Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT

Clark W. Barrett, David L. Dill, and Aaron Stump

barrett|dill|stump@cs.stanford.edu, http://verify.stanford.edu

Abstract. In the past few years, general-purpose propositional satisfiability (SAT) solvers have improved dramatically in performance and have been used to tackle many new problems. It has also been shown that certain simple fragments of first-order logic can be decided efficiently by first translating the problem into an equivalent SAT problem and then using a fast SAT solver. In this paper, we describe an alternative but similar approach to using SAT in conjunction with a more expressive fragment of first-order logic. However, rather than translating the entire formula up front, the formula is incrementally translated during a search for the solution. As a result, only that portion of the translation that is actually relevant to the solution is obtained. We describe a number of obstacles that had to be overcome before developing an approach which was ultimately very effective, and give results on verification benchmarks using CVC (Cooperating Validity Checker), which includes the Chaff SAT solver. The results show a performance gain of several orders of magnitude over CVC without Chaff and indicate that the method is more robust than the heuristics found in CVC's predecessor, SVC.

Key Words: Satisfiability, Decision Procedures, Propositional Satisfiability, First-Order Logic.

1 Introduction

Automated tools to check the satisfiability (or dually, the validity) of formulas are of great interest because of their versatility. Many practical problems can be reduced to the question of whether some formula is valid in a given logical theory.

Different approaches have been taken to developing general-purpose decision procedures. At one extreme, propositional satisfiability (SAT) solvers are blazingly fast, but only operate on propositional formulas, a very limited input language. At another extreme, general purpose first- or higher-order theorem provers are capable of proving some sophisticated results, but since their logics are undecidable, a result cannot be guaranteed.

A middle road is to develop fast decision procedures for specific decidable first-order theories. One interesting way to do this which has recently seen a lot of research activity is to translate the problem to SAT and then use a fast

SAT solver to obtain a solution. By using appropriate tricks to reduce the time and space required for the translation, this approach seems to work well for simple theories such as the theory of equality with uninterpreted functions [3, 12]. However, it is not clear how or whether such an approach would work for other decidable theories.

We propose a method designed to be more generally applicable: given a satisfiability procedure Sat_{FO} for a conjunction of literals in some first-order theory, a fast SAT-based satisfiability procedure for *arbitrary* quantifier-free formulas of the theory can be constructed by abstracting the formula to a propositional approximation and then incrementally refining the approximation until a sufficiently precise approximation is obtained to solve the problem. The refinement is accomplished by using Sat_{FO} to diagnose conflicts and then adding the appropriate *conflict clauses* to the propositional approximation.

In Section 2, we briefly review propositional satisfiability. We then describe the problem in Section 3. Section 4 describes our approach to solving the problem using SAT, and Section 5 describes a number of difficulties that had to be overcome in order to make the approach practical. Section 6 describes some related work, and in Section 7, we give results obtained using CVC [15], a new decision procedure for a combination of theories in a quantifier-free fragment of first-order logic which includes the SAT solver Chaff [10]. We compare with results using CVC without Chaff and with our best previous results using SVC [1], the predecessor to CVC. The new method is generally faster, requires significantly fewer decisions, and is able to solve examples that were previously too difficult.

2 Propositional Satisfiability

The SAT problem is the original classic NP-complete problem of computer science. A propositional formula is built as shown in Fig. 1 from propositional variables (i.e. variables that can either be assigned *true* or *false*) and Boolean operators ($\wedge$, $\vee$, $\neg$). Given such a formula, the goal of SAT is to find an assignment of *true* or *false* to each variable which results in the entire formula being *true*.

Instances of the SAT problem are typically given in conjunctive normal form (CNF). As shown in Fig. 1, CNF requires that the formula be a conjunction of *clauses*, each of which is a disjunction of propositional literals. In Section 4.1, we describe a well-known technique for transforming any propositional formula into an equisatisfiable propositional formula in conjunctive normal form.

