Theorem Proving using Lazy Proof Explication

Cormac Flanagan[1], Rajeev Joshi[1], Xinming Ou[2], and James B. Saxe[1]

[1] Systems Research Center, HP Labs, Palo Alto, CA
[2] Princeton University, Princeton, NJ

Abstract

Many verification problems reduce to proving the validity of formulas involving both propositional connectives and domain-specific functions and predicates. This paper presents an *explicating* theorem prover architecture that leverages recent advances in propositional SAT solving and the development of proof-generating domain-specific procedures. We describe the implementation of an explicating prover based on this architecture that supports propositional logic, the theory of equality with uninterpreted function symbols, linear arithmetic, and the theory of arrays. We have applied this prover to a range of processor, cache coherence, and timed automata verification problems. We present experimental results on the performance of the prover, and on the performance impact of important design decisions in our implementation.

1 Introduction

In 1979, Nelson and Oppen [?] introduced a scheme for combining a collection of decision procedures for disjoint underlying theories, together with backtracking search, to obtain a theorem prover for formulas incorporating both propositional connectives and arbitrarily mixed application of functions and predicates of the various theories. In this paper, we propose a prover architecture based on a new style of interaction between propositional and theory-specific decision procedures. Unlike the traditional Nelson-Oppen method, our architecture separates the propositional search from the work done by decision procedures for the underlying theories. It thereby allows us to gain efficiency by taking advantage of recent advances in propositional SAT solving and the development of proof-generating decision procedures.

In the rest of this introduction, we illustrate the key ideas of our approach by means of an example. In later sections, we describe our architecture in more detail, report on our experience with a prototype implementation, discuss various design choices that impact performance, and compare our work with related approaches.

1.1 Our Approach

To simplify the exposition, we consider the problem of determining whether a given formula, called the *query*, is *satisfiable* (this is the dual of the theorem-proving problem). For instance, consider checking the satisfiability of:

$$(a = b) \;\wedge\; (\neg(f(a) = f(b)) \;\vee\; b = c) \;\wedge\; \neg(f(a) = f(c)) \tag{1}$$

Our approach uses a propositional SAT solver together with suitable decision procedures. For this example, we need only one theory-specific decision procedure, for the theory of equality with uninterpreted function symbols (EUF). A more useful prover would employ a larger collection of theories, cooperating according to the Nelson-Oppen equality-sharing protocol.

We translate the given problem into a purely propositional formula by introducing propositional variables $v_1 \ldots v_4$, called *proxies*, as shown below:

$$(\overbrace{a = b}^{v_1}) \;\wedge\; (\,\neg(\overbrace{f(a) = f(b)}^{v_2}) \;\vee\; \overbrace{b = c}^{v_3}) \;\wedge\; \neg(\overbrace{f(a) = f(c)}^{v_4}) \tag{2}$$

Replacing each atomic formula in the query with the corresponding propositional proxy, we obtain the propositional formula:

$$(v_1) \;\wedge\; (\neg v_2 \vee v_3) \;\wedge\; (\neg v_4) \tag{3}$$

We refer to the atomic formula associated with a proxy as its *interpretation*; thus $a = b$ is the interpretation of v_1.

Formula (3) is an *abstraction* of query (1) in the sense that, given any satisfying assignment for (1) we can obtain a satisfying assignment for (3) simply by assigning to each proxy the truth value of its interpretation. Clearly, however, the converse does not hold in general, since the truth values are assigned to the proxies without considering their interpretations. Our strategy is to use the underlying theories to produce a sequence of successively stronger propositional abstractions until either (a) the propositional abstraction becomes unsatisfiable (in which case the original query was itself unsatisfiable), or (b) the abstraction remains satisfiable even when the proxies are interpreted as atomic formulas (in which case we have a satisfying assignment for the query).

