Noname manuscript No.
(will be inserted by the editor)

Formalization and Implementation of Modern SAT Solvers

Filip Marić

Received: date / Accepted: date

Abstract Most, if not all, state-of-the-art complete SAT solvers are complex variations of the DPLL procedure described in the early 1960's. Published descriptions of these modern algorithms and related data structures are given either as high-level (rule-based) transition systems or, informally, as (pseudo) programming language code. The former, although often accompanied with (informal) correctness proofs, are usually very abstract and do not specify many details crucial for efficient implementation. The latter usually do not involve any correctness argument and the given code is often hard to understand and modify. This paper aims at bridging this gap: we present SAT solving algorithms that are formally proved correct, but at the same time they contain information required for efficient implementation. We use a tutorial, top-down, approach and develop a SAT solver, starting from a simple design that is subsequently extended, step-by-step, with the requisite series of features. Heuristic parts of the solver are abstracted away, since they usually do not affect solver correctness (although they are very important for efficiency). All algorithms are given in pseudo-code. The code is accompanied with correctness conditions, given in Hoare logic style. Correctness proofs are formalized within the Isabelle theorem proving system and are available in the extended version of this paper. The given pseudo-code served as a basis for our SAT solver ARGO-SAT.

This work was partially supported by Serbian Ministry of Science grant 144030.

Filip Marić
Faculty of Mathematics
University of Belgrade
Serbia
E-mail: filip@matf.bg.ac.yu

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, SAT solvers are becoming a tool suitable for attacking more and more practical problems. Some of the solvers are complete, while others are stochastic. For a given SAT instance, complete SAT solvers can either find a solution (i.e., a satisfying variable assignment) or show that no solution exists. Stochastic solvers, on the other hand, cannot prove that an instance is unsatisfiable although they may be able to find a solution for certain kinds of large satisfiable instances quickly. The majority of the state-of-the-art complete SAT solvers are based on the branch and backtracking algorithm called Davis-Putnam-Logemann-Loveland, or DPLL [DP60, DLL62]. Starting with the work on the GRASP and SATO systems [MSS99, Zha97], and continuing with Chaff, Berk-Min and MiniSAT [MMZ$^+$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*, *conflict-driven lemma learning*, and *restarts*, and (ii) better implementation techniques, such as the *two-watch literals* scheme for unit propagation. These advances make it possible to decide the satisfiability of industrial SAT problems with tens of thousands of variables and millions of clauses.

While SAT solvers have become complex, describing their underlying algorithms and data structures has become a nontrivial task. Some papers describe conceptual, higher level concepts, while some papers describe system level architecture with smart implementation techniques and tricks. Unfortunately, there is still a large gap between these two approaches. Higher level presentations, although clean and accompanied with correctness proofs, omit many details that are vital to efficient solver implementation. Lower level presentations usually give SAT solver algorithms in a form of pseudo-code. The open source SAT solvers themselves are, in a sense, the most detailed presentations or specifications of SAT solving techniques. The success of MiniSAT [ES04], and the number of its re-implementations, indicate that detailed descriptions of SAT solvers are needed and welcome in the community. However, in order to achieve the highest possible level of efficiency, these descriptions are far from the abstract, algorithmic level. Often, one procedure in the code contains several higher level concepts or one higher level algorithm is spread across several code procedures. The resulting pseudo-code, although almost identical to the award winning solvers, is, in our opinion, hard to understand, modify, and reason about. This paper is an attempt to bring these two approaches together. We claim that SAT solvers can be implemented so that (i) the code follows higher level descriptions that make the solver easy to understand, maintain, modify, and to prove correct, and (ii) it contains lower level implementation tricks, and therefore achieves high efficiency. We support this claim by (i) our SAT solver ARGO-SAT, that represents a rational reconstruction of MiniSAT, obeying the given two requirements, and (ii) our correctness proofs (formalized in Isabelle) for the presented algorithms, accompanying our SAT solver.[1]

Complicated heuristics (e.g., for literal selection, for determining the appropriate clause database size, restart strategy) represent important parts of modern SAT solvers and are crucial for solver efficiency. While a great effort is put on developing new heuristics, and researchers compete to find more and more effective ones, we argue that those can be abstracted from the core part of the solver (the DPLL algorithm itself) and represented as just a few additional function calls (or separated into external classes in object-oriented setting). No matter how complicated these heuristics are, they do not affect the solver correctness as long as they meet a several, usually trivial, conditions. Separating the core DPLL algorithm from complicated, heuristic parts of the solver leads to simpler solver design, and to more reliable and flexible solvers.

In the rest of the paper, we develop the pseudo-code of a SAT solver from scratch and outline its correctness arguments along the way. We take a top-down approach, starting with the description of a very simple solver, and introduce advanced algorithms and data structures one by one. The partial correctness of the given code is proved using the proof system Isabelle (in the Hoare style). Isabelle proof documents containing the proofs of correctness conditions are available in [Mar08]. A longer version of this paper, available from `http://argo.matf.bg.ac.rs` contains these proofs presented in a less formal, but more readable manner. In this paper we do not deal with termination issues. Still, it can be shown that all presented algorithms are terminating[2].

Overview of the paper. In §2 we briefly describe the DPLL algorithm, present two rule-based SAT solver descriptions, and describe the basics of program verification and Hoare logic. In §3 we introduce the background theory in which we will formalize and prove the properties of a modern SAT solver, and describe the pseudo-code language used to describe the implementation. The bulk of the paper is in §4: it contains descriptions of SAT solver algorithms and data structures and outlines their correctness proofs. We start from the basic backtrack search (§4.1), then introduce unit propagation (§4.2), backjumping, clause learning and firstUIP conflict analysis (§4.3), conflict clause minimization (§4.4), clause forgetting (§4.5), restarts (§4.6), exploiting literals asserted at zero level of assertion trail (§4.7), and introduce efficient detection of conflict and unit clauses using watch literals (§4.8, §4.9). In §5 we give a short history of SAT solver development, and in §6 we draw final conclusions.

[1] Web page of ARGO-SAT is `http://argo.matf.bg.ac.rs/`

[2] Formal termination proofs of rule-based systems on which our implementation is based are available in [Mar08].

```
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($F[l \rightarrow \bot]$)
   end
end
```

Fig. 1 DPLL algorithm - recursive definition

2 Background

Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Most of the complete modern SAT solvers are based on the *DPLL algorithm* [DP60, DLL62]. Its recursive version is shown in the 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 $\overline{l}$ with $\bot$, and simplifying afterwards. A literal is pure if it occurs in the formula but its opposite does not. A clause is unit if it contains only one literal. This recursive implementation is practically unusable for larger formulae and therefore it will not be used in the rest of this paper.

Rule-based SAT solver descriptions. During the last few years, two transition rule systems which model the DPLL-based SAT solvers and related SMT solvers have been published [NOT06, KG07]. These descriptions define the top-level architecture of solvers as a mathematical object that can be grasped as a whole and fruitfully reasoned about. Both systems are accompanied with pen-and-paper correctness and termination proofs. Although they succinctly and accurately capture all major aspects of the solvers' global operation, they are high level and far from the actual implementations. Both systems model the solver behavior as transitions between states. States are determined by the values of solver's global variables. 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 more transition rules apply (i.e., when final states are reached).

The system given 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 an implementation based on the given set of rules.

Decide:

$$\frac{l \in F \qquad l, \bar{l} \notin M}{M := M\ l^d}$$

UnitPropag:

$$\frac{l \vee l_1 \vee \ldots \vee l_k \in F \qquad \bar{l}_1, \ldots, \bar{l}_k \in M \qquad l, \bar{l} \notin M}{M := M\ l}$$

Conflict:

$$\frac{C = no\_cflct \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \in M}{C := \{l_1, \ldots, l_k\}}$$

Explain:

$$\frac{l \in C \qquad l \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad l_1, \ldots, l_k \prec l}{C := C \cup \{l_1, \ldots, l_k\} \setminus \{l\}}$$

Learn:

$$\frac{C = \{l_1, \ldots, l_k\} \qquad \bar{l}_1 \vee \ldots \vee \bar{l}_k \notin F}{F := F \cup \{\bar{l}_1 \vee \ldots \vee \bar{l}_k\}}$$

Backjump:

$$\frac{C = \{l, l_1, \ldots, l_k\} \qquad \bar{l} \vee \bar{l}_1 \vee \ldots \vee \bar{l}_k \in F \qquad \text{level } l > m \geq \text{level } l_i}{C := no\_cflct \qquad M := M^{[m]}\ \bar{l}}$$

Forget:

$$\frac{C = no\_cflct \qquad c \in F \qquad F \setminus c \vDash c}{F := F \setminus c}$$

Restart:

$$\frac{C = no\_cflct}{M := M^{[0]}}$$

Fig. 2 Rules of DPLL as given in [KG07]. ($l_i \prec l_j$ denotes that the literal l_i precedes l_j in M, and $M^{[m]}$ denotes the prefix of M up to the level m)

The system given in [KG07] gives a more detailed description of some parts of the solving process (particularly the conflict analysis phase) than the one given in [NOT06]. Since the system of [KG07] 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 a set of literals, or the distinguished symbol no_cflct. The input to the system is an arbitrary set of clauses F_0. Solving starts from the initial state in which $F = F_0$, $M = [\,]$, and $C = no\_cflct$. The rules have guarded assignment form: above the line is the condition that enables the rule application, below the line is the update to the state variables.

Formal proofs. Over the last years, in all areas of mathematics and computer science, machine checkable formal proofs have gained more and more importance. There are growing efforts in this direction, with many extremely complex theorems formally proved and with many software tools for producing and checking formal proofs. Some of them are Isabelle, HOL, Coq, PVS, Mizar, etc. A comparison of these tools can be found in [Wie03].

Program verification. Program verification is the process of formally proving that a computer program meets its specification (that formally describes the expected program behavior). Following the lessons from major software failures, an increasing amount of effort is being invested in this field. Many fundamental algorithms and properties of data structures have been formalized and verified. Also, a lot of work has been devoted to formalization of compilers, program semantics, communication protocols, security protocols, etc. Formal verification is important for SAT and SMT solvers and the first steps towards this direction have been made [KG07,NOT06,Bar03].

Hoare logic. Verification of imperative programs is usually done in Floyd-Hoare logic [Hoa69], a formal system that provides a set of logical rules in order to reason about the correctness of computer programs with the rigor of mathematical logic. The central object in Hoare logic is Hoare triple which describes how the execution of a piece of code changes the state of a computation. A Hoare triple is of the form $\{P\}$ `code` $\{Q\}$, where P (the *precondition*) and Q (the *postcondition*) are formulae of a meta-logic and `code` is a programming language code. Hoare triple should be read as: "Given that the assertion P holds at the point before the `code` is executed, and the `code` execution terminates, the assertion Q will hold at the point after the `code` was executed".

3 Notation and Definitions

In this section we introduce the notation and definitions that will be used in the rest of the paper.

3.1 Background Theory

In order to reason about the correctness of SAT solver implementations, we have to formally define the notions we are reasoning about. This formalization will be made in higher-order logic of the system Isabelle. Formulae and logical connectives of this logic ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) are written in the usual way. The symbol $=$ denotes syntactical identity of two expressions. Function and predicate applications are written in prefix form, as in ($\mathsf{f}\ x_1\ \ldots\ x_n$). Existential quantifier is written as $\exists$ and universal quantifier is written as $\forall$.

We assume that the background theory we are defining includes the built-in theory of lists and (finite) sets. Syntax of these operations is summarized in the first column of the Figure 3, and the semantics is informally described in the second column.

