Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 269 (2011) 3–17
www.elsevier.com/locate/entcs

The Mechanical Verification of a DPLL-Based Satisfiability Solver [1]

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar}@csl.sri.com
URL: http://www.csl.sri.com/~shankar

Marc Vaucher

École Polytechnique
marc.vaucher@polytechnique.org

Abstract

Recent years have witnessed dramatic improvements in the capabilities of propositional satisfiability procedures or SAT solvers. The speedups are the result of numerous optimizations including conflict-directed backjumping. We use the Prototype Verification System (PVS) to verify a satisfiability procedure based on the Davis–Putnam–Logemann–Loveland (DPLL) scheme that features these optimizations. This exercise is a step toward the verification of an efficient implementation of the satisfiability procedure. Our verification of a SAT solver is part of a larger program of research to provide a secure foundation for inference using a verified reference kernel that contains a verified SAT solver. Our verification exploits predicate subtypes and dependent types in PVS to capture the specification and the key invariants.

Keywords: SAT solver, backlumping, predicate subtype, dependent type, PVS

1 Introduction

Inference procedures have a number of important applications in programming as well as in other disciplines. In recent years, there have been rapid advances in the power and efficiency of inference procedures, particularly with solvers for propositional satisfiability (SAT) and satisfiability modulo theories (SMT). These procedures are used in assertional verification, bounded model checking, unbounded model checking, and planning. In addition to making these procedures efficient and

[1] This work was completed in the Summer of 2008 when the second author was a student at École Polytechnique. He now works at Autorité de Contrôl Prudentiel in Paris. The work was supported by NSF Grants CSR-EHCS(CPS)-0834810 and CNS-0917375 and by NASA Cooperative Agreement NNX08AY53A.

doi:10.1016/j.entcs.2011.03.002

powerful, we are interested in a deeper theoretical understanding of how and why these procedures work. This understanding can lead to more efficient inference procedures. It can also be used to achieve a significant level of trust in the correctness of their computations without restricting their efficiency by using an offline verified *reference* implementation to check the results of an online untrusted procedure [17]. As a step toward the construction of such a verified reference, we describe the verification of an abstract SAT solver based on the Davis–Putnam-Logemann–Loveland method [9,6] using the Prototype Verification System (PVS) [13].

Our work falls within a long tradition of work in the mechanized verification of metatheoretic procedures. Many of the early results were carried out by Boyer and Moore using their celebrated theorem prover [1]. Examples of these include the McCarthy-Painter proof of correctness of a compiler for arithmetic expressions, the correctness of a satisfiability solver for conditional expressions, and the Turing-completeness of pure Lisp [2]. Shankar [14] describes proofs of the soundness, completeness, and decidability (by means of a satisfiability solver) of a propositional logic, as well as proofs of the Church–Rosser theorem for untyped lambda calculus [15] and Gödel's incompleteness theorem [16], that have been mechanized using the Boyer–Moore prover. Since then, a variety of decision procedures have been mechanically verified in a wide range of systems. Some of these have even been used reflectively to automate proofs. Propositional satisfiability procedures have become extremely powerful in the last decade starting with systems such as SATO [20], GRASP [12], and Chaff [11]. As a result of these improvements, SAT solvers have found a wide and growing range of applications in hardware and software verification, constraint solving, and test generation. The increased efficiency of modern SAT solvers comes from fast Boolean constraint propagation using clever data structures, heuristics for variable selection, and the the use of conflict-based learning and backjumping. Although our treatment of DPLL-based SAT solving is somewhat abstract, the only non-executable operation in our formalization is the one for variable selection, primarily because this involves heuristic rather than logical considerations. Our formalization covers clause learning and backjumping, but does not include the two-literal watching method for Boolean constraint propagation. Our verification is a first step toward the verification of a fully executable and efficient SAT solver [11]. Such a SAT solver forms the key component in a verified reference kernel architecture that can be used to certify the results of untrusted, highly engineered proof tools. A trusted SAT solver can be used for verifying the results of other untrusted verifiers such as binary decision diagrams (BDDs) [3], model checkers [4], and SMT solvers [7].

The verification of a SAT solver poses significant challenges for mechanization. Our verification, though interactive, exploits several features of the PVS language and inference mechanisms. In particular, we used PVS to interactively explore the details of the formalization so that our proofs at this point involve a fair amount of manual guidance. We plan to use our development to explore strategies for greater mechanization, particularly through the greater use of solvers for propositional satisfiability and satisfiability modulo theories.