We start by invoking a propositional SAT solver to solve the initial propositional abstraction (3). Suppose that our solver returns with the satisfying assignment that assigns `true` to v_1 and `false` to v_2, v_3 and v_4. Next, we assert the associated interpretations of these proxies to the underlying EUF theory, which detects that they are contradictory.

At this point, a conventional Nelson-Oppen prover like Simplify [12] would backtrack and search for a different satisfying assignment for (3), perhaps coming up with the assignment in which v_1 and v_3 are assigned `true`, while v_2 and v_4 are assigned `false`. This assignment would again be found inconsistent with EUF, and so on. Note, however, that $a = b$ and $\neg(f(a) = f(b))$ are mutually inconsistent by themselves. Thus, there is no point in considering any further assignments in which v_1 is assigned `true` and v_2 is assigned `false`.

To exploit this observation, we depart from the conventional approach and assume the existence of a decision procedure for EUF that, given a conjunction of inconsistent literals, is capable of producing a proof of the inconsistency.

Returning to our example, when the interpretations of the proxies are asserted to the EUF procedure, it reports the inconsistency of $a = b$ and $\neg(f(a) = f(b))$ by *explicating* the following "proof", which is an instance of the congruence axiom,

$$\overbrace{a = b}^{v_1} \;\Rightarrow\; \overbrace{f(a) = f(b)}^{v_2}$$

We use this proof to refine our propositional abstraction by adding the additional clause $(\neg v_1 \vee v_2)$, obtaining

$$(v_1) \;\wedge\; (\neg v_2 \vee v_3) \;\wedge\; (\neg v_4) \;\wedge\; (\neg v_1 \vee v_2) \tag{4}$$

Note that addition of this "explicated proof" to the propositional abstraction allows the SAT solver to refute the SAT assignment using purely propositional reasoning. Next, we invoke the SAT solver on (4). This time, it finds the satisfying assignment in which v_1, v_2, v_3 are assigned `true`, and v_4 is assigned `false`. As before, we assert the associated interpretations to the underlying EUF theory. The theory finds the assignment to be inconsistent, and explicates the following proof of inconsistency:

$$(\overbrace{a = b}^{v_1} \;\wedge\; \overbrace{b = c}^{v_3}) \;\Rightarrow\; \overbrace{f(a) = f(c)}^{v_4}$$

Using this proof, we refine our propositional abstraction to

$$(v_1) \;\wedge\; (\neg v_2 \vee v_3) \;\wedge\; (\neg v_4) \;\wedge\; (\neg v_1 \;\vee\; v_2) \;\wedge\; (\neg v_1 \;\vee\; \neg v_3 \;\vee\; v_4) \tag{5}$$

The SAT solver finds that (5) is now unsatisfiable, so we conclude that the original query (1) was itself unsatisfiable.

1.2 Motivation

The ideas described in this paper grew out of experience with the design and use of the theorem prover "Simplify" [12], which is based on the traditional Nelson-Oppen design. During our use of Simplify, we observed two serious performance problems. First, the backtracking search algorithm that we used for propositional inference had been far surpassed by recently developed fast SAT solvers [21, 24, 14]. Second, if the prover was in the midst of deeply nested case splits when it detected an inconsistency with an underlying theory, the backtracking search engine gained no information about which tentative truth assignments (besides the most recent) actually contributed to the inconsistency. Consequently the decision procedure was often forced repeatedly to rediscover the same inconsistency in later branches of the search, which differed only in the truth assignments to other, irrelevant atomic formulas. This led to our exploration of proof-generating decision procedures; such procedures essentially identify useful theory-specific facts, which are then projected into the propositional domain. We thus leverage the efficiency of modern SAT solvers, which can reuse the explicated clauses much more efficiently than the theory-specific decision procedure could regenerate them.

