A Proof-Producing Boolean Search Engine*

Clark Barrett[1] and Sergey Berezin[2]

[1] New York University, barrett@cs.nyu.edu
[2] Stanford University, berezin@stanford.edu

Abstract. We present a proof-producing search engine for solving the Boolean satisfiability problem. We show how the proof-producing infrastructure can be used to track the dependency information needed to implement important optimizations found in modern SAT solvers. We also describe how the same search engine can be extended to work with decision procedures for quantifier-free first-order logic. Initial results indicate that it is possible to extend a state-of-the-art SAT solver with proof production in a way that both preserves the algorithmic performance (e.g. the number of decisions to solve a problem) and does not incur unreasonable overhead for the proofs.

1 Introduction

Decision procedures for domain-specific first-order theories have become important tools for many verification applications. Two of the primary challenges in creating a practical implementation of such decision procedures are ensuring correctness and achieving adequate performance. The addition of proof production can help accomplish both of these goals.

Many arguments have been made for adding proof production to automated theorem provers. For example, proofs provide additional reliability and the ability to check a result independently using a trusted proof-checker. We advocate proof production for an additional reason: the proof infrastructure tracks dependencies among assumed and derived facts during the proof search. These dependencies capture exactly the information that is needed to to determine the cause of each conflict during the proof search, making it easy to generate *conflict clauses*. As described in Section 3, conflict clauses are an essential ingredient of efficient SAT algorithms.

Although other methods exist for generating conflict clauses when the input to the SAT solver is a Boolean formula, our algorithm can produce conflict clauses when extended to quantifier-free first order logic.

The paper is organized as follows. Following a survey of related work, Section 2 describes our proof system and a simple proof-producing SAT solver.

* This research was supported by GSRC contract DABT63-96-C-0097-P00005, by National Science Foundation CCR-0121403, and by a grant from Intel Corporation. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, Intel, or the Government, and no official endorsement should be inferred.

Then, in Section 3, we give an overview of the performance enhancements used in modern SAT solvers. Section 4 gives a detailed implementation of an efficient proof-producing SAT solver, and Section 5 discusses the extension to quantifier-free first-order logic. Section 6 concludes.

1.1 Related Work

In the past few years, there has been significant interest in combining decision procedures for first-order theories with SAT [1–3, 8, 16, 7]. The implementations described in these approaches treat the SAT solver more or less as a black box. In particular, the first-order problem must first be translated into a purely Boolean problem. Often, valuable structural information is lost during such a translation. In contrast, our approach integrates the SAT solver with the first-order decision procedures, allowing structural information to be preserved. In addition, hybrid systems which use available SAT solvers are unable to produce proofs. Our integrated solver does produce proofs.

Recently, there has been some work done on proof-producing SAT solvers [10, 17]. However, the "proof" produced by these solvers is really just a script which enables another, presumably trusted, solver to duplicate the steps taken by the original solver. Though this does increase confidence in the solution, it does not actually produce a proof object which can be checked by a theorem prover. In our approach, an actual proof object can be produced. This proof can then be checked by a small trusted theorem prover which does not need to include a SAT solver.

Previous work at Stanford on proof-producing decision procedures culminated with the proof-producing tool CVC [14, 15]. CVC includes two options for solving the Boolean part of the problem: a slow SAT solver that produces proofs and a fast SAT solver that does not produce proofs. The current research aims to combine these approaches, resulting in solver which is both fast and able to produce proofs.

2 A Simple Proof-Producing SAT solver

Consider the simple propositional logic described in Fig. 1. A propositional formula is built from the constant formulas $\top$ (always true) and $\bot$ (always false), propositional variables (i.e. variables that can either be assigned true or false), and Boolean operators ($\wedge$, $\vee$, $\neg$). A *literal* is a propositional variable or the negation of a propositional variable. We also define $\phi_1 \rightarrow \phi_2$ to be an abbreviation for $\neg\phi_1 \vee \phi_2$, and $\phi_1 \leftrightarrow \phi_2$ to be an abbreviation for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_1 \rightarrow \phi_2)$.

Given such a formula, the goal of SAT is to find an assignment of true or false to each variable such that the formula evaluates (under the obvious standard semantics) to true.