Although the SAT problem is NP-complete, a wide variety of techniques have been developed that enable many examples to be solved very quickly. A large number of publicly distributed algorithms and benchmarks are available [14]. Chaff [10] is a SAT solver developed at Princeton University. As with most other SAT solvers, it requires that its input be in CNF. It is widely regarded as one of the best performing SAT solvers currently available.

```
propositional formula ::= true | false | propositional variable
      | propositional formula ∧ propositional formula
      | propositional formula ∨ propositional formula
      | ¬propositional formula

CNF formula ::= (clause ∧ ... ∧ clause)

clause ::= (propositional literal ∨ ... ∨ propositional literal)

propositional literal ::= propositional variable
      | ¬propositional variable
```

Fig. 1. Propositional logic and CNF

3 The Problem

```
formula ::= true | false | literal
      | term = term
      | predicate symbol(term, ..., term)
      | formula ∧ formula
      | formula ∨ formula
      | ¬formula

literal ::= atomic formula | ¬atomic formula

atomic formula ::= atomic term = atomic term
      | predicate symbol(atomic term, ..., atomic term)

term ::= atomic term
      | function symbol(term, ...,term)
      | ite(formula,term,term)

atomic term  ::= variable | constant symbol
      | function symbol(atomic term, ...,atomic term)
```

Fig. 2. A quantifier-free fragment of first-order logic.

We will show how to use SAT to aid in determining the satisfiability of a formula ϕ in a language which is much more expressive than propositional logic: the basic variant of quantifier-free first-order logic shown in Fig. 2. Note that in the remainder of the paper, the term "literal" by itself will be used to refer to an atomic formula or its negation, as defined in Fig. 2. This differs from the term "propositional literal" which we will use as in the previous section to mean a propositional variable or its negation. A small difference between this logic and conventional first-order logic is the inclusion of the **ite** (*if-then-else*) operator which makes it possible to compactly represent a term which may have one of two values depending on a Boolean condition, a situation which is common in applications. An **ite** expression contains a formula and two terms. The semantics are that if the formula is true, then the value of the expression is the first term, otherwise the value of the expression is the second term. Note that while both formulas and terms may contain proper Boolean sub-expressions, *atomic* formulas and *atomic* terms do not.

Formulas in the logic of Fig. 2 are intended to be interpreted with respect to some first-order theory which gives meaning to the function, predicate, and constant symbols in the formula. The theory of integer linear arithmetic, for example, defines function symbols like "+" and "-", predicate symbols like "$<$", and "$>$", and arbitrary integer constant symbols. For a given theory and formula, the formula is *satisfiable* if it is possible to assign values to the variables in the formula from elements of the domain associated with the theory in a way that makes the formula true.

Significant research has gone into fast algorithms for determining the satisfiability of conjunctions of literals with respect to some logical theory (or combination of theories) [2, 11, 13]. CVC, for example, is such a decision procedure which includes theories for arithmetic, arrays, abstract data types, and uninterpreted functions. We do not address the issue of constructing such decision procedures here, but rather assume that we are given a decision procedure Sat_{FO} for determining the satisfiability, with respect to a theory of interest, of a conjunction of literals in the logic of Fig. 2.

The problem we will address is how to use such a decision procedure to construct an efficient SAT-based decision procedure for the satisfiability of arbitrary formulas (i.e. not just conjunctions of literals).

4 Checking Satisfiability of Arbitrary Formulas using SAT

Suppose we have, as stated, a core decision procedure Sat_{FO} for determining the satisfiability of conjunctions of literals, and we wish to determine whether an arbitrary formula ϕ is satisfiable.

An obvious approach would be to use propositional transformations (such as distributivity and DeMorgan's laws) to transform ϕ into a logically equivalent disjunction of conjunctions of literals and then test each conjunct for satisfiability using Sat_{FO}. Unfortunately, this transformation can increase the size of the formula exponentially, and is thus too costly in practice.