Basic types. The logic used is typed, and we define the basic types.

Definition 1

`Boolean`	`true` or `false`
`Nat`	natural number
`Variable`	natural number.
`Literal`	either a positive variable (+*vbl*) or a negative variable (−*vbl*)
`Clause`	a list of literals
`Formula`	a list of clauses
`Valuation`	a list of literals

(a, b)	the ordered pair of elements a and b.
$[]$	the empty list.
$[e_1, \ldots, e_n]$	the list of n given elements $e_1, \ldots, e_n$.
$e \# list$	the list obtained by prepending element e to the list $list$.
$list_1 @ list_2$	the list obtained by appending lists $list_1$ and $list_2$.
$e \in list$	e is a member of the list $list$.
$list_1 \subseteq list_2$	all elements of $list_1$ are also elements of $list_2$.
$list \setminus e$	the list obtained by removing all occurrences of the element e from the list $list$.
$list_1 \setminus list_2$	the list obtained from the list $list_1$ by removing all elements of the list $list_2$ from it.
$(\mathsf{first}\ list)$	the first element of the list $list$. (assuming the $list$ is not empty).
$(\mathsf{second}\ list)$	the second element of the list $list$ (assuming the $list$ has at least two elements).
$(\mathsf{head}\ list)$	a synonym for $(\mathsf{first}\ list)$.
$(\mathsf{tail}\ list)$	the list obtained by removing the first element of the list $list$.
$(\mathsf{last}\ list)$	the last element in the nonempty list $list$.
$(\mathsf{length}\ list)$	the length of the list $list$.
$(\mathsf{unique}\ list)$	true iff the list $list$ contains no repeating elements.
$\{\}$	the empty set.
$e \in set$	element e is a member of the set set.
$set_1 \cup set_2$	the set union of set_1 and set_2
$\|set\|$	the number of elements in the set set.
$\{a_1 \mapsto b_1, \ldots, a_k \mapsto b_k\}$	the mapping of elements $\{a_1, \ldots, a_k\}$ to elements $\{b_1, \ldots, b_k\}$
$H(a_i)$	the image of the element a_i in the mapping H (provided that it has been defined).

Fig. 3 List and set operations

Although we use typed logic, for the sake of readability we sometimes omit sorts and use the following naming convention: Literals (i.e., variables of the type `Literal`) are denoted by l (e.g. $l, l', l_0, l_1, l_2, \ldots$), variables by vbl, clauses by c, formulae by F, and valuations by v.

Although most of the following definitions are formalized using the primitive recursion, in order to simplify the presentation and improve readability we give them as natural language descriptions.

Definition 2 *The opposite literal of a literal* l, denoted $\overline{l}$, is defined by: $\overline{+vbl} = -vbl$, $\overline{-vbl} = +vbl$.

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, formula, or valuation.

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

The semantics (satisfaction and falsification relations) is defined by:

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

We will write $v \nvDash l$ to denote that l is not true in v, $v \nvDash c$ to denote that c is not true in v, and $v \nvDash F$ to denote that F is not true in v.

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

We will write $v \nvDash\!\neg\, l$ to denote that l is not false in v, $v \nvDash\!\neg\, c$ to denote that c is not false in v, and $v \nvDash\!\neg\, F$ 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 ($\mathsf{inconsistent}\ v$), iff it contains both literal and its opposite i.e., $\exists l.\ v \vDash l \wedge v \vDash \overline{l}$. A valuation is *consistent*, denoted ($\mathsf{consistent}\ v$), 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 ($\mathsf{sat}\ F$) iff it has a model i.e., $\exists v.\ (\mathsf{consistent}\ v) \wedge v \vDash F$

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

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

Definition 11 A clause c is *unit* in a valuation v with a *unit literal* l, denoted ($\mathsf{isUnit}\ c\ l\ v$) iff $l \in c$, $v \nvDash l$, $v \nvDash\!\neg\, l$ and $v \vDash\!\neg\, (c \setminus l)$ (i.e., $\forall l'.\ l' \in c \wedge l' \neq l \Rightarrow v \vDash\!\neg\, l'$).

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

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

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 background theory with the following type:

Definition 14

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

A trail can be implemented, for example, as a list of `(Literal, Boolean)` ordered pairs. We will denote trails by M (e.g. $M, M', M_0, \ldots$).

Example 1 A trail M could be $[+1, |-2, +6, |+5, -3, +4, |-7]$. The decision literals are marked with the symbol | on their left hand sides.

Definition 15 $(\mathsf{decisions}\ M)$ is the list of all marked elements (i.e., of all decision literals) from a trail M.

Definition 16 $(\mathsf{lastDecision}\ M)$ is the last element in a trail M that is marked.

Definition 17 $(\mathsf{decisionsTo}\ M\ l)$ 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.

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

Definition 18 The *current level* for a trail M, denoted $(\mathsf{currentLevel}\ M)$, is the number of marked literals in M, i.e., $(\mathsf{currentLevel}\ M) = (\mathsf{length}\ (\mathsf{decisions}\ M))$.

Definition 19 The *decision level* of a literal l in a trail M, denoted $(\mathsf{level}\ l\ M)$, is the number of marked literals in the trail that precede the first occurrence of l, including l if it is marked, i.e., $(\mathsf{level}\ l\ M) = (\mathsf{length}\ (\mathsf{decisionsTo}\ M\ l))$.

Definition 20 $(\mathsf{prefixToLevel}\ M\ \mathit{level})$ is the prefix of a trail M containing all elements of M with levels less or equal to level.

Definition 21 $(\mathsf{prefixBeforeLastDecision}\ M)$ is the prefix of a trail M up to the last element that is marked, not including that element.[3]

Example 3 For the trail in Example 1, $(\mathsf{level}\ \ +1\ M) = 0$, $(\mathsf{level}\ \ +4\ M) = 2$, $(\mathsf{level}\ \ -7\ M) = 3$, $(\mathsf{currentLevel}\ M) = 3$, $(\mathsf{prefixToLevel}\ M\ 1) = [+1, |+2, +6]$, $(\mathsf{prefixBeforeLastDecision}\ M) = [+1, |-2, +6, |+5, -3, +4]$.

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

[3] Note that some of the defined functions are partial, and are not defined for all trails. For example, $(\mathsf{prefixBeforeLastDecision}\ M)$ is defined only for trails that contain a decision literal.

Definition 23 The *maximal decision level for a clause* c in a trail M, denoted ($\mathsf{maxLevel}\ c\ M$), is the maximum of all decision levels of all literals from c that belong to M, i.e., ($\mathsf{maxLevel}\ c\ M$) = (level ($\mathsf{lastAssertedLiteral}\ c\ M$) M).

Example 4 Let c is $[+4, +6, -3]$, and M is the trail from the example 1. Then, ($\mathsf{maxLevel}\ c\ M$) $= 2$, and ($\mathsf{lastAssertedLiteral}\ c\ M$) $= +4$.

3.2 Pseudo-code Language

All algorithms in the following text will be specified in a PASCAL-like pseudo-code language. An algorithm specification consists of a program state declaration, followed by a series of function definitions.

The program state is specified by a set of global variables. The state is given in the following form, where $\mathtt{Type}_i$ can be any type of the background theory.

```
var
    var_1 : Type_1
    ...
    var_k : Type_k
```

A *block* is a sequence of *statements* separated by the symbol `;`, where a *statement* is one of the following:

```
begin block end
x := expression
if condition then statement
if condition then statement else statement
while condition do statement
repeat statement until condition
function_name(arg_1 , ..., arg_n)
return
```

Conditions and *expressions* can include variables, functions calls, and even background theory expressions.

Function definitions are given in the following form, where $\mathtt{Type}_i$ is the type of the argument $\mathtt{arg}_i$, and `Type` is the return type of the function.

```
function name (arg_1 : Type_1 , ..., arg_k : Type_k) : Type
begin
    block
end
```

If a function does not return a value, then its return type is omitted. A function returns a value by assigning it to a special variable `ret`. An explicit `return` statement is supported. Parameters are passed by value. If a parameter in the parameter list is marked with the keyword `var`, then it is passed by reference. Functions marked as `const` do not change the program state. In order to save some space, local variable declarations will be omitted when their type is clear from the context.

We allow the use of meta-logic expressions within our algorithm specifications. The reason for this is twofold. First, we want to simplify the presentation by avoiding the need for explicit implementation of some trivial concepts. For example, we assume that lists are supported in the language and we directly use the meta-logic notation for the list operations, having in mind that their correct implementation can be easily provided. Second, during the algorithm development, we intentionally leave some conditions at a high level and postpone their implementation. In such cases, we use meta-logic expressions in algorithm descriptions to implicitly specify the intended (post)conditions for the unimplemented parts of the code. For example, we can write `if` $M \vDash F$ `then`, without specifying how the test for $M \vDash F$ should effectively be implemented. These unimplemented parts of the code will be highlighted. Alternatively, a `function satisfies(M : Trail, F : Formula) : Boolean` could be introduced, which would have the postcondition $\{ret \iff M \vDash F\}$. The above test would then be replaced by `if satisfies(M, F) then`.

4 SAT Solver Algorithms and Data Structures

In this section, we present SAT solver algorithms and data structures, and outline their correctness proofs. Our implementation will follow the rules of the framework described in [KG07]. We give several variants of SAT solver, labeled as *SAT solver v.n*. To save space, instead of giving the full code for each SAT solver variant, we will only print changes with respect to the previous version of code. These changes will be highlighted. Lines that are added will be marked by + on the right hand side, and lines that are changed will be marked by * on the right hand side. Each solver variant contains a function:

```
function solve (F_0 : Formula) : {SAT, UNSAT}
```

which determines if the given formula F_0 is satisfiable or unsatisfiable. This function sets the global variable *satFlag* and returns its value.

The partial correctness of the SAT solver *v.n* is formalized by the following soundness theorem:

Theorem 1 *SAT solver v.n satisfies the Hoare triple:*
$\{\top\}$ solve(F_0) $\{(satFlag = UNSAT \wedge \neg(\mathsf{sat}\ F_0)) \ \vee\ (satFlag = SAT \wedge M \vDash F_0)\}$

This theorem will usually be proved by proving two lemmas:

1. Soundness for satisfiable formulae states that if the solver returns the value SAT, then the formula F_0 is satisfiable.
2. Soundness for unsatisfiable formulae states that if the solver returns the value $UNSAT$, then the formula F_0 is unsatisfiable.

Notice that, under the assumption that solver is terminating (which is not proved in this paper), these two soundness lemmas imply solver completeness.

To prove soundness, a set of conditions that each solver variant satisfies will be formulated. Also, preconditions that have to be satisfied before each function call will be given. All proofs that these conditions are invariants of the code and that preconditions are met before each function call are available in the longer version of this paper.

4.1 Basic Backtrack Search

The simplest, but still sound and complete, SAT solver can be implemented using the truth-table method which enumerates and checks all valuations. In this section we give a solver that is based on iterative backtrack search. The only improvement over the basic truth-table satisfiability checking is the use of early search-tree pruning. This code can be also seen as a non-recursive implementation of a simplified DPLL algorithm (pure literal and unit clause rules are omitted). All successive solver refinements are based on this simple code, so we will present it and prove its correctness for methodological reasons.