2 The DPLL Satisfiability Procedure

Given a set of propositional variables P, a propositional formula has the grammar

$$\Phi := P \mid \neg\Phi \mid \Phi_1 \vee \Phi_2$$

An *assignment* M for a formula ϕ maps propositional variables to truth values $\{\bot, \top\}$. Let $\sqsupset$ and $\veebar$ correspond to the truth table interpretation of the $\neg$ and $\vee$ connectives. The truth value $M[\![\phi]\!]$ of a formula ϕ with respect to an assignment M can be computed from the truth table interpretation of the connectives. If $M[\![\phi]\!] = \top$, then M is a *model* of ϕ.

Propositional formulas can be transformed into an equivalent conjunctive normal form (CNF). A literal is a propositional variable p or its negation $\neg p$. If k is a literal p (respectively, $\neg p$), then its complement $\overline{k}$ is $\neg p$ (respectively, p). A clause κ is a disjunction $k_1 \vee \ldots \vee k_n$ where $n \geq 0$ and each k_i for $1 \leq i \leq n$ is a literal. We assume, without loss of generality, that a clause does not contain duplicate literals and never contains both a literal and its complement. A formula in conjunctive normal form is a set of clauses. A CNF formula $\bigwedge_{i=1}^{n} \kappa_i$ is satisfiable if it has a model. A clause κ is the *consequence* of a set of clauses K if any model M of K is also a model of κ. For example, the resolution rule of inference where the clause $\kappa \vee \kappa'$ is derived from, and is a consequence of, the clauses $k \vee \kappa$ and $\overline{k} \vee \kappa'$. In the informal presentation, we assume that the order of appearance of literals in a clause is irrelevant and can be freely permuted.

The Davis–Putnam–Logemann–Loveland procedure [9,6] (DPLL) inference system searches for a satisfying assignment for a set of n clauses K over m propositional variables [7]. It does this by building a *partial assignment* M in levels l and a set of implied *conflict lemmas* C. A partial assignment M up to level l has the form $M_0; M_1; \ldots; M_l$. The partial assignment M_0 is a set of pairs $k_i[\gamma_i]$ with $\gamma_i \in K \cup C$ with *source clause* γ_i from $K \cup C$. For $0 < i < l$, each M_i has the form $d_i : k_1[\gamma_1], \ldots, k_n[\gamma_n]$ with decision literal d_i and *implied literals* k_i and their corresponding source clause γ_i. When k occurs in M, let $M_{<k}$ be the prefix of the partial assignment preceding the occurrence k in M. We can view M as a partial assignment since $M(p) = \top$ if p occurs in M, $M(p) = \bot$ if $\neg p$ occurs in M, and $M(p)$ is undefined, otherwise.

The DPLL procedure searches for a satisfying assignment by applying four subprocedures: *propagate*, *analyze*, *backjump*, and *select*, to the partial assignment and the clause set $K \cup C$. The *propagate* procedure checks if there is a clause γ in $K \cup C$ where at most one literal k is not falsified by M. If this literal k is unassigned, then it is an *implied literal* with respect to M and $k[\gamma]$ is added to M at the current level. If the literal k is valid in M, then no action is taken. If the literal k is falsified by M, then γ is an initial conflict clause. The *propagate* procedure is applied repeatedly until it finds a conflict or there are no more implied literals.

When *propagate* finds a conflict, if $l = 0$, then the DPLL procedure terminates by asserting the unsatisfiability of the input K. Otherwise, when $l > 0$, it applies the *analyze* procedure to construct a conflict lemma θ. It does this by repeatedly

replacing γ with the result of resolving γ with γ' where $k[\gamma']$ occurs in M and $\overline{k}$ is a literal at the current level (but not the unique such literal) occurring in γ. The new value of γ is also falsified in M since γ' is of the form $k \vee \gamma''$, where γ'' is falsified in M. When γ contains a unique literal $\overline{k}$ at the current level, then we let the conflict lemma θ be γ.

The conflict lemma θ constructed by *analyze* is of the form $\overline{k} \vee \theta'$. Let l' be the lowest level with $l' < l$ such that $M_0; \ldots; M_{l'}$ falsifies θ' but leaves k unassigned. The *backjump* procedure replaces M with $M_0; \ldots; M_{l'}, \overline{k}[\theta]$ while adding θ to C. When M is not a total assignment and the *propagate* procedure does not detect a conflict or generate any new implied literals, then the *select* procedure increments the level l by one and adds an unassigned literal k to M as the decision literal at this new level. Otherwise, if M is a total assignment and no conflict is detected, the input K must be satisfiable.