Our explicated clauses resemble the *conflict clauses* generated by modern SAT solvers such as GRASP [21], SATO [24], and Chaff [14, 25], in that (i) after being generated once they can be reused many times, and (ii) they are generated on the basis of their demonstrated utility in refuting some attempted satisfying assignment. The difference is that our explicated clauses are theory-specific logical consequences of the interpretations of the proxy variables, and they are discovered by the theory-specific decision procedures. In contrast, a SAT solver's conflict clauses are propositional consequences of the given clauses, and they are discovered by analyzing contradictions detected during boolean constraint propagation in the SAT solver.

We are by no means the only ones to notice that modern SAT solvers can be usefully integrated into the Nelson-Oppen design. Similar ideas have been recently proposed by Barrett, Dill and Stump [5] and by de Moura and Rueß [11]. Our approach differs from their work in a number of critical design decisions. This paper discusses these design decisions and evaluates their performance impact.

2 Architecture

2.1 Terminology

We use terminology that is standard in the literature. A *term* is a variable or an application of a function to terms. Thus, x, $x + 3$ and $f(x, y)$ are all terms. An *atomic formula* is a propositional variable or an application of a predicate symbol to some terms. Thus, $x + 3 < 5$ and $f(x) = f(y)$ are atomic formulae. A *literal* is either an atomic formula or its negation, and a *clause* is a disjunction of literals. A *monome* is a conjunction of literals in which no atomic formula is both affirmed and negated. We identify a monome M with the partial truth assignment that assigns `true` to atomic formulae that are conjuncts of M and assigns `false` to atomic formulae whose negations are conjuncts of M.

The task of a satisfier is to decide whether an input formula, called a *query*, is satisfiable for a given set of underlying theories. The underlying theories may assume particular semantics for some predicate and function symbols, such as $>$ and $+$, while leaving others uninterpreted. A *satisfying assignment* for a query is a monome that is consistent with the underlying theories of the satisfier and entails the query by propositional inference alone (i.e., treating all syntactically distinct atomic formulae as if they were distinct propositional variables).

2.2 Architecture

Figure 1 sketches the main algorithm for our satisfier. As shown, given a query F, we introduce proxies for the atomic formulae of F, along with a mapping Γ relating these proxies to the atomic formulae. This results in the SAT problem $\Gamma^{-1}(F)$. We invoke the SAT solver on $\Gamma^{-1}(F)$ by invoking *SAT-solve*, which is

```
Input: A query F
Output: A monome satisfying F, or null, indicating that F is unsatisfiable

function satisfy(Formula F) : Monome {
    while (true) {
        allocate proxy propositional variables for atomic formulae in F, and
            create mapping Γ from proxies to atomic formulae;
        TruthAssignment  A := SAT-solve(Γ^-1(F));
        if (A = null) {        Γ^-1(F) is unsatisfiable, hence so is F
            return null;
        } else {
            Monome  M := Γ(A);
            Formula  E := check(M);
            if (E = null) {        E is satisfiable, and so is F
                return Γ(A);
            } else {      decision procedure found M inconsistent and explicated E
                F := F ∧ E;
            }
        }
    }
}
```

Fig. 1. A satisfiability algorithm using proof explication

expected to return either **null** (if the given SAT problem is unsatisfiable) or a satisfying *TruthAssignment* A. If *SAT-solve* returns **null**, it means $\Gamma^{-1}(F)$ is unsatisfiable, so we can deduce that F itself was unsatisfiable, and we are done. Otherwise, we use the satisfying assignment A provided by the SAT solver, and invoke the procedure *check* which determines if the monome M is consistent with the underlying theories. If *check* returns **null**, it means that M is consistent with the underlying theories, so we have obtained a satisfying assignment to our original F and we are done. Otherwise, *check* determines that M is inconsistent with the underlying theories, and returns a proof E of the inconsistency. We update the query F by conjoining E to it, and continue by mapping the query using Γ and reinvoking the SAT solver as described above. We assume that E is (1) entailed by the axioms of the underlying theories, and (2) propositionally entails the negation of the given monome M. Condition (2) suffices to show that the algorithm terminates, since it ensures that at each step, we strictly strengthen the query. Condition (1) suffices to show soundness, so that if the updated query becomes propositionally unsatisfiable, we may conclude that the original query was unsatisfiable. Thus, we can view the iterative process as transfering information from the underlying theories into the propositional domain. Eventually we either strengthen the query until it becomes propositionally unsatisfiable, or the SAT solver finds a satisfying assignment whose interpretation is consistent with the underlying theories.