The approach taken by CVC's predecessors is equivalent to the recursive algorithm shown in Fig. 3. The algorithm takes two parameters: the decisions made so far, and the formula whose satisfiability is in question. The formula is first simplified relative to the decisions. Then, a number of base cases are checked: if the formula is *false* or the decisions are inconsistent, the algorithm returns $\emptyset$ (indicating that no satisfying assignment was found); if the formula is *true*, then the set *decisions* describes a consistent state in which the formula is satisfied. If none of the base cases hold, then a case-split is done on the first

```
CheckSat(decisions,φ)
  φ := Simplify(decisions,φ);
  IF φ ≡ false THEN RETURN ∅;
  IF ¬Sat_FO(decisions) THEN RETURN ∅;
  IF φ ≡ true THEN RETURN decisions;
  Choose the first atomic formula α appearing in φ.
  result := CheckSat(decisions ∪ {α},φ);
  IF result = ∅ THEN
    result := CheckSat(decisions ∪ {¬α},φ);
  RETURN result;
```

Fig. 3. Simple recursive algorithm for checking satisfiability

atomic formula α in ϕ. The algorithm is then called recursively: first considering the case when α is *true* and then considering the case when α is *false*. Although this approach is straightforward and works well in some cases, it is not very robust: small changes or differences in formulas can cause a dramatic change in the number of decisions made and the amount of time taken.

Our new approach is designed to be fast and robust. The key idea is to incrementally form a propositional abstraction of a first-order formula. Consider an abstraction function *Abs* which maps first-order formulas to propositional formulas. It is desirable that the abstraction have the following two properties:

1. For any formula ϕ, if $Abs(\phi)$ is unsatisfiable, then ϕ is unsatisfiable.
2. If $Abs(\phi)$ is satisfiable, then the abstract solution can either be translated back into a solution for ϕ or be used to refine the abstraction.

We first describe a process for determining an appropriate initial propositional abstraction *Abs*. We then describe how to refine the abstraction if the proof attempt is inconclusive.

4.1 Computing a Propositional Abstraction of a First-Order Formula

The basic idea of the process is to replace non-propositional formulas with propositional variables. Each syntactically distinct atomic formula α is replaced with a new propositional variable, p_α. Syntactically identical atomic formulas are replaced with the same propositional variable.

The result would be a purely propositional formula if not for the **ite** operator. Handling this operator requires a bit more work. We use a transformation which preserves satisfiability and eliminates the **ite** expressions. First, each **ite** term t is replaced with a new term variable v_t. Again, syntactically identical terms are replaced with the same variable. Then for each syntactically distinct term $t \equiv \mathbf{ite}(a, b, c)$ that is replaced, the following formula is conjoined to the original formula: $(a \rightarrow v_t = b) \wedge (\neg a \rightarrow v_t = c)$. By repeating this process, all **ite** operators can be eliminated (in linear time), and in the resulting formula, all terms are atomic. Atomic formulas can then be replaced by propositional variables, as described above, and the resulting formula is purely propositional.

To convert the resulting propositional formula to CNF in linear time, we employ a standard technique [8]: a new propositional variable is introduced for each syntactically distinct non-variable sub-formula. Then, a set of CNF clauses is produced for each sub-formula which describes the relationship of the formula to its children. The translations for each of the standard Boolean operators are as follows.

$$\begin{aligned} a &:= \neg b \longrightarrow (a \vee b) \wedge (\neg a \vee \neg b) \\ a &:= b \wedge c \longrightarrow (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b) \wedge (\neg a \vee c) \\ a &:= b \vee c \longrightarrow (\neg a \vee b \vee c) \wedge (a \vee \neg b) \wedge (a \vee \neg c) \end{aligned}$$

Now, suppose that $Abs(\phi)$ is satisfiable and that the solution is given as a conjunction ψ of propositional literals. This solution can be converted into an equivalent first-order solution by inverting the abstraction mapping on the solution (replacing each propositional variable p_α in ψ with α). Call the result $Abs^{-1}(\psi)$. Since $Abs^{-1}(\psi)$ is a conjunction of literals, its satisfiability can be determined using Sat_{FO}. If $Abs^{-1}(\psi)$ is satisfiable, then in the interpretation which satisfies it, the original formula ϕ must reduce to *true*, and thus ϕ is satisfiable. Otherwise, the result of the experiment is inconclusive, meaning that the abstraction must be refined. We describe how to do this next.