An example of the procedure is shown in Figure 1. The input clause set K is $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$. Since there are no unit (single literal) clauses, there are no implied literals at level 0. We therefore select an unassigned literal, in this case s as the decision literal at level 1. Again, there are no implied literals at level 1, and we select an unassigned literal r as the decision literal at level 2. Now, we can add the implied literals $\neg q$ from the input clause $\neg q \vee \neg r$ and p from the input clause $p \vee q$. At this point, propagation identifies a conflict where the partial assignment M falsifies the input clause $\neg p \vee q$. The conflict is analyzed by replacing $\neg p$ with q to get the unit clause q. Since the maximal level of the empty clause is 0, backjumping yields a partial assignment q at level 0 while adding the unit clause q to the conflict clause set C. Propagation then yields the implied literals p from the input clause $p \vee \neg q$ and r from the input clause $\neg p \vee r$, which leads to the falsification of the input clause $\neg q \vee \neg r$. Since this conflict occurs at level 0, we report unsatisfiability.

The correctness of the procedure can be established by observing that each step preserves the satisfiability of $M_0 \cup K \cup C$, so that when a conflict is detected at level 0, then $M_0 \cup K \cup C$ must be unsatisfiable, and hence K must be unsatisfiable. If M is a total assignment such that no clause in K generates a conflict with M, then K must be satisfiable. The procedure terminates because in each step, the value of $\Sigma_{i=0}^{m} |M_i| * (m+1)^{(m-i)}$ increases toward a bound $(m+1)^{(m+1)}$.

3 Formalizing DPLL in PVS

PVS is a specification and verification framework based on higher-order logic and interactive proof. The PVS specification language enriches simply typed higher-order logic with predicate subtypes, dependent types, abstract datatypes and co-datatypes, inductive definitions, type judgements, parametric theories, and theory interpretations. Proofs in PVS are constructed interactively by combining a variety of powerful automated tools such as Boolean simplification, ground decision procedures, rewriting, symbolic model checking, heuristic quantifier instantiation, and induction. New proof strategies can be defined in terms of old ones using a strategy

step	l	M	K	C	γ
select s	1	$; s$	K	$\emptyset$	_
select r	2	$; s; r$	K	$\emptyset$	_
propagate	2	$; s; r, \neg q[\neg q \vee \neg r]$	K	$\emptyset$	_
propagate	2	$; s; r, \neg q, p[p \vee q]$	K	$\emptyset$	_
conflict	2	$; s; r, \neg q, p$	K	$\emptyset$	$\neg p \vee q$
analyze	0	$\emptyset$	K	q	_
backjump	0	$q[q]$	K	q	_
propagate	0	$q, p[p \vee \neg q]$	K	q	_
propagate	0	$q, p, r[\neg p \vee r]$	K	q	_
conflict	0	q, p, r	K	q	$\neg q \vee \neg r$

Fig. 1. The DPLL procedure with input $\{p \vee q, \neg p \vee q, p \vee \neg q, s \vee \neg p \vee q, \neg s \vee p \vee \neg q, \neg p \vee r, \neg q \vee \neg r\}$

language. We provide examples of specific features of PVS as these are used in the formalization.

The PVS formalization consists of a formalization of the resolution rule of inference, a representation for partial assignments, the definitions of the basic operations used in the formalization, and the three key procedures: `propagate`, `analyze`, and `dpll`. We make heavy use of predicate subtypes and dependent types to capture the key invariants. Typechecking with these types generates proof obligations called type-correctness conditions (TCCs) that must be discharged using the PVS theorem prover. We only present the key definitions. The full formalization is available from the corresponding author.

3.1 Resolution

The resolution inference rule is introduced and justified in a PVS theory (a collection of declarations of types, constants, and formulas) that is parameterized by a bound `m` on the index of variables that can appear in a clause.

```
resolution [m: posnat ] : THEORY
  BEGIN
   :
  END resolution
```

In the `resolution` theory, a propositional atom is modeled as an integer in the subrange $[1, m]$, and a literal is a predicate subtype of the integers that contains an atom `a` and its negation `-a`.

```
atom: TYPE = subrange(1, m)
a, b: VAR atom
literal: TYPE = {i: subrange(-m, m) | i /= 0}
k, l: VAR literal
```

A clause, given by the predicate subtype `bclause`, is a list of literals that is not a tautology.

```
nontautology?(ll: list[literal]): bool =
  (FORALL k: NOT (member[literal](k, ll) AND
                  member[literal](-k, ll)))

bclause: TYPE = (nontautology?)
```

A model is a (total) assignment which is the function type from `atom` to the Booleans `bool`.

```
model: TYPE = [atom -> bool]

M, N: VAR model
```