First we give an informal description of the solver. Formula F_0 is tested for satisfiability. The trail M represents the current partial valuation. Initially it is empty. Literals are added to the trail and marked as decision literals. Whenever a literal is added to the trail, F_0 is checked for inconsistency. When F_0 is inconsistent with the current trail, we say that a *conflict* occurred. The clause $c \in F_0$, such that $M \vDash \neg\, c$ is called a *conflict clause*. When a conflict occurs, the last decision literal on the trail is flipped, i.e., it and all literals after it are backtracked, and its opposite is added to the trail, but this time as a non-decision literal. If a conflict occurs with no decision literals on the trail, then we say that a *conflict at decision level zero* occurred and F_0 is determined to be unsatisfiable. If M contains all variables that occur in F_0 and conflict does not occur, then F_0 is determined to be satisfiable and M is its model.

SAT solver v.1

```
var                                                              +
   $satFlag$ : {UNDEF, SAT, UNSAT}                               +
   $F_0$ : Formula                                               +
   $M$ : Trail                                                   +

function solve ($F_0$ : Formula) : {SAT, UNSAT}                  +
begin                                                            +
   $satFlag$ = UNDEF;                                            +
   $M$ := [];                                                    +
   while $satFlag$ = UNDEF do begin                              +
      if $M \vDash \neg\, F_0$ then                              +
         if (decisions $M$) = [] then                            +
            $satFlag$ := UNSAT                                   +
         else                                                    +
            applyBacktrack()                                     +
      else                                                       +
         if (vars $M$) = (vars $F_0$) then                       +
            $satFlag$ := SAT                                     +
         else                                                    +
            applyDecide()                                        +
   end                                                           +
end                                                              +

function applyDecide()                                           +
begin                                                            +
   l := selectLiteral();                                         +
   assertLiteral(l, true)                                        +
end                                                              +
function applyBacktrack()                                               +
begin                                                                   +
    l := (lastDecision M);                                              +
    $M$ := (prefixBeforeLastDecision $M$);                              +
    assertLiteral($\overline{l}$, false)                                +
end                                                                     +
```

```
function assertLiteral(l : Literal, decision : Boolean)                 +
begin                                                                   +
    $M$ := $M$ @ [(l, decision)]                                        +
end                                                                     +
```

```
$\{$(vars $M) \neq$ (vars $F_0)\}$
const function selectLiteral() : Literal                                +
$\{$(var $ret) \in$ (vars $F_0) \wedge$ (var $ret) \notin$ (vars $M)\}$
```

Almost all meta-logic expressions used in the given pseudo-code can be easily made effective and implemented in a real programming language. For instance, the test (decisions M) = [ ] can be changed to the equivalent test (currentLevel M) = 0. The exception is the test for $M \vDash\neg F_0$ because it is not trivial to make its efficient implementation. We postpone addressing this issue until §4.8.

We leave the function selectLiteral undefined, and only require it to satisfy the Hoare triple specification:

$$\{(\mathsf{vars}\ M) \neq (\mathsf{vars}\ F_0)\}\ \ \mathsf{selectLiteral}()\ \ \{(\mathsf{var}\ ret) \in (\mathsf{vars}\ F_0) \wedge (\mathsf{var}\ ret) \notin (\mathsf{vars}\ M)\}$$

The selection of decision literals is irrelevant for the solver correctness, as long as this condition is met. On the other hand, the literal selection strategy is usually crucial for the solver efficiency. Many different strategies have been developed [ZM02, MS99] and all of them meet the given condition. Since these strategies can be implemented relatively independently from the rest of the solver, we will not further investigate this issue.

Example 5 Let $F_0 = [[-1, 2], [-3, 4], [-1, -3, 5], [-2, -4, -5], [-2, 3, 5, -6], [-1, 3, -5, -6], [1, -6], [1, 7]]$. Figure 4 lists one possible execution trace (where by *trace* we mean the list of steps applied accompanied with the program states). Since we do not provide a concrete literal selection strategy, this is only one of the many possible execution traces. Since applyBacktrack is called only in a conflict situation, with each its call we have printed a conflict clause. Notice that this conflict clause need not be unique.

Correctness. First we formulate some of the conditions that hold at each line of the code and therefore are its invariants:

$Invariant_{consistentM}$: (consistent M) — ensures that M is always in consistent state so it is a potential model of a formula.

$Invariant_{uniqueM}$: (unique M) — ensures that there are no duplicate literals in the trail (which is important for termination).

$Invariant_{varsM}$: (vars M) $\subseteq$ (vars F_0) — ensures that trail does not contain variables that do not occur in the formula (which is important for termination). As a consequence of this invariant, the test (vars M) = (vars F_0) can be replaced by the test $|$(vars M)$|$ = $|$(vars F_0)$|$ which is easier to implement.

Function applied	satFlag	M
	UNDEF	$[\,]$
applyDecide (l = -6)	UNDEF	$[\vert -6]$
applyDecide (l = 1)	UNDEF	$[\vert -6, \vert 1]$
applyDecide (l = 3)	UNDEF	$[\vert -6, \vert 1, \vert 3]$
applyDecide (l = -4)	UNDEF	$[\vert -6, \vert 1, \vert 3, \vert -4]$
applyBacktrack ($M \vDash\neg [-3, 4]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4]$
applyDecide (l = 2)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2]$
applyDecide (l = -5)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2, \vert -5]$
applyBacktrack ($M \vDash\neg [-1, -3, 5]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, \vert 2, 5]$
applyBacktrack ($M \vDash\neg [-2, -4, -5]$)	UNDEF	$[\vert -6, \vert 1, \vert 3, 4, -2]$
applyBacktrack ($M \vDash\neg [-1, 2]$)	UNDEF	$[\vert -6, \vert 1, -3]$
applyDecide (l = -2)	UNDEF	$[\vert -6, \vert 1, -3, \vert -2]$
applyBacktrack ($M \vDash\neg [-1, 2]$)	UNDEF	$[\vert -6, \vert 1, -3, 2]$
applyDecide (l = 7)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7]$
applyDecide (l = 4)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4]$
applyDecide (l = 5)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, \vert 5]$
applyBacktrack ($M \vDash\neg [-2, -4, -5]$)	UNDEF	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, -5]$
$M \nvDash\neg F_0$, $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$	SAT	$[\vert -6, \vert 1, -3, 2, \vert 7, \vert 4, -5]$

Fig. 4 Execution trace for the Example 5

$Invariant_{impliedLiterals}$: $\forall l.\ l \in M \implies F_0 \ @\ (\mathsf{decisionsTo}\ M\ l) \vDash l$ — ensures that all implied literals on the trail are entailed from the formula F_0 and all decision literals that precede them.

The non-trivial preconditions for function calls are:

applyDecide	$(\mathsf{vars}\ M) \neq (\mathsf{vars}\ F_0)$
applyBacktrack	$M \vDash\neg F_0$, $(\mathsf{decisions}\ M) \neq [\,]$

Now we outline the soundness proof. The variable $satFlag$ is initialized to UNDEF and it is changed only in two lines of code. The following lemmas ensure that this is the case only when the formula F_0 is determined to be satisfiable or to be unsatisfiable. From this, it is easy to prove that the required postcondition $\{(satFlag = UNSAT \wedge \neg(\mathsf{sat}\ F_0)) \ \vee\ (satFlag = SAT \wedge M \vDash F_0)\}$ defined in Theorem 1 holds.

In case when F_0 is not false in M and when all variables from F_0 have been assigned (when $M \nvDash\neg F_0$ and $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$), $satFlag$ is set to SAT. $Invariant_{consistentM}$ ensures that in that case F_0 is satisfiable and its model has been found, hence the procedure is sound for satisfiable formulae:

Lemma 1 *If*

(a) $Invariant_{consistentM}$ *holds,*

(b) $M \nvDash\neg F_0$,

(c) $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$,

then M *is a model for* F_0.

Proof Since M is a total valuation with respect to the variables from F_0, the formula F_0 is either true (i.e., $M \vDash F_0$) or false (i.e., $M \vDash\neg F_0$) in it. Since $M \nvDash\neg F_0$, it must be the case that $M \vDash F_0$. Since the condition $(\mathsf{consistent}\ M)$ holds, M is a model for F_0, hence F_0 is satisfiable. □

When a conflict at decision level zero occurs (when (decisions M) = [] and $M \vDash\neg F_0$), $satFlag$ is set to $UNSAT$. Then, $Invariant_{impliedLiterals}$ ensures that F_0 is unsatisfiable, so the procedure is sound for unsatisfiable formulae:

Lemma 2 *If*

(a) $Invariant_{impliedLiterals}$ *holds,*
(b) $M \vDash\neg F_0$*,*
(c) (decisions M) $= [\,]$*,*

then F_0 *is not satisfiable, i.e.,* $\neg$(sat F_0)*.*

Proof Since (decisions M) $= [\,]$, it holds that (decisionsTo M l) $= [\,]$ and for all literals l such that $l \in M$, it holds that $F_0 \vDash l$. Thus, the formula F_0 is false in a valuation it entails, so it is unsatisfiable. □

4.2 Unit Propagation

The simple implementation given in the previous section is based entirely on search and does not use any inference to find a satisfying assignment. For example, in the previous trace, a decision -4 was made, even though 3 was on the trail and there was a clause $[-3, 4]$ in F_0. For this clause to be true, the literal 4 must be true when the literal 3 is asserted, and 4 should be asserted as an implied literal immediately after 3 was asserted. If 3 is asserted in the trail M, and neither 4 nor -4 are, then $[-3, 4]$ is a unit clause in M with the unit literal 4, i.e., (isUnit $[-3, 4]$ 4 M). Exploiting unit clauses leads to huge reductions of the search space. In order to achieve this, the code from the previous section can be changed in the following way.

SAT solver v.2

```
function solve ($F_0$ : Formula) : (SAT, UNSAT)
..................
   while $satFlag$ = UNDEF do begin
      exhaustiveUnitPropagate();                                   +
      if $M \vDash\neg F_0$ then
..................
function exhaustiveUnitPropagate()                                 +
begin                                                              +
   repeat                                                          +
      ret := applyUnitPropagate();                                 +
   until $M \vDash\neg F_0$ $\vee$ ret = false                     +
end                                                                +
function applyUnitPropagate() : Boolean                            +
begin                                                              +
   if $\exists$c.$\exists$l. c $\in$ $F_0$ $\wedge$ (isUnit c l $M$) then begin   +
      assertLiteral (l, false);                                    +
      ret := true                                                  +
   end    else                                                     +
      ret := false                                                 +
end
```

Function applied	satFlag	M
	UNDEF	$[\,]$
applyDecide (l = 6)	UNDEF	$[\|6]$
applyUnitPropagate (c = $[1,-6]$, l = 1)	UNDEF	$[\|6,1]$
applyUnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[\|6,1,2]$
applyDecide (l = 7)	UNDEF	$[\|6,1,2,\|7]$
applyDecide (l = 3)	UNDEF	$[\|6,1,2,\|7,\|3]$
applyUnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[\|6,1,2,\|7,\|3,4]$
applyUnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[\|6,1,2,\|7,\|3,4,5]$
applyBacktrack ($M \vDash\neg [-2,-4,-5]$)	UNDEF	$[\|6,1,2,\|7,-3]$
applyUnitPropagate (c = $[-2,3,5,-6]$, l = 5)	UNDEF	$[\|6,1,2,\|7,-3,5]$
applyBacktrack ($M \vDash\neg [-1,3,-5,-6]$)	UNDEF	$[\|6,1,2,-7]$
applyDecide (l = 3)	UNDEF	$[\|6,1,2,-7,\|3]$
applyUnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[\|6,1,2,-7,\|3,4]$
applyUnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[\|6,1,2,-7,\|3,4,5]$
applyBacktrack ($M \vDash\neg [-2,-4,-5]$)	UNDEF	$[\|6,1,2,-7,-3]$
applyUnitPropagate (c = $[-2,3,5,-6]$, l = 5)	UNDEF	$[\|6,1,2,-7,-3,5]$
applyBacktrack ($M \vDash\neg [-1,3,-5,-6]$)	UNDEF	$[-6]$
applyDecide (l = 1)	UNDEF	$[-6,\|1]$
applyUnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[-6,\|1,2]$
applyDecide (l = 7)	UNDEF	$[-6,\|1,2,\|7]$
applyDecide (l = 3)	UNDEF	$[-6,\|1,2,\|7,\|3]$
applyUnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[-6,\|1,2,\|7,\|3,4]$
applyUnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[-6,\|1,2,\|7,\|3,4,5]$
applyBacktrack ($M \vDash\neg [-2,-4,-5]$)	UNDEF	$[-6,\|1,2,\|7,-3]$
applyDecide (l = 4)	UNDEF	$[-6,\|1,2,\|7,-3,\|4]$
applyUnitPropagate (c = $[-2,-4,-5]$, l = -5)	UNDEF	$[-6,\|1,2,\|7,-3,\|4,-5]$
$M \not\vDash\neg F_0$, $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$	SAT	$[-6,\|1,2,\|7,-3,\|4,-5]$

Fig. 5 Execution trace for the Example 6

Example 6 Let F_0 be as in Example 5. A possible execution trace is given in Figure 5.