If no satisfying assignment exists, a formula is said to be *unsatisfiable*. Given an unsatisfiable propositional formula ϕ, a proof system should be able to produce a proof of $\neg\phi$. We now introduce a proof system for accomplishing this task which is based on natural deduction.

propositional formula ::= $\top$ | $\bot$ | *propositional variable*
| *propositional formula* $\wedge$ *propositional formula*
| *propositional formula* $\vee$ *propositional formula*
| $\neg$*propositional formula*

Fig. 1. Propositional logic

2.1 Proof System

A proof is a tree, each of whose nodes is labeled with a formula. The formulas at the leaves of the tree are called *assumptions*. Assumptions may be designated as *open* or *closed*.

A *sequent* is a pair $\Gamma \vdash \phi$, where Γ is a set of formulas and ϕ is a formula. Since we are often interested only in the assumptions and the conclusion (the formula labeling the root) of a proof, the sequent $\Gamma \vdash \phi$ is used to represent any proof whose open assumptions are among Γ and whose conclusion is ϕ.

A *proof rule* or *inference rule* is a function which takes one or more proofs (called premises) and returns a new proof whose root node has each of the input proofs as its successors. A proof rule specifies the formula which should label the new root node and may also change the designation of one or more assumptions from open to closed.

Proof rules depend only on the assumptions and conclusions of their premises and can thus be described using sequents. We denote a proof rule as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i's are sequents for the premises and C is a sequent representing the new proof tree. The proof rule takes any set of proofs which match the P_i's and returns a new proof whose root is labeled by the right-hand side of C. If an assumption appears in some P_i but not in C, then that assumption is closed in the proof tree constructed by the proof rule. If there are no premises, the rule is called an *axiom*. We will describe the proof rules of our system below.

A sequent $\Gamma \vdash \phi$ is *valid* if the conjunction of the assumptions in Γ implies ϕ. A proof rule is *sound* if the validity of all its premises implies the validity of the conclusion. The set of *valid* proofs are those which can be constructed using the proof rules.

Though we will not prove it here, it is straightforward to show that if all the proof rules are sound, then the sequent for a valid proof is indeed valid.

2.2 Proof Rules

To simplify the notation, we write Γ, α to denote $\Gamma \cup \{\alpha\}$ for the assumptions. This notation also implies that $\alpha \notin \Gamma$.

The most basic rule is the assumption axiom. This and other rules needed for a simple proof-producing SAT solver are shown below.

$$\frac{}{\phi \vdash \phi}\,\text{assume} \qquad \frac{\Gamma_1,\, \alpha \vdash \phi \quad \Gamma_2,\, \neg\alpha \vdash \phi}{\Gamma_1 \cup \Gamma_2 \vdash \phi}\,\text{caseSplit}$$

$$\frac{\Gamma \vdash \phi \leftrightarrow \top}{\Gamma \vdash \phi}\,\text{iffTrueElim} \qquad \frac{\Gamma \vdash \phi \leftrightarrow \bot}{\Gamma \vdash \neg\phi}\,\text{iffFalseElim}$$

$$\frac{\Gamma_0 \vdash \alpha_0 \quad \Gamma_1 \vdash \alpha_1 \quad \ldots \quad \Gamma_n \vdash \alpha_n}{\Gamma_0 \cup \Gamma_1, \ldots, \Gamma_n \vdash \phi \leftrightarrow \phi'}\,\text{simplify}$$

$$\frac{\Gamma_1 \vdash \phi \leftrightarrow \psi \quad \Gamma_2 \vdash \psi \leftrightarrow \theta}{\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta}\,\text{trans}$$

In the pseudo-code, these rules are used as function calls which take the premises and, possibly, some additional parameters, and return the conclusion sequent. For instance, `assume`(ϕ) takes ϕ as an argument and returns the sequent $\phi \vdash \phi$ as the result. Similarly, `caseSplit`(s_1, s_2, α) is an application of the caseSplit rule with premises s_1 and s_2. The parameter α identifies which assumptions to eliminate from s_1 and s_2. A call to `simplify`(Δ, ϕ) takes the set of premises $\Delta = \{\Gamma_i \vdash \alpha_i \mid i \in \{0 \ldots n\}\}$ as its first parameter, and the formula ϕ to be simplified as its second parameter. It returns a sequent for $\phi \leftrightarrow \phi'$ where ϕ' is obtained by replacing all instances of the literals in Δ by true (and their negations by false) and applying obvious Boolean simplifications to the result.