3 Implementation and Evaluation

To explore the performance benefits of generating explicated clauses, we implemented a satisfier, called **Verifun**, based on the architecture of Figure 1. Verifun consists of approximately 10,500 lines of Java and around 800 lines of C code. The bulk of the Java code implements explicating decision procedures for EUF, rational linear arithmetic, and the theory of arrays. The decision procedure for EUF is based on the *E-graph* data structure proposed by Nelson and Oppen [19], which we adapted to explicate proofs of transitivity and congruence. The decision procedure for linear arithmetic is based on a variation [17] of the Simplex algorithm that we have modified to support proof explication. Finally, the decision procedure for arrays uses pattern matching to produce ground instances of select and store axioms. The C code implements the interface to the SAT solver.

The Verifun implementation extends the basic explicating architecture of Figure 1 with a number of improvements, which we describe below.

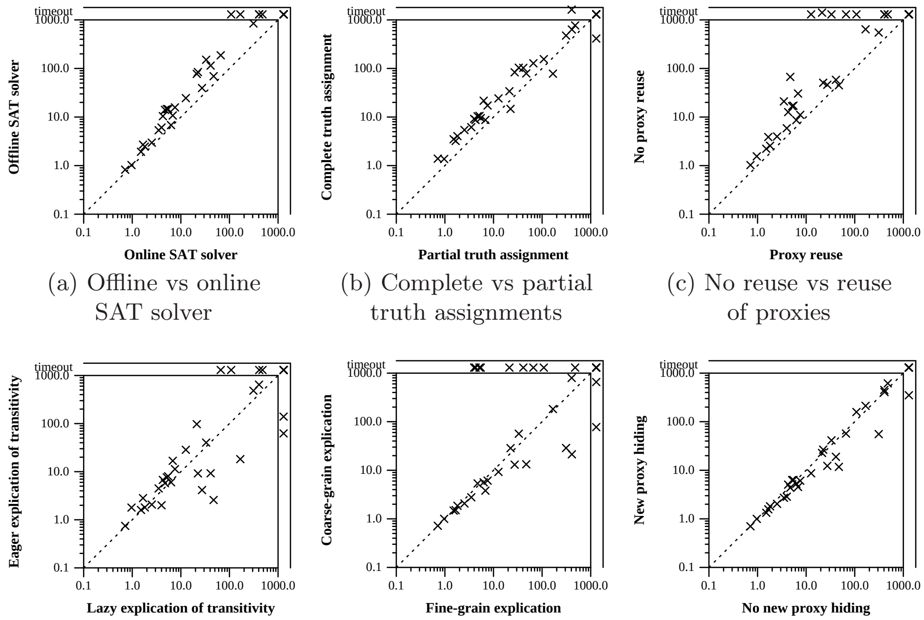
3.1 Online vs. Offline Propositional SAT Solving

The architecture of Figure 1 uses an *offline* SAT solver, which is invoked from scratch each time the SAT problem is extended with additional explicated clauses. Each sucessive invocation is likely to repeat much of the work of the previous invocation. To avoid this performance bottleneck, Verifun uses a customized *online* variant of the zChaff SAT solver [14]. After reporting a satisfying assignment, this online SAT solver accepts a set of additional clauses and then continues its backtracking search from the point at which that assignment was found, and thus avoids reexamining the portion of the search space that has already been refuted.