Again, most meta-logic expressions can be easily implemented in a real programming language. An exception is the test $\exists$c.$\exists$l. c $\in F_0 \wedge$ (isUnit c l M), used to detect unit clauses. We postpone addressing this issue until §4.9.

Correctness. Unit propagation does not compromise the code correctness, as the code preserves all invariants listed in Section 4.1. The proofs of lemmas 1 and 2 still apply.

4.3 Backjumping and Learning

A huge improvement in SAT solver development was gained when the simple backtracking was replaced by the conflict driven backjumping and learning. Namely, two problems are visible from the trace given in Example 6:

1. The series of steps taken after the decision 7 was made showed that neither 3 nor -3 are compatible with previous decisions. Because of that, the backtrack operation implied that -7 must hold. Then, exactly the same series of steps was performed to show that, again, neither 3 nor -3 are compatible with previous decisions. Finally, the backtrack operation implied that

-6 must hold. A careful analysis of the steps that produced the inconsistency would show that the variable 7 is totally irrelevant for the conflict, and the repetition of the process with 7 flipped to -7 was a waste of time. This redundancy was the result of the fact that the backtrack operation always undoes only the last decision made, regardless of the actual reason that caused the inconsistency. The operation of *conflict-driven backjumping* which is a form of more advanced, non-chronological backtracking, allows solvers to undo several decisions at once, down to the latest decision literal which actually participated in the conflict. This eliminates unnecessary redundancies in the execution trace.

2. In several steps, the decision 3 was made after the literal 1 was already on the trail. The series of steps taken afterwards showed that these two are incompatible. Notice that this occurred in two different contexts (both under the assumption 6, and under the assumption -6). The fact that 1 and 3 are incompatible can be represented by the clause $[-1, -3]$, which is a logical consequence of F_0. If this clause was a part of the formula F, then it would participate in the unit propagation and it would imply the literal -3 immediately after 1 occurred on the trail. The *clause learning* mechanism allows solvers to extend the clause set of the formula during the search process with (redundant) clauses implied by F_0.

Backjumping is guided by a *backjump clause*[4] (denoted in what follows by C), which is a consequence of the formula F_0 and which corresponds to variable assignment that leads to the conflict. When a backjump clause is constructed, the top literals from the trail M are removed, until the backjump clause becomes 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*. This process is sometimes described using a graph-based framework and the backjump clauses are constructed by traversing the implication graph [ZMMM01]. The conflict analysis process can be also described as a backward resolution process that starts from the conflict clause and performs a series of resolutions with clauses that are reasons for propagation of conflict literals [ZM02].

Notice that backtracking can be seen as a special case of backjumping. This happens when advanced conflict analysis is not explicitly performed, but the backjump clause always contains the opposites of all decision literals from M. This clause becomes unit clause when the last decision literal is backtracked, and then it implies the opposite of the backtracked last decision literal.

There are several strategies for conflict analysis [SS96, ZMMM01, ZM02]. Their variations include different ways of choosing the clause that guides backjumping and that is usually learnt. Some strategies allow multiple clauses to be learnt from a single conflict. Still, most of conflict analysis strategies are based on the following technique:

- The conflict analysis process starts with a conflict clause itself (the clause of F that is false in M), and the backjump clause C is initialized to it.

[4] Sometimes also called the *assertive clause*.

- Each literal contained in the current backjump 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 forced this propagation to occur. These clauses are called *reason clauses*, and (isReason c l M) holds. Propagated literals from the current backjump clause C are then replaced (we say *explained*) by other literals from reason clauses, continuing the analysis backwards. The explanation step can be seen as a resolution between the backjump and reason clauses.
- The procedure is repeated until some termination condition is fulfilled, resulting in the final backjump clause.

Different strategies determine termination conditions for the conflict analysis process. We will only describe the one called the *first unique implication point (firstUIP)*, since it is used in most leading SAT solvers and since it outperforms other strategies on most benchmarks [ZM02]. Using the firstUIP strategy, the learning process is terminated when the backjump clause contains exactly one literal from the current decision level.

The following version of the solver works similarly to SAT solver v.2 up to the point when $M \vDash \neg F$ i.e., when a conflict occurs. Then, the conflict analysis is performed, implemented through applyConflict and applyExplain functions. The function applyConflict initializes the backjump clause C to a conflict clause. The function applyExplain resolves out a literal l from C by performing a single resolution step between C and a clause that is the reason for propagation of $\overline{l}$. When the conflict is resolved, the solver continues to work as SAT solver v.2.

If a conflict occurs at a decision level other then zero, then a backjump clause is constructed using the function applyExplainUIP. It iteratively resolves out the last asserted literal of $\overline{C}$ using the applyExplain function until C satisfies the firstUIP condition. The function applyLearn adds the constructed backjump clause C to the current clause set F. The function applyBackjump backtracks literals from the trail M until C becomes a unit clause, and after that asserts the unit literal of C. Notice that this last step does not have to be done by applyBackjump, but it can be handled by the function applyUnitPropagate. So, some implementations of applyBackjump omit its last two lines given here.

If a conflict at decision level zero occurs, then the empty clause C is effectively constructed using the function applyExplainEmpty. It always resolves out the last asserted literal from $\overline{C}$ by calling applyExplain, until C becomes empty. It is possible to extend the solver with the possibility of generating resolution proofs for unsatisfiability and this explicit construction of empty clause makes that process more uniform.

SAT solver v.3

```
...
F : Formula                                                              +
C : Clause                                                               +
reason : Literal => Clause                                               +
function solve ($F_0$ : Formula) : (SAT, UNSAT)
begin
    $satFlag$ = UNDEF;
    $M$ := [];
    $F$ := $F_0$;                                                     +
    while $satFlag$ = UNDEF do begin
        exhaustiveUnitPropagate();
        if $M \vDash \neg F$ then begin
            applyConflict();                                          +
            if (currentLevel $M$) = 0 then begin
                applyExplainEmpty();                                  +
                applyLearn();                                         +
                $satFlag$ = UNSAT
            end else begin
                applyExplainUIP();                                    +
                applyLearn();                                         +
                applyBackjump()                                       +
            end
        end else
            if |(vars $M$)| = |(vars $F_0$)| then
                $satFlag$ = SAT
            else
                applyDecide()
    end
end
function applyUnitPropagate() : Boolean
begin
    if $\exists$c.$\exists$l. c $\in$ $F_0$ $\wedge$ (isUnit c l $M$) then begin
        assertLiteral (l, false);
        setReason(l, c);                                              +
        ...

function applyConflict()                                              +
begin                                                                 +
    $C$ := getConflictClause()                                        +
end                                                                   +
function applyExplainUIP()                                            +
begin                                                                 +
    while $\neg$ isUIP()                                              +
        applyExplain((lastAssertedLiteral $\overline{C}$ $M$))        +
end                                                                   +
const function isUIP() : Boolean                                      +
begin                                                                 +
    l := (lastAssertedLiteral $\overline{C}$ $M$);                    +
    if $\exists$l'. l'$\in \overline{C}$ $\wedge$ l'$\neq$l $\wedge$ (level l' M) = (level l M) then     +
        ret := false                                                  +
    else                                                              +
        ret := true                                                   +
end                                                                   +
function applyExplainEmpty()                                          +
begin                                                                 +
    while $C \neq$ []                                                 +
        applyExplain((lastAssertedLiteral $\overline{C}$ $M$))        +
end                                                                   +
function applyExplain(l : Literal)                                  +
begin                                                               +
   reason := getReason(l);                                          +
   $C$ := (resolvent $C$ reason $\overline{l}$)                     +
end                                                                 +
function applyLearn()                                               +
begin                                                               +
   $F$ = $F$ @ $C$                                                  +
end                                                                 +
function applyBackjump()                                            +
begin                                                               +
   l := (lastAssertedLiteral $\overline{C}$ $M$);                   +
   level := getBackjumpLevel();                                     +
   $M$ := (prefixToLevel $M$ level);                                +
   assertLiteral($\overline{l}$, false)                             +
   setReason($\overline{l}$, $C$);                                  +
end                                                                 +
const function getBackjumpLevel : int                               +
begin                                                               +
   l := (lastAssertedLiteral $\overline{C}$ $M$);                   +
   if $\overline{C} \setminus l \neq$ [] then                       +
      ret := (maxLevel $\overline{C} \setminus l$ M)                +
   else                                                             +
      ret := 0                                                      +
end                                                                 +
const function getReason(l : Literal) : Clause                      +
begin                                                               +
   ret := $reason$(l)                                               +
end                                                                 +
function setReason(l : Literal, c : Clause)                         +
begin                                                               +
   $reason$(l) := c                                                 +
end                                                                 +
$\{M \vDash\neg F\}$
const function getConflictClause() : Clause                         +
$\{M \vDash\neg ret\}$
```

The function getConflictClause returns an arbitrary clause of F that is false in M. It satisfies the Hoare triple $\{M \vDash\neg F\}$ getConflictClause() $\{M \vDash\neg ret\}$.

All non-decision literals are asserted as a result of unit propagation or backjumping, and the reason clauses are memorized in the mapping *reason* using the function setReason.[5] The function getReason retrieves the clause that caused the assertion of a given implied literal. It satisfies the Hoare triple $\{l \in M \ \wedge \ l \notin$ (decisions M)$\}$ `getReason(l)` $\{ret \in F \ \wedge$ (isReason $ret\ l\ M$)$\}$.

Example 7 Let F_0 be as in Example 5. One possible execution trace is shown in Figure 6. Resolution trees corresponding to conflict analyses from Example 7 are shown in Figure 7.

Correctness. The partial correctness proof for the SAT solver v.3 is more involved. The soundness theorem (Theorem 1) needs to be proved again.