4.2 Refining the Abstraction

An obvious approach to refining the abstraction is to add a clause to the propositional formula that rules out the solution determined to be invalid by Sat_{FO}. Since ψ is a conjunction of propositional literals, applying de Morgan's law to $\neg\psi$ yields a standard propositional clause. Thus, $Abs(\phi) \wedge \neg\psi$ is a refinement of the original abstraction which rules out the invalid solution ψ. Furthermore, the refinement is still in CNF as required. We call the clause $\neg\psi$ a a *conflict clause* because it captures a set of propositional literals which conflict, causing an inconsistency. This is in accordance with standard SAT terminology. However, in standard SAT algorithms, conflict clauses are obtained by analyzing a clause which has become false to see which decisions made by the SAT solver are responsible. In our approach, the conflict clause is obtained by an agent outside of the SAT solver. After refining the abstraction by adding a conflict clause, the SAT algorithm can be restarted. By repeating this process, the abstraction will hopefully be refined enough so that it can either be proved unsatisfiable by the SAT solver or the solution ψ provided by SAT can be shown to map to a satisfying assignment for the original formula.

5 The Difficult Path to Success

There are a surprising number of roadblocks on the way from the previous idea to a practical algorithm. In this section we describe some of these and our solutions.

5.1 Redundant Clauses

The most severe problem with the naive approach outlined above is that it tends to produce an enormous number of redundant clauses. To see why, suppose that SAT computes a solution consisting of $n + 2$ propositional literals, but that only the last two propositional literals contribute to the inconsistency of the equivalent first-order set of literals. Then, for each assignment of values to the other n propositional variables which leads to a satisfying solution, the refinement loop will have to add another clause. In the worst case, the refinement loop will have to add 2^n clauses. This is particularly troubling because a single clause, one containing just the two contributing propositional literals would suffice.

In order to avoid the problem just described, the refinement must be more precise. In particular, when Sat_{FO} is given a set of literals to check for consistency, an effort must be made to find the *smallest* possible subset of the given set which is still inconsistent. Then, a clause derived from *only these* literals can be added to the propositional formula.

One possible way to implement this is to minimize the solution by trial and error: starting with n literals, pick one of the literals and remove it from the set. If the set is still inconsistent, leave that literal out; otherwise, return it to the set. Continue with each of the other literals. At the end, the set will contain a minimal set of literals. Unfortunately, this approach requires having Sat_{FO} process $O(n)$ literals n times for each iteration of the refinement loop (where n is the number of variables in the abstract formula). A few experiments with this approach quickly reveal that it is far too costly to give a practical algorithm.

A more practical solution, though one which is not trivial to implement, is to have the decision procedure Sat_{FO} maintain enough information to be able to report directly which subset of a set of inconsistent literals is responsible for the inconsistency.

Fortunately, through a discussion with Cormac Flanagan [6], we realized that this is not difficult to do in CVC. This is because CVC is a proof-producing decision procedure, meaning that it is possible to have CVC generate an actual proof of any fact that it can prove. Using the infrastructure for proof production in CVC, we implemented a mechanism for generating *abstract proofs*. In abstract proof mode, CVC just tracks the external assumptions that are required for each proof. The result is that when a set of literals is reported by CVC to be inconsistent, the abstract proof of inconsistency contains exactly the *subset* of those literals that would be used to generate a proof of the inconsistency. The abstract proof thus provides a subset which is known to be inconsistent. This subset is not guaranteed to be minimal, but we found that in most cases it is very close to minimal. Since the overhead required to keep track of abstract proofs is small (typically around 20%), abstract proofs provide an efficient and practical solution for eliminating the problem of redundant clauses.

5.2 Lazy vs. Eager Notification

The approach described in the previous section is *lazy* (see the note in Section 6 below) in the sense that the SAT solver is used as a black box and the first-order procedure Sat_{FO} is not invoked until a solution is obtained from the SAT solver. Unfortunately, as shown in Table 3, the lazy approach becomes impractical for problems which require many refinements. In contrast, an *eager* approach is to notify the first-order procedure Sat_{FO} of every decision that is made (or unmade) by the SAT solver. Then, if an inconsistency is detected by Sat_{FO}, it is immediately diagnosed, providing a new conflict clause for SAT. The SAT algorithm then continues, never having to be restarted.

The performance advantages of the eager approach are significant. The disadvantages are that it requires more functionality of both the SAT solver and the decision procedure Sat_{FO}. The SAT solver is required to give notification every time it makes (or revokes) a decision. Furthermore, it must be able to accept new clauses in the middle of solving a problem (CVC includes a modified version of Chaff which has this functionality). The eager approach also requires Sat_{FO} to be *online*: able quickly to determine the consistency of incrementally more or fewer literals. Fortunately, CVC has this property.