Note that the premises must be of a certain form in order for the rule to be sound. Our implementation includes an option which causes each of these rules to verify at run-time that its arguments are of the right form and generates an error if the check fails. This provides a very efficient "on-the-fly" internal soundness check in the tool which can often be used to detect soundness bugs without the need for an external proof checker.

2.3 Naïve SAT solver with proof production.

A simple SAT solver can be constructed using an algorithm which first picks a propositional variable α called a *splitter*, assigns it true or false, and then calls itself recursively until the formula evaluates to true or false.

A simple proof-producing SAT solver using the proof rules just described is shown in Fig. 2. The procedure `checkSAT` takes as input a formula ϕ and returns either a theorem of the form $\vdash \neg\phi$ or a theorem of the form $\Gamma \vdash \phi$, where Γ is a set of literals appearing in ϕ.

The `checkSAT` procedure calls `checkSAT`$_r$, a recursive procedure which takes as input a set of assumptions Δ (theorems of the form $\alpha \vdash \alpha$ where α is a literal), and a formula ϕ, and returns either a theorem of the form $\Gamma \vdash \phi \leftrightarrow \top$ or $\Gamma \vdash \phi \leftrightarrow \bot$.

Both procedures use a helper function called `getRHS`. This function takes a proof of $\phi \leftrightarrow \phi'$ as input (for some ϕ and ϕ') and returns the formula ϕ'. The `checkSAT`$_r$ procedure also makes use of the `findSplitter` function which takes a formula and returns a propositional variable appearing in the formula.

3 Efficient SAT algorithms

The algorithm in the previous section is essentially an implementation of the standard Davis-Putnam-Logemann-Loveland (DPLL) algorithm [5, 6]. Modern

```
checkSAT(φ) {
  s := checkSAT_r(∅, φ);
  φ' := getRHS(s);
  if (φ' = ⊤) return iffTrueElim(s);
  return iffFalseElim(s);
}

checkSAT_r(Δ, φ) {
  s_0 := simplify(Δ, φ);
  φ' := getRHS(s_0);
  if (φ' ∈ {⊤, ⊥}) return s_0;
  α := findSplitter(φ');
  s_1 := trans(s_0,checkSAT_r(Δ ∪ assume(α),φ'));
  if (getRHS(s_1)= ⊤) return s_1;
  s_2 := trans(s_0,checkSAT_r(Δ ∪ assume(¬α),φ'));
  if (getRHS(s_2)= ⊤) return s_2;
  return caseSplit(s_1, s_2, α);
}
```

Fig. 2. Naïve SAT solver.

SAT solvers like GRASP [12] and Chaff [13] are also based on this same fundamental algorithm, but include significant refinements and optimizations.

Efficient SAT algorithms are based on fast manipulation of *clauses*. A clause is a disjunction of one or more literals. Most SAT solvers assume that the formula to be checked is given in *Conjunctive Normal Form* (CNF), that is, as a conjunction of clauses.

Fig. 3 shows pseudo-code for an enhanced SAT solver (without proofs). It is similar to the algorithms in [12, 13], but is organized slightly differently. The `checkSAT` procedure takes as input a formula ϕ and returns either $\emptyset$, indicating that the formula is unsatisfiable, or a satisfying assignment. The satisfying assignment is represented as a set of literals θ with the property that if each literal in θ is `true`, then the formula ϕ is also true. The formula to be checked is first converted to CNF. We do not address how to do this here, but the conversion is straightforward and discussed in other papers, such as [3, 11]. The CNF clauses are stored in the global variable Φ, and then the recursive procedure $\texttt{checkSAT}_r$ is called.

$\texttt{checkSAT}_r$ takes as input a partial assignment (an assignment to some subset of the variables appearing in Φ) again represented as a set θ of literals. It returns an assignment θ' which extends θ if there exists such an assignment satisfying Φ. Otherwise, it returns $\emptyset$. The first step in $\texttt{checkSAT}_r$ is *Boolean Constraint Propagation* (BCP). BCP uses the structure of the clauses in Φ to deduce additional assignments that must hold in order to obtain a satisfying assignment for Φ, and is described in more detail below. It returns a new partial assignment. BCP may discover that some variable v is required to take on two different values by

different clauses in Φ. In this case, the returned partial assignment has both v and $\neg v$ and is said to be *inconsistent*.