[5] In a real programming language implementation, one would store only the pointers to clauses instead of clauses themselves.

function applied	*satFlag*	M	F	C
applyDecide (l = 6)	UNDEF	$[\vert 6]$	F_0	-
applyUnitPropagate (c = $[1,-6]$, l = 1)	UNDEF	$[\vert 6,1]$	F_0	-
applyUnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[\vert 6,1,2]$	F_0	-
applyDecide (l = 7)	UNDEF	$[\vert 6,1,2,\vert 7]$	F_0	-
applyDecide (l = 3)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3]$	F_0	-
applyUnitPropagate (c = $[-3,4]$, l = 4)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4]$	F_0	-
applyUnitPropagate (c = $[-1,-3,5]$, l = 5)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4,5]$	F_0	-
applyConflict ($M \vDash\lnot [-2,-4,-5]$)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4,5]$	F_0	$[-2,-4,-5]$
applyExplain (l = 5, reason = $[-1,-3,5]$)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4,5]$	F_0	$[-1,-2,-3,-4]$
applyExplain (l = 4, reason = $[-3,4]$)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4,5]$	F_0	$[-1,-2,-3]$
applyLearn (C = $[-1,-2,-3]$)	UNDEF	$[\vert 6,1,2,\vert 7,\vert 3,4,5]$	$F_0@[-1,-2,-3]$	$[-1,-2,-3]$
applyBackjump (C = $[-1,-2,-3]$, l = 3, level = 1)	UNDEF	$[\vert 6,1,2,-3]$	$F_0@[-1,-2,-3]$	-
applyUnitPropagate (c = $[-2,3,5,-6]$, l = 5)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	-
applyConflict ($M \vDash\lnot [-1,3,-5,-6]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	$[-1,3,-5,-6]$
applyExplain (l = 5, reason = $[-2,3,5,-6]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	$[-1,-2,3,-6]$
applyExplain (l = -3, reason = $[-1,-2,-3]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	$[-1,-2,-6]$
applyExplain (l = 2, reason = $[-1,2]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	$[-1,-6]$
applyExplain (l = 1, reason = $[-6,1]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]$	$[-6]$
applyLearn (C = $[-6]$)	UNDEF	$[\vert 6,1,2,-3,5]$	$F_0@[-1,-2,-3]@[-6]$	$[-6]$
applyBackjump (C = $[-6]$, l = 6, level = 0)	UNDEF	$[-6]$	$F_0@[-1,-2,-3]@[-6]$	-
applyDecide (l = 1)	UNDEF	$[-6,\vert 1]$	$F_0@[-1,-2,-3]@[-6]$	-
applyUnitPropagate (c = $[-1,2]$, l = 2)	UNDEF	$[-6,\vert 1,2]$	$F_0@[-1,-2,-3]@[-6]$	-
applyUnitPropagate (c = $[-1,-2,-3]$, l = -3)	UNDEF	$[-6,\vert 1,2,-3]$	$F_0@[-1,-2,-3]@[-6]$	-
applyDecide (l = 7)	UNDEF	$[-6,\vert 1,2,-3,\vert 7]$	$F_0@[-1,-2,-3]@[-6]$	-
applyDecide (l = 4)	UNDEF	$[-6,\vert 1,2,-3,\vert 7,\vert 4]$	$F_0@[-1,-2,-3]@[-6]$	-
applyUnitPropagate (c = $[-2,-4,-5]$, l = -5)	UNDEF	$[-6,\vert 1,2,-3,\vert 7,\vert 4,-5]$	$F_0@[-1,-2,-3]@[-6]$	-
$M \nvDash\lnot F_0$, (vars M) = (vars F_0)	SAT	$[-6,\vert 1,2,-3,\vert 7,\vert 4,-5]$	-	-

Fig. 6 Execution trace for the Example 7

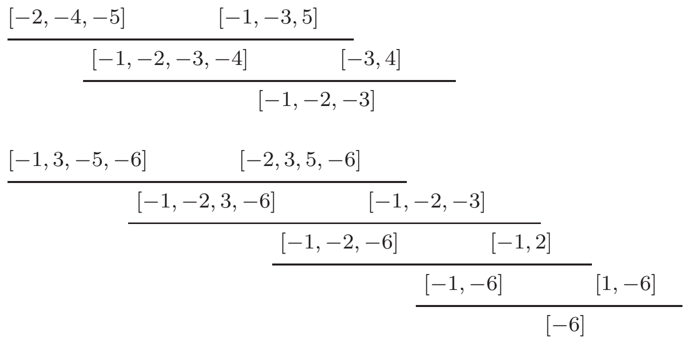

Fig. 7 Resolution trees corresponding to conflict analyses in Example 7

Along with $Invariant_{consistentM}$, $Invariant_{uniqueM}$, $Invariant_{varsM}$, and $Invariant_{impliedLiterals}$ the code also satisfies the following conditions:

$Invariant_{equiv}$: $F \equiv F_0$ — ensures that when new clauses are learnt, the current formula F remains equivalent to the initial formula F_0.
$Invariant_{varsF}$: $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0)$ — ensures that clauses in F do not introduce new variables.
$Invariant_{reasonClauses}$: $\forall l.\ l \in M \wedge l \notin (\mathsf{decisions}\ M) \implies \exists\ c.\ c \in F \wedge (\mathsf{isReason}\ c\ l\ M)$ — ensures that there is a reason clause for each propagated literal in M.

During the conflict analysis process (in the `if` $M \vDash\neg F$ `then` branch) the following conditions hold.

$Invariant_{Cfalse}$: $M \vDash\neg C$ — ensures that C becomes an assertive clause (i.e., unit in the prefix) after backtracking.
$Invariant_{Cimplied}$: $F_0 \vDash C$ — is used to support $Invariant_{equiv}$, since C is the clause that is being learned.

These conditions are preserved by applyConflict and applyExplain functions.

The non-trivial preconditions for function calls are:

applyConflict	$M \vDash\neg F$
applyExplainUIP	$M \vDash\neg C$, $(\mathsf{decisions}\ M) \neq []$
applyExplainEmpty	$M \vDash\neg C$, $(\mathsf{decisions}\ M) = []$
applyExplain(l)	$M \vDash\neg C$, $\bar{l} \in C$, $l \notin (\mathsf{decisions}\ M)$
applyLearn	$F \vDash C$, $C \notin F$
applyBackjump	$(\mathsf{isUIP}\ C\ M)$, $C \in F$
isUIP	$M \vDash\neg C$
getBackjumpLevel	$(\mathsf{isUIP}\ C\ M)$
getReason(l)	$l \in M$, $l \notin (\mathsf{decisions}\ M)$
setReason(l, c)	$(\mathsf{isReason}\ c\ l\ M)$

The soundness for satisfiable formulae follows from the following analogue of Lemma 1.

Lemma 3 *If*
(a) $Invariant_{consistentM}$, $Invariant_{varsF}$, *and* $Invariant_{equiv}$ *hold,*

(b) $M \not\models \neg F$,
(c) $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$,
then M *is a model for* F_0.

Proof Since $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0)$ and $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$, it holds that $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ M)$, and M is a total valuation with respect to variables from F. So, the formula F is either true (i.e., $M \models F$) or false (i.e., $M \models \neg F$) in it. Since $M \not\models \neg F$, it must be the case that $M \models F$. Since the condition $(\mathsf{consistent}\ M)$ holds, M is a model for F. Since F and F_0 are logically equivalent by $Invariant_{equiv}$, M is also a model for F_0. □

Regarding the soundness for unsatisfiable formulae, an analouge of Lemma 2 could be formulated and proved. However, we give a simpler, alternative proof of soundness for unsatisfiability, that does not use $Invariant_{impliedLiterals}$ (in contrast to the given proof of Lemma 2). After a conflict at the decision level zero has been detected (when $M \models \neg F$ and $(\mathsf{decisions}\ M) = [\,]$), the function applyExplainEmpty is called. It performs a series of applications of the explain rule, and resolves out all the literals from the conflict clause, leaving the empty clause C. In this case, the procedure reports unsatisfiability of the formula, if and only if the empty clause has been derived. The postcondition of applyExplainEmpty guarantees that $C = [\,]$, and the soundness is a consequence of $Invariant_{Cimplied}$ (i.e., $F_0 \models C$), as stated in the following trivial lemma (given without proof).

Lemma 4 *If the following conditions hold*
(a) $C = [\,]$,
(b) $F_0 \models C$ *(i.e.,* $Invariant_{Cimplied}$ *holds),*
then F_0 *is unsatisfiable.*

Soundness for satisfiability and soundness for unsatisfiability together imply Theorem 1 for SAT Solver v.3.

4.3.1 Efficient Data Structures

The code given in §4.3 leaves the function resolvent unspecified. We will now describe how it can be efficiently implemented, roughly following MiniSAT [ES04]. Without a great loss of generality, it is assumed that the conflict clause (and therefore the clause C) contains a literal from the current decision level i.e., $(\mathsf{currentLevel}\ M) = (\mathsf{maxLevel}\ \overline{C}\ M)$. This holds whenever there is a guarantee that no new decisions are made when $M \models \neg F$, and, clearly, this is the case in the implementation we provided.[6] It is also reasonable to require that the clause C does not contain repeated elements. So, in each resolvent step, when a union of two clauses is created, duplicates have to be removed. In order to achieve an efficient implementation, instead of ordinary list of literals, some more suitable representation for the clause C has to be used. We use a map C_H

[6] We will see that this is an important requirement for the correctness of the two-watch literal propagation scheme that we explain in §4.9.

that maps literals to booleans such that $C_H(l) = \mathit{true}$ iff $l \in C$. Notice that this representation allows constant time check whether a literal is contained in the clause C so it is suitable for a range of operations performed with this clause. When the conflict analysis process is done, the list of literals contained in the clause C can be constructed by traversing the map C_H looking for literals which are true in it. Notice that this expensive traversal can be avoided using the fact that during the firstUIP conflict analysis only the literals from the highest decision level of C (i.e., the current decision level of M) are explained. So, when a literal from a decision level lower then the current decision level of M is added into the clause C, it cannot get removed from it until the end of the firstUIP resolution process. Also, since the UIP condition is met at the end of the resolution process, the backjump clause contains exactly one literal from the current (highest) decision level. Therefore, it is useful to keep the list C_P of the literals from the lower decision levels from C and the last asserted literal C_l of $\overline{C}$, because these two can give the list of literals contained in the clause C at the end, avoiding the traversal of C_H. In order to optimize isUIP check, the procedure also keeps track of the number C_n of literals from the highest decision level (that is (currentLevel M)).

The list of literals C from §4.3 used in SAT solver v.3, and the variables C_H, C_P, C_l and C_n used in SAT solver v.4 are related as follows:

$$\begin{aligned}
C_H(l) = \mathit{true} &\Leftrightarrow l \in C \\
C_l &= (\mathsf{lastAssertedLiteral}\ \overline{C}\ M) \\
l \in C_P &\Leftrightarrow l \in C\ \wedge\ (\mathsf{level}\ \overline{l}\ M) < (\mathsf{maxLevel}\ \overline{C}\ M) \\
C_n &= |\{l\ :\ l \in C\ \wedge\ (\mathsf{level}\ \overline{l}\ M) = (\mathsf{maxLevel}\ \overline{C}\ M)\}|
\end{aligned}$$

The following code is a modification of SAT solver v.3 adapted to use the data structures just described. The clause C (that is its components C_H, C_P, C_n) can be changed only by the functions addLiteral and removeLiteral, while C_l is set by findLastAssertedLiteral.