A clause `ll` is a positive clause (`pclause?`) for literal `k` if it contains k as a `member` of the list, a negative clause (`nclause?`) if it contains `-k`, and is a neutral clause if it contains neither `k` nor `-k`.

```
pclause?(k)(ll): bool = member[literal](k, ll);
nclause?(k)(ll): bool = member[literal](-k, ll);
oclause?(k)(ll): bool = NOT (pclause?(k)(ll)
                             OR nclause?(k)(ll));
```

A clause `ll` is compatible with clause `kk` with respect to literal `k` if the only literal occurring positively in `kk` and negatively in `ll` is `k` itself.

```
compatible?(k, kk)(ll): bool =
  (FORALL l: member[literal](l, kk) AND
             member[literal](-l, ll)
     IMPLIES l = k)
```

The `resolve` operation applies the resolution inference over literal `k` to the clause `kk` (in which `k` occurs positively) and the clause `ll` (which is compatible with `kk` and in which `k` occurs negatively). It returns a clause that is neutral with respect to k and is obtained by deleting all occurrences of `k` and `-k` from `kk` and `ll`, respectively, and appending the resulting clauses.

```
resolve(k)(kk : (pclause?(k)),
           (ll | nclause?(k)(ll) AND compatible?(k, kk)(ll)))
 : (oclause?(k)) =
 append(delete(k, kk), delete(-k, ll))
```

Typechecking the `resolution` theory generates several proof obligations including the key one requiring that the result returned by `resolve` be a non-tautology that is neutral with respect to `k`. The main property that we need of `resolve` is that its result is a consequence of the clauses `kk` and `ll`.

```
resolve_sat: LEMMA
  (FORALL (M: model,  kk: (pclause?(k)),
          (ll | nclause?(k)(ll) AND compatible?(k, kk)(ll))):
    satisfies(M, kk) AND satisfies(M, ll)
  IMPLIES satisfies(M, resolve(k)(kk, ll)))
```

3.2 Partial Assignments

Modeling partial assignments is the trickiest part of the formalization. A partial assignment must consist of a sequence of levels l with $l \geq 0$. Each level $l > 0$ contains a decision literal and a sequence of implied literals. Each implied literal has a source clause that must be falsified in the preceding assignment. We capture partial assignments with four separate constructs. The first construct is a model M as above which is a total mapping atoms to Booleans. The second construct is a counter `index` that maintains the number of assigned atoms and a partially injective map I from the atoms to the subrange $[0, index]$. If $I(a) = 0$ for some variable, then that variable is unassigned. The map I is partially injective in the sense that if $0 \neq I(a) = I(b) \neq 0$, then $a = b$. We actually need I to be partially bijective so that for every i up to `index`, the cardinality of the set $\{a | 1 \leq I(a) \leq i\}$ is exactly i. The third construct is a stack of atoms ls corresponding to the decision literals. This *decision stack* must be ordered by I so that if atom a is below atom b in the stack, then $0 < I(a) < I(b)$. The fourth construct maps non-decision (implied) atoms to the source clause.

Given a partial assignment with a model `M`, the truth value of a literal can be found by the operation `lookup` which looks up the truth value of the atom and compares it to the sign of the literal.

```
lookup(M, l): bool = (M(abs(l)) IFF l > 0)
```

A clause `lk` is falsified by a partial assignment containing `index`, `I`, and `M` if each literal is assigned false in the model. Since a clause is just a list of literals, the `falsifies` operation is defined recursively over the list `lk`.

```
falsifies(index, (I: imap(index)), M, lk): RECURSIVE bool =
 (CASES lk OF
   null : TRUE,
   cons(k, lm): (LET a = abs(k)
                 IN (I(a) > 0 AND
                      (M(a) IFF k<0) AND
                           falsifies(index, I, M, lm)))
  ENDCASES)
 MEASURE length(lk)
```

The index of a literal k is `I(abs(k))`. A recursion over the list of literals in a clause is used to compute the maximal index `maxindex(index, I, lk)` of a clause.

Another similar recursion computes the maximal literal `maxliteral(index, I, lk)` in a clause `lk`. The maximal level `maxlevel(index, I, ls, i)` of an index `i`, the highest index of a decision literal in `ls` that is below `i`, is computed by means of a recursion on the decision stack.

A clause table `K` is a dependent record where the field entry `K‘numclause` is the number of clauses and the field entry `K‘clauses` is a mapping from clause indices in the subrange $[0, \texttt{numclause}]$ to nonempty clauses.

```
clause_table: TYPE =
  [# numclause: upfrom(n),
     clauses: [below(numclause) -> ne_bclause]
  #]
```

Now we can describe the last remaining component of the partial assignment which is the source table `src` which maps atoms to clause indices corresponding to the clauses (in the clause table) in which the atom appears as an implied literal. The latter condition is again ensured by means of dependent typing and predicate subtyping. In summary, predicate subtypes and dependent typing in PVS are used to capture the critical datatype invariants associated with the partial assignment.