If BCP discovers a conflict, the conflict is analyzed to produce a new *conflict clause*. This clause identifies a subset of θ which is responsible for the conflict and is (permanently) added to Φ, ensuring that the conflict will not reoccur. We discuss this more below. When a conflict is discovered, $\texttt{checkSAT}_r$ returns $\emptyset$, indicating that the given partial assignment θ cannot be extended to a satisfying assignment.

If BCP does not discover a conflict, then the search for a satisfying assignment can continue. $\texttt{checkSAT}_r$ calls `findSplitter` which searches Φ for a variable not already assigned by θ. If such a variable is found, it is returned in α. Otherwise, $\emptyset$ is returned, in which case all variables are assigned, so the current assignment is a satisfying assignment.

If a splitter is found, it is added to the current assignment. Then $\texttt{checkSAT}_r$ is called recursively with the current assignment. The result is either a satisfying assignment which is then returned, or $\emptyset$, indicating that the current assignment does not have a satisfying assignment. In the latter case, execution continues at the top of the loop where BCP is called again. Because of the conflict clause just added by the most recent call to $\texttt{checkSAT}_r$, we are guaranteed that this call to BCP will detect an inconsistency.

```
checkSAT(φ) {
  Φ := convertToCNF(φ);
  return checkSAT_r(∅);
}

checkSAT_r(θ) {
  while (true) {
    θ := BCP(Φ,θ);
    if (inconsistent(θ)) {
      conflictClause := analyzeConflict(Φ,θ);
      Φ := Φ ∪ {conflictClause};
      return ∅;
    }
    α := findSplitter(Φ,θ);
    if (α = ∅) return θ;
    θ := θ ∪ α;
    θ' := checkSAT_r(θ);
    if (θ' ≠ ∅) return θ';
  }
}
```

Fig. 3. Enhanced SAT solver.

We now discuss the importance and implementation of several components of enhanced SAT algorithm shown in Fig. 3: fast BCP, conflict clauses, so called

non-chronological backtracking, or *intelligent backjumping*, and finally, good *decision heuristics* for picking splitters.

3.1 Fast Boolean Constraint Propagation (BCP).

Intuitively, the purpose of BCP is to derive all the assignments to variables that logically follow from the current assignments without having to split on any variable. The complexity of BCP is polynomial, and can be implemented very efficiently. Since the worst-case complexity of SAT is exponential in the number of variables, reducing the number of variables by BCP is one of the most important aspects of the algorithm.

When the formula is represented as a set of clauses, BCP amounts to finding *unit clauses*, those that have exactly one literal unassigned, and assigning the corresponding variable to make the literal true. This assignment may result in more unit clauses, and the process continues until either a contradiction is detected (one of the clauses gets all of its literals assigned to false), or no more unit clauses remain.

GRASP and Chaff implement unit clause detection by having two *watched literals* in each clause. As long as both literals stay unassigned, the clause is guaranteed not to become a unit clause. If either of the watched literals gets assigned, the clause is searched for another unassigned literal, and if one is found, it becomes the new watched literal. Each variable maintains two lists of clauses in which the variable appears as a positive or negative literal respectively. So, for each variable assignment, only those clauses are processed in which a watched literal becomes false.

Learning the Conflict Clauses. When a SAT solver detects a conflict, it is often the case that only a small subset of the variable assignments is responsible for the contradiction, and therefore, the same assignment will appear in many branches of the decision tree. A SAT solver takes advantage of this fact by learning such *conflict assignments*, so that when they show up again, it immediately backtracks without having to derive the contradiction again.

Typically, a conflict assignment contains exactly one variable assigned at the last level of recursion (often the most recent splitter). Other variables may be either previous splitters themselves, or assignments derived from those splitters by BCP.

A conflict assignment can be expressed as a formula $c \equiv \ell_1 \wedge \cdots \wedge \ell_n$, where ℓ_i's are literals. Since we know that the original problem Φ is unsatisfiable when c is true, we can state that Φ is satisfiable only if $\overline{c} \equiv \overline{\ell}_1 \vee \cdots \vee \overline{\ell}_n$ is true.