SAT solver v.4

```
...
$C_H$ : Literal => Boolean
$C_P$ : Clause
$C_l$ : Literal
$C_n$ : nat
...
function applyConflict()
begin
   $C_H$ := {};                                                      +
   foreach l : l $\in$ getConflictClause() do                        *
      addLiteral(l)                                                  *
   findLastAssertedLiteral();                                        +
end
function applyExplainUIP()
begin
   while $\neg$ isUIP() do
      applyExplain($C_l$);                                           *
   buildC()                                                          +
end
```

```
function buildC()                                                       +
    $C$ := $C_P$ @ $\overline{C_l}$                                     +
end                                                                     +
const function isUIP() : Boolean
begin
    if $C_n$ = 1 then ret := true else ret := false                     *
end
function applyExplain(l : Literal)
begin
    reason := getReason(l);
    resolve(reason, l)                                                  *
    findLastAssertedLiteral();                                          +
end
function resolve(clause : Clause, l : Literal)                          +
begin                                                                   +
    removeLiteral($\overline{l}$);                                      +
    foreach l': l' $\in$ clause $\wedge$ l' $\neq$ l do                 +
        addLiteral(l')                                                  +
end                                                                     +
function applyBackjump()
begin
    level := getBackjumpLevel();
    $M$ := (prefixToLevel $M$ level);
    assertLiteral($\overline{C_l}$, false)                              *
    setReason($\overline{C_l}$, $C$);                                   *
end
const function getBackjumpLevel : int
begin
    if $C_P \neq$ [] then                                               *
        ret := (maxLevel $\overline{C_P}$ M)                            *
    else
        ret := 0
end
function addLiteral(l : Literal)                                        +
begin                                                                   +
    if $C_H$(l) = false then begin                                      +
        $C_H$(l) := true;                                               +
        if (level $\overline{l}$ $M$) = (currentLevel $M$) then         +
            $C_n$ := $C_n$ + 1                                          +
        else                                                            +
            $C_P$ := $C_P$ @ l                                          +
    end                                                                 +
end                                                                     +
function removeLiteral(l : Literal)                                     +
begin                                                                   +
    $C_H$(l) := false;                                                  +
    if (level $\overline{l}$ $M$) = (currentLevel $M$) then             +
        $C_n$ := $C_n$ - 1                                              +
    else                                                                +
        $C_P$ := $C_P$ $\setminus$ l                                    +
end                                                                     +
```

function applied	C_H[7]	C_P	C_l	C_n
...				
`applyConflict` $(M \vDash\neg [-2,-4,-5])$	$\{-2 \mapsto \top, -4 \mapsto \top, -5 \mapsto \top\}$	$[-2]$	5	2
`applyExplain` (`l = 5, reason =` $[-1,-3,5]$)	$\{-1 \mapsto \top, -2 \mapsto \top, -3 \mapsto \top, -4 \mapsto \top\}$	$[-1,-2]$	4	2
`applyExplain` (`l = 4, reason =` $[-3,4]$)	$\{-1 \mapsto \top, -2 \mapsto \top, -3 \mapsto \top\}$	$[-1,-2]$	3	1
...				

Fig. 8 The conflict analysis using efficient data-structures.

```
function findLastAssertedLiteral()                               +
begin                                                            +
   repeat                                                        +
      $C_l$ := (last $M$)                                        +
   until $C_H(\overline{C_l})$ = true                            +
end                                                              +
```

Example 8 Figure 8 show one conflict analysis trace, from Example 7, for $M = [|6, 1, 2, |7, |3, 4, 5]$.

Notice that some minor optimizations can be made in the given code. For example, the function resolve is always called with a current level literal as the clashing literal. Therefore, the call to the generic removeLiteral function, could be replaced by a call to function removeCurrentLevelLiteral that would always be called for a literal at the current level. This would save a check of the level of the literal that is being removed. Also, it is possible to change the map C_H, so that it maps variables instead of literals to booleans. In that case, $l \in C$ would imply that $C_H(\text{var } l)$ must hold. This change is possible, because invariants (consistent M) and $M \vDash\neg C$ imply that C can not be a tautological clause i.e., it can not contain both a literal and its opposite. So, since it holds that $l \in C \Leftrightarrow C_H(l) = true$, for each literal l it must hold that $C_H(l) = true \Rightarrow C_H(\overline{l}) = false$.

4.4 Conflict Clause Minimization

In some cases, clauses obtained by the firstUIP heuristic can further be minimized. Smaller clauses prune larger parts of the search tree and lead to faster unit propagation. Several clause minimization techniques have been proposed; as an illustration, we present the *subsumption resolution* of [ES04]. In Example 7, the learnt backjump clause is $[-1, -2, -3]$, but since 2 is implied by 1 (because of the clause $[-1, 2]$), one more resolution step could be performed to get a smaller backjump clause $[-1, -3]$.

This variant of SAT solver augments SAT solver v.3.

SAT solver v.5

```
function solve ($F_0$ : Formula) : (SAT, UNSAT)
.........................
```

[7] For simplicity, literals that map to $\perp$ are omitted.

```
   applyExplainUIP();
   applyExplainSubsumption();                                                +
   applyLearn();
   applyBackjump()
. . . . . . . . . . . . . . . . . . . . . . . . .
function applyExplainSubsumption()                                           +
begin                                                                        +
   foreach l: l ∈ $\overline{C}$ ∧ l ∉ (decisions M) do                      +
      if subsumes($C$, getReason(l) \ l) then                                +
         applyExplain(l)                                                     +
end                                                                          +
const function subsumes (c1 : Clause, c2 : Clause) : Boolean                 +
begin                                                                        +
   if c2 ⊆ c1 then ret := true else ret := false                             +
end                                                                          +
```

The subsumption check has to be carefully implemented so that it does not become the bottleneck of this part of the solver. An implementation that uses the efficient conflict analysis clause representation given in §4.3.1 is available in the extended version of this paper.

A more advanced conflict clause minimization, based on subsumption, can also be performed. Namely, the subsumption check fails if the reason clause for l contains at least one literal l' different from l that is not in C. However, by following the reason graph backwards for l', we might find that l' became true as a consequence of assigning only literals present in the conflict clause, in which case l' can be ignored. If this is true for all literals l' of the reason clause that are not in C, then the literal l can still be explained.

4.5 Forgetting

During the solving process with clause learning, the number of clauses in F increases. If the number of clauses becomes too large, then the boolean constraint propagation becomes unacceptably slow and some (redundant) clauses from F should be removed. It is necessary to take special care and ensure that the reduced formula is still equivalent with the initial formula F_0 (i.e., that $Invariant_{equiv}$ is preserved). Solvers usually ensure this by allowing only the removal of learnt clauses while the initial clauses never get removed. It is also required that clauses that are reasons for propagation of some literals in M are not removed (i.e., that $Invariant_{reasonClauses}$ is preserved).

The following modifications can be made to any given clause learning SAT solver (solvers after variant 3).

SAT solver v.6

```
function solve ($F_0$ : Formula) : (SAT, UNSAT)
. . . . . . . . . . . . . . . . . . . . . . . . .
                if shouldForget() then                                       +
                   applyForget()                                             +
               applyDecide()
. . . . . . . . . . . . . . . . . . . . . . . . .
```

```
function applyForget()
begin
   newF := [];
   foreach c : c $\in$ F do
      if shouldForget(c) $\wedge$ isLearnt(c) $\wedge$ $\neg$isReason(c) then
         removeClause(c)
      else
         newF := newF @ c;
   F := newF
end
```

```
$\{\top\}$
const function shouldForget() : Boolean                                  +
$\{\top\}$
```

```
$\{\top\}$
const function shouldForget(Clause : c) : Boolean                        +
$\{\top\}$
```

The function `isLearnt` is left unspecified. Its implementation can be very simple. For example, the list F can be naturally split into the initial clauses F_0 and the learnt clauses F_l.

The function `isReason` can be implemented as checking if c is the reason for propagation of its last asserted literal.

The function `shouldForget` determines when to apply the forget rule and the function `shouldForget` determines which clauses to forget. Together, they form a forget strategy. Usually, a forget strategy is based on the number of learnt clauses, but other criteria (e.g., the total number of literals in the clause database) can be used. Clauses are usually forgotten as a result of their poor activity in conflicts and in unit propagation, or as a result of their length.

4.6 Restarting

Another important feature of state-of-the-art SAT solvers are *restarts*. From time to time, solvers start the search from scratch by backtracking the trail M down to the decision level zero, but keeping all the knowledge accumulated in the learnt clauses. This can improve performance, because it can lead the solver to a new (usually easier) search path. It has been shown that clause learning, as practiced in today's SAT solvers, assuming unlimited restarts, corresponds to a proof system exponentially more powerful than that of DPLL [BKS04]. However, unlimited restarts can jeopardize termination, so strategies that determine when to restart must be carefully designed. There is a strong experimental evidence that clause learning SAT solvers could benefit substantially from a carefully designed restart policy [Hua07b].

The following modification can be made to any given solver variant that uses clause learning.

SAT solver v.7

```
function solve ($F_0$ : Formula) : (SAT, UNSAT)
.........................
```

```
                if shouldRestart() then                                      +
                    applyRestart()                                           +
               applyDecide()
........................

function applyRestart()                                                     +
begin                                                                       +
    $M$ := (prefixToLevel $M$ 0)                                            +
end                                                                         +
{⊤}
const function shouldRestart() : Boolean                                    +
{⊤}
```

The function shouldRestart() is intentionally left unspecified and it represents the restart strategy of the solver. For a survey of restart strategies see, for instance, [Hua07b].

4.7 Zero Level Literals

Since (level l M) $= 0 \implies$ (decisionsTo M l) $= [\,]$, the $Invariant_{impliedLiterals}$ ensures that literals at the decision level zero are consequences of the formula itself. These literals have a special role during the solving process.

4.7.1 Single Literal Clauses

Under the assumption that literals from the decision level zero never get removed from the trail, adding single literal clauses $[l]$ to the current clause set F can be avoided. Instead, their literals l are added to the decision level zero of the trail M. This change helps in implementation of the two-watch literals scheme that will be described in §4.9. The initialization instruction F := F_0 in the previous code, should be changed in a way that guarantees that all clauses in F are at least two-literal clauses. The SAT solver v.8 is based on the SAT solver v.5 regardless of whether the modifications from v.6 or v.7 have been made, and is given through several function modifications.

SAT solver v.8

```
function solve ($F_0$ : Formula) : (SAT, UNSAT)
...
    $M$ := [];
    foreach clause: clause ∈ $F_0$ do                                        *
        addClause(clause);                                                   *
...
function addClause(clause : Clause)                                         +
begin                                                                       +
    clause := removeDuplicateLiterals(clause);                              +
    clause := removeFalsifiedLiterals(clause);                              +
    if containsTrueLiteral(clause) then                                     +
        return;                                                             +
    else if (length clause) = 0 then                                        +
        satFlag := UNSAT                                                    +
    else if (length clause) = 1 then begin                                  +
      assertLiteral((head clause), false);                                +
      exhaustiveUnitPropagate()                                           +
   end else if isTautological(clause) then                                +
      return;                                                             +
   else                                                                   +
      $F$ := $F$ @ clause                                                 +
end                                                                       +

const function containsTrueLiteral(clause: Clause) : Boolean              +
begin                                                                     +
   ret := false;                                                          +
   foreach l : l $\in$ clause do                                          +
      if $M \vDash$ l then ret := true                                    +
end                                                                       +

const function removeDuplicateLiterals(clause : Clause) : Clause          +
begin                                                                     +
   ret := [];                                                             +
   foreach l : l $\in$ clause do                                          +
      if l $\notin$ ret then ret := ret @ l;                              +
end                                                                       +
const function removeFalsifiedLiterals(clause : Clause) : Clause          +
begin                                                                     +
   ret := [];                                                             +
   foreach l : l $\in$ clause do                                          +
      if $M \nvDash \neg$l then ret := ret @ l;                           +
end                                                                       +
const function isTautological(c : clause)                                 +
   if $\exists$ l. l$\in$c $\wedge$ $\overline{\text{l}}$$\in$c then ret:=true else ret:=false     +
end                                                                       +
```

The given initialization procedure also ensures that no clause in F contains duplicate literals or both a literal and its negation. This property is preserved throughout the code and is another invariant.