3.3 Inference Subprocedures

The `backjump(index, I, M, ls, ito)` operation takes a partial assignment $\langle$`index, I, M, ls`$\rangle$ and a target index `ito` and returns a restriction of partial assignment up to and including the level containing the index `ito`. The operation returns a record containing the fields `newindex` which is the index of the new assignment between `ito` and `index`, the new index map `newI` whose (dependent) type is a single set containing the expected value, and the new decision stack `newls` which must be a suffix of the original stack `ls` where all the decision literals with index greater than `ito` have been deleted.

The *propagate* operation `propagate(index, I, ls, M, K, src)` returns a new inference state `index'`, `I'`, `M'`, and `src'`, and returns an index to a initial conflict clause in `K` when a conflict has been detected. The definition repeatedly scans the clauses to identify a new implied literal or a conflict clause. Each time a new implied literal is added to `M`, the clauses are rescanned. Implied literals are found by scanning the literals in a clause to determine if all but one of the literals are falsified.

Recall that the *analyze* operation backchains on a conflict clause to produce a conflict lemma with a unique level-maximal literal. The operation `analyze(index, I, ls, M, K, src, kk)` takes a partial assignment consisting of `index`, `I`, `M`, and `src`, and a conflict clause `kk` that is falsified by the partial assignment. It returns either an empty conflict clause, a conflict lemma whose maximal literal is assigned at level 0, or a conflict lemma with a unique level-maximal literal. As expected, `analyze` is defined recursively to repeatedly resolve the clause `kk` with the source clause for the maximal literal in `kk` until one of the conditions enumerated in the previous sentence holds. It is easy to check that the clauses involved in the resolution are compatible and the resulting resolvent clause is falsified by the partial assignment. The termination of `analyze` follows from the fact that `maxlit` is the maximal literal in `kk` and its negation is the maximal literal in the source clause `K`‘`clauses(src(abs(maxlit)))`, and hence the index of the maximal literal in `kk` decreases with each recursive call.

3.4 DPLL Search

Before we define the main DPLL search procedure, we introduce two other operations that are used in this definition. The parametric `lift` datatype from the PVS prelude library essentially adds, by way of a disjoint union, a bottom element to the given type `T`. It is used to represent the result of the search.

```
lift[T: TYPE]: DATATYPE
 BEGIN
  bottom: bottom?
  up(down: T): up?
 END lift
```

The operation `select` picks an atom that is unassigned according to `I`. It uses Hilbert's `epsilon`, axiomatized in the PVS prelude library so that for any predicate `p` over a nonempty type `T`, if there is any `a` such that `p(a)` holds, then `p(epsilon(p))` holds

```
select(index, (I: imap(index))): atom = epsilon(LAMBDA a: I(a) = 0)
```

The main DPLL search procedure is also defined recursively. In addition to the partial assignment, it takes the input clause set `K_in`, which is included for specification purposes, and the current clause set `K` which extends `K_in` with the conflict clauses learned through analysis during the search. The signature of this procedure `dpllr` is given below.

```
dpllr(index,
    (I: bimap(index)),
    (ls : lstack(index, I)),
    M,
    (K_in : clause_table),
    (K : (extends_ct(K_in))),
    src: source(index, I, M, ls, K)
    ):  RECURSIVE lift[model] =  ...
```

The `dpllr` function returns a lifted model. The value `bottom` indicates that the input is unsatisfiable, and the value `up(M')` indicates that `M'` is a satisfying model for the input. This interpretation of the result of `dpllr` is justified in the next section.

The first step in the search is the application of propagation to obtain a record with `newindex`, `newI`, `newmodel`, and `newsrc`.

```
(LET rr = propagate(index, I, ls, M, K, src),
     newindex = rr`newindex,
     newI = rr`newI,
     newmodel = rr`newmodel,
     newsrc = rr`newsrc
 IN ...
```