Notice that $\overline{c}$ has the syntax of a clause, so it can simply be added to Φ. When the same assignment is made again, the conflict will be immediately detected by BCP due to the newly added clause $\overline{c}$, which we call a *conflict clause*.

Intelligent Backjumping. Each variable assignment has an associated *decision level*, which is the corresponding depth in the decision tree where the assignment is made.

When a conflict occurs, the SAT solver returns to the previous decision level, effectively undoing all the assignments made at the current decision level. Notice that if the conflict clause includes the most recent splitter, then the negation of the splitter will be derived by BCP from the conflict clause. Therefore, there is no need to consider the opposite assignment of the splitter explicitly, it will happen automatically.

In some cases, the SAT solver can backtrack beyond the previous decision level. If the conflict clause does not include any variables from the previous decision level, then the negation of the splitter is still implied in the decision level before that. Thus, we can backtrack to the most recent decision level in which a variable from the conflict clause is assigned.

Decision Heuristics. It is well-known that the order in which the splitters are chosen can dramatically affect the performance of the SAT algorithm. Modern SAT solvers have developed sophisticated splitter heuristics that work amazingly well in practice. Examples of these heuristics are detailed in [9, 13].

4 An Efficient Proof-Producing SAT solver

We now give a relatively detailed description of a proof-producing SAT solver with the enhancements described above. Before describing the algorithm, we describe the data structures used in the algorithm. Some additional proof rules used by the algorithm are shown in Fig. 4.

4.1 Basic Data Structures

Expressions All formulas and terms are represented as DAGs with maximal sharing of subexpressions. That is, if two expressions e_1 and e_2 are syntactically the same, then they are physically stored in the same location. In particular, checking expressions for (syntactic) equality is a constant time operation (comparison of pointers).

Theorems Theorems hold a sequent $\Gamma \vdash \phi$, where Γ is a set of formula expressions and ϕ is a formula expression. Besides the sequent, theorems may also carry the actual proof tree corresponding to the sequent (as a special *proof term* expression, not discussed here). The proofs are only generated when the tool is requested to produce an externally checkable proof. Otherwise, the sequents are sufficient for the functionality of the algorithm. The metavariable s (for "sequent") is used to represent theorems. If s is a theorem whose sequent is $\Gamma \vdash \phi$, then `getAssumptions(`s`)` returns the set of formulas in Γ, and `getConc(`s`)` returns the formula ϕ. To simplify the algorithm, we also allow a special "NULL" theorem, without a sequent or a proof, denoted by $\emptyset$.

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge_E\text{-left} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge_E\text{-right} \qquad \frac{\Gamma,\ \alpha \vdash \bot}{\Gamma \vdash \neg\alpha}\ \neg_I$$

$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \leftrightarrow \psi}{\Gamma_1 \cup \Gamma_2 \vdash \psi}\ \text{iffMP} \qquad \frac{(\Gamma_i \vdash \neg\alpha_i)_{i \neq j} \quad \Gamma \vdash \bigvee_i(\alpha_i)}{\Gamma \cup \bigcup_{i \neq j}(\Gamma_i) \vdash \alpha_j}\ \text{unitProp}$$

$$\frac{\Gamma,\ \alpha_1,\ \ldots,\ \alpha_n \vdash \bot}{\Gamma \vdash \neg\alpha_1 \vee \cdots \vee \neg\alpha_n}\ \text{conflictClause}$$

Fig. 4. Additional Proof Rules.

4.2 Program State

The state of the SAT solver consists of the following components. Some components are *backtracked*, meaning that when `pop()` is called, they revert to the value they had when the corresponding call to `push()` was made. Backtracked components are marked with the † sign.

†**Assumptions:** Δ is a set of theorems called *assumptions*, each of whose conclusions is a literal. These literals (and the corresponding variables) are said to be *assigned*. All other variables and literals are *unassigned*. Δ corresponds to decisions and derived literals.

†**Literals:** `literals` is a queue containing theorems whose conclusions are literals (which are waiting to be added as assumptions). The function call `pushBack(literals,`s`)` inserts s into the queue and the corresponding call to `popFront(literals)` returns and removes the first item in the queue. Initially, `literals` is empty.