To illustrate the benefit of online SAT solving, Figure 2(a) compares the performance of two versions of Verifun, which use online and offline versions of zChaff, respectively. We used a benchmark suite of 38 processor and cache coherence verification problems provided by the UCLID group at CMU [7]. These problems are expressed in a logic that includes equality, uninterpreted functions, simple counter arithmetic (increment and decrement operations), and the usual ordering over the integers. All experiments in this paper were performed on a machine with dual 1GHz Pentium III processors and 1GB of RAM, running Redhat Linux 7.1. Since Verifun is single-threaded, it uses just one of the two processors. We consider an invocation of the prover to *timeout* if it took more than 1000 seconds, ran out of memory, or otherwise crashed. As expected, the online SAT solver significantly improves the performance of Verifun and enables it to terminate on more of the benchmark problems. Note that the the 'x' in the top right of of this graph (and in subsequent graphs) covers several benchmarks that timed out under both Verifun configurations.

3.2 Partial vs. Complete Truth Assignments

As described in Section 2.2, the SAT solution A is converted to a monome M that entails the query by propositional reasoning (although M may not be consistent



(a) Offline vs online SAT solver

(b) Complete vs partial truth assignments

(c) No reuse vs reuse of proxies

(d) Eager vs lazy transitivity

(e) Fine vs coarse explication

(f) New proxy hiding vs no hiding

Fig. 2. Scattergraph of running times (in seconds) comparing versions of Verifun on the UCLID benchmarks. Except where labeled otherwise, Verifun used the online SAT solver, partial truth assignments, proxy reuse, lazy transitivity, fine-grain explication, and no hiding of new proxy variables.

with the semantics of atomic formulas as defined by the theories). By default, M is *complete*, in that it associates a truth value with every atomic formula in the query. An important optimization is to compute from M a minimal sub-monome M' that still entails the query. Since any monome extending M' also entails the query, M can be considered a random extension of M' that assigns arbitrary truth values to atomic formulas not mentioned in M', which in turn may cause *check* to explicate clauses that are not very useful. Therefore, we instead apply *check* to the partial monome or truth assignment M'. Figure 2(b) illustrates the benefit of this optimization.

3.3 Proxy Reuse

Since standard SAT solvers require their input to be in conjunctive normal form (CNF), Verifun first converts the given query to CNF. To avoid exponential blow-up, Verifun introduces additional *proxy* variables for certain terms in the query, as necessary. If a particular term appears multiple times in the query, an important optimization is to reuse the same proxy variable for that term, instead of introducing a new proxy variable for each occurrance of the term. Figure 2(c) illustrates the substantial performance benefit of this optimization.

3.4 Eager Transitivity

By default, Verifun explicates clauses in a *lazy* manner, in response to satisfying assignments produced by the SAT solver. An alternative approach proposed by Velev [9] and others [7, 8, 13] is to perform this explication *eagerly*, before running the SAT solver. The relative performance of the two approaches is unclear, in part because lazy explication generates fewer clauses, but invokes the SAT solver multiple times.

As a first step in comparing the two approaches, we extended Verifun to perform eager explication of clauses related to transitivity of equality. These clauses are generated by maintaining a graph whose vertices are the set of all terms, and whose edges are the set of all equalities that appear (negated or not) in the current query. At each step, before the SAT solver is invoked, we add edges to make the graph chordal, using the well-known linear-time algorithm of Tarjan and Yannakakis [23]. Next, we enumerate all triangles in this graph that contain at least one newly added edge. For each such triangle, we generate the three possible instances of the transitivity axiom. Figure 2(d) illustrates the effect of eager explication on Verifun's running time on the UCLID benchmarks. Interestingly, although eager explication significantly reduces the number of iterations though the main loop of Verifun, often by an order of magnitude, the timing results indicate that eager explication does not produce a consistent improvement in Verifun's performance over lazy explication.