If propagation turns up a conflict and the decision stack is empty, then `dpllr` reports unsatisfiability. Otherwise, if the decision stack is not empty, `dpllr` applies `analyze` to the conflict clause. If `analyze` returns an empty clause or a clause at level 0, then `dpllr` reports unsatisfiability.

```
    (IF rr‘conflict < K‘numclause
     THEN (IF null?(ls)
           THEN bottom
           ELSE
            (LET kk = analyze(newindex, newI, ls, newmodel, K,
                              newsrc, K‘clauses(rr‘conflict))
             IN (IF null?(kk)
                 THEN bottom
                 ELSE
                  LET newlit = maxliteral(newindex, newI, kk),
                      nextindex = maxindex(newindex, newI,
                                           delete(newlit, kk)),
                      maxlevel = maxlevel(newindex, newI, ls,
                                          nextindex)
                  IN
                  IF maxlevel(newindex, newI, ls,
                              newI(abs(newlit))) = 0
                  THEN bottom
                  ELSE
                    ...
                  ENDIF
                 ENDIF))
           ENDIF)
```

Otherwise, the conflict clause returned by `analyze` has a unique level-maximal literal. Backjumping is applied to scale back the partial assignment, the conflict clause is added to `K`, and the maximal literal is added to this scaled back assignment with the conflict clause as its source.

```
       LET result = backjump(newindex, newI, newmodel, ls,
                             maxlevel)
       IN dpllr(result‘newindex + 1,
                result‘newI WITH [(abs(newlit)) :=
                                  result‘newindex + 1],
                result‘newls,
                newmodel WITH [(abs(newlit)) := (newlit > 0)],
                K_in,
                K WITH [‘numclause := K‘numclause + 1,
                        ‘clauses(K‘numclause) := kk],
                newsrc WITH [(abs(newlit)) := K‘numclause])
```

In the remaining case when `propagate` does not return a conflict, we select an unassigned literal as the decision literal for the next level of the search.

```
    ELSE (LET M1 = rr‘newmodel,
              I1 = rr‘newI,
              a  = select(newindex, I1)
          IN IF I1(a)>0
             THEN up(M1)
             ELSE
              LET I2 = I1 WITH [(a) := newindex + 1]
               IN dpllr(newindex + 1, I2, cons(a, ls),
                        M1, K_in, K, newsrc)
             ENDIF)
    ENDIF))
MEASURE (expt(m+1,m+1) - dweight(index, I, ls))
```

The termination measure is that `dweight(index, I, ls)` increases to the bound `expt(m+1, m+1)`, where `expt` is the exponentiation operation for natural number exponents. The operation `dweight(index, I, ls)` is recursively defined to compute the expression $\Sigma_{i=0}^{n}|M_i| * (m+1)^{(m-i)}$.

The termination of the `dpllr` procedure is the major part of the correctness argument. As the informal proof showed, the soundness and completeness are not as challenging as termination. We describe the proof obligations in the correctness proof in the next section.

4 Some Highlights of the Proof

We now briefly summarize the more important proof obligations in the verification of `dpllr`. There are a number of modest lemmas about partial assignments, and the operations `maxindex` and `maxliteral` the we omit.

The definition of `backjump` generates some nontrivial proof obligations. The main claim about `backjump` is that given a partial assignment, the `backjump` operation deletes all assignments for indices that are equal or exceed the least index of an atom in the decision stack that is strictly larger than the target index given to `backjump`. This lemma requires an inductive proof with about 50 interactions.

In the definition of `analyze`, there is a resolution step where a clause κ that is falsified in a given partial assignment is resolved on its maximal literal with the source clause of the assignment falsifying the maximal literal. There are two proof obligations associated with this step. The first one is to show that the resolution step is compatible, and the second one is to show that the resulting clause has a smaller maximal index than the original clause. Both proofs are nontrivial.

In the definition of the `propagate` operation, there are two interesting proof obligations. One to ensure that the source clause corresponding to the newly added implied literal satisfies the type constraints of the source clause table, and the second to check that the type of the result of the recursive call matches the expected type in the signature.

A recursive judgement in PVS is a typing judgement on a recursive definition that assumes that the expected type given by the judgement holds of the recursive calls. For the definition of `analyze`, a recursive judgement is used to show that the clause returned is a consequence of the clause set `K`. Another lemma demonstrates that the conflict lemma returned by `analyze` is falsified by the current assignment.

The weight of a partial assignment given by `dweight` is used to construct a termination measure for the `dpllr` procedure. There are some important lemmas associated with this definition. First, a recursive judgement `dweight_bounded` is used to show that the value of `dweight` is bounded by $(m+1)^{(m+1)}$. This claim generates 4 TCCs of which one has a proof that involves nearly 50 interactions.

```
dweight_bounded: RECURSIVE JUDGEMENT
  dweight(index, (I: imap(index)), (ls: lstack(index, I)))
  HAS_TYPE upto(expt(m+1,m+1) - expt(m+1, m - length(ls)))
```

Then a bit of complicated arithmetic is used to bound the decrease in the `dweight` for a backjumping step. This is a very delicate proof requiring about 100 interactions and only a few of the subproofs are just cut-and-paste copies of proofs on other branches. Another lemma shows that the loss in `dweight` from backjumping can be more than compensated by the gain in weight from adding the implied literal to the partial assignment. This proof requires about 30 interactions.