Clauses: Φ is a set of theorems, each of whose conclusions is a clause. Φ contains clauses that are part of the original formula to be checked for satisfiability as well as derived conflict clauses. Given a clause expression c, the function `setupWatchPointers(`c`)` selects two of its literals to be *watched literals* and associates with c two *watch pointers* indexed by $i \in \{0, 1\}$ which point to these literals. Given a clause c and an index $i \in \{0, 1\}$, the function `updateWatchPointer(`c`,`i`)` is called when the i^{th} watch pointer in c is assigned. It searches for an unassigned literal in c and updates the i^{th} watch pointer to point to the new literal.

Non-clauses: Θ is a set of theorems, each of whose conclusions is neither a literal nor a clause. It contains parts of the original formula which are in non-clausal form.

Watch lists: Associated with each literal is a list of the clauses where the literal is being watched. For a literal l, the function `getWatchPointers(`l`)` returns a set of pairs (c, i) where c is a clause in which l is being watched and $i \in \{0, 1\}$ is the index corresponding to the watched literal l in c. Initially these lists are empty.

4.3 The Algorithm

The code for an efficient proof-producing SAT solver is shown in Fig. 5 and Fig. 6.

checkSAT. The main function in the SAT checking algorithm is `checkSAT` which, as in the algorithm of Fig. 2, takes as input a formula ϕ and returns either a theorem of the form $\vdash \neg\phi$ or a theorem of the form $\Gamma \vdash \phi$, where Γ is a set of literals appearing in ϕ.

`checkSAT` begins by initializing the theorem sets to be empty. It then takes the formula to be checked and passes it as a trivial theorem (created using the `assume` rule) to `addFact`, which partitions ϕ into literals, clauses, and non-clauses. Next, `checkSAT` calls $\texttt{checkSAT}_r$ which does the main recursive search. If $\texttt{checkSAT}_r$ returns $\emptyset$, then this means that ϕ is satisfiable under the set of assumptions contained in Δ, so the `simplify` rule (followed by `iffTrueElim`) can be used to get a theorem of the appropriate form. If $\texttt{checkSAT}_r$ does not return $\emptyset$, then it must return a theorem whose conclusion is $\bot$. Since ϕ is assumed in the first call to `addFact`, any derivation of $\bot$ returned by $\texttt{checkSAT}_r$ will also contain ϕ as an assumption, so we can use the `notIntro` rule to derive a theorem whose conclusion is $\neg\phi$.

addFact. `addFact` takes as input a theorem s and figures out where to put it. It first assigns ϕ to be the conclusion of the theorem. If ϕ is a conjunction, then s is split using the conjunction elimination rules and `addFact` calls itself recursively. If ϕ is a literal, s gets pushed onto the literal queue. If ϕ is a clause, s gets added to the set Φ of clauses. Otherwise, s is added to the set Θ of non-clauses.

$\texttt{checkSAT}_r$. $\texttt{checkSAT}_r$ is similar to the procedure of the same name in Fig. 3. It starts by calling BCP which figures out additional assignments which are implied by the current set of assignments. If BCP does not return $\emptyset$, it means that an inconsistency was detected and the return value is a proof of $\bot$ from the current set of assumptions. In this case, the current context is inconsistent, so there is no need to search further along the current branch. The theorem proving $\bot$ is returned.

If BCP does not detect an inconsistency, then we continue the search for a satisfying assignment by finding a splitter. The function `findSplitter` can look for a splitter in either the set of clauses Φ or the set of non-clauses Θ. In our implementation, findSplitter first uses a depth-first search to find splitters from Θ. When all the literals in Θ have been assigned, it then uses a Chaff-like heuristic to pick splitters from Φ. There are certainly more sophisticated heuristics that could be used and investigating these is part of our current research.