5.3 Decision Heuristics

The decision heuristics used by Chaff and other SAT solvers consider every variable a possible target when choosing a new variable to do a case split on. However, in the abstracted first-order formula, not all variables are created equally. For example, consider an **ite** expression: $\mathbf{ite}(\alpha, t_1, t_2)$, and suppose that t_1 and t_2 are both large non-atomic terms. If the propositional variable associated with α is set to *true*, then all of the clauses generated by the translation of t_2 can be ignored since they can no longer affect the value of the original formula. Unfortunately, the SAT solver doesn't have this information, and as a result it can waste a lot of time choosing irrelevant variables. This problem has been addressed by others [5], and our solution is similar. We annotate the propositional variables with information about the structure of the original formula (i.e. parent/child relationships). Then, rather than invoking the built-in heuristic for variable selection, a depth-first search (DFS) is performed on the portion of the original formula which is relevant. The first variable corresponding to an atomic formula which is not already assigned a value is chosen. Although this can result in sub-optimal variable orders in some cases, it avoids the problem of splitting on irrelevant variables. Table 4 compares results obtained using the built-in Chaff decision heuristic with those obtained using the DFS heuristic. These are discussed in Section 7.

5.4 SAT Heuristics and Completeness

A somewhat surprising observation is that some heuristics used by SAT solvers must be disabled or the method will be incomplete. An example of this is the "pure literal" rule. This rule looks for propositional variables which have the property that only one of their two possible propositional literals appears in the formula being checked for satisfiability. When this happens, all instances of the propositional literal in question can immediately be replaced with *true*, since if a solution exists, a solution will exist in which that propositional literal is *true*.

However, if the formula is an abstraction of a first-order formula, it may be the case that a solution exists when the propositional literal is *false* even if a solution does not exist when the propositional literal is *true*. This is because the propositional literal is actually a place-holder for a first-order literal whose truth may affect the truth of other literals. Propositional literals are guaranteed to be independent of each other, while first-order literals are not. Because of this, there is no obvious way to take advantage of pure literals and the rule must be disabled. Fortunately, this was the only such rule that had to be disabled in Chaff.

5.5 Theory-specific Challenges

Finally, a particularly perplexing difficulty is dealing with first-order theories that need to do case splits in order to determine whether a set of literals is satisfiable. For example, consider a theory of arrays with two function symbols, *read* and *write*. In this theory, $read(a, i)$ is a term which denotes the value of array a at index i. Similarly, the term $write(a, i, v)$ refers to an array which is identical to a everywhere except possibly at index i, where its value is v. Now, consider the following set of literals in this theory: $\{read(write(a, i, v), j) = x, x \neq v, x \neq a[i]\}$. In order for the array decision procedure to determine that such a set of literals is inconsistent, it must first do a case split on $i = j$. However, such additional case splits by the theories can cost a lot of time. Furthermore, they may not even be necessary to solve the problem. We found it difficult to find a strategy for integrating such case splits without adversely affecting performance. Our solution was to preprocess the formulas to try to eliminate such case splits. In particular, for the array theory, every instance of $read(write(a, i, v), j)$ is rewritten to $\mathbf{ite}(i = j, v, read(a, i))$. Furthermore, in order to increase the likelihood of being able to apply this rewrite, every instance of $read(\mathbf{ite}(a, b, c), v)$ is rewritten to $\mathbf{ite}(a, read(b, v), read(c, v))$. These rewrites were sufficient to obtain reasonable performance for our examples. However, we suspect that for more complicated examples, something more sophisticated may be required.

6 Related Work

Flanagan, Joshi, and Saxe at Compaq SRC have independently developed a very similar approach to combining first-order decision procedures with SAT [7]. Their translation process is identical to ours. Furthermore, their approach to generating conflict clauses is somewhat more sophisticated than ours. However, their prototype implementation is lazy (the nomenclature of "lazy" versus "eager" is theirs). Also it only includes a very limited language and its performance is largely unknown. Unfortunately, we have not been able to compare directly with their implementation.