Finally, we show that whenever the partial assignment is extended, the dweight increases. This proof requires about 100 interactions, but several of these are just copies of proofs from other branches.

```
dweight_extends: LEMMA
  FORALL (index : below(m)), (I: bimap(index)),
         (newindex: below(m)), (newI: bimap(newindex)),
        (ls: lstack(index, I)):
    index <= newindex AND
    bextendsI?(index, I, newindex)(newI)
   IMPLIES dweight(index, I, ls)
           <= dweight(newindex, newI, ls)
```

Now, we are ready to tackle the termination and correctness of the main `dpllr` procedure. The definition of `dpllr` generates 25 TCCs of which 15 are proved trivially. The remaining 10 TCCs require proofs ranging from 10 to 90 interactions. Only two of these are termination TCCs corresponding to the two recursive cases of the definition. The first of the termination TCCs requires 40 interactions, and the second one about 50 interactions.

Having proved the type correctness and termination of the definition of `dpllr`, we are left with the task of proving that it is sound and complete. A key lemma asserts that if the decision stack is empty, then all of the literals in the partial assignment are implied by the clauses in the clause table `K`. This proof is nontrivial and requires nearly 60 interactions.

```
source_empty_lstack: LEMMA
 FORALL index, (I:imap(index)),  (K: clause_table),
        (src: source(index, I, M, null, K)):
    satisfies(N, K)
  IMPLIES (FORALL a: I(a) > 0
           IMPLIES N(a) = M(a))
```

Finally, we express the soundness and completeness of `dpllr` as a recursive judgement that asserts that if `dpllr` returns a model, then this model satisfies the input clauses. Conversely, if it does not return a model, then the input clauses are not satisfiable. Note that the latter condition holds regardless of the given partial assignment `M`, and this is because the invariant conditions on the partial assignment ensure that any unretractable assignment in `M` is a consequence of `K` and hence `K_in`. The recursive judgement claim generates 4 TCCs which have proof ranging from 20 to 50 steps. Only one of these proofs, corresponding to the case when `analyze` returns a conflict clause at level 0, is genuinely challenging.

```
dpll_conservation: RECURSIVE JUDGEMENT
  dpllr(index,
        (I: bimap(index)),
        (ls : lstack(index, I)),
        M,
        (K_in : clause_table),
        (K : (extends_ct(K_in))),
        src: source(index, I, M, ls, K))
   HAS_TYPE {LL : lift[model] |
    (IF up?(LL) THEN satisfies(down(LL), K_in)
                ELSE (FORALL N: NOT satisfies(N, K_in))
     ENDIF)}
```

Altogether, the proof involves 331 lemmas or proof obligations. All the proofs can be rechecked in about 275 CPU seconds on a 2.16 GHz Intel Core 2 Duo MacBook with 1GB of RAM.

5 Discussion

The proof of correctness of the DPLL search procedure defined as `dpllr` was mostly developed while the second author, then an undergraduate at École Polytechnique, was on a two-month internship at SRI International. The first author is a co-developer of PVS whereas the second author was familiar with functional programming but had no prior experience with interactive proof checking . Both authors worked jointly on the definition of the `dpllr` intermittently over the first two weeks and carried out a few initial proofs. The second author, working on his own, completed a proof of termination for the procedure. This proof revealed a large number of moderately serious bugs in the definitions including missing invariants, incorrect invariants, and missing cases in the initial body of definitions. The first author subsequently redid the termination proof to use the measure shown here over about four days, and then completed the soundness and completeness arguments over a couple of days.

As should be clear from our presentation, we exploited a number of features of the PVS language such as functions, records, predicate subtypes, dependent types, recursive datatypes, parametric theories, and typing judgements (including recursive judgements). Predicate subtyping and dependent typing in PVS provide a high level of safety in the sense that, viewed as a higher-order functional language, a well-typed PVS program is safe. Apart from the `select` operation, our formalization is in an executable fragment of PVS that we have recently expanded to include the dynamic arrays used here for the clause table. In addition to the usual type safety, PVS execution will not encounter nontermination, array bounds violations, division by zero, missing cases, or any other execution error provided we assume unlimited resources.

Dependent typing in PVS is used for a lot more than type safety. It is not uncommon for users to exploit dependent typing and proof obligation generation to capture large parts of the specification and structure proofs. For example, a number of interesting termination examples in PVS exploit higher-order dependent typing [18]. PVS has a lot of features for managing type constraints. The type-checker does not generate TCCs that are trivial, and checks for subsumed TCCs. It also automatically applies type judgements to compute multiple types for an expression. The proof checker has strategies for automatically discharging TCCs and the decision procedures exploit any known type constraints on expressions. Even so, there are still some significant challenges with using dependent typing as heavily as we have in this proof. Many proofs require the explicit introduction of type constraints for specific expressions. The generated TCCs are often too specific which limits their reuse as lemmas. Quite often the same proof obligation can be generated many times within a single proof. Of course, these problems can easily be overcome with a little bit of planning. However, since we use PVS to explore the formal space and to discover the proof, we are not actually checking a pre-planned proof. Type judgements, particularly recursive judgements are very powerful since they generate proof obligations that correspond to just the relevant cases of a definition. In summary, the PVS type system can be quite effective for capturing the

datatype invariants but we need to identify better ways of using type constraints and more efficient mechanisms for reasoning with them.

The bulk of our proof is devoted to proving the termination of `dpllr`. The challenge here is in showing that the partial assignment, which grows and shrinks, must eventually stop growing. The challenging parts of the proof are not in the orderings themselves, but in showing that the ordering is bounded and that the datatype invariants are satisfied.

Recently and independently, Maric [10] has used Isabelle/HOL to verify a a SAT solver that is quite similar to the one described here. His verification required about a man-year of effort and a 30,000-line proof script. His formalization also covers efficient unit propagation using the two-literal watching technique. He uses a list representation for both clauses and the clause database, whereas we have used an array representation for the clause database.

Could we have achieved a better level of automation in the proof? Clearly, the distance between the informal argument and the formal proof is quite large. A lot of this has to do with the missing details in the informal proof, but we did find areas such as quantifier instantiation where the automation was inadequate. The SMT solver Yices [5] is integrated with PVS, and the language of Yices is quite similar to that of PVS. However, we tend to not use it in exploratory proof construction. This is because we have to carefully identify the definitions and lemmas that are needed to complete the proof. Once there is a completed proof that identifies the theorems and lemmas, it is possible to make use of Yices to achieve greater automation to construct proofs that are more robust to changes. It is not so easy to exploit the ideas from automated SMT solvers within interactive proof without sacrificing readability. However, it is possible to better exploit the availability of *precise explanations* that excludes the formulas that are irrelevant to a proof. For improved quantifier instantation, first-order or higher-order proof search might be useful in some instances, but heuristics based on e-graph matching are likely to be more helpful [8]. We also plan to investigate a declarative proof style centered around an SMT solver. We find the goal-directed proof style of PVS quite conducive to exploration, but a declarative proof style would yield proofs that are more robust and comprehensible.

6 Conclusions

We have described a mechanical verification using PVS of a DPLL-based search procedure for propositional satisfiability. This procedure poses interesting challenges for verification and can used as a challenge for various automated and semi-automated tools. It is also a good example for experimenting with different styles of formalization. Our verification is a initial step toward the construction of a trusted/verified reference kernel for checking the results of other untrusted verifiers.

The formalization and proof of the DPLL SAT solver exploit several features of PVS. For example, typing judgements and recursive typing judgements play an important role in decomposing proofs into manageable proof obligations that cor-

respond to the individual cases of a larger proof. While our proof represents a preliminary and exploratary attempt, we plan to investigate avenues for better formalization and greater automation, and to examine the verification of satisfiability procedures, including SMT solvers, that are more efficient and expressive. While it is encouraging that a relatively untrained user could build complex proofs with only a modest amount of effort, we believe that there is plenty of room for improvement.

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[2] R. S. Boyer and J. S. Moore. A mechanical proof of the Turing completeness of pure Lisp. *Contemporary Mathematics*, 29:133–167, 1984.

[3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[5] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. `http://yices.csl.sri.com/`, 2006.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962. Reprinted in Siekmann and Wrightson [19], pages 267–270, 1983.

[7] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, 2007.

[8] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Hewlett-Packard Systems Research Center, 2003.

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[10] Filip Maric. Formalization and implementation of modern SAT solvers. *J. Autom. Reasoning*, 43(1):81–119, 2009.

[11] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[12] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[13] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[14] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[15] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988.

[16] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.

[17] Natarajan Shankar. Trust and automation in verification tools. In Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *Lecture Notes in Computer Science*, pages 4–17. Springer-Verlag, October 2008.

[18] Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 37–52, Toulouse, France, September 1999. Springer-Verlag.

[19] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag, 1983.

[20] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997.