If no splitter can be found, $\texttt{checkSAT}_r$ returns $\emptyset$ to indicate that the current set of assumptions constitutes a satisfying assignment. Otherwise, the backtracked state (Δ and `literals`) is saved by calling `push()`. Then, the splitter α is added to the set of assigned literals by calling `addFact` and $\texttt{checkSAT}_r$ is

```
checkSAT(φ) {
  Θ := Φ := Δ := ∅;
  addFact(assume(φ));
  s := checkSAT_r();
  if (s = ∅) return iffTrueElim(simplify(Δ,φ));
  return notIntro(s,φ);
}

addFact(s) {
  φ := getConc(s);
  if (isConjunction(φ)) {
    addFact(andElimLeft(s));
    addFact(andElimRight(s));
  }
  else if (isLiteral(φ)) pushBack(literals, s);
  else if (isClause(φ)) {
    Φ := Φ ∪ {s};
    setupWatchPointers(getConc(s));
  }
  else Θ := Θ ∪ {s};
}

checkSAT_r() {
  while (true) {
    s := BCP();
    if (s ≠ ∅) return s;
    α := findSplitter();
    if (α = ∅) return ∅;
    push();
    addFact(assume(α));
    s := checkSAT_r();
    if (s = ∅) return s;
    pop();
    if (backJump(s)) return s;
    addFact(notIntro(s,getLastAssumption(s)));
  }
}
```

Fig. 5. Enhanced SAT solver with proofs.

called recursively. If the result is $\emptyset$ indicating that a satisfying assignment was found, then checkSAT$_r$ returns without calling pop() to preserve the satisfying assignment.

The other possibility is that the recursive call results in a proof of $\perp$ from some subset of the current assumptions. In this case, we first check for the possibility of intelligent backjumping. The function backJump(s) returns true if none of the assumptions in s (excluding the most recently assigned assumption) were assigned in the current recursion level (since the most recent active call to push). In this case, the theorem s can be used to derive a literal in the previous recursion level, and is thus returned.

If backJump(s) returns false, then the notIntro rule can be used to derive the negation of the most recent assumption in s from the others, and checkSAT$_r$ starts over at the top of the loop.

```
BCP() {
  inconsistent := false;
  while (¬inconsistent) {
    s := popFront(literals);
    Δ := Δ ∪ {s};
    l := getConc(s);
    w := getWatchPointers(l);
    foreach (c,i)∈ w {
      updateWatchPointers(c,i);
      if (isUnsat(c)) {
        inconsistent := true;
        unsatClause := c;
      }
      else if (isUnit(c)) addFact(unitProp(c));
    }
  }
  if (¬inconsistent) {
    foreach s_θ ∈ Θ {
      s := iffMP(s_θ,simplify(Δ,getConc(s_θ)));
      if (getConc(s)= ⊥) return s;
    }
    return ∅;
  }
  conflict = processImplGraph(unsatClause);
  addFact(conflictClause(conflict));
  return conflict;
}
```

Fig. 6. Enhanced BCP with proofs.

BCP. The last part of the SAT solver is the BCP code. BCP begins by processing the queue of literals. Each theorem in the queue is added to the set of

assumptions. Then the watch pointers for all clauses which are watching the literal are updated. If any of these clauses become unsatisfiable (all its literals are assigned false), the `inconsistent` flag is set and the inconsistent clause is stored in `unsatClause`. If any of these clauses becomes a unit clause, then the `unitProp` rule is used to derive the remaining unassigned literal.

Once all of the literals have been processed, if no inconsistency has been detected, then the non-clauses are processed. This is done by checking if any of them simplify to $\bot$. If so, a theorem deriving $\bot$ is returned. Otherwise, $\emptyset$ is returned indicating no inconsistency.

If an inconsistent clause is detected, then the so-called "implication graph" can be used to generate a conflict clause. The implication graph is stored implicitly by the theorems in Φ. We start by looking up the theorem for the inconsistent clause in Φ and collecting all of its assumptions. We then look in Δ for the theorems justifying these assumptions. This process is continued until we have a set of assumptions, at most one of which was assigned at the current recursion level (since the last active call to `push`). The call to `processImplGraph` returns a theorem which derives $\bot$ from these assumptions.

Finally, the `conflictClause` rule is used to turn the conflict theorem into a conflict clause, which is then added to the set of clauses (implementing conflict clause learning), and the conflict is returned.

5 Extension to Quantifier-Free First-Order Logic

Although the algorithm just presented only handles Boolean logic, it can easily be extended to cooperate with quantifier-free first-order decision procedures. In first-order logic, the atomic formulas are no longer required to be propositional variables, but can also be predicates applied to terms over object constants and variables. Therefore, we redefine the notion of a *literal* to be an atomic formula (a predicate or a propositional variable) or its negation.