3.5 Granularity of Explication

When the *check* decision procedure detects an inconsistency, there is generally a choice about which clauses to explicate. To illustrate this idea, suppose the decision procedure is given the following inconsistent collection of literals:

$$a = b\ ,\ b = c\ ,\ f(a) \neq f(c)$$

When the decision procedure detects this inconsistency, it could follow the *coarse-grain* strategy of explicating this inconsistency using the single clause:

$$a = b \wedge b = c \ \Rightarrow\ f(a) = f(c)$$

A second strategy is *fine-grain* explication, whereby the decision procedure explicates the proof of inconsistency using separate instances of the transitivity and congruence axioms:

$$\begin{aligned} a = b \wedge b = c \ &\Rightarrow\ a = c \\ a = c \ &\Rightarrow\ f(a) = f(c) \end{aligned}$$

Fine-grained explication produces more and smaller explicated clauses than coarse-grained explication. This is likely to result in the SAT solver doing more unit propagation, but may allow the SAT solver to refute more of the search space without reinvoking the decision procedure. In the example above, the clause $a = c \Rightarrow f(a) = f(c)$ might help prune a part of the search space where $a = c$ holds, even if the transitivity chain $a = b = c$ does not hold. By comparison, the coarse-grain explicated clause would not have been useful.

Figure 2(e) compares the performance of coarse-grain verses fine-grain explication. On several benchmarks, the coarse-grain strategy times out because it generates a very large number of clauses, where each clause is very specific and only refutes a small portion of the search space. Fine-grained explication terminates more often, but is sometimes slower when it does terminate.

We conjecture this slowdown is because fine-grain explication produces clauses containing atomic formulas (such as $a = c$ in the example above) that do not occur in the original query, which causes the SAT solver to assign truth values to these new atomic formulas. Thus, subsequent explication may be necessary to refute inconsistent assignments to the new atomic formulas.

To avoid this problem, we extended Verifun to hide the truth assignment to these new atomic formulas from the decision procedure. In particular, when the SAT solver returns a satisfying assignment, Verifun passes to the decision procedure only the truth assignments for the original atomic formulas, and not for the new atomic formulas. Figure 2(f) shows that this *hiding new proxies* strategy produces a significant performance improvement, without introducing the problems associated with the coarse-grain strategy.

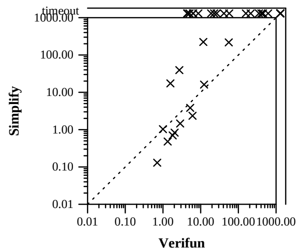
3.6 Comparison to Other Theorem Provers

We next compare the performance of Verifun with three comparable provers. Figure 3(a) compares Verifun with the Simplify theorem prover [12] on the UCLID benchmarks, and shows that Verifun scales much better to large problems. In several cases, Verifun is more than two orders of magnitude faster than Simplify, due to its use of explicated clauses and a fast SAT solver.

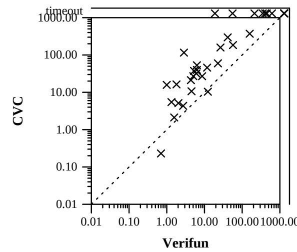
Figure 3(b) compares Verifun to the Cooperating Validity Checker (CVC) [5] on the UCLID benchmarks. The results show that Verifun performs better than CVC on these benchmarks, perhaps because CVC uses coarse-grain explication, which our experiments suggest is worse than Verifun's fine-grain explication.

Figure 3(c) and (d) compare Verifun with the Stanford Validity Checker (SVC) [**?**]. Figure 3(c) uses the UCLID benchmarks plus an additional benchmark provided by Velev that uses EUF and the theory of arrays. Figure 2 (d) uses the Math-SAT postoffice suite[3] of 41 timed automata verification problems [2, 1]. Interestingly, SVC performs better than Verifun on the UCLID benchmarks, but worse on the postoffice benchmarks, perhaps because these tools have been tuned for different problem domains. In addition, SVC is a relatively mature and stable prover written in C, whereas Verifun is a prototype written in Java, and still has significant opportunities for further optimization. For example, our decision procedures are currently non-backtracking and therefore repeat work across invocations. In most cases, a large percentage of Verifun's total running time

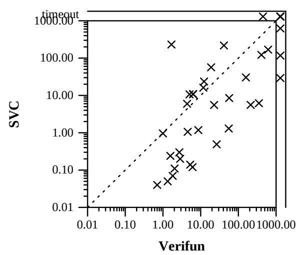
[3] We have not been able to run either Simplify or CVC on the postoffice benchmarks, due to incompatible formats.