Learning should be changed so that it is performed only for clauses with at least two (different) literals:

```
function applyLearn()
begin
   if (length $C$) > 1 then                                               +
      $F$ = $F$ @ $C$
end
```

Since learning is not performed for single literal clauses, for some zero level literals there is no reason clause stored in F. This means that those literals cannot be explained and removed from C using the applyExplain function (it is still safe to call getReason(l) for all non-decision literals asserted at higher decision levels). To handle this, backjump clauses are generated so that they do not contain literals from the decision level zero. These literals are skipped during the conflict analysis process and applyExplain is modified so that after its application C becomes $(\mathsf{resolvent}\ C\ c\ \overline{l}) \setminus \overline{(\mathsf{prefixToLevel}\ M\ 0)}$. When using the efficient representation of conflict analysis clause, defined in §4.3.1, the function addLiteral is modified in the following way.

```
function addLiteral(l : Literal)
begin
    if $C_H$(l) = false then begin
        if (level $\overline{l}$ $M$) $\neq$ 0 then begin                                  +
            $C_H$(l) := true;
            if (level $\overline{l}$ $M$) = (currentLevel $M$)
                $C_n$ := $C_n$ + 1
            else
                $C_P$ := $C_P$ @ l
        end                                                                +
    end
end
```

Notice that this change complicates generating unsatisfiability proofs, because some literals are removed from C without the explicit call of applyExplain.

Correctness. If single literal clauses are not added to F, then $Invariant_{equiv}$ does necessarily hold anymore. Instead, the condition

$$Invariant_{equiv'} : F \mathbin{@} (\mathsf{prefixToLevel}\ M\ 0) \ \equiv\ F_0$$

holds.

Also $Invariant_{reasonClauses}$ does not necessarily hold anymore because reason clauses are not stored in F for some literals at decision level zero. Instead,

$$Invariant_{reasonClauses'} : \forall l.\ l \in M \ \wedge\ l \notin (\mathsf{decisions}\ M) \ \wedge\ (\mathsf{level}\ l\ M) > 0 \implies \exists\ c.\ c \in F \ \wedge\ (\mathsf{isReason}\ c\ l\ M)$$

holds. Because of this, the precondition for applyExplain is strengthened with the condition $(\mathsf{level}\ l\ M) > 0$.

Soundness lemmas must be modified since their original proofs rely on the invariants that had to be modified.

Again, a lemma similar to Lemma 1 and Lemma 3 shows that M is a model for F_0 when $satFlag$ is set to SAT.

Lemma 5 *If*
(a) $Invariant_{consistentM}$, $Invariant_{varsF}$, *and* $Invariant_{equiv'}$ *hold,*
(b) $M \not\models \neg F$,
(c) $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$,
then M *is a model for* F_0.

Proof Since $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ F_0)$ and $(\mathsf{vars}\ M) = (\mathsf{vars}\ F_0)$, it holds that $(\mathsf{vars}\ F) \subseteq (\mathsf{vars}\ M)$, and M is a total valuation with respect to variables from F. Therefore, the formula F is either true (i.e., $M \models F$) or false (i.e., $M \models \neg F$) in it. Since $M \not\models \neg F$, it must be the case that $M \models F$. Since the condition $(\mathsf{consistent}\ M)$ holds, M is a model for F. Since M is trivially a model for $(\mathsf{prefixToLevel}\ M\ 0)$, it holds that M is a model for $F \mathbin{@} (\mathsf{prefixToLevel}\ M\ 0)$ which is logically equivalent to F_0, so M is also a model for F_0. □

Lemma 4 still holds and it shows that F_0 is not satisfiable when $satFlag$ is set to UNSAT. Therefore, Theorem 1 holds.

4.7.2 Clause Set Simplification.

Whenever a literal l is added to the decision level zero of M, to reduce memory consumption, a clause set simplification could be performed. All clauses that contain l could be removed as they are satisfied. Further, the literal $\overline{l}$ could be removed from all remaining clauses. However, solvers usually do not perform these simplifications eagerly, but do this only from time to time. Details of this technique are available in the extended version of this paper.

4.8 One-Watch Literal Scheme

To get a functional solver, the meta-logic condition $M \vDash\neg F$ has to be effectively implemented. A naive implementation which evaluates each clause from F would be extremely inefficient. A clause is false in a valuation M if and only if all of its literals are false in M. If a clause c contains a literal l which is unfalsified in M (i.e., $M \nvDash\neg l$), then c cannot be false in M, whatever values of its other literals are. This property motivates the *one-watch literal scheme.* The idea is to "put a watch" to an arbitrary unfalsified literal in each clause of F. When a literal l gets asserted in M, to check if $M \vDash\neg F$ it is sufficient to check only the clauses which have $\overline{l}$ as their watched literal (because if their watch is some other literal l', it would remain unfalsified after asserting l). The question that remains is how to find clauses from F that have $\overline{l}$ as their watched literal. Traversal of all clauses and checking their watch literal would be expensive. Instead, *watch lists* are used to index and store this information. The watch list of a literal l contains all clauses in which the literal l is the watched literal.

This scheme is not used in state-of-the-art solvers, because it is subsumed by a more powerful, but more complicated, *two-watch literals scheme* that also enables efficient and complete detection of unit clauses. We mention it here as an aid to understand the two-watch literals scheme. More details on the one-watch scheme can be found in the extended version of this paper.

4.9 Two-Watch Literals Scheme

For an efficient unit propagation implementation, the test `∃c.∃l. c` $\in F \wedge$ `(isUnit c l` M`)` has to be made effective and efficient. Again, a straightforward checking of each clause in F for the presence of unit literals is out of question, for efficiency reasons. The *two-watch literals scheme*, which efficiently detects both falsified and unit clauses, follows the ideas from §4.8. A clause c cannot be unit in a valuation v if it contains a true literal or contains at least two unfalsified literals. So, two different literals from each clause are marked as its watched literals, and the clause has to be checked only when one of its watches becomes falsified.

Watch literals data structure can be seen as a mapping from clauses to literals. Still, most implementations do not store watches in a separate map.

Usually, either the data type `Clause` is augmented and is a record containing both the list of literals and the watched literals, or, more often, the convention is that the first and second literal in the clause are its watches. Regardless of the actual watch literal representation, we will denote them by (watch1 c) and (watch2 c). As described in §4.7, the current set of clauses F can contain only clauses with two or more literals, which significantly simplifies implementation. The unit clauses and the corresponding unit literals found by this procedure are put in the unit propagation queue Q, from where they are picked and asserted. The flag $conflictFlag$ is used to inform if a conflict has been detected.

Clauses are accessed only when one of their watched literals gets falsified, and then their other literals are examined to determine if the clause has become unit, falsified, or (in the meanwhile) satisfied. If neither of these is the case, then its watch literals are updated. To simplify implementation, if (watch1 c) gets falsified, then watches are swapped, so we can assume that the falsified literal is always (watch2 c). The following cases are possible:

1. If it can be quickly detected that the clause contains a true literal t, there is no need to change the watches. The rationale is the following; for this clause to become unit or false t must be backtracked from M. At this point, since the watch became false after t, it would also become unfalsified. It remains open how to check if a true literal t exists. Older solvers checked only if (watch1 c) is true in M. Newer solvers cache some arbitrary literals and check if they are true in M. These literals are stored in a separate data structure that is most of the time present in cache. This avoids accessing the clause itself and leads to significantly better performance.
2. If a quick check did not detect a true literal t, then other literals are examined.
   (a) If there exist a non-watched literal l not false in M, then it becomes the new (watch2 c). At this point, (watch1 c) cannot be false in M. Indeed, if it was false, at the time when it became falsified, then it would be first swapped with (watch2 c) (which could not be false for the same reasons), and then l would become (watch2 c).
   (b) If all non-watched literals are false in M, but (watch1 c) is undefined, then the clause just became a unit clause and (watch1 c) is enqueued in Q for propagation. Watches are not changed. The rationale for this is that after unit propagation watches would be the last two literals that defined in M and for this clause to become unit or falsified again, the trail must be backtracked, and the watches would become undefined.
   (c) If all non-watched literals and (watch1 c) are false in M, then the whole clause is false and $conflictFlag$ is raised. Watches are not changed. The rationale for this the same as in the previous case.

Under the assumption that unit propagation is eagerly performed and no decisions are made after conflict clause is detected, after taking prefix of the trail during the backjump operation, there would be no falsified clauses and the only unit clause would be the backjump clause that is being learned. This is an important feature of this scheme, as it allows constant-time backjumping.

The following table summarizes the effect of the notifyWatches function. Letters T, F and U denote true, false and undefined literals, respectively.

before assert(M)			after assert(M')			after notify(M')			*effect*
(w1 c)	(w2 c)	*other*	(w1 c)	(w1 c)	*other*	(w1 c)	(w2 c)	*other*	
T	U	U/T	T	F	U/T	T	F	U/T	
U	T	U/T	F	T	U/T	T	F	U/T	swap((w1 c), (w1 c))
U	U	U/T	U	F	U/T	U	U/T	?	(w2 c) := *other*
U	U	U/T	F	U	U/T	U	U/T	?	swap((w1 c), (w1 c)), (w2 c) := *other*
T	U	F	T	F	F	T	F	F	
U	T	F	F	T	F	T	F	F	swap((w1 c), (w1 c))
U	U	F	U	F	F	U	F	F	$Q := Q$ @ (w1 c)
U	U	F	F	U	F	U	F	F	swap((w1 c), (w1 c)), $Q := Q$ @ (w1 c)
U	F	F	F	F	F	F	F	F	swap((w1 c), (w1 c)), $conflictFlag := true$

The described process is the essence of the notifyWatches procedure given in the pseudo-code that follows.

SAT solver v.9

```
...
$Q$ : Literal list                                                    +
$conflictFlag$ : Boolean                                              +


function addClause(clause : Clause)
begin
...
   else begin
      setWatch1(clause, (first clause));                            +
      setWatch2(clause, (second clause));                           +
      $F$ := $F$ @ clause
   end
end
function assertLiteral(l : Literal, decision : Boolean)
begin
   $M$ := $M$ @ [(l, decision)];
   notifyWatches($\overline{l}$)                                     +
end
function notifyWatches (l : Literal)                                +
begin                                                               +
   foreach clause : clause $\in F \wedge$                            +
            ((watch1 clause) = l $\vee$ (watch2 clause) = l) do begin +
      if (watch1 clause) = l then                                   +
         swapWatches(clause);                                       +
      if $M \nvDash$ (watch1 clause) then                            +
         if ($\exists$l'. isNonWatchedUnfalisfiedLiteral(l', clause)) then +
            setWatch2(clause, l')                                   +
         else if $M \vDash \neg$ (watch1 clause) then begin          +
            $conflictFlag$ := true;                                 +
            $conflictClause$ := clause                              +
         end else                                                   +
            if (watch1 clause) $\notin Q$ then                       +
               $Q$ := $Q$ @ (watch1 clause)                          +
   end                                                              +
end                                                                 +
const function isNonWatchedUnfalsifiedLiteral(l : Literal, clause : Clause)     +
begin                                                                           +
   ret := l $\in$ clause  $\wedge$                                              +
         l $\neq$ (watch1 clause)  $\wedge$  l $\neq$ (watch2 clause)  $\wedge$ +
         $M \not\vDash\neg$ l                                                   +
end                                                                             +
function buildC()
begin
   $C$ := $C_P$ @ $\overline{C_l}$
   if $C_P \neq$ [] then begin                                                  +
      setWatch1($C$, $\overline{C_l}$);                                         *
      setWatch2($C$, $\overline{(\text{lastAssertedLiteral}\ \overline{C_P}\ M)}$)  *
   end                                                                          +
end
function applyUnitPropagate() : Boolean
begin
   if $Q$ = [] then                                                             *
      ret := False                                                              *
   else begin                                                                   *
      assertLiteral ((head $Q$), false);                                        *
      $Q$ := (tail $Q$);                                                        *
      ret := True                                                               *
   end                                                                          *
end
function applyBackjump()
begin
   level := getBackjumpLevel();
   $M$ := (prefixToLevel $M$ level);
   $conflictFlag$ := False;
   $Q$ := [$\overline{C_l}$]                                                    +
end
function isReason(c : Clause) : Boolean
begin
   if getReason((watch1 c)) = c then ret := true else ret := false
end
```

Some implementations treat the unit propagation queue Q as an unprocessed part of the trail M. This modification leads to earlier detection of conflicts. We could implement it by replacing all tests of the type $M \vDash$ and $M \vDash\neg$ in the code of SAT solver v.9 with $M\,@\,Q \vDash$ and $M\,@\,Q \vDash\neg$, respectively. We will not do it, however, in order to avoid complicating the proofs.