Since the atomic formulas are no longer independent from each other, purely Boolean constraint propagation is not sufficient. To overcome this problem, we extend BCP to *first-order constraint propagation* (FOCP) as follows. Every time a new literal ϕ is added to Δ, it is also submitted to the first-order decision procedure. Similarly, whenever a decision procedure derives a new literal l, `addFact(`l`)` is called. In fact, if a decision procedure derives non-literals, these can also be handled just by calling `addFact`. However, in this case the additional clauses or non-clauses will have to be backtracked.

Additionally, after each literal is submitted to the first-order decision procedure, the first-order decision procedure may report an inconsistency. If the first-order decision procedure is instrumented with proof production, then it can return a theorem deriving $\bot$ from some subset of the current assumptions. This theorem can be then be used to generate a conflict clause just as with purely Boolean conflicts.

The ability to produce proofs from the first-order decision procedure is thus essential for enabling the efficiency gains that come from learning conflict clauses.

6 Conclusion

We have implemented a prototype of our algorithm in an emerging tool called *CVC Lite*. CVC Lite is a smaller, more light-weight implementation of CVC, designed for experimentation and rapid prototyping.

For our experiments, we adopted the philosophy of using families of benchmarks as advocated in [4]. We compared the average number of decisions required to solve the Boolean benchmarks in the *PC* families (each containing 33 benchmarks) of the *hole4* and *hole5* benchmarks [4]. We compared zchaff, CVC (in its standard proof-producing mode), and CVC Lite. Though CVC has a fast SAT-based mode which is essentially equivalent to zchaff, this mode cannot produce proofs.

benchmark	zchaff	CVC Lite	CVC
hole4	30	31	800
hole5	149	163	25832

Fig. 7. Comparison of the number of decisions.

Clearly, CVC Lite does not quite capture all of the intelligence in zchaff, but it is close. When proofs are enabled, CVC does a poor job on Boolean examples. CVC Lite demonstrates that it is possible to implement a proof-producing Boolean solver whose algorithmic performance is close to that of a highly tuned non-proof-producing solver.

Because our prototype implementation has not been tuned for run-time performance, we did not compare run-times. However based on [17] as well as our previous experience with CVC, proof-production only adds a small constant overhead to the underlying algorithm (the reason CVC is so much slower in proof-production mode is that it is not using the efficient SAT algorithms, not because proof-production adds a lot of overhead). Also, the optimizations enabled by proof production typically lead to an exponential speedup, so a small amount of overhead is reasonable.

As already mentioned, one important area of ongoing research is investigating decision heuristics for combined clausal and non-clausal formulas. An obvious question is: why not just convert everything to CNF? Although this is possible for purely Boolean formulas, it often degrades performance of some non-Boolean testcases (this problem is also mentioned in [3]). We are working on decision heuristics which perform well on both Boolean and non-Boolean testcases.

References

1. Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-Based Procedures for Temporal Reasoning. In S. Biundo and M. Fox, editors, *Proceedings of*

the 5th European Conference on Planning (Durham, UK), volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.

2. Gilles Audemard, Piergiorgio Bertoli, and Alessandro Cimatti. A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Reiner Hähnle, editor, *Proceedings of the 18th International Conference on Automated Deduction (Copenhagen, Denmark)*, Lecture Notes in Artificial Intelligence. Springer, 2002.
3. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, 2002. Copenhagen, Denmark.
4. Franc Brglez, Xiao Yu Li, and Matthias F. Stallman. The role of a skeptic agent in testing and benchmarking of SAT algorithms. In *Fifth Int. Symp. on the Theory and Applications of Satisfiability Testing*, Cincinnati, Ohio, USA, May 2002.
5. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.
6. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
7. 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.
8. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James Saxe. Theorem Proving using Lazy Proof Explication. In *15th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
9. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.
10. E. Goldberg and Y. Novikov. Verification of Proofs of Unsatisfiability for CNF formulas. In *Proceedings of Design, Automation and Test in Europe (Munich, Germany)*, 2003.
11. Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):4–15, January 1992.
12. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
13. M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.
14. A. Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002. In preparation: check http://verify.stanford.edu/~stump/ for a draft.
15. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
16. Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
17. L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proceedings of Design, Automation and Test in Europe (Munich, Germany)*, 2003.