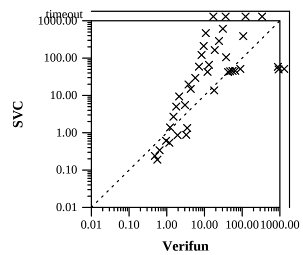
(a) Verifun vs Simplify on UCLID



(b) Verifun vs CVC on UCLID



(c) Verifun vs SVC on UCLID and Velev



(d) Verifun vs SVC on postoffice

Fig. 3. Scattergraph of running times (in seconds) comparing Verifun with (a) Simplify, (b) CVC, and (c) SVC on the UCLID benchmarks, and (d) comparing Verifun with SVC on the postoffice benchmarks. Verifun used the online SAT solver, lazy transitivity, fine-grain explication, and hiding of new proxy variables.

is spent inside these decision procedures. We expect that backtracking decision procedures would significantly improve Verifun's performance.

Math-SAT [2, 1] performs extremely well on the postoffice benchmarks, partly because these benchmarks include hints that Math-SAT uses to direct the search performed by the SAT solver. Verifun cannot currently exploit such hints, but its performance is superior to Math-SAT if these hints are not provided. In particular, on a 700MHz Pentium III with 1GB of RAM, Math-SAT is unable to solve any of the 5 largest problems within an hour, whereas Verifun can solve the largest one in 26 minutes.

4 Related Work

The idea of theorem proving by solving a sequence of incrementally growing SAT problems occurs as early as the 1960 Davis and Putnam paper [10]. However their algorithm simply enumerated all instantiations of universally quantified formulas in increasing order of some complexity measure, testing ever larger sets of instantiations for propositional consistency with the initial query. While their prover was, in principle, complete for first-order predicate calculus, it was unlikely to complete any proof requiring quantified variables to be instantiated with large terms before getting overwhelmed with numerous simpler but irrelevant instances.

The idea of adding support for proof explication to decision procedures has been explored by George Necula in the context of his work on "proof-carrying-code"(PCC) [15, 16]. However, in PCC, proof-generation is used with a different purpose, viz., to allow code receivers to verify the correctness of code with respect to a safety policy. Our concerns, on the other hand, are different: we are interested in proof-generation in order to produce a sufficient set of clauses to rule out satisfying assignments that are inconsistent with the underlying theory. In particular, the quality of the explicated proofs is not as crucial in the context of PCC.

More similar in nature to our work is the Cooperating Validity Checker (CVC) [5]. CVC also uses explicated clauses in order to control the boolean search. However, the CVC approach differs from Verifun's in some crucial ways:

- CVC invokes its decision procedures incrementally as the SAT solver assigns truth values to propositional variables.
- Explication in CVC is coarser-grained than in Verifun.
- CVC generates proofs as new facts are inferred. On the other hand, our prover merely records sufficient information in the data structures so that proofs can be generated if needed.

Recent work by de Moura and Rueß [11] is closely related to ours, in that they too propose an architecture in which the decision procedures are invoked *lazily*, after the SAT solver has produced a complete satisfying assignment. They note the impact of proof-explicating decision procedures in pruning the search space more quickly. They also contrast the lazy decision procedure invocation approach with the eager invocation approach of provers like CVC. Our work differs from theirs in that we have focused on studying the performance impact of various design choices within the space of lazy invocation of decision procedures.

Another system employing a form of lazy explication is Math-SAT [2]. This system is specialized to the theory of linear arithmetic, for which it incorporates not only a full decision procedure but three partial decision procedures. Each decision procedure is invoked only when all weaker ones have failed to refute a potential satisfying assignment.