De Moura, Ruess, and Sorea at SRI have also developed a similar approach using their ICS decision procedure [4]. However, ICS is unable to produce minimal conflict clauses, so they use an optimized variation of the trial and error method described in Section 5.1 to minimize conflict clauses. Also, as with the Compaq approach, their implementation is lazy and its performance unknown. Though they do not report execution times, they do provide their benchmarks, and our implementation using CVC with Chaff was able to solve all of them easily.

It would also be interesting to compare with the approach for solving problems in the logic of equality with uninterpreted functions by translating them (up front) to SAT problems. We made an attempt to perform direct comparisons with [12], but their benchmarks are not provided in the language of equality with uninterpreted functions, and unfortunately, it is not clear how to translate them. As a result, we were unable to run their benchmarks. We suspect that our approach would be competitive with theirs. However, since the logic is so simple, it is not clear that a more general approach like ours would be better.

7 Results

We implemented the approach described above in the CVC decision procedure using the Chaff SAT solver, and tested it using a suite of processor verification benchmarks. The first three benchmarks are purely propositional formulas from Miroslav Velev's superscalar suite (http://www.ece.cmu.edu/∼mvelev). The next three are also from Velev's DLX verification efforts, but they include array and uninterpreted function operations. The rest are from our own efforts in processor verification and also include array and uninterpreted function operations.

These were run using gcc under linux on an 800MHz Pentium III with 2GB of memory. The best overall results were obtained by using an eager notification strategy and the DFS decision heuristic. Table 1 compares these results to results obtained by using CVC without Chaff (using the recursive algorithm of Fig. 3). As can be seen, the results are better, often by several orders of magnitude, in every case but one (the easiest example which is solved by both methods very quickly). These results show that CVC with Chaff is a significant improvement over CVC alone.

Our goal in integrating Chaff into CVC was not only to test the feasibility of the approach, but also to produce a tool that could compete with and improve upon the best results obtained by our previous tool, SVC. SVC uses a set of clever but somewhat ad hoc heuristics to improve on the performance obtained by the algorithm of Fig. 3 by learning which atomic formulas are best to split on [9]. Table 2 compares the results obtained by SVC with the results obtained by CVC with Chaff.

SVC performs particularly well on the last 6 examples, a fact which is not too surprising since these are old benchmarks that were used to tune SVC's heuristics. However, SVC's performance on the first six examples shows that it's

Table 1. Results comparing CVC without Chaff to CVC combined with Chaff

Example	CVC without Chaff		CVC+Chaff	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	?	> 10000	2522	1.14
bool-dlx2-aa	?	> 10000	792	0.81
bool-dlx2-cc-bug01	?	> 10000	573387	833
v-dlx-pc	8642456	5082	6137	6.10
v-dlx-dmem	2888268	2820	2184	3.48
v-dlx-regfile	29435	37.6	3833	6.64
dlx-pc	515	0.68	529	1.04
dlx-dmem	6031	4.50	1276	1.90
dlx-regfile	6386	5.27	2739	4.12
pp-bloaddata-a	93714	79.1	1193	1.80
pp-bloaddata	345569	338	4451	4.51
pp-dmem2	367877	338	2070	1.52

heuristics are simply not flexible enough to handle a large variety of formulas. CVC, on the other hand produces good results fairly consistently. Even in the four cases where CVC is slower than SVC, the number of decisions is comparable, and in all other cases the number of decisions required by CVC is much less. This is encouraging because it means that CVC is finding shorter proofs, and additional performance gains can probably be obtained by tuning the code. Thus, overall, CVC seems to perform better and to be more robust than SVC, which is the goal we set out to accomplish.