Correctness. The invariants that describe $conflictFlag$ and Q are:

$Invariant_{conflictFlag}$: $conflictFlag \iff M \vDash\neg\, F$
$Invariant_{unitQueue}$: $\neg conflictFlag \implies \big(\forall l.\ l \in Q \iff \exists c.\ c \in F\ \wedge\ (isUnit\ c\ l\ M)\big)$

Note that the $Invariant_{unitQueue}$ also guarantees completeness of the unit propagation which is not needed for correctness but is important for efficiency.

In order to prove that $Invariant_{conflictFlag}$ and $Invariant_{unitQueue}$ hold, several other invariants are formulated. First, two watched literals have to be different:

$Invariant_{watchesDiffer}$: $\forall\ c.\ c \in F \implies (\text{watch1}\ c) \neq (\text{watch2}\ c)$

All clauses in F satisfy $Invariant_{watch}$ (with $(\mathsf{watch\_}i\ c)$ set both to $(\mathsf{watch1}\ c)$ and $(\mathsf{watch2}\ c)$):

$$
\begin{array}{l}
M \models\neg (\mathsf{watch\_}i\ c) \implies \\
(\exists l.\ M \models l \wedge \mathsf{level}\ l \leq \mathsf{level}\ \overline{(\mathsf{watch\_}i\ c)}) \vee \\
(\forall l.\ l \neq (\mathsf{watch1}\ c) \wedge l \neq (\mathsf{watch2}\ c) \implies M \models\neg l \wedge \mathsf{level}\ \overline{l} \leq \mathsf{level}\ \overline{(\mathsf{watch\_}i\ c)})
\end{array}
$$

These invariants can become temporarily invalid when a literal is asserted in M, but they are restored after the $\mathsf{notifyWatches}$ function call. The precondition of the $\mathsf{assertLiteral}(l, d)$ function call is that decisions are made only if there are no false or unit clauses in F with regards to M, i.e., $d \implies \neg conflictFlag \wedge Q = [\,]$. The proofs of soundness lemmas from the previous sections remain valid and Theorem 1 holds.

4.9.1 Watch Lists

In order to make an efficient implementation, watch lists are used:

SAT solver v.10

```
function setWatch1(clause : Clause, l : Literal)
begin
   (watch1 clause) := l
   $W$(l) := $W$(l) @ clause                                          +
end
function setWatch2(clause : Clause, l : Literal)
begin
   (watch2 clause) := l
   $W$(l) := $W$(l) @ clause                                          +
end
function notifyWatches (l : Literal)
begin
   newWL := [];                                                       +
   foreach clause : clause $\in$ $W$(l)                               *
      if (watch1 clause) = l then
         swapWatches(clause);
      if $M \models$ (watch1 clause) then                             +
         newWL := newWL @ clause                                      +
      else
         if ($\exists$l'. isNonWatchedUnfalisfiedLiteral(l', clause)) then
            setWatch2(clause, l')
         else if $M \models\neg$ (watch1 clause) then begin
            $conflictFlag$ := true;
            $conflictClause$ := clause;
            newWL := newWL @ clause                                   +
         end else begin
            $Q$ := $Q$ @ (watch1 clause);
            newWL := newWL @ clause                                   +
         end
   end;
   $W$(l) := newWL                                                    +
end
```

```
function removeClause(c : Clause)
begin
    $W((\mathsf{watch1}\ c)) := W((\mathsf{watch1}\ c)) \setminus c$;                         *
    $W((\mathsf{watch2}\ c)) := W((\mathsf{watch2}\ c)) \setminus c$                          *
end
```

When a literal stops being watched, its watch list must be updated, and the corresponding clause should be removed from the list. In one notifyWatches function call, this list is traversed and many such remove operations can happen. It turns out that it is more efficient to regenerate the watch list of the falsified literal, than to perform these remove operations, so the clauses for which this watch did not change, are reinserted in the watch list `newWl`. This looks as a strange solution, but is in fact an important feature. Being the heart of boolean constraint propagation, the notifyWatches function is crucial for the solver's efficiency, because the solver can spend up to 80% of time in it [ES04].

Correctness. Invariant that describes watch lists is:

$$Invariant_{watchLists} : c \in W(l) \ \Leftrightarrow\ c \in F \ \wedge\ ((\mathsf{watch1}\ c) = l \vee (\mathsf{watch2}\ c) = l)$$

5 Related work

Detailed surveys of SAT solver development can be found in [GKSS07, ZM02, BHZ06]. There is about forty-five years of research invested in DPLL-based SAT solvers. Earlier SAT solvers based on DPLL include Tableau (NTAB), POSIT, 2cl and CSAT, among others[GKSS07]. In the last fifteen years, there has been a significant growth and success in SAT solver research based on the DPLL framework. Many practical applications emerged, which pushed these solvers to their limits and provided strong motivation for finding even more efficient algorithms. In the mid 1990's, this led to a new generation of solvers such as SATO [Zha97], Chaff [MMZ^{+}01], and BerkMin [GN02] which pay a lot of attention to optimizing various aspects of the DPLL algorithm. Annual SAT competitions have led to the development of dozens of clever implementations of such solvers, exploration of many new techniques, and the creation of an extensive suite of real-world instances as well as challenging hand-crafted benchmark problems. Some of the most successful DPLL-based solvers in recent competitions are Rsat [PD07], Picosat[Bie08], Minisat [ES04], Tinisat [Hua07a], etc. SAT4J is a solver implemented in JAVA in which attention has been put on the code design.

Non-chronological backtracking (conflict-directed backjumping), was proposed first in the Constraint Satisfaction Problem (CSP) domain [BHZ06]. This, together with conflict-driven learning were first incorporated into a SAT solver in the mid 1990's by Silva and Sakallah in GRASP [MSS99], and by Bayardo and Schrag in rel_sat [BS97]. Conflict clause minimization was introduced by Eén and Sörensson [ES04] in their solver Minisat. Randomized restarts were introduced by Gomes et al. [GSK98] and further developed by Baptista and Marques-Silva [BMS00]. The watch literals scheme by Moskewicz et al. was introduced in their solver zChaff [MMZ$^+$01], and is now a standard method used by most SAT solvers for efficient constraint propagation.

The first correctness proof of DPLL with clause learning is given in [ZM03]. The given proof is informal and omits a lot of important details. Formal descriptions of SAT solvers in a form of state-transition systems were given in [NOT06,KG07], together with descriptions of related SMT Solvers. These papers contain correctness proof, although presented systems hide many important implementation aspects.

6 Conclusions

In this paper we have developed the core of SAT solver implementation based on a modified DPLL algorithm. We tried to make a clean separation between different concepts, and all techniques, algorithms, and data-structures were introduced one-by-one, in separate sections. The code follows higher level solver descriptions [KG07] and is, in our opinion, easier to understand, modify and reason about than the code available in existing presentations (e.g., [ES04]). Heuristic components of the solver were not investigated. We hope that this methodological approach makes this paper usable as a tutorial and that it could significantly help interested reader in understanding and learning details of modern SAT solver implementation, especially to researchers that are new to this field. The solver descriptions, although given in pseudo-code, could be converted to a real programming language implementation.

The main part of the paper also provides conditions that have to be proved to ensure solver's partial correctness. Proofs of correctness conditions are available in the extended version of this paper and original Isabelle proof documents [Mar08]. To the best of our knowledge, this paper gives the first formalization and correctness proof for some low-level implementation techniques, most notably the two-watch literal scheme. We hope that these proofs could help in better understanding what conditions are sufficient for solver correctness. For example, in [NOT06] the paper [ZM03] is criticized for claiming that the new decisions should not be made in the presence of conflict or unit clauses. From our formalization, it is clear that this is an important invariant of the two-watch literal propagation scheme, and that, although it need not hold for the more abstract rule-based system, it has to hold to ensure the correctness if the two-watch literal propagation is used.

Acknowledgements The author would like to thank Dr. Predrag Janičić and anonymous reviewers for carefully reading the manuscript and giving valuable suggestions and advices.

References

[Bar03] C. Barrett. Checking validity of quantifier-free formulas in combinations of first-order theories. In *Ph.D. thesis, Stanford University*, 2003.

[Bie08] A. Biere. PicoSAT essentials. In *JSAT*, vol. 4, pp. 75-97, 2008.

[BHZ06] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. In *ACM Surveys*, 2006.

[BKS04] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[BMS00] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *6th CP*, Singapore, 2000.

[BS97] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *14th AAAI*, pp. 203–208, Providence, USA, 1997.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *3rd STOC*, pp. 151–158, New York, USA, 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. Een and N. Sorensson. An extensible SAT solver. *Theory and Applications of Satisfiability Testing.* pp. 502–518. 2004.

[GKSS07] C P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*. Elsevier, 2007. to appear.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver, *Design Automation and Test in Europe (DATE)*, pp. 142–149, 2002.

[GSK98] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *15th AAAI*, pp. 431–437, Madison, WI, USA, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hua07a] J. Huang. A case for simple SAT solvers. In *CP*, pp. 839–846, 2007.

[Hua07b] J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pp. 2318–2323, 2007.

[KG07] S. Krstić and A. Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *FroCos*, pp. 1–27, 2007.

[Mar08] F. Marić, SAT solver verification. *The Archive of Formal Proofs,* `http://afp.sf.net/entries/SATSolverVerification.shtml`.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th DAC*, 2001.

[MS99] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA '99*, pp. 62–74, London, UK, 1999.

[MSS99] J P. Marques-Silva and K A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions 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, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A proof assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PD07] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pp. 294–299, 2007.

[SS96] J. P. Marques Silva and K. A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *8th ICTAI*, pp. 467, Washington DC, USA, 1996.

[Wie03] F. Wiedijk. Comparing mathematical provers. In *MKM03*, pp. 188–202, volume 2594 of LNCS, 2003.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *CADE-14*, pp. 272–275, London, UK, 1997.

[ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *14th CAV*, pp. 17–36, London, UK, 2002.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using independent resolution-based checker. In *DATE'03*, pp. 10880, Washington DC, USA, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01*, pp. 279–285, Piscataway, NJ, USA, 2001.