Verifun produces propositional projections of theory-specific facts lazily, on the basis of its actual use of those facts in refuting proposed satisfying assignments. An alternative approach is to identify at the outset and project to the propositional domain all theory-specific facts that might possibly be needed for testing a particular query. This "eager" approach has been applied by Bryant, German, and Velev [6] to the domain of equality with uninterpreted function symbols and arrays, and extended by Bryant, Lahiri and Seshia [8] to incorporate counter arithmetic. Ofer Strichman [22] has investigated the eager preojection to SAT for Presburger and linear arithmetic. When employing the eager approach it is important not to explicate the exact theory-specific constraint on the atomic formulas in the query, but to identify a set of theory-specific facts guaranteed to be sufficient for deciding a query without being excessively large [9]. Where this has been possible, the eager approach has been impressively successful. For richer theories (in particular for problems involving quantification), it is unclear whether it will be possible to identify and project all necessary theory-specific facts at the outset without also including irrelevant facts that swamp the SAT solver.

5 Conclusion

Our experience suggests that lazy explication is a promising strategy to harness recent developments in SAT solving and proof-generating decision procedures. Our comparisions of Verifun and Simplify indicate that this approach is more efficient than the traditional Nelson-Oppen approach. Comparisions with other approaches like SVC, though promising (as shown in Figure 3(c)), are not as conclusive. This is partly because several obvious optimizations (such as backtracking theories) are not yet implemented in Verifun.

One advantage of our approach over that of CVC is that there is less dependence on the SAT solver, which makes it easier to replace that SAT solver with the current world champion. A potential advantage of lazy explication is that it is easier to extend to additional theories than the eager explication approaches of Bryant et al. and Strichman. In particular, we have recently extended our implementation to handle quantified formulas. By comparison, it is unclear how to extend eager explication to handle quantification.

Acknowledgments
We are grateful to Sanjit Seshia for providing us with the UCLID benchmarks in SVC and CVC formats, to Alessandro Cimatti for helping us understand the format of the Math-SAT postoffice problems, and to Rustan Leino for comments on an earlier version of this paper.

References

1. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Input files for Math-SAT case studies. http://www.dit.unitn.it/~rseba/Mathsat.html.
2. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In *Proceedings of the 18th Conference on Automated Deduction*, July 2002.
3. Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166, pages 187–201. Springer, 1996.
4. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, July 2002.
5. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In preparation, 2002.

6. Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Proceedings 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 85–98. Springer, July 1999.
7. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings. In *Proceedings of the First International Workshop on Constraints in Formal Verification*, September 2002.
8. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, July 2002.
9. Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. In *Proceedings 12th International Conference on Computer Aided Verification*, pages 85–98, July 2000.
10. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.
11. Leonardo de Moura and Harald Ruess. Lemmas on Demand for Satisfiability Solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002.
12. David Detlefs, C. Greg Nelson, and James B. Saxe. A theorem-prover for program checking. Technical report, Hewlett-Packard Systems Research Center, 2003. In preparation.
13. Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and Verification of Out-of-Order Microprocessors in UCLID. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer, November 2002.
14. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, June 2001.
15. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Mellon University, 1998. Also available as CMU Computer Science Technical Report CMU-CS-98-154.
16. George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 25–44, June 2000.
17. C. Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979. A revised version of this thesis was published as Xerox PARC Computer Science Laboratory Research Report CSL-81-10.
18. C. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, October 1979.
19. Greg Nelson and Derek C. Oppen. Fast Decision Algorithms based on congruence closure. *JACM*, 27(2), October 1979.
20. Robert E. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, January 1984.
21. João Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositionsal satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
22. Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In *Proceedings Formal Methods in Computer-Aided Design, 4th International Conference*, pages 160–170, 2002.

23. Robert E. Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal of Computing*, 13(3):566–579, August 1984.
24. Hantao Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, 1997.
25. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, November 2001.