Table 2. Results comparing SVC to CVC

Example	SVC		CVC+Chaff	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	11228452	776	2522	1.14
bool-dlx2-aa	?	> 10000	792	0.81
bool-dlx2-cc-bug01	?	> 10000	573387	833
v-dlx-pc	4620149	503	6137	6.10
v-dlx-dmem	199540	31.7	2184	3.48
v-dlx-regfile	74600	18.2	3833	6.64
dlx-pc	384	0.15	529	1.04
dlx-dmem	655	0.21	1276	1.90
dlx-regfile	936	0.27	2739	4.12
pp-bloaddata-a	902	0.66	1193	1.80
pp-bloaddata	35491	5.35	4451	4.51
pp-dmem2	47989	7.54	2070	1.52

7.1 Comparing Different Strategies

Finally, we show experimental results for some of the different strategies discussed in the previous section. First, just to drive the point home, we show a simple comparison of the naive (lazy without minimal conflict clauses), lazy (with minimal conflict clauses), and eager (with minimal conflict clauses) implementations on some simple examples. As can be seen, the naive and lazy approaches quickly become impractical.

Table 3. Results comparing naive, lazy, and eager implementations

Example	Naive		Lazy		Eager
	Iterations	Time (s)	Iterations	Time (s)	Time (s)
read0	77	0.14	17	0.09	0.07
pp-pc-s2i	?	> 10000	82	1.36	0.10
pp-invariant	?	> 10000	239	5.81	0.22
v-dlx-pc	?	> 10000	6158	792	3.22
v-dlx-dmem	?	> 10000	?	> 10000	4.12

Next, we compare two versions of the eager approach with minimal conflict clauses: one using the standard Chaff decision heuristics, and one using the DFS heuristic discussed in Section 5.3. The results are shown in Table 4. As can be seen, DFS outperforms the standard technique on all but four examples. Two of these are purely Boolean test cases, and so the DFS method wouldn't be expected to provide any advantage. For purely propositional formulas, then, (or first-order formulas that are *mostly* propositional), the standard Chaff technique is probably better. It is particularly interesting to note how badly DFS does on the example "bool-dlx2-cc-bug01". One area for future work is trying to find a way to automatically choose between or combine these two methods.

More information about these and other benchmarks (as well as the benchmarks themselves) is available from http://verify.stanford.edu/barrett/CAV02. CVC is available from http://verify.stanford.edu/CVC.

Acknowledgments

We'd like to thank the anonymous reviewers for many helpful suggestions. This work was partially supported by the National Science Foundation Grant CCR-9806889, and ARPA/AirForce contract number F33615-00-C-1693.

References

1. C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, pages 187–201, 1996.

Table 4. Variable selection by Chaff vs. by depth-first search

Example	Chaff		DFS	
	Decisions	Time (s)	Decisions	Time (s)
bool-dlx1-c	1309	0.69	2522	1.14
bool-dlx2-aa	4974	2.36	792	0.81
bool-dlx2-cc-bug01	10903	11.4	573387	833
v-dlx-pc	4387	3.22	6137	6.10
v-dlx-dmem	5221	4.12	2184	3.48
v-dlx-regfile	6802	5.85	3833	6.64
dlx-pc	39833	19.0	529	1.04
dlx-dmem	34320	18.8	1276	1.90
dlx-regfile	47822	35.5	2739	4.12
pp-bloaddata-a	8695	5.47	1193	1.80
pp-bloaddata	9016	5.56	4451	4.51
pp-dmem2	3167	2.24	2070	1.52

2. Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories.* PhD thesis, Stanford University, 2002.
3. R. Bryant, S. German, and M. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *11th International Conference on Computer-Aided Verification*, pages 470–482, 1999.
4. Leonardo de Moura, Harald Ruess, and Maria Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *18th International Conference on Automated Deduction*, 2002.
5. L. e Silva, L. Silveira, and J. Marques-Silva. Algorithms for Solving Boolean Satisfiability in Combinational Circuits. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, March 1999.
6. C. Flanagan. Private Communication, 2000.
7. Cormac Flanagan, Rajeev Joshi, and James B. Saxe. The Design of An Efficient Theorem Prover using Explicated Clauses. 2002. In Preparation.
8. Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
9. Jeremy R. Levitt. *Formal Verification Techniques for Digital Systems.* PhD thesis, Stanford University, 1999.
10. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 39th Design Automation Conference*, 2001.
11. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
12. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding Equality Formulas by Small-Domain Instantiations. In *11th International Conference on Computer-Aided Verification*, pages 455–469, 1999.
13. H. Ruess and N. Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, June 2001.
14. Laurent Simon. The Sat-Ex Site. http://www.lri.fr/~simon/satex/satex.php3.
15. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.