

# Formal Verification of a Modern SAT Solver

Filip Marić<sup>a</sup>

<sup>a</sup>*Faculty of Mathematics, Belgrade*

---

## Abstract

We present a formalization and a formal total correctness proof of a MiniSAT-like SAT solver within the system Isabelle/HOL. The solver is based on the DPLL procedure and employs most state-of-the-art SAT solving techniques, including the conflict-guided backjumping, clause learning, and the two-watch unit propagation scheme. A shallow embedding into HOL is used and the solver is expressed as a set of recursive HOL functions. Based on this specification, the Isabelle's built-in code generator can be used to generate executable code in several supported functional languages (Haskell, SML, and OCaml). The SAT solver implemented in this way is, to our knowledge, the first fully formally and mechanically verified modern SAT solver.

*Key words:* formal program verification, SAT problem, DPLL procedure, Isabelle

---

## 1. Introduction

Propositional satisfiability problem (SAT) is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It is a canonical NP-complete problem [Coo71] and it holds a central position in the field of computational complexity. SAT problem is also important in many practical applications such as electronic design automation, software and hardware verification, artificial intelligence, and operations research. Thanks to recent advances in propositional solving technology, SAT solvers are becoming the tool for attacking more and more practical problems. Most modern SAT solvers are based on the *Davis-Putnam-Logemann-Loveland (DPLL) procedure* [DP60, DLL62] and its modifications.

Since SAT solver are used in applications that are very sensitive (e.g., software and hardware verification), their misbehavior could be both financially expensive and dangerous from the aspect of security. Clearly, having a trusted SAT solving system is vital. This can be achieved in two different ways.

1. One approach is to extend an *online* SAT solver with the possibility of generating models of satisfiable formulas and proofs of unsatisfiability for unsatisfiable formulas. The generated models and proofs are then checked *offline* by an independent trusted checker [ZM03, Gel07].
2. Another approach is to apply software verification techniques and verify the implementation of the SAT solver itself, so that it becomes trusted [Mar08a, SV08].

The first approach has successfully been used in recent years. It is relatively easy to implement, but it has some drawbacks. Generating object-level proofs introduces about 10% overhead to the solver’s running time and proof checking can also take significant amount of time [Gel07]. More importantly, since proofs are very large objects, they can consume up to several gigabytes of storage space. Since proof checkers have to be trusted, they must be very simple programs so that they could be „verified” only by manually inspecting their source code [Gel07]. On the other hand, in order to handle large proof objects, checkers must use specialized functionality of the underlying operating system, which reduces the level of their confidence.<sup>1</sup>

In this work we take the second, harder, approach and formally verify a full implementation of a SAT solver. There are several reasons for doing this.

1. We believe that this verification effort could help in better theoretical understanding of how and why modern SAT solver procedures work.
2. Although the overheads of generating unsatisfiability proofs during solving are not unmanageable, they can still be avoided if the SAT solver itself is trusted.
3. Verified SAT solvers can serve as the trusted kernel checkers for verifying results of other untrusted verifiers such as BDDs, model checkers, and SMT solvers [SV08]. Also, verification of some SAT solver modules (e.g., Boolean constraint propagation) can serve as a basis for creating a verified, yet efficient, proof checker for SAT.
4. Finally, we want to demonstrate that, thanks to the recent advances in both automated and semi-automated software verification technology, the time has finally come when it is possible to have a non-trivial software fully verified. We hope that this work contributes to the *Verification Grand Challenge* [VSTTE].

In order to prove correctness of a SAT solver implementation, it needs to be formalized in some meta-theory so its properties can be analyzed by using an appropriate mathematical apparatus. In order to achieve the desired, highest level of trust, formalization in a classical “pen-and-paper” fashion is not satisfactory and a mechanized and machine-checkable formalization is required. All formalizations presented here were made within the system Isabelle/HOL [NPW02]. A shallow embedding is used, i.e., the SAT solver is expressed as a set of recursive functions in HOL. Original proof documents are available in [Mar08b].

**Overview of the paper.** The rest of the paper is structured as follows. In §2 we give some background information about the DPLL procedure and its modifications. We also give some background on program verification. In §3 we introduce basic notions of the system Isabelle and formulate an underlying theory for our formalization. The central section of the paper is §4 in which we present the specification of the SAT solver and introduce correctness conditions along the way. In §5 we outline the correctness proof of our implementation and in §6 we discuss some aspects of the proof management. In §7 we list some of

---

<sup>1</sup>For example, proof checker used in SAT competitions uses Linux’s `mmap` functionality [Gel07].

```

function dpll ( $F$  : Formula) : (SAT, UNSAT)
begin
  if  $F$  is empty then
    return SAT
  else if there is an empty clause in  $F$  then
    return UNSAT
  else if there is a pure literal  $l$  in  $F$  then
    return dpll( $F[l \rightarrow \top]$ )
  else there is a unit clause  $[l]$  in  $F$  then
    return dpll( $F[l \rightarrow \top]$ )
  else begin
    select a literal  $l$  occurring in  $F$ 
    if dpll( $F[l \rightarrow \top]$ ) = SAT then
      return SAT
    else
      return dpll( $F[l \rightarrow \perp]$ )
  end
end
end

```

Figure 1: DPLL algorithm — recursive definition

the related work, in §8 we list some possible directions for further work, and in §9 we draw final conclusions.

## 2. Background

***DPLL Procedure and its Modifications.*** Most modern SAT solvers are based on the *Davis-Putnam-Logemann-Loveland (DPLL) procedure*. Its original recursive version is shown in Figure 1, where  $F$  denotes a set of propositional clauses tested for satisfiability and  $F[l \rightarrow \top]$  denotes the formula obtained from  $F$  by substituting the literal  $l$  with  $\top$ , its opposite literal  $\bar{l}$  with  $\perp$ , and simplifying afterwards. A literal is *pure* if it occurs in the formula but its opposite literal does not occur. A clause is *unit* if it contains only one literal. This recursive implementation is practically unusable for larger formulae and therefore it is not used in modern SAT solvers, nor in this paper.

Starting with the work on the GRASP and SATO systems [MSS99, Zha97] and continuing with Chaff, BerkMin and MiniSAT [MMZ<sup>+</sup>01, GN02, ES04], the spectacular improvements in the performance of DPLL-based SAT solvers achieved in the last years are due to (i) several conceptual enhancements of the original DPLL procedure, aimed at reducing the amount of explored search space, such as *backjumping* (a form of non-chronological backtracking), *conflict-driven lemma learning*, and *restarts*, and (ii) better implementation techniques, such as the *two-watched literals* scheme for unit propagation. These advances make it possible to decide satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

***Rule-based descriptions of the DPLL procedure.*** During the last few years two transition rule systems which model modern DPLL-based SAT solvers and related SMT solvers have been published [NOT06, KG07]. These descriptions define the top-level architecture of solvers as mathematical object that can be grasped as a whole and fruitfully reasoned about. Both systems are accompanied by pen-and-paper correctness and termination proofs. Although they succinctly and accurately capture all major aspects of the solvers' global operation, they

$$\begin{array}{l}
\text{Decide:} \\
\frac{l \in F \quad l, \bar{l} \notin M}{M := M \bar{l}^d} \\
\text{UnitPropag:} \\
\frac{l \vee l_1 \vee \dots \vee l_k \in F \quad \bar{l}_1, \dots, \bar{l}_k \in M \quad l, \bar{l} \notin M}{M := M l} \\
\text{Conflict:} \\
\frac{C = \text{no\_cflct} \quad \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad l_1, \dots, l_k \in M}{C := \{l_1, \dots, l_k\}} \\
\text{Explain:} \\
\frac{l \in C \quad l \vee \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad l_1, \dots, l_k \prec l}{C := C \cup \{l_1, \dots, l_k\} \setminus \{l\}} \\
\text{Learn:} \\
\frac{C = \{l_1, \dots, l_k\} \quad \bar{l}_1 \vee \dots \vee \bar{l}_k \notin F}{F := F \cup \{\bar{l}_1 \vee \dots \vee \bar{l}_k\}} \\
\text{Backjump:} \\
\frac{C = \{l, l_1, \dots, l_k\} \quad \bar{l} \vee \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad \text{level } l > m \geq \text{level } l_i}{C := \text{no\_cflct} \quad M := M^{[m]} \bar{l}} \\
\text{Forget:} \\
\frac{C = \text{no\_cflct} \quad c \in F \quad F \setminus c \models c}{F := F \setminus c} \\
\text{Restart:} \\
\frac{C = \text{no\_cflct}}{M := M^{[0]}}
\end{array}$$

Figure 2: Rules of DPLL as given in [KG07]

are still high level and far from the actual implementations. Both systems model the solver behavior as transitions between states that represent the values of global variables of the solver. These include the set of clauses  $F$  and the corresponding assertion trail  $M$ . Transitions between states are performed only by using precisely defined transition rules. The solving process is finished when no transition rule applies and final state is reached.

The system presented in [NOT06] is very coarse. It can capture many different strategies seen in the state-of-the art SAT solvers, but this comes at a price. Several important aspects still have to be specified in order to build the implementation based on the given set of rules.

The system presented in [KG07] gives a more detailed description of some parts of the solving process (particularly the conflict analysis phase) than the previous one. Since this system is used as a basis of the implementation given in this paper, we list its transition rules in Figure 2. Together with the formula  $F$  and the trail  $M$ , the state of the solver is characterized by the conflict analysis set  $C$  which is either the set of literals or the distinguished symbol *no\_cflct*. The input to the system is an arbitrary set of clauses  $F_0$ , modeled as initial state in which  $F = F_0$ ,  $M = []$ , and  $C = \text{no\_cflct}$ . The rules have guarded assignment form: above the line is the condition that enables the application of the rule, below the line is the update to the state variables.

**Formal program verification.** Formal program verification is the process of proving that a computer program meets its specification which formally describes the expected program behavior. Early results date back to 1950's and pioneers in this field were A. Turing, J. von Neumann and J. McCarthy. In the late 1960's R. Floyd introduced equational reasoning on flowcharts for proving

program correctness and T. Hoare introduced axiomatic semantics for programming constructs. Following the lessons from major software failures in recent years, an increasing amount of effort is being invested in this field.

To achieve the highest level of trust, mechanically checkable formal proofs of correctness are required. Many fundamental algorithms and properties of data structures have been formalized and verified in this way. Also, lot of work has been devoted to formalization of programming language semantics, compilers, communication protocols, security protocols, etc. Many of early results in mechanical program verification were carried out by Boyer and Moore using their theorem prover. Theorem provers that are most commonly used for program verification nowadays are Isabelle, HOL, Coq, PVS, Nuprl, etc. A large collection of formalized theories (of both pure mathematics and computer science) mechanically checked by the theorem prover Isabelle is available in *Archive of formal proofs* (<http://afp.sourceforge.net>).

**Formal program verification by shallow embedding into HOL.** Shallow embedding into higher-order logic is a technique that is widely used for verification, despite its well-known limitations [BKH<sup>+</sup>08]. This success is due in part to the simplicity of the approach: a formal model of the operational or denotational semantics of the language is not required and many technical difficulties (e.g., the representation of binders) are avoided altogether. Furthermore, the proof methods used are just standard induction principles and equational reasoning, and no specialized program logic (e.g., Hoare logic) is necessary. The specifications may be turned into executable code directly by means of code generation [Haf08]. The main drawback of this approach is that all programs must be expressed as purely functional. As the notion of side-effect is alien to the world of HOL functions, programs with imperative updates of references or arrays cannot be expressed directly which heavily effects the efficiency of the generated code. Still, approaches to overcome these difficulties have been proposed recently [BKH<sup>+</sup>08].

### 3. Underlying Theory

In order to create and reason about the correctness of a SAT solver, we have to formally define some basic notions of propositional logic. The full formalization has been made in higher-order logic of the system Isabelle and basic knowledge about this system is assumed in the rest of the paper. We will use a syntax similar to the syntax used in Isabelle. Formulas and logical connectives of this logic ( $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\longrightarrow$ ,  $\longleftarrow$ ) are written in the usual way. The symbol  $=$  denotes syntactical identity of two expressions. Function applications are written in prefix form, as in  $(f\ x_1\ \dots\ x_n)$ . Existential quantification is denoted by  $\exists x. \dots$  and universal quantification by  $\forall x. \dots$

We assume that the underlying theory we are defining includes the theory of ordered pairs, lists, (finite) sets, and optional data-types (all of them are built-in in Isabelle). We also assume that record data-types are available. Syntax of these operations is summarized in the first column of Figure 3 and the semantics is informally described in the second column.

**Basic types.** Apart from the basic built-in types, we introduce the types used in propositional logic of CNF formulas as given by Definition 1.

<code>bool</code>	the Boolean type with values <code>True</code> and <code>False</code>
<code>ExtendedBool</code>	the extended Boolean type with values <code>True</code> , <code>False</code> and <code>Undef</code>
<code>nat</code>	the type of natural numbers
<code>('a × 'b)</code>	the type of ordered pairs with elements of types <code>'a</code> and <code>'b</code>
<code>(a, b)</code>	the ordered pair of elements <code>a</code> and <code>b</code>
<code>'a list</code>	the type of lists with elements of type <code>'a</code>
<code>[]</code>	the empty list
<code>[e<sub>1</sub>, ..., e<sub>n</sub>]</code>	the list of <code>n</code> given elements <code>e<sub>1</sub>, ..., e<sub>n</sub></code>
<code>e # list</code>	the list obtained by prepending the element <code>e</code> to the list <code>list</code>
<code>list<sub>1</sub>@list<sub>2</sub></code>	the list obtained by appending the lists <code>list<sub>1</sub></code> and <code>list<sub>2</sub></code>
<code>e ∈ list</code>	<code>e</code> is a member of the list <code>list</code>
<code>(removeAll e list)</code>	the list obtained by removing all occurrences of the element <code>e</code> from the list <code>list</code>
<code>(list_diff list<sub>1</sub> list<sub>2</sub>)</code>	the list obtained from the list <code>list<sub>1</sub></code> by removing all elements of the list <code>list<sub>2</sub></code> from it
<code>(fst list), (hd list)</code>	the first element of the list <code>list</code>
<code>(tl list)</code>	the list obtained by removing the first element of the list <code>list</code>
<code>list ! n</code>	the <code>n</code> -th element of the list <code>list</code>
<code>(last list)</code>	the last element in the nonempty list <code>list</code>
<code>(length list)</code>	the length of the list <code>list</code>
<code>(distinct list)</code>	check if the list <code>list</code> contains no repeating elements
<code>(remdups list)</code>	the list obtained from the list <code>list</code> by removing all its duplicate elements
<code>(filter P list)</code>	the list obtained from the list <code>list</code> by taking all its elements that satisfy the condition <code>P</code>
<code>(map f list)</code>	the list obtained from the list <code>list</code> by applying the function <code>f</code> to all its elements
<code>(prefixToElement e list)</code>	the prefix of the list <code>list</code> up to the first occurrence of the element <code>e</code> (including it)
<code>a &lt;<sup>list</sup> b</code>	the element <code>a</code> precedes the element <code>b</code> in the list <code>b</code>
<code>'a set</code>	the type of sets with elements of type <code>'a</code>
<code>{}</code>	the empty set
<code>e ∈ set</code>	<code>e</code> is a member of the set <code>set</code>
<code>set<sub>1</sub> ∪ set<sub>2</sub></code>	the set union of <code>set<sub>1</sub></code> and <code>set<sub>2</sub></code>
<code> set </code>	the number of elements in the set <code>set</code>
<code>'a option</code>	the type of optional values of the type <code>'a</code>
<code>Some a</code>	the optional value exists and is <code>a</code>
<code>None</code>	the optional value does not exist
<code>f(x := y)</code>	the mapping obtained from the mapping <code>f</code> by setting the value of <code>x</code> to <code>y</code>
<code>rec(f<sub>1</sub> := a<sub>1</sub>, ..., f<sub>k</sub> := a<sub>k</sub>)</code>	the record obtained from the record <code>rec</code> by setting the values of fields <code>f<sub>1</sub>, ..., f<sub>k</sub></code> to values <code>a<sub>1</sub>, ..., a<sub>k</sub></code> , respectively

Figure 3: Summary of Isabelle's basic types and operations

**Definition 1.**


---

Variable	natural number.
Literal	either a positive variable ( $\text{Pos } vbl$ ) or a negative variable ( $\text{Neg } vbl$ )
Clause	a list of literals
Formula	a list of clauses
Valuation	a list of literals

---

Or in Isabelle's syntax:

```

types      Variable = nat
datatype Literal = Pos Variable | Neg Variable
types      Clause   = "Literal list"
types      Formula  = "Clause list"
types      Valuation= "Literal list"

```

For the sake of readability, we will sometime omit printing types and use the following naming convention: literals (i.e., variables of the type `Literal`) are denoted by  $l$  (e.g.,  $l, l', l_0, l_1, l_2, \dots$ ), variables by  $vbl$ , clauses by  $c$ , formulae by  $F$ , and valuations by  $v$ .

Most of the following definitions are formalized by using the primitive recursion, so that they can be used to generate executable code. However, in order to simplify the presentation and improve readability we give their characterizations in an informal way and omit the Isabelle code.

**Definition 2.** The opposite literal of a literal  $l$ , denoted  $\bar{l}$ , is defined by:  $\overline{\text{Pos } vbl} = \text{Neg } vbl$ ,  $\overline{\text{Neg } vbl} = \text{Pos } vbl$ .

We abuse the notation and overload some symbols. For example, the symbol  $\in$  denotes both set membership and list membership. It is also used to denote that a literal occurs in a formula.

**Definition 3.** A formula  $F$  contains a literal  $l$  (i.e., a literal  $l$  occurs in a formula  $F$ ), denoted  $l \in F$ , iff  $\exists c. c \in F \wedge l \in c$ .

Symbol `vars` is also overloaded and denotes the set of variables occurring in a clause, in a formula, or in a valuation.

**Definition 4.** The set of variables that occur in a clause  $c$  is denoted by  $(\text{vars } c)$ . The set of variables that occur in a formula  $F$  is denoted  $(\text{vars } F)$ . The set of variables that occur in a valuation  $v$  is denoted  $(\text{vars } v)$ .

The semantics is introduced by the following definitions.

**Definition 5.** A literal  $l$  is true in a valuation  $v$ , denoted  $v \models l$ , iff  $l \in v$ . A clause  $c$  is true in a valuation  $v$ , denoted  $v \models c$ , iff  $\exists l. l \in c \wedge v \models l$ . A formula  $F$  is true in a valuation  $v$ , denoted  $v \models F$ , iff  $\forall c. c \in F \Rightarrow v \models c$ .

We will write  $v \not\models l$  to denote that  $l$  is not true in  $v$  (note that it does not mean that  $l$  is false in  $v$ ),  $v \not\models c$  to denote that  $c$  is not true in  $v$ , and  $v \not\models F$  to denote that  $F$  is not true in  $v$ . We will say that  $l$  (or  $c$ , or  $F$ ) is *unsatisfied* in  $v$ .

**Definition 6.** A literal  $l$  is false in a valuation  $v$ , denoted  $v \models \neg l$ , iff  $\bar{l} \in v$ . A clause  $c$  is false in a valuation  $v$ , denoted  $v \models \neg c$ , iff  $\forall l. l \in c \Rightarrow v \models \neg l$ . A formula  $F$  is false in a valuation  $v$ , denoted  $v \models \neg F$ , iff  $\exists c. c \in F \wedge v \models \neg c$ .

We will write  $v \not\models l$  to denote that  $l$  is not false in  $v$ ,  $v \not\models \neg c$  to denote that  $c$  is not false in  $v$ , and  $v \not\models \neg F$  to denote that  $F$  is not false in  $v$ . We will say that  $l$  (or  $c$ , or  $F$ ) is *unfalsified* in  $v$ .

**Definition 7.** A valuation  $v$  is inconsistent, denoted (inconsistent  $v$ ), iff it contains both literal and its opposite i.e., iff  $\exists l. v \models l \wedge v \models \bar{l}$ . A valuation is consistent, denoted (consistent  $v$ ), iff it is not inconsistent.

**Definition 8.** A model of a formula  $F$  is a consistent valuation under which  $F$  is true. A formula  $F$  is satisfiable, denoted ( $\text{sat } F$ ), iff it has a model i.e.,  $\exists v. (\text{consistent } v) \wedge v \models F$ .

**Definition 9.** A formula  $F$  entails a clause  $c$ , denoted  $F \models c$ , iff  $c$  is true in every model of  $F$ . A formula  $F$  entails a literal  $l$ , denoted  $F \models l$ , iff  $l$  is true in every model of  $F$ . A formula  $F$  entails valuation  $v$ , denoted  $F \models v$ , iff it entails all its literals i.e.,  $\forall l. l \in v \Rightarrow F \models l$ . A formula  $F_1$  entails a formula  $F_2$  denoted  $F_1 \models F_2$ , if every model of  $F_1$  is a model of  $F_2$ .

**Definition 10.** Formulae  $F_1$  and  $F_2$  are logically equivalent, denoted  $F_1 \equiv F_2$ , iff any model of  $F_1$  is a model of  $F_2$  and vice versa, i.e., iff  $F_1 \models F_2$  and  $F_2 \models F_1$ .

**Definition 11.** A clause  $c$  is unit in a valuation  $v$  with a unit literal  $l$ , denoted ( $\text{isUnit } c \ l \ v$ ) iff  $l \in c$ ,  $v \not\models l$ ,  $v \not\models \neg l$  and  $v \models \neg(c \setminus l)$  (i.e.,  $\forall l'. l' \in c \wedge l' \neq l \Rightarrow v \models \neg l'$ ).

**Definition 12.** A clause  $c$  is a reason for propagation of literal  $l$  in valuation  $v$ , denoted ( $\text{isReason } c \ l \ v$ ) iff  $l \in c$ ,  $v \models l$ ,  $v \models \neg(c \setminus l)$ , and for each literal  $l' \in (c \setminus l)$ , the literal  $\bar{l}'$  precedes  $l$  in  $v$ .

**Definition 13.** The resolvent of clauses  $c_1$  and  $c_2$  over the literal  $l$ , denoted ( $\text{resolvent } c_1 \ c_2 \ l$ ) is the clause  $(c_1 \setminus l) @ (c_2 \setminus \bar{l})$ .

**Definition 14.** A clause  $c$  is a tautological clause, denoted ( $\text{clauseTautology } c$ ), if it contains both a literal and its opposite (i.e.,  $\exists l. l \in c \wedge \bar{l} \in c$ ).

**Definition 15.** The conversion of a valuation  $v$  to a formula  $\langle v \rangle$  is the list that contains all single literal clauses made of literals from  $v$ .

**Assertion Trail.** In order to build a non-recursive implementation of the DPLL algorithm, the notion of valuation should be slightly extended. During the solving process, the solver should keep track of the current partial valuation. In that valuation, some literals are called *decision literals*. Non-decision literals are called *implied literals*. These check-pointed sequences that represent valuations with marked decision literals will be stored in the data structure called *assertion trail*. All literals that belong to the trail will be called *asserted literals*. Assertion trail operates as a stack and literals are always added and removed from its top. We extend the underlying theory with the type `LiteralTrail`, as given by Definition 16:

**Definition 16.**

---

`LiteralTrail` a list of literals, with some of them marked as decision literals.

---



We will denote variables of the type `LiteralTrail` by  $M$  (e.g.,  $M, M', M_0, \dots$ ).

**Example 1.** A trail  $M$  could be  $[+1, |-2, +6, |+5, -3, +4, |-7]$ . The symbol  $+$  is written instead of the constructor `Pos`, the symbol  $-$  instead of `Neg` and the decision literals are marked with the symbol  $|$  on their left hand sides.

A trail can be implemented, for example, as a list of `(Literal, bool)` ordered pairs and all following definitions will be based on this specific implementation. Our SAT solver implementation effectively uses the `LiteralTrail` datatype and so we also show its Isabelle formalization.

```
types LiteralTrail = "(Literal × bool) list"
```

**Definition 17.** For a trail element  $a$ ,  $(\text{element } a)$  denotes the first (*Literal*) component and  $(\text{isDecision } a)$  denotes the second (*Boolean*) component. For a trail  $M$ ,  $(\text{elements } M)$  (abbreviated as  $\widehat{M}$ ) denotes the list of all its elements and  $(\text{decisions } M)$  denotes the list of all its marked elements (i.e., of all its decision literals).

```
definition element :: "(Literal × bool) ⇒ Literal"
where "element x = fst x"
```

```
definition isDecision :: "(Literal × bool) ⇒ bool"
where "isDecision x = snd x"
```

```
definition elements :: "LiteralTrail ⇒ Literal list"
where "elements M = map element M"
```

```
definition decisions :: "LiteralTrail ⇒ Literal list"
where "decisions trail = filter (λ e. isDecision e) trail"
```

**Definition 18.**  $(\text{decisionsTo } M \ l)$  is the list of all marked elements from a trail  $M$  that precede the first occurrence of the element  $l$ , including  $l$  if it is marked.

```
definition decisionsTo :: "Literal ⇒ LiteralTrail ⇒ Literal list"
where
"decisionsTo e trail = decisions (prefixToElement e trail)"
```

**Example 2.** For the trail given in Example 1,  $(\text{decisions } M) = [-2, +5, -7]$ ,  $(\text{decisionsTo } M \ +4) = [-2, +5]$ , and  $(\text{decisionsTo } M \ -7) = [-2, +5, -7]$ .

**Definition 19.** The current level for a trail  $M$ , denoted  $(\text{currentLevel } M)$ , is the number of marked literals in  $M$ , i.e.,  $(\text{currentLevel } M) = (\text{length } (\text{decisions } M))$ .

```
definition currentLevel :: "LiteralTrail ⇒ nat"
where
"currentLevel trail = length (decisions trail)"
```

**Definition 20.** The decision level of a literal  $l$  in a trail  $M$ , denoted  $(\text{level } l \ M)$ , is the number of marked literals in the trail that precede the first occurrence of  $l$ , including  $l$  if it is marked, i.e.,  $(\text{level } l \ M) = (\text{length } (\text{decisionsTo } M \ l))$ .

```
definition elementLevel :: "Literal ⇒ LiteralTrail ⇒ nat"
where
"elementLevel e trail = length (decisionsTo e trail)"
```

**Definition 21.** ( $\text{prefixToLevel } M \text{ level}$ ) is the prefix of a trail  $M$  containing all elements of  $M$  with levels less or equal to  $\text{level}$ .

```
definition prefixToLevel :: "nat  $\Rightarrow$  LiteralTrail  $\Rightarrow$  LiteralTrail"
```

**Example 3.** For the trail in Example 1,  $(\text{level } +1 \ M) = 0$ ,  $(\text{level } +4 \ M) = 2$ ,  $(\text{level } -7 \ M) = 3$ ,  $(\text{currentLevel } M) = 3$ ,  $(\text{prefixToLevel } M \ 1) = [+1, |+2, +6]$ .

**Definition 22.** The last asserted literal of a clause  $c$ , denoted  $(\text{lastAssertedLiteral } c \ \widehat{M})$ , is the literal from  $c$  that is in  $\widehat{M}$ , such that no other literal from  $c$  comes after it in  $\widehat{M}$ .

The function `isLastAssertedLiteral` is used to check if the given literal is the last asserted literal of the given clause in the given valuation.

```
definition isLastAssertedLiteral::"Literal  $\Rightarrow$  Literal list  $\Rightarrow$  Valuation  $\Rightarrow$  bool"
where
"isLastAssertedLiteral literal clause valuation =
  literal  $\in$  clause  $\wedge$  valuation  $\models$  literal  $\wedge$ 
  ( $\forall$  literal'. literal'  $\in$  clause  $\wedge$  literal'  $\neq$  literal  $\longrightarrow$ 
    literal  $\not\stackrel{\text{valuation}}{\prec}$  literal')"
```

The function `getLastAssertedLiteral` is used to detect the last asserted literal of the given clause in the given valuation.

```
definition getLastAssertedLiteral :: "Clause  $\Rightarrow$  Valuation  $\Rightarrow$  Literal"
where
"getLastAssertedLiteral clause valuation =
  last (filter ( $\lambda$  l. l  $\in$  clause) valuation)"
```

**Example 4.** Let  $c$  is  $[+4, +6, -3]$  and  $M$  is the trail from Example 1. Then,  $(\text{lastAssertedLiteral } c \ \widehat{M}) = +4$ .

## 4. SAT Solver Formalization

In this section we will present formalized implementation of a SAT solver within the underlying theory introduced in Section 3. Different concepts and algorithms will be described in separate subsections. Together with the solver implementation we will give conditions that describe its variables and their relationships that must be invariant for the solver functions. These invariants fully characterize the role of some variables in the system and help understanding the whole system. Because invariants are listed simultaneously with the implementation, the style used can be seen as implementation driven by its specification.

Note that the following solver description is very formal and concise, and that some previous knowledge about the SAT solving technology is assumed. Useful tutorial descriptions can be found, for example, in [GKSS07, Mar08a].

### 4.1. Solver State

In an imperative or object-oriented language, the state of the solver is represented by using global or class variables. Functions of the solver access and change these variables as their side-effects. In HOL, functions cannot have side-effects, so the solver state must be wrapped up in a record and passed around

with each function call. Therefore, all functions in our functional implementation will receive the current solver state as their last parameter and return the modified state as their result. This cumbersome feature can be avoided if monadic programming is used, as it will be described in Section 8.

The state of the solver is represented by the following record:

```
record State =
  "getSATFlag"      :: ExtendedBool
  "getF"            :: Formula
  "getM"            :: LiteralTrail
  "getConflictFlag" :: bool
  "getConflictClause" :: pClause
  "getQ"            :: "Literal list"
  "getReason"       :: "Literal => pClause option"
  "getWatch1"       :: "pClause => Literal option"
  "getWatch2"       :: "pClause => Literal option"
  "getWatchList"    :: "Literal => pClause list"
  "getC"            :: Clause
  "getCl"           :: Literal
  "getCl1"          :: Literal
```

The data-type `pClause` is just a synonym for `nat` and it indicates “pointers” to clauses i.e., indices of clauses in the clause list representing the formula.

Basic variables of the solver state are the following.

- The variable *SATFlag* reflects *the status of the solving process* and it remains *Undef* until the formula which is being solved is detected to be satisfiable (when *SATFlag* is set to *True*) or to be unsatisfiable (when *SATFlag* is set to *False*). Its characterization will be the main partial correctness result and it will be proved in Section 5.

*Invariant*<sub>SATFlag</sub><sup>2</sup>:

$$SATFlag = True \leftrightarrow (\text{sat } F_0) \wedge SATFlag = False \leftrightarrow \neg(\text{sat } F_0),$$

where  $F_0$  is the formula tested for satisfiability.

- The literal trail  $M$  contains *the current partial valuation* (i.e.,  $\widehat{M}$  is the current partial valuation). It is characterized by the following invariants:

*Invariant*<sub>Mconsistent</sub>:

$$(\text{consistent } \widehat{M})$$

*Invariant*<sub>Mdistinct</sub>:

$$(\text{distinct } \widehat{M}),$$

which ensure that  $M$  also represents a mapping of some variables to their truth values.

The trail  $M$  contains literals whose variables are in the initial formula  $F_0$  and literals whose variables are in the special set of decision variables (denoted by *decisionVars* and used in *decide* operation formalized in Section

---

<sup>2</sup>We will say that *a state satisfies an invariant* and that *invariant holds in a state* if the components (`getXXX`) of the state satisfy the condition given by the invariant.

4.6). Note that these two sets usually coincide, but this is not necessarily the case. This *domain property* of  $M$  is given by the following invariant.

*Invariant<sub>Mvars</sub>*:

$$(\text{vars } M) \subseteq (\text{vars } F_0) \cup \text{decisionVars}$$

- The formula  $F$  will be referred to as *the current set of clauses*. It changes during the solving process and its clauses are either (simplified) clauses of the initial formula  $F_0$  or its consequences that are learned during the solving process. Since initial clauses are built from literals of  $F_0$  and learned clauses are built from literals of  $M$ , the formula  $F$  satisfies the following domain property.

*Invariant<sub>Fvars</sub>*:

$$(\text{vars } F) \subseteq (\text{vars } F_0) \cup \text{decisionVars}$$

All clauses in  $F$  will have at least two different literals. Single literal clauses  $[l]$  will never be added to  $F$ , but instead their only literal  $l$  will be immediately added to  $M$ . Indeed, adding a single literal clause  $[l]$  to  $F$  would be useless because its only literal  $l$  must be contained in every satisfying valuation and  $[l]$  is automatically satisfied when  $l$  is asserted. To ensure correctness, once these literals are added to  $M$ , they must never get removed from it. This is the case in the implementation we provide, since all these literals will be asserted at the decision level zero of the trail  $M$  which never gets backtracked.

As said, all clauses in  $F$  are logical consequences of  $F_0$ . Also, the decision level zero of the trail  $M$  contains literals that are logical consequences of the formula  $F_0$ . The following invariant describing the relation between the initial formula  $F_0$ , the formula  $F$ , and the trail  $M$  plays a very important role in the soundness and completeness of the solving process. It states that the formula  $F_0$  is fully characterized by the formula  $F$  and the decision level zero of the trail  $M$ .

*Invariant<sub>equivalent</sub>*:

$$F_0 \equiv F @ (\widehat{\text{prefixToLevel } 0} M)$$

The fact that  $F$  contains only clauses with two or more different literals also simplifies the implementation of the two-watch literal scheme (see Section 4.4.1).

Other components of the solver state are used in specific phases of the solving process and will be explained in the following sections.

#### 4.2. Initialization

In this section we describe the process of initializing the solver state by the given formula  $F_0$  tested for satisfiability.

The function `initialize` calls `addClause` for each clause in  $F_0$  which appropriately updates the solver state.

```
primrec initialize :: "Formula ⇒ State ⇒ State"
where
  "initialize [] state = state" |
  "initialize (clause # formula) state =
    initialize formula (addClause clause state)"
```

The function `initialize` is initially called only for `initialState`, so there are no decision literals in  $M$  when it is called.

```

definition initialState :: "State"
where
initialState =
  (| getSATFlag = UNDEF,
    getF = [],
    getM = [],
    getConflictFlag = False,
    getConflictClause = 0,
    getQ = [],
    getWatch1 = λ c. None,
    getWatch2 = λ c. None,
    getWatchList = λ l. [],
    getReason = λ l. None,
    getC = arbitrary,
    getCl = arbitrary,
    getCll = arbitrary
  |)

```

Before we introduce the function `addClause`, we define an auxiliary function `removeFalseLiterals` used to simplify clauses. It removes all literals from the given clause that are false in the given valuation.

```

definition removeFalseLiterals :: "Clause ⇒ Valuation ⇒ Clause"
where
removeFalseLiterals clause valuation =
  filter (λ l. valuation  $\not\models$  l) clause

```

The function `addClause` (called only by `initialize`) preprocesses the clause by removing its repeated literals and removing its literals that are false in the current trail  $M$ . After this, several cases arise.

- If the clause is satisfied in the current trail  $M$ , it is just skipped. The rationale for this is that if there is a satisfying valuation for  $F_0$ , it will be an extension of the current trail  $M$ , so it will also satisfy the clause that was skipped.
- If the clause is empty after preprocessing, the formula  $F_0$  is unsatisfiable and *SATFlag* is set to *False*, since the empty clause cannot be satisfied in any valuation.
- Tautological clauses (i.e., clauses containing both a literal and its opposite) are also skipped since they can always be satisfied.

The two remaining cases actually update  $F$  or  $M$ .

1. As described, clauses  $[l]$  containing only a single literal  $l$  are treated in a special way. Since they can only be satisfied if their literal  $l$  is true in  $M$ ,  $l$  is immediately added to  $M$ . Then a round of unit propagation (see Section 4.5) is performed, which can infer further consequences of asserting  $l$ .
2. Clauses containing more than one literal are added to  $F$  and data structures related to the two-watch literal scheme are appropriately initialized (see Section 4.4.1).

```

definition addClause :: "Clause  $\Rightarrow$  State  $\Rightarrow$  State"
where
"addClause clause state =
  (let clause' =
    (remdups (removeFalseLiterals clause (elements (getM state)))) in
  (if elements (getM state)  $\models$  clause' then
    state
  else (if clause'=[] then
    state (| getSATFlag := False |)
  else (if length clause' = 1 then
    let state' = (assertLiteral (hd clause') False state) in
    exhaustiveUnitPropagate state'
  else (if clauseTautology clause' then
    state
  else
    let clauseIndex = length (getF state) in
    let state' = state(| getF := (getF state) @ [clause'] |) in
    let state'' = setWatch1 clauseIndex (clause' ! 0) state' in
    let state''' = setWatch2 clauseIndex (clause' ! 1) state'' in
    state''')
  )))
)"

```

### 4.3. Top Level Solver Operation

The only function of the solver that end-users are expected to call is the function `solve`. First it performs initialization and then it performs the main solver loop while the status of the solving process (given by the variable *SATFlag*) is *UNDEF*. The first time *SATFlag* changes, the main solver loop stops and the current value of *SATFlag* is the final solver result.

```

definition solve :: "Formula  $\Rightarrow$  ExtendedBool"
where
"solve F0 = getSATFlag
  (solve_loop
    (initialize F0 initialState)
    (vars F0)
  )
"

function (domintros, tailrec) solve_loop :: "State  $\Rightarrow$  Variable set  $\Rightarrow$  State"
where
"solve_loop state decisionVars =
  (if (getSATFlag state)  $\neq$  UNDEF then
    state
  else
    let state' = solve_loop_body state decisionVars in
    solve_loop state' decisionVars
  )
"
by pat_completeness auto

```

Note that the `solve_loop` is defined by general recursion, so its termination is not trivial.

The body of the solver loop begins with a round of exhaustive unit propagation. After that, four different cases arise.

1. It has been detected that  $M \models \neg F$ . In that case we say that a *conflict* occurred.

- (a) If there are no decision literals in  $M$ , we say that a *conflict at decision level zero* occurred and it is determined that the formula  $F_0$  is unsatisfiable. In that case,  $SATFlag$  is set to  $FALSE$ .
  - (b) If there are some decision literals in  $M$ , then the *conflict analysis and resolving procedure* is performed (see Section 4.7).
2. It has been detected that  $M \not\models \neg F$ .
- (a) If all variables from the fixed variable set  $decisionVars$  are defined in the current trail  $M$ , it is determined that the formula is satisfiable. In that case,  $SATFlag$  is set to  $TRUE$ . The set  $decisionVars$  must meet additional requirements in order to guarantee soundness of this conclusion. For example, it suffices that  $(vars F_0) \subseteq decisionVars$ , as it is the case in our implementation.
  - (b) If there are some decision variables that are undefined in  $M$ , a new decision is made (see Section 4.6) and a decision literal is asserted.

The detection of clauses of  $F$  that are false in  $\widehat{M}$  or unit in  $\widehat{M}$  must be done efficiently so that it does not become the bottleneck of the whole solver. An optimized way to achieve this is given in Section 4.4.

```

definition solve_loop_body :: "State  $\Rightarrow$  Variable set  $\Rightarrow$  State"
where
"solve_loop_body state decisionVars =
  (let state_up = exhaustiveUnitPropagate state in
   (if (getConflictFlag state_up) then
     (if (currentLevel (getM state_up)) = 0 then
       state_up(| getSATFlag := False |)
     else
       let state_c = applyConflict state_up in
       let state_e = applyExplainUIP state_c in
       let state_l = applyLearn state_e in
       let state_b = applyBackjump state_l in
       state_b
     )
   else
     (if (vars (elements (getM state')))  $\supseteq$  decisionVars) then
       state'(| getSATFlag := TRUE |)
     else
       applyDecide state' decisionVars
     )
  ))
"

```

#### 4.4. Conflict and Unit Clause Detection

Each time a literal is added to  $M$ , the formula  $F$  is checked for existence of unit or false clauses. Results of this check are stored in the following state variables.

- The variable  $conflictFlag$  is set when it is determined that the current set of clauses  $F$  is false in the valuation  $\widehat{M}$ . The invariant that fully characterizes it is:

*Invariant*<sub>conflictFlagCharacterization</sub>:

$$conflictFlag \longleftrightarrow \widehat{M} \models \neg F$$

- The number *conflictClause* is the index of a clause in  $F$  that is false in the valuation  $\widehat{M}$ . Its defining invariant is:

*Invariant<sub>conflictClauseCharacterization</sub>*:

$$\text{conflictFlag} \longrightarrow \text{conflictClause} < |F| \wedge \widehat{M} \models \neg (F! \text{conflictClause})$$

- The list  $Q$  is a list of all literals that are unit literals for clauses in  $F$  which are unit clauses wrt. the valuation  $\widehat{M}$ . These literals are ready to be asserted in  $M$  as a result of the *unit propagation* operation. The unit propagation queue  $Q$  is fully characterized by the following invariant.

*Invariant<sub>QCharacterization</sub>*:

$$\neg \text{conflictFlag} \longrightarrow (\forall l. l \in Q \iff (\exists c. c \in F \wedge (\text{isUnitClause } c \ l \ \widehat{M})))$$

Note that this condition guarantees the *completeness for unit propagation* i.e., it guarantees that all unit literals for unit clauses in  $F$  are contained in  $Q$ . This is not necessary for the soundness nor completeness of the whole procedure, but, if satisfied, leads to better efficiency.

Also, there should be no repeated elements in  $Q$ .

*Invariant<sub>Qdistinct</sub>*:

$$(\text{distinct } Q)$$

As  $Q$  is built of literals of  $F$  its domain (its set of variables) is the same as the domain of  $F$ .

*Invariant<sub>Qvars</sub>*

$$(\text{vars } Q) \subseteq (\text{vars } F_0) \cup \text{decisionVars}$$

- The mapping *reason* maps literals in  $Q$  to indices of clauses in  $F$  for which they are the unit literals. Since this mapping does not change when the literals from  $Q$  get asserted in  $M$ , it continues to map non-decision literals of  $M$  to indices of clauses in  $F$  that are reasons for their propagation. Notice that no reason clauses can be attached to the literals at the decision level zero. This is because literals at the decision level zero have a special role in the solving process, as they can get asserted by propagating single literal clauses which are not explicitly stored in  $F$ , as described in Section 4.1. All this is characterized by the following complex invariant.

*Invariant<sub>reasonCharacterization</sub>*:

$$\begin{aligned} & ((\text{currentLevel } M) > 0 \longrightarrow \forall l. l \in Q \longrightarrow \\ & \quad (\exists c. (\text{reason } l) = (\text{Some } c) \wedge c < |F| \wedge (\text{isUnit } (F!c) \ l \ \widehat{M}))) \wedge \\ & (\forall l. l \in \widehat{M} \wedge l \notin (\text{decisions } M) \wedge (\text{level } l) > 0 \longrightarrow \\ & \quad (\exists c. (\text{reason } l) = (\text{Some } c) \wedge c < |F| \wedge (\text{isReason } (F!c) \ l \ \widehat{M}))) \end{aligned}$$



#### 4.4.1. Two-watched Literal Scheme

An efficient way to check for false and unit clauses is by using the *two-watch literal scheme*. It introduces the following variables to the state.

- Mappings  $watch_1$  and  $watch_2$  assign two distinguished literals to each clause of  $F$ . This condition is imposed through the following invariants.

*Invariant<sub>watchesEl</sub>*:

$$\forall c. c < |F| \longrightarrow \exists w1 w2. \quad (watch_1 c) = (\text{Some } w1) \wedge w1 \in F!c \wedge \\ (watch_2 c) = (\text{Some } w2) \wedge w2 \in F!c$$

*Invariant<sub>watchesDiffer</sub>*:

$$\forall c. c < |F| \longrightarrow (watch_1 c) \neq (watch_2 c)$$

- The mapping  $watchList$  assigns to each literal  $l$  a list of clause indices in  $F$  that represent clauses in which  $l$  is a watched literal. This is imposed by the following invariants.

*Invariant<sub>watchListsCharacterization</sub>*:

$$\forall l c. \quad c \in (watchList l) \longleftrightarrow \\ c < |F| \wedge ((watch_1 c) = (\text{Some } l) \vee (watch_2 c) = (\text{Some } l))$$

It also holds that watch lists do not contain repeated clauses.

*Invariant<sub>watchListsDistinct</sub>*:

$$\forall l. (\text{distinct } (watchList l))$$

Next, we describe the function `assertLiteral` that adds the given literal (either decision or implied) to the trail  $M$ . The variables `conflictFlag`, `conflictClause`,  $Q$ , and `reason` are then updated by using the two watched literal propagation scheme encoded by the function `notifyWatches`.

```
definition assertLiteral :: "Literal  $\Rightarrow$  bool  $\Rightarrow$  State  $\Rightarrow$  State"
where
"assertLiteral literal decision state =
  let state' = state| getM := (getM state) @ [(literal, decision)] | in
  notifyWatches (opposite literal) state'
"
```

Before we introduce and explain the function `notifyWatches`, we introduce several auxiliary functions.

Functions `setWatch1` and `setWatch2` promote the given literal to be a new watched literal of the given clause and then add that clause to its watch list<sup>3</sup>.

```
definition addToWatchList :: "Literal  $\Rightarrow$  pClause  $\Rightarrow$  State  $\Rightarrow$  State"
"addToWatchList literal clause state =
  state| getWatchList :=
    (getWatchList state)"
```

---

<sup>3</sup>Only `setWatch1` is listed since `setWatch2` is similar.

```

        (literal := clause # (getWatchList state literal))
    )
"
definition setWatch1 :: "pClause ⇒ Literal ⇒ State ⇒ State"
where
"setWatch1 clause literal state =
    let state' = state( getWatch1 :=
        (getWatch1 state)(clause := Some literal) ) in
    addToWatchList literal clause state' "

```

The function `swapWatches` swaps the two watched literals of the given clause.

```

definition swapWatches :: "pClause ⇒ State ⇒ State"
where
"swapWatches clause state =
    state( getWatch1 :=
        (getWatch1 state)(clause := (getWatch2 state clause)),
        getWatch2 :=
        (getWatch2 state)(clause := (getWatch1 state clause))
    )
"

```

The function `getNonWatchedUnfalsifiedLiteral` checks if there is a literal in the given clause, other than its watched literals, which is not false in  $M$ .

```

primrec getNonWatchedUnfalsifiedLiteral ::
    "Clause ⇒ Literal ⇒ Literal ⇒ LiteralTrail ⇒ Literal option"
where
"getNonWatchedUnfalsifiedLiteral [] w1 w2 M = None" |
"getNonWatchedUnfalsifiedLiteral (literal # clause) w1 w2 M =
    (if literal ≠ w1 ∧ literal ≠ w2 ∧ (elements M) ⊭¬ literal then
        Some literal
    else
        getNonWatchedUnfalsifiedLiteral clause w1 w2 M
    )
"

```

The function `setReason` updates the mapping *reason* by assigning the given clause index to the given literal.

```

definition setReason :: "Literal ⇒ pClause ⇒ State ⇒ State"
where
"setReason literal clause state =
    state( getReason := (getReason state)(literal := Some clause) )
"

```

Next, we explain the essence of the two-watch literal scheme encoded in the functions `notifyWatches` and `notifyWatches_loop`. The two-watch literal scheme relies on the fact that a watched literal of a clause can be false in  $\widehat{M}$  only when the clause is either true, false or unit in  $M$ . In all other cases (when it is undefined and is not unit), both watched literals of the clause are known to be unfalsified. This is formalized by the following invariant (with two instances for  $i = 1$  and  $i = 2$ ).

$$\begin{aligned}
 \forall c. c < |F| \longrightarrow M \models \neg(\text{watch}_i c) \longrightarrow \\
 & (\exists l. l \in c \wedge M \models l \wedge \text{level } l \leq \text{level } \overline{(\text{watch}_i c)}) \vee \\
 & (\forall l. l \in c \wedge l \neq (\text{watch}_1 c) \wedge l \neq (\text{watch}_2 c) \longrightarrow \\
 & \quad M \models \neg l \wedge \text{level } \bar{l} \leq \text{level } \overline{(\text{watch}_i c)}).
 \end{aligned}$$

Note that the additional conditions imposed on the literal levels are required only for the correctness of backjumping, as described in Section 4.7.

During the `assertLiteral` operation, the trail  $M$  gets extended by a literal  $l$ . When this happens, all clauses that do not have  $\bar{l}$  as their watched literal still satisfy the condition of *Invariant<sub>watchCharacterization</sub>* and they cannot be unit nor false in the extended trail. The only clauses that could have become unit or false are the ones that have  $\bar{l}$  as their watched literal. These clauses are exactly the ones whose indices are contained in  $(watchList \bar{l})$ . The function `notifyWatches` calls the function `notifyWatches_loop` which traverses this list and processes all clauses represented by it. In order to simplify the implementation, for each processed clause index  $c$ , watches are swapped if necessary so that it is ensured that  $(watch_2 c) = \bar{l}$  and so  $(watch_2 c)$  is false. The following cases may further arise:

1. If it can be quickly detected that the clause  $F!c$  contains a true literal  $t$ , there is no need to change its watches, since it satisfies the condition of *Invariant<sub>watchCharacterization</sub>* for the extended trail. In order to achieve high performance, this check should be done only by using the clause index and other data structures which are most of the time present in the processor cache, without accessing the clause itself. The older solvers checked only if  $(watch_1 c)$  is true in  $M$  and this is the case in the implementation we provide. Some new solvers sometimes cache some arbitrary literals of the clause and check if they are true in  $M$ .
2. If a quick check does not detect a true literal  $t$ , then the clause is accessed and its other literals are examined by the function `getUnfalsifiedNon-WatchedLiteral`.
  - (a) If there exist a non-watched literal  $l$  that is not false in  $M$ , it becomes a new  $(watch_2 c)$ .
  - (b) If all non-watched literals and  $(watch_1 c)$  are false in  $M$ , then the whole clause is false and *conflictFlag* is raised. The watches are not changed, since they will both become undefined in  $M$ , if the backjump operation is performed (see Section 4.7).
  - (c) If all non-watched literals are false in  $M$ , but  $(watch_1 c)$  is undefined, then the clause just became a unit clause and  $(watch_1 c)$  is enqueued in  $Q$  for propagation (if it is not already present there). The reason for its propagation is set to  $c$ . The watches are not changed, as the clause will have a true literal  $(watch_1 c)$  after propagation.

When a literal which was not watched becomes a new  $(watch_2 c)$ , the literal  $\bar{l}$  stops being the watched literal of  $c$  and the clause index  $c$  should be removed from its watch list. Since this happens many time during the traversal performed by the `notifyWatches_loop`, it turns out that it is more efficient to regenerate the new watch list for the literal  $\bar{l}$ , then to do successive remove operations instead. This is the role of *newWl* parameter in the `notifyWatches_loop` function.

```

definition notifyWatches :: "Literal  $\Rightarrow$  State  $\Rightarrow$  State"
where
  "notifyWatches literal state =
    notifyWatches_loop literal (getWatchList state literal) [] state
  "

```

```

primrec
notifyWatches_loop :: "Literal  $\Rightarrow$  pClause list  $\Rightarrow$  pClause list  $\Rightarrow$  State  $\Rightarrow$  State"
where
"notifyWatches_loop literal [] newWl state =
  state(| getWatchList := (getWatchList state)(literal := newWl) |)" |
"notifyWatches_loop literal (clause # list') newWl state =
  (let state' = (if Some literal = (getWatch1 state clause) then
    (swapWatches clause state)
    else
      state) in
  case (getWatch1 state' clause) of Some w1  $\Rightarrow$  (
  case (getWatch2 state' clause) of Some w2  $\Rightarrow$  (
  (if (literalTrue w1 (elements (getM state')))) then
    notifyWatches_loop literal list' (newWl @ [clause]) state'
  else
  (case (getNonWatchedUnfalsifiedLiteral
    ((getF state') ! clause) w1 w2 (getM state')) of
  Some l'  $\Rightarrow$ 
    notifyWatches_loop literal list' newWl (setWatch2 clause l' state') |
  None  $\Rightarrow$ 
    (if (literalFalse w1 (elements (getM state')))) then
      let state'' = (state'(| getConflictFlag := True,
        getConflictClause := clause |) in
      notifyWatches_loop literal list' (newWl @ [clause]) state''
    else
      let state'' = state'(| getQ := (if w1 el (getQ state') then
        (getQ state')
        else
          (getQ state') @ [w1]
        )) in
      let state''' = (setReason w1 clause state'') in
      notifyWatches_loop literal list' (newWl @ [clause]) state'''
    )
  )))
  )
"

```

The invariants *Invariant<sub>watchListsCharacterization</sub>* and *Invariant<sub>watchesEl</sub>* together guarantee that for each clause there will always be two watched literals (hence, the missing `None` branches in the case expressions are indeed not needed).

#### 4.5. Unit Propagation

The operation of unit propagation asserts unit literals of unit clauses of  $F$ . Since the two-watch literal scheme is complete for false and unit clause detection (as the function `assertLiteral` preserves *Invariant<sub>conflictFlagCharacterization</sub>* and *Invariant<sub>QCharacterization</sub>*), all unit literals of clauses in  $F$  can be found in  $Q$ . This makes unit propagation a rather trivial operation — literals are picked from  $Q$  and asserted until  $Q$  is not emptied or until a conflict is detected.

```

definition applyUnitPropagate :: "State  $\Rightarrow$  State"
where
"applyUnitPropagate state =
  (let state' = (assertLiteral (hd (getQ state)) False state) in
  state'(| getQ := tl (getQ state') |)
"

```

```

function (domintros, tailrec) exhaustiveUnitPropagate :: "State  $\Rightarrow$  State"
where

```

```

"exhaustiveUnitPropagate state =
  (if (getConflictFlag state) | (getQ state) = [] then
    state
  else
    exhaustiveUnitPropagate (applyUnitPropagate state)
  )
"
by pat_completeness auto

```

Notice that the termination of the `exhaustiveUnitPropagate` function is non-trivial, since it is defined by using the general recursion.

#### 4.6. Decision Heuristics

When unit propagation exhausts, no new literal can be inferred and a kind of backtracking search must be performed. This search is driven by the guesses made by the *decision heuristics*. The heart of the decision heuristics is the `selectLiteral` function whose role is to pick a literal whose variable is in the fixed set of decision variables *decisionVars*, but which is not yet asserted in  $M$ . The literals are selected based on some given criteria. Many different criteria can be used and experimental evidence shows that this heuristic is often crucial for a solver's performance. However, in this paper we will specify it only by its effect given by the following postcondition.

```

consts selectLiteral :: "State  $\Rightarrow$  Variable set  $\Rightarrow$  Literal"
axioms selectLiteral_def:
"let diff = decisionVars \ vars (elements (getM state)) in
  diff  $\neq$   $\emptyset$   $\longrightarrow$  var (selectLiteral state decisionVars)  $\in$  diff"

definition applyDecide :: "State  $\Rightarrow$  Variable set  $\Rightarrow$  State"
where
"applyDecide state decisionVars =
  assertLiteral (selectLiteral state decisionVars) True state
"

```

#### 4.7. Conflict Handling

The conflict handling procedure consists of the *conflict analysis*, *learning* and *backjumping* and it is executed whenever a conflict occurs at a decision level higher than zero (when the conflict occurs at the decision level zero, then the formula is determined to be unsatisfiable). After the conflict handling procedure, a top portion of trail is removed and a non-conflicting state is restored. Unlike the classic backtrack operation which would remove only the last decision made, the backjump operation performs a form of non-chronological backtracking which undoes as many decisions as possible. Backjumping is guided by a *backjump clause*, which is a consequence of the formula  $F_0$  and which corresponds to variable assignment that lead to the conflict. When the backjump clause is constructed, the top literals from the trail  $M$  are removed, until the backjump clause becomes a unit clause in  $M$ . From that point, its unit literal is propagated and the search process continues. Backjump clauses are constructed in the process called *conflict analysis*.

Several components of the solver state are used during the conflict handling procedure.

- The clause  $C$  represents the current conflict analysis clause, which becomes the backjump clause once the conflict analysis process is finished. This clause is characterized by the following invariants<sup>4</sup>.

*Invariant<sub>CFalse</sub>:*

$$\text{conflictFlag} \longrightarrow M \models \neg C$$

*Invariant<sub>CEntailed</sub>:*

$$\text{conflictFlag} \longrightarrow F_0 \models C$$

The following variables represent different aspects of the clause  $C$  and are cached in the solver state only for performance reasons.

- The literal  $C_l$  is the last asserted literal of  $\overline{C}$  in the trail  $M$ .

*Invariant<sub>ClCharacterization</sub>:*

$$\text{conflictFlag} \longrightarrow (\text{isLastAssertedLiteral } C_l \overline{C} M)$$

- The literal  $C_{ll}$  is the last asserted literal of  $\overline{C} \setminus C_l$ .

*Invariant<sub>ClCharacterization</sub>:*

$$\text{conflictFlag} \wedge \overline{C} \setminus C_l \neq [] \longrightarrow (\text{isLastAssertedLiteral } C_{ll} (\overline{C} \setminus C_l) M)$$

- The number  $C_n$  is the number of literals on the highest decision level of the trail  $M$ .

*Invariant<sub>CnCharacterization</sub>:*

$$\begin{aligned} \text{conflictFlag} \longrightarrow \\ C_n = (\text{length } (\text{filter } (\lambda l. \text{level } \overline{l} M = \text{currentLevel } M) (\text{remdups } C))) \end{aligned}$$

#### 4.7.1. Conflict Analysis

In order to implement the conflict analysis procedure, we introduce several auxiliary functions.

The function `findLastAssertedLiteral` is used to set the value of  $C_l$  based on the current values of  $C$  and  $M$ .

```
definition findLastAssertedLiteral :: "State  $\Rightarrow$  State"
where
"findLastAssertedLiteral state =
  state (| getCl := getLastAssertedLiteral (oppositeLiteralList (getC state))
        (elements (getM state)) )"

```

The function `countCurrentLevelLiterals` is used to set the value of  $C_n$  based on the current values of  $C$  and  $M$ .

---

<sup>4</sup>All invariants that are relevant for the conflict handling process need to hold only until the conflict has been resolved. Therefore, they are guarded with the condition  $\text{conflictFlag} \longrightarrow$  so that they can be treated as other global invariants.

```

definition countCurrentLevelLiterals :: "State  $\Rightarrow$  State"
where
"countCurrentLevelLiterals state =
  (let filterFn = ( $\lambda$  l. elementLevel (opposite l) (getM state) =
                    currentLevel (getM state)) in
   state ( $\mid$  getCn := length (filter filterFn (getC state))  $\mid$ ))"

```

Since for some literals asserted at the decision level zero there are no reason clauses in  $F$ , it is required that the clause  $C$  does not contain literals from the decision level zero. Also, it is reasonable to require that the clause  $C$  does not contain repeated literals. The function `setConflictAnalysisClause` sets the clause  $C$  to the given one, but first it preprocesses it by removing duplicates and literals asserted at decision level zero. It also caches the values of  $C_l$  and  $C_n$ .

```

definition setConflictAnalysisClause :: "Clause  $\Rightarrow$  State  $\Rightarrow$  State"
where "setConflictAnalysisClause clause state =
  (let oppM0 = oppositeLiteralList (elements (prefixToLevel 0 (getM state))) in
   let state' = state ( $\mid$  getC := remdups (list_diff clause oppM0)  $\mid$ ) in
   let state'' = findLastAssertedLiteral state' in
   let state''' = countCurrentLevelLiterals state'' in
   state''')"

```

The conflict analysis algorithm can be described as follows:

- The conflict analysis process starts with a conflict clause itself (the clause of  $F$  that is false in  $M$ ) and the clause  $C$  is initialized to it. The function `applyConflict` initializes the clause  $C$  to the current conflict clause.

```

definition applyConflict :: "State  $\Rightarrow$  State"
where
"applyConflict state =
  (let conflictClause = (getF state) ! (getConflictClause state) in
   setConflictAnalysisClause conflictClause state)
"

```

- Each literal contained in the current clause  $C$  is false in the current trail  $M$  and is either a decision made by the search procedure or the result of some propagation. For each propagated literal  $l$ , there is a clause  $c$  that caused the propagation. These clauses are called *reason clauses* and  $(\text{isReason } c \ l \ \widehat{M})$  holds. Propagated literals from the current clause  $C$  are then replaced (we say *explained*) by other literals from the reason clauses, continuing the analysis backwards. The explanation step can be seen as a resolution between the backjump and the reason clause. The function `applyExplain` performs this resolution.

```

definition applyExplain :: "Literal  $\Rightarrow$  State  $\Rightarrow$  State"
where
"applyExplain literal state =
  (case (getReason state literal) of Some reason  $\Rightarrow$ 
   let res = resolve (getC state)
                 (getF state) ! reason
                 (opposite literal) in
   setConflictAnalysisClause res state)"

```

Notice that *Invariant<sub>reasonCharacterization</sub>* guarantees that each propagated literal has an assigned reason clause and that the missing `None` branch in the case expression is not necessary.

- The conflict analysis procedure we implemented always explains the last asserted literal of  $\overline{C}$  and the procedure is repeated until the isUIP condition is fulfilled, i.e., until there is exactly one literal in  $\overline{C}$  such that all other literals of  $\overline{C}$  are asserted at strictly lower decision levels. This condition can be easily checked by examining the value of  $C_n$ . The implementation of this technique is given by the function `applyExplainUIP`.

```

function (domintros, tailrec) applyExplainUIP :: "State  $\Rightarrow$  State"
where
"applyExplainUIP state =
  (if getCn state = 1 then
    state
  else
    applyExplainUIP (applyExplain (getC1 state) state))
"
by pat_completeness auto

```

Notice that this function is defined by general recursion so its termination is non-trivial and must be appropriately addressed.

#### 4.7.2. Learning

During the learning process, the formula  $F$  (or the level zero of the trail  $M$  in case of single literal clauses) is extended by learned (redundant) clauses that are logical consequences of the formula  $F_0$ . In our implementation (as it is often the case in modern SAT solvers), the only clauses that are being learned are the backjump clauses. Since we require that all clauses in  $F$  have more than two different literals, if a backjump clause  $C$  contains only one literal, then learning is not explicitly performed (it is performed implicitly as a part of the backjumping operation). The implementation of learning is given by the function `applyLearn`. After extending  $F$  by  $C$ , the watch literals for the clause  $C$  are set in a way which ensures *Invariant<sub>watchCharacterization</sub>*. In the same time, the literal  $C_l$  is computed and cached.

```

definition applyLearn :: "State  $\Rightarrow$  State"
where
"applyLearn state =
  (if getC state  $\neq$  [opposite (getC1 state)] then
    let state' = state (| getF := (getF state) @ [getC state] |) in
    let C1 = getC1 state in
    let C1l = getLastAssertedLiteral
      (removeAll 1 (oppositeLiteralList (getC state)))
      (elements (getM state)) in
    let clauseIndex = length (getF state) in
    let state'' = setWatch1 clauseIndex (opposite C1) state' in
    let state''' = setWatch2 clauseIndex (opposite C1l) state'' in
    state''' (| getC1l := C1l |)
  else
    state
  )
"

```

#### 4.7.3. Backjumping

The backjump operation consists of removing literals from  $M$  up to a minimal level in which the backjump clause  $C$  becomes a unit clause, after which its unit literal  $\overline{C}_l$  is propagated. This level is found by using the function `getBackjumpLevel`.



```

definition getBackjumpLevel :: "State  $\Rightarrow$  nat"
where
"getBackjumpLevel state =
  (if getC state  $\neq$  [opposite (getCl state)] then
    elementLevel (getCl1 state) (getM state)
  else
    0
  )"

```

The function `applyBackjump` performs the backjump operation itself.

```

definition applyBackjump :: "State  $\Rightarrow$  State"
where
"applyBackjump state =
  (let l = (getCl state) in
    let level = getBackjumpLevel state in
    let state' = state(| getConflictFlag := False,
                       getQ := [],
                       getM := (prefixToLevel level (getM state)) |) in
    let state'' = (if level > 0 then
                  setReason (opposite l) (length (getF state) - 1) state'
                  else
                  state') in
    assertLiteral (opposite l) False state'' )"

```

Notice that after taking the prefix of  $M$ , it is concluded that conflict has been successfully resolved (so *conflictFlag* is unset), and that there are no unit clauses in  $F$  with respect to the taken prefix of  $M$  (so  $Q$  is cleared). For these conclusions to be valid, it is required that no new decisions are made once  $M$  is in a conflicting state. Also, unit propagation has to be exhaustive and no new decisions should be made while there are unit clauses in  $F$ . These conditions are imposed by the following invariants.

*Invariant<sub>noDecisionsWhenConflict</sub>:*

$$\forall level' < (\text{currentLevel } M) \longrightarrow (\text{prefixToLevel } level' \ M) \not\models \neg F$$

*Invariant<sub>noDecisionsWhenUnit</sub>:*

$$\forall level' < (\text{currentLevel } M) \longrightarrow \\ \neg \exists c \ l. \ c \in F \wedge (\text{isUnitClause } c \ l \ (\text{prefixToLevel } level' \ M))$$

## 5. Highlights of the Total Correctness Proof

The invariants listed in Section 4 are sufficient to prove the total correctness of the procedure. Proving that they are preserved by all solver functions was the most involved part of the total correctness proof. These proofs are available in [Mar08b] and we will not list them here.

Next we will describe the techniques used to prove the termination of our main solver function `solve`. We will also prove its total correctness theorem.

### 5.1. Termination

In the code presented in this paper, only the functions `exhaustiveUnitPropagate`, `explainUIP`, and `solve_loop` are defined by using general recursion and it is not obvious if they are terminating. The only function that end-users of the

solver are expected to call directly is the function `solve` as it is the solver’s only entry-point. This means that all three functions defined by general recursion are called only indirectly by the function `solve` and all parameters that are passed to them are computed by the solver. Therefore, these functions can be regarded to be partial functions and it is not necessary to show that they terminate for all possible values of their input parameters. It suffices to show that they terminate for those values of their input parameters that could actually be passed to them during a solver’s execution starting from the initial state.

We use Isabelle’s built-in features to model this kind of partiality [Kra08].

1. Notice that all three functions are defined by using the tail recursion and annotated by the directive `tailrec`. This is a very important feature, because it enables the code generator to work with these partial functions.
2. When an n-ary function `f` is defined by using a general recursion, a predicate `f_dom` which tests if an n-tuple  $(a_1, \dots, a_n)$  is in the domain of `f` (i.e., if `f` terminates on input  $(a_1, \dots, a_n)$ ) is automatically generated. If the function definition is annotated by the directive `domintros`, Isabelle generates a theorem of the form

$$g \longrightarrow (\text{f\_dom } (f_1(a_1), \dots, f_n(a_n))) \longrightarrow (\text{f\_dom } (a_1, \dots, a_n)),$$

for each recursive call  $f(f_1(a_1), \dots, f_n(a_n))$  in the definition of `f`, where `g` is a guard for this recursive call. Until the termination of `f` is proved i.e., until `f` is proved to be total, the usual induction scheme theorem for the function `f` (which would be called `f.induct`) cannot be proved and used. However, when `f` is defined a weaker, partial induction scheme theorem (called `f.pinduct`) is automatically proved. It differs from the usual induction scheme only because it adds the domain predicate `f_dom` both to the induction base and to the induction steps. These domain predicates are then carried over and assumed in all lemmas about the function `f` which are proved by (the partial) induction. Still, in order to complete the whole correctness proof, at one point they have to be discharged. This is done by proving that all inputs passed to the function `f` imply the domain predicate.

In our case, we know that invariants are preserved throughout any solver’s run and that each state for which our solver functions are called satisfies all given invariants. We show that some of these invariants imply the domain predicates, i.e., that our three functions defined by general recursion terminate for states in which these invariants hold.

As an illustration, we will outline the proof that the function `exhaustive-UnitPropagate` (p20) terminates if its input satisfies certain invariants.

In order to prove this, we introduce a well-founded ordering of trails such that applications of `applyUnitPropagate` advances, i.e., decreases the trail, in that ordering. So, let us first define an ordering  $\prec^{lit}$  of marked literals (it is trivially well-founded).

**Definition 23.**  $l_1 \prec^{lit} l_2 \iff (\text{isDecision } l_1) \wedge \neg(\text{isDecision } l_2)$

Now we can introduce an ordering of trails, which will be used as a basis for the ordering that we are constructing.

**Definition 24.**

$$M_1 \succ_M M_2 \iff M_1 \prec_{lex}^{lit} M_2,$$

where  $\prec_{lex}^{lit}$  is a lexicographic extension of relation  $\prec^{lit}$ .

The function `applyUnitPropagate` advances in this ordering (trivially, by the definition of lexicographic extension), but, unfortunately, this ordering need not be well-founded. However, since invariants hold in every state during the solver's operation, we can make a restriction of  $\succ_M$  that is also well-founded.

**Definition 25.**

$$M_1 \succ_M^r M_2 \iff \begin{aligned} & (\text{consistent } M_1) \wedge (\text{unique } M_1) \wedge (\text{vars } M_1) \subseteq Vbl \\ & (\text{consistent } M_2) \wedge (\text{unique } M_2) \wedge (\text{vars } M_2) \subseteq Vbl \\ & M_1 \succ_M M_2 \end{aligned}$$

This is the ordering we were looking for and now we can prove a lemma saying that if the state satisfies certain invariants, then it is in the domain of the `applyUnitPropagate` function (i.e., that this function terminates when applied to that state).

**Lemma 1.** *If the set `decisionVars` is finite and the state `state` is such that:*

- (a) *`InvariantMconsistent` (p11) and `InvariantMdistinct` (p11) hold in state,*
- (b) *`InvariantMvars` (p12), `InvariantFvars` (p12), and `InvariantQvars` (p16) hold in state,*
- (c) *`InvariantconflictFlagCharacterization` (p15), `InvariantQCharacterization` (p16), and `InvariantQdistinct` (p16) hold in state,*
- (d) *`InvariantwatchListsCharacterization` (p17) and `InvariantwatchListsDistinct` (p17) hold in state,*
- (e) *`InvariantwatchesEl` (p17), `InvariantwatchesDiffer` (p17) and `InvariantwatchCharacterization` (p18) hold in state,*

*then the function `exhaustiveUnitPropagate` terminates when applied to state i.e., (`exhaustiveUnitPropagate_dom state`).*

*Proof:* If `Q` is empty or `conflictFlag` is raised in the state `state`, then the function `exhaustiveUnitPropagate` terminates and `state` is trivially in its domain. So, let us assume that `Q` is not empty and `conflictFlag` is false.

The proof is carried by well-founded induction on the ordering  $\succ_M^r$ . Assume, as an inductive hypothesis, that the statement holds for all states `state'` for which `state`  $\succ_M^r$  `state'`. Let `state' = (applyUnitPropagate state)`. Since invariants hold in `state` and are preserved by the `applyUnitPropagate` function, they hold in `state'` as well<sup>5</sup>. Since the trail `M` in `state'` is extended by a single literal, it holds that the `state`  $\succ_M^r$  `state'`. So, by inductive hypothesis, it holds that (`exhaustiveUnitPropagate_dom state'`). The lemma then follows from the domain introduction theorem `exhaustiveUnitPropagate_domintros`:

$$\begin{aligned} \neg \text{conflictFlag}_{state} \wedge Q_{state} \neq [] & \longrightarrow \\ (\text{exhaustiveUnitPropagate\_dom } (\text{applyUnitPropagate } state)) & \longrightarrow \\ (\text{exhaustiveUnitPropagate\_dom } state). & \end{aligned}$$

---

<sup>5</sup>Note that only `InvariantconsistentM`, `InvariantdistinctM`, and `InvariantvarsM` need to hold in order to use the ordering  $\succ_M^r$ . However, we had to assume many additional invariants in the premises of this lemma, because they are needed to show that these three key invariants are preserved when `applyUnitPropagate` is applied.

□

Termination (on relevant inputs) of the `applyExplainUIP` and `solve_loop` functions is proved in a similar way. The termination proof for `solve_loop` function uses the same ordering  $\succ_M^r$  and the termination proof for `applyExplainUIP` uses the following well-founded ordering of clauses  $\succ_C^M$  parametrized by the trail  $M$ .

**Definition 26.**

$$C_1 \succ_C^M C_2 \iff \{\text{remdups } \overline{C_2}\} \prec_{mult}^{\widehat{M}} \{\text{remdups } \overline{C_1}\},$$

where  $\{\dots\}$  denotes the multiset of list elements and  $\prec_{mult}^{\widehat{M}}$  is the multiset extension of the order  $\prec^{\widehat{M}}$  induced by the list  $\widehat{M}$ .

### 5.2. Total Correctness

Total correctness of the `solve` function is given by the following theorem.

**Theorem 1.**

$$((\text{solve } F_0) = \text{TRUE} \wedge (\text{sat } F_0)) \vee ((\text{solve } F_0) = \text{FALSE} \wedge \neg(\text{sat } F_0))$$

Assuming that all invariants hold in each state reached during the `solve` function execution, the proof of Theorem 1 relies on the following two soundness lemmas, which correspond to the two places in the solver code where `SATFlag` is changed.

**Lemma 2.** *If in some state state it holds that:*

- (a) *Invariant<sub>equivalent</sub> (p12) holds in state,*
- (b) *Invariant<sub>conflictFlagCharacterization</sub> (p15) holds in state,*
- (c) *conflictFlag is true in state,*
- (d) *(currentLevel  $M$ ) = 0 in state,*

*then it holds that  $\neg(\text{sat } F_0)$ .*

*Proof:* From  $(\text{currentLevel } M) = 0$  it follows that  $(\text{prefixToLevel } 0 \ M) = M$ . Hence, from *Invariant<sub>equivalent</sub>* it follows that  $F @ \langle \widehat{M} \rangle \equiv F_0$ . Since from *conflictFlag* and *Invariant<sub>conflictFlagCharacterization</sub>* it holds that  $M \models \neg F$ , by monotonicity it also holds that  $M \models \neg F @ \langle \widehat{M} \rangle$ . Since  $F @ \langle \widehat{M} \rangle \equiv F_0$ , the formula  $F @ \langle \widehat{M} \rangle$  is false in a valuation that it entails, so is unsatisfiable. Since  $F_0$  is logically equivalent to  $F @ \langle \widehat{M} \rangle$ , it is also unsatisfiable. □

**Lemma 3.** *If in some state state it holds that:*

- (a) *(vars  $F_0$ )  $\subseteq$  decisionVars,*
- (b) *Invariant<sub>Mconsistent</sub> (p11) holds in state,*
- (c) *Invariant<sub>Fvars</sub> (p12) holds in state,*
- (d) *Invariant<sub>equivalent</sub> (p12) holds in state,*
- (e) *Invariant<sub>conflictFlagCharacterization</sub> (p15) holds in state,*
- (f) *conflictFlag is false in state,*
- (g) *(vars  $\widehat{M}$ )  $\supseteq$  decisionVars in state,*

*then (sat  $F_0$ ) and (model  $\widehat{M}$   $F_0$ ) hold.*

*Proof:* From  $Invariant_{Fvars}$ , it follows that  $(vars F) \subseteq (vars F_0) \cup decisionVars$ . With  $(vars F_0) \subseteq decisionVars$ , it holds that  $(vars F) \subseteq decisionVars$ . With  $(vars \widehat{M}) \supseteq decisionVars$ , it holds that  $(vars F) \subseteq (vars \widehat{M})$  and  $\widehat{M}$  is a total valuation wrt. the variables of  $F$ . Therefore, it is either the case that  $\widehat{M} \models \neg F$  or  $\widehat{M} \models F$ . Since  $conflictFlag$  is false, by  $Invariant_{conflictFlagCharacterization}$  it holds that  $\widehat{M} \not\models \neg F$ , so it must be the case that  $\widehat{M} \models F$ . It trivially holds that  $\widehat{M} \models \langle prefixToLevel 0 M \rangle$  and  $\widehat{M}$  is consistent by  $Invariant_{Mconsistent}$ . Therefore  $\widehat{M}$  is a model for  $F @ \langle prefixToLevel 0 M \rangle$ . Since  $F_0 \equiv F @ \langle prefixToLevel 0 M \rangle$ , it holds that  $\widehat{M}$  is also a model for  $F_0$  and  $(sat F_0)$  holds.  $\square$

## 6. Discussion on Proof Management

Although it is hard to quantify the efforts invested in this work, we can estimate it to be around one man-year. The proof scripts are around 30000 lines of Isabelle code and the generated PDF proof documents are around 700 pages long. These numbers are of course heavily dependent on the indentation style used. Proof-checking time by Isabelle is under 5 minutes on a 1.6GHz/512Mb RAM machine running Linux. We estimate that careful investigation of the proof text and its reorganization mainly by extracting some common parts of different proofs into lemmas could lead to 10-20 percent reductions.

During this verification effort some interesting technical issues arose. In order to make such a large-scale verification effort possible, it was necessary to introduce some kind of modularity to the formalization. The crucial step in this direction was to prove the properties of abstract rule-based DPLL systems of [NOT06, KG07] and then use these proofs in the correctness proof of low-level implementation presented here. A good direction to follow would be to define internal data-structures (for example the assertion trail) as abstract data-types (ADT) with some desired properties given axiomatically. Although, unfortunately, this has not been explicitly done in our formalization, this idea has been followed to some extent. Namely, after introducing basic definitions, we showed lemmas that could be regarded as axioms of the ADT and all further proofs relied only on those lemmas, without using the low-level properties of the implementation. This, of course, enables changing the low-level implementation into a more efficient one without changing much of the whole correctness proof. We think that explicit encoding the ADT approach (for example by using type-classes or locales [NPW02]) would lead to even more flexible formalization and is a step in a right direction.

When proving properties about recursively defined functions we had a dilemma whether to repeat the same induction scheme in proofs of many similar lemmas (one for each property of the recursive function) or to formulate one bigger lemma that groups all assumptions and conclusions for several properties that are being shown. We took the second approach and reduced the total number of lemmas and the total size of proofs, but the price that had to be paid is that we lost track of which assumptions are effectively used for proving a specific conclusion. For example, most of our high-level lemmas that show that invariants are preserved by the function calls assume that all invariants hold before the function call and show that all invariants hold after the function call. The only way to find out which invariants are necessary to hold before the

function call so that a specific invariant holds after it is reading the proof texts which can be very tedious and unpleasant.

## 7. Related work

First steps towards verification of SAT and SMT solvers have been recently made. Shankar has formally and mechanically proved soundness, completeness, and decidability of propositional logic (by means of a satisfiability solver) [Sha85]. Zhang and Malik have informally proved correctness of a modern SAT solver [ZM03]. Barret, in his PhD thesis has formally (but not mechanically) proved correctness of the Stanford framework for cooperating decision procedures [Bar03]. Rule based descriptions of [NOT06, KG07] formally describe high-level operation of modern SAT solvers and authors have informally proved their correctness. Shankar and Vaucher have formally and mechanically verified a higher level description of a modern DPLL procedure within the system PVS. Although these descriptions include most state-of-the-art SAT algorithms, lower-level implementation techniques (e.g., two-watch unit propagation scheme) are not covered by any of these descriptions. In [Mar08a], the author gives a tutorial exposure of the modern SAT solving techniques (both high and low level) with correctness properties formulated in a Hoare-style framework and proved (to some extent) mechanically within the system Isabelle. The present paper is a sequel of work described in [Mar08a], but without the tutorial aspect. More importantly, it covers program termination and provides fully mechanized correctness proofs of both higher and lower level SAT techniques. As we know, this is the first fully mechanized formalization of several low-level techniques, most notably the two-watched literal propagation scheme.

## 8. Further Work

The specification of the SAT solver given in this paper is such that a fully executable code in a functional language can be automatically generated from it, providing that an executable decision heuristic is supplied. However, the efficiency of the generated code must still be improved, if we want to get a competitive solver.

First, there are several low-level algorithmic improvements that have to be made. For example, in the current implementation, checking if a literal is true in a trail  $M$  requires performing a linear-time scan through the list. Most real-world solvers cache truth values of all literals in an array and so allow a constant time check. Also, the conflict analysis phase is expressed here in a bit more abstract way than in implementations of MiniSat style solvers.

Next, some higher-level heuristics have to be implemented more carefully. For example, we have only made tests with a trivial decision heuristics that selects a random undefined literal, but in order to have a usable solver, a more involved decision heuristics (e.g., the MiniSat one) should be used. It would also be useful to implement forgetting and restarting techniques [KG07, NOT06].

Although these modifications require to invest more work, we believe that they are straightforward. However, the most problematic issue is the fact that because of the pure functional nature of HOL no side-effects are possible and there can be no *destructive updates* of data-structures. We will now explain

one possible way to overcome this problem. In order to avoid explicit passing of the solver state throughout function calls, the Haskell-like monadic style of programming can be used. In this way, all functions are regarded to be `StateTransformers` of some type, i.e., their results are *actions* that transform `State` along with returning a result of some given type.

```
types 'a StateTransformer = "State ⇒ ('a × State)"
```

Basic state transformers for accessing and modifying each component of the state can be introduced. For example:

```
definition readWatch1 :: "nat ⇒ Literal option StateTransformer"
where
"readWatch1 clause = (λ state. (getWatch1 state clause, state))"
```

```
definition updateWatch1 :: "nat ⇒ Literal option ⇒ unit StateTransformer"
where
"updateWatch1 clause literal =
  (λ state. ((), state(| getWatch1 := (getWatch1 state)(clause := literal) |)))"
```

Standard monadic bind and return operators can be introduced and they can be used to combine `StateTransformers`. Isabelle allows to introduce a convenient Haskell-like `do` syntax. In this way, the function `setWatch1` (p17) becomes:

```
definition setWatch1 :: "nat ⇒ Literal ⇒ unit StateTransformer"
where
"setWatch1 clause literal =
  do
    updateWatch1 clause (Some literal);
    addToWatchList literal clause
  done
"
```

which is much easier to read and resembles imperative style of programming.

It can be easily shown that these new definitions, written in monadic style, are in their essence equivalent to definitions given throughout this paper. However, apart from the better readability, there is one huge difference — it is possible to adapt the code generator to generate monadic Haskell and imperative ML code which would lead to huge efficiency benefits since it allows mutable references and arrays. We hope that with these modifications, the generated code could become comparable to real-world SAT solvers and this would be the main direction of our further work.

## 9. Conclusions

In this paper, we have presented a formalization and a total correctness proof of a MiniSAT-like SAT solver within the system Isabelle/HOL. The solver is based on the DPLL procedure and employs most state-of-the-art SAT solving techniques including the conflict-guided backjumping, clause learning and the two-watch unit propagation scheme. The described solver specification can serve as a basis for implementation of an efficient and correct SAT solver. One possible approach for that would be to manually implement a SAT solver (in an imperative programming language) by strictly following the descriptions of the solver given in this paper. However, the highest possible level of trust could be

achieved only if fully executable code (in a functional programming language) is automatically generated by using the Isabelle’s built-in code generator. Although this can already be done, the efficiency of generated code should further be improved and that is the field of our future research. We hope that this work can facilitate better understanding of modern SAT solvers. The final product of this research will be a trusted and efficient SAT solver that can be used either independently or as a kernel for checking results of other untrusted verifiers. We also hope that this work will show that it is possible to have a fully verified implementation of a very non-trivial software system.

*Acknowledgements.* The author wishes to thank Dr. Predrag Janičić for helpful discussions and valuable assistance in writing this paper.

## References

- [Bar03] C. Barrett. Checking validity of quantifier-free formulas in combinations of first-order theories. *Ph.D. thesis, Stanford University*, 2003.
- [BKH<sup>+</sup>08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkok, and J. Matthews. Imperative functional programming with Isabelle/HOL, In *TPHOLs ’08*, Montreal, 2008.
- [Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *STOC ’71*, pp. 151–158, New York, 1971.
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [ES04] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT ’03*, pp. 502–518, Santa Margherita Ligure, 2003.
- [Gel07] A. Van Gelder. Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In *SAT ’07*, pp. 328–333, Lisbon, 2007.
- [GKSS07] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*. Elsevier, 2007.
- [GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, In *DATE ’02*, Paris, 2002.
- [Haf08] F. Haftmann. Code generation from Isabelle/HOL theories. <http://isabelle.in.tum.de/documentation.html>, 2008.
- [KG07] S. Krstic and A. Goel. Architecting solvers for sat modulo theories: Nelson-oppen with dpll. In *FroCos*, pp. 1–27, Liverpool, 2007.
- [Kra08] A. Krauss. Defining recursive functions in Isabelle/HOL. <http://isabelle.in.tum.de/documentation.html>, 2008.
- [Mar08a] F. Marić. Formalization and implementation of modern sat solvers. Manuscript submitted for publication. 2008.



- [Mar08b] F. Marić. Sat solver verification. *The Archive of Formal Proofs*, 2008.
- [MMZ<sup>+</sup>01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC '01*, Las Vegas, 2001.
- [MSS99] J. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, 1999.
- [NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
- [NPW02] T. Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [Sha85] N. Shankar. Towards mechanical metamathematics. *J. Autom. Reason.*, 1(4):407–434, 1985.
- [SV08] N. Shankar and M. Vaucher. The mechanical verification of a DPLL-based satisfiability solver. Manuscript submitted for publication. 2008.
- [Zha97] H. Zhang. SATO: An efficient propositional prover. In *CADE-14*, pp. 272–275, London, 1997.
- [VSTTE] *Verified Software: Theories, Tools, Experiments*. Conference. <http://vstte.ethz.ch/>
- [ZM03] L. Zhang and S. Malik. Validating sat solvers using independent resolution-based checker. In *DATE '03*, p10880, Washington DC, 2003.