A Pearl on SAT Solving in Prolog (extended abstract)*

Jacob M. Howe† and Andy King‡

1 Introduction

The Boolean satisfiability problem, SAT, is of continuing interest because a variety of problems are naturally expressible as a SAT instance. Much effort has been expended in the development of algorithms for, and implementations of, efficient SAT solvers. This has borne fruit with a number of solvers that are either for specialised applications or are general purpose.

Recently, it has been demonstrated how a dedicated external SAT solver coded in C can be integrated with Prolog [1] and this has been utilised for a number of applications. This work elegantly uses of Prolog to transform propositional formulae to Conjunctive Normal Form (CNF). The work of [1] begs the question of the suitability of Prolog as a medium for coding a SAT solver, either for use in a stand-alone fashion or in tandem with a constraint solver. In this abstract it is argued that a SAT solver can not only be coded in Prolog, but that this solver is a so-called natural pearl. That is, the key concepts of efficient SAT solving can be formulated in a logic program using a combination of logic and control features that lie at the heart of the logic programming paradigm. This pearl was discovered when implementing an efficient groundness analyser [3], naturally emerging from the representation of Boolean functions using logical variables.

The logic and control features exemplified in this pearl are the use of logical variables, backtracking and the suspension and resumption of execution via delay declarations. A delay declaration is a control mechanism that provides a way to delay the selection of an atom in a goal until some condition is satisfied. They provide a way to handle, for example, negated goals and non-linear constraints. Delay declarations are now an integral part of Prolog systems. This abstract demonstrates just how good the match between Prolog and SAT is, when implementing the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [2] with watched literals [6]. Watched literals are one of the most powerful features in speeding up SAT solvers. The resulting solver is elegant and concise, coded in twenty-two lines of Prolog, it is self-contained and it is efficient enough for solving some interesting, albeit modest, SAT instances [3].

The full version of this note is [5], which extends [4].

2 SAT solving

This section briefly outlines the SAT problem and the DPLL algorithm [2] with watched literals [6] that the solver implements.

The Boolean satisfiability problem is the problem of determining whether or not, for a given Boolean formula, there is a truth assignment to the variables in the formula under which the formula evaluates to true. Most recent Boolean satisfiability solvers have been based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [2]. Figure 1 presents a recursive formulation of the algorithm adapted from that given in [8]. The first argument of the function DPLL is a

---

---

*The work was supported by EPSRC projects EP/E033105/1 and EP/E034519/1 and a Royal Society Industrial Fellowship. This research was conducted in part whilst the first author was on sabbatical leave at the University of St Andrews and the second author was on secondment to Portcullis Computer Security Limited

†Department of Computing, City University London, EC1V 0HB, UK

‡School of Computing, University of Kent, Canterbury, CT2 7NF, UK
propositional formula, \( f \), defined over a set of propositional variables \( X \). As usual \( f \) is assumed to be in CNF. The second argument, \( \theta : X \rightarrow \{\text{true}, \text{false}\} \), is a partial (truth) function. The call \( \text{DPLL}(f, \emptyset) \) decides the satisfiability of \( f \) where \( \emptyset \) denotes the empty truth function. If the call returns the special symbol \( \bot \) then \( f \) is unsatisfiable, otherwise the call returns a truth function \( \theta \) that satisfies \( f \).

### 2.1 Unit propagation

At line (3) the function extends the truth assignment \( \theta \) to \( \theta_1 \) by applying so-called unit propagation on \( f \) and \( \theta \). For instance, suppose \( f = (\neg x \lor z) \land (u \lor \neg v \lor w) \land (\neg w \lor y \lor \neg z) \) so that \( X = \{u, v, w, x, y, z\} \) and \( \theta \) is the partial function \( \theta = \{x \mapsto \text{true}, y \mapsto \text{false}\} \). Unit