiiity programs splits, without checking if any of the branches of the case split is satisfiable.

In addition to ASSERT, the second simplification algorithm uses the functions PUSH, POP, and SIMPATOM. PUSH and POP simply call the push and pop functions **for** each of the satisfiabiiity programs. SIMPATOM takes an arbitrary term or literal and simplifies it using the information in CONTEXT_S and $\mathbf{CONTEXT}_T$ by calling the appropriate SIMPATOM functions. A record is kept of the individual variables generated as labels for terms in step 1, so that SIMPATOM can put the iiterais back together, replacing generated labels by the terms they represent. We omit the details.

The following simplification algorithm is a refinement of the one given above.

SIMPLIFY(F):

1. If F is not of the form **(cond** P A B), return **SIMPATOM(F).**

2. F is of the form **(cond** P A B). Call **PUSH().** Set Q ← ASSERT(P).

If Q = false, then **POP()** and return SIMPLIFY(B).

If Q= true then set A ← SIMPLIFY(A), **POP()** and go to step 3.

Otherwise, Qis a disjunction El v . . . v E,.

Set A ← SIMPLIFY((cond E_1 A . . . **(cond** E_k A omega). . .)), **POP()** and go to step 3.

3. Call **PUSH().** Set Q ← ASSERT{-P).

If Q = false, then **POP()** and return A.

If Q = true then set B ← SIMPLIFY(B), **POP()** and go to step 4.

Otherwise, Qis a disjunction El v . . . v E_k.

Set B ← SIMPLIFY((cond E_1 B . . . **(cond** E_k B omega). . .)), **POP()** and go to step 4.

4. If A = omega, return B. If B = omega, return A. If A = B, return A. Otherwise, let 'P = SIMPATOM(P). If A = true and B = false, return P. Otherwise return the expression **(cond** P A B).

We sketch the proof of the completeness of the algorithm. Whenever $CONTEXT_s$ or $\mathbf{CONTEXT_T}$ are non-convex, SIMPLIFY calls itself recursively on some **cond** expression. Thus whenever its argument is not a **cond** expression, $CONTEXT_s$ and $\mathbf{CONTEXT_T}$ are convex. By the definition of ASSERT, $CONTEXT_s$ and $\mathbf{CONTEXT_T}$ entail the same set of equalities when ASSERT returns. It follows from lemma 5 that if $CONTEXT_s$ and $\mathbf{CONTEXT_T}$ are convex, satisfiable, and entail the same set of equalities, then their conjunction is satisfiable. Therefore whenever SIMPLIFY returns from step **1,** the context is consistent. If F is valid, every leaf of its **cond** tree with a consistent context is iabeiied with true, so every term returned in step 1 is true. **It** follows by induction that A and B are always true, and therefore that the algorithm is complete.

3.4 Comparison with DNF-style Theorem Proving

We do not know how to give an adequate analysis of our simplifier. Its behaviour in practice is much better than its worst case behaviour. Instead, we will compare our approach, using **cond** normal form, with an obvious alternative approach, using disjunctive normal form, which we call a DNF-style approach. The DNF-style approach is not suited to arbitrary simplification, but only to proving the validity of formulas.

Let F be a formula represented as a **cond** tree with n internal nodes. The most obvious algorithm to determine if F is provable is to put its negation into disjunctive normal form and test each disjunct for unsatisfiabiiity. This corresponds to testing that the context of each leaf iabeiied with false is unsatisfiable. The standard DNF-style approach builds up the context for each leaf from scratch, that is, from the root of the **cond** tree. The number of calls to ASSERT equals the sum, taken over ail leaf nodes iabeiied with false, of the length of the branch to the leaf. This sum varies from O(n) to $O(n^2)$, and has an average value of $O(n^{1.5})$, if one considers all binary trees with n internal nodes and ail external node iabeiiings with true or false to be equally likely. There are no calls to PUSH or POP. A non-resettable satisfiabiiity program can be used.

Our algorithm makes n calls to PUSH, n calls to POP, and 2n calls to ASSERT. Therefore, DNF-style algorithms minimize (to zero) the number of calls to PUSH and POP, while our algorithm minimizes the number of calls to ASSERT. To determine which method is better, we would need to know the expected number of calls to ASSERT which each algorithm makes on realistic input distributions and the relative costs of resettable satisfiability programs and non-resettable ones.

The formulas which arise in the Stanford Verifier are often implications between conjunctions of iiterais. (Formulas with this structure arise in program verification whenever the invariant assertion on a simple loop is a conjunction of iiterais.) If there are n conjuncts in the antecedent of such a formula and m conjuncts in the consequent, then the disjunctive normal form of the negation of the formula has length $m(n+1)$, while the **cond** tree has only m + n internal nodes. A DNF-style algorithm can therefore make as many as $m(n+1)$ calls to ASSERT, while our algorithm can make at most m + n calls to ASSERT, PUSH and POP.

3.5 Finding the Simplest Form

In this section, we note some problems with our simplifier. The problems do not arise when our simplifier is used as a theorem prover, but only when it is being used to simplify expressions which do not simplify to an atomic symbol such as true. These problems arise in the design of any simplification algorithm.

First, a problem common to ail normal forms is that they may lose some of the structure of the original expression. It is hard to recover this structure if the expression does not significantly simplify. For instance, using **cond** normal form, the formula (A v B v C) ∧ (D v E v F) is "simplified" to

```
(cond A (cond E true (cond D true F))
        (cond B (cond E true (cond D true F))
                (cond C (cond E true (cond D true F))
                        false)))
```

and (cond E true (cond D true F)) is duplicated in three places. Our simplifier converts this formula back to a formula involving the usual boolean connectives, but the present version of the simplifier does not find the original, simplest form of the expression.

Another problem occurs when simplifying conjunctions like $x \leq y \wedge y \leq x \wedge x = y$. The simplifier discovers that the last equality is redundant and simplifies the conjunction to $x \leq y \wedge y \leq x$ instead of to $x = y$. (Had the equality appeared first, both inequalities would have been removed as redundant.) Handling this problem requires extending the set of primitives for manipulating contexts. For example, if a call to ASSERT made earlier conjuncts in the context redundant, this might be detected and exploited.

A significant problem concerns implementing the test A = B in step 4 of our simplification algorithm. This is intended to collapse branches of the cond tree which lead to identical results. If A or B are atomic symbols, there is no problem. If they contain conds, testing for logical equivalence is possible but probably impractical. If they contain no conds, then testing them for equality (using the lisp EQUAL) will usually be sufficient, if SIMPATOM puts expressions into a canonical form. There is, however, a difficulty: consider (cond (= X 1) (F I) (F X)), which we would like to simplify to (F X). Our SIMPATOM chooses (F 1), not (F X), as the canonical form when X = 1 is known, so in step 4 A is (F 1) and B is (F X). A completely adequate test for collapsing the two branches would require testing whether $Q \wedge P$ entailed A = B, in which case B should be returned, otherwise whether $Q \wedge \neg P$ entailed A = B, in which case A should be returned. (Qis the context of F, which is of the form (cond P A B).) Again the overhead may be prohibitive. This problem actually arises frequently and is more troublesome in practice than any of the other problems we have mentioned in this section.

4. Notes

The language accepted by the simplifier is richer than that described in section 1. Ail predicates (including =) and boolean connectives are considered boolean-valued functions (that is, functions which evaluate to the booieans true and false). Terms are allowed to contain arbitrary boolean-valued expressions. Expressions are allowed as functions. The following simplifications illustrate this generality.

```
F(true) ⊃ F(X ∨ ¬X);
true;

cond(true, F, G)(X)
F(X);
```

Our simplifier does not enforce strict typing. For instance, cons(X, Y) + store(V, I, E) is an acceptable expression (that the simplifier will simplify to itself). We plan to add type predicates (or type constants and a type function) to the next version of our simplifier.

The simpiifer does not store conjunctions of atomic formulas as strings or LISP s-expressions, but instead in a graph with one vertex for each term and **subterm** in the conjunction. Another data structure is used to represent an equivalence relation on the vertices. Two vertices are equivalent if the terms they represent are known to be equal in this context. To propagate an equality, a satisfiabiiity procedure merges two equivalence classes; this can be done very efficiently. More details of this representation are given in [Nelson and Oppen 19781.

Using this representation, it is not necessary to generate "labels" for terms which appear in in homogeneous iiterais.

This representation also allows the implementation of other routines in our simplifier to be more efficient, such as PUSH and POP. Obviously, one way to implement PUSH would be to have it make a physical copy of the existing context; this is not very satisfactory. The approach we take is to keep a ***history*** of all changes we make to our global data structure; popping then involves undoing these changes until we reach the context of the last call to PUSH.

Our simplifier is not a general purpose theorem prover; it cannot prove quantified theorems of the predicate calculus. However, in the Stanford Verifier, it is used in conjunction with a program called the rule ***handler*** which accepts user-supplied lemmas. During a simplification, the rule handier instantiates the free variables of the lemmas and sends the instantiated lemmas to the simplifier. In our system, the rule handier stands in the same relation to the simplifier as the satisfiabiiity programs. The rule handier can be viewed as a satisfiability program driven by user-supplied axioms.

Acknowledgment

We thank the Stanford Verification group for their patience in waiting two years for this simplifier.

References

[Craig 1957] W. Craig, "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory", Journal of Symbolic Logic, volume 22.

[Downey and Sethi 1976] P. Downey and R. Sethi, "Assignment Commands and Array Structures", manuscript.

[Johnson and Tarjan 19771 D. S. Johnson and R. E. Tarjan, "Finding Equivalent Expressions", manuscript.

[Kaplan 1968] D. M. Kaplan, "Some Completeness Results in the Mathematical Theory of Computation", Journal of the ACM, volume 15.

[McCarthy 19631 J. McCarthy, "A Basis for a Mathematical Theory of Computation", in Computing Programming and Formal Systems, edited by P. Braffort and D. Hirshberg, North-Holland.

[Nelson 1978] C. G. Nelson, "The Simplex Algorithm in Mechanical Theorem Proving", in preparation.

[Nelson and Oppen 1978] C. G. Nelson and D. C. Oppen, "Fast Decision Algorithms based on Congruence Closure'", AI Memo AIM309, CS Report No. STAN-CS-77-646, Stanford University. (An earlier version appeared in the Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science, October 1977.)

[Oppen 1978] D. C. Oppen, "Reasoning about Recursively Defined Data Structures", Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, January 1978.

[Shostak 1978] R. Shostak, "An Efficient Decision Procedure for Arithmetic with Function Symbols", to appear JACM.

[Suzuki and Jefferson 1977] N. Suzuki and D. Jefferson, "Verification Decidability of Presburger Array Programs," Proceedings of a Conference on Theoretical Computer Science, University of Waterloo, A ugust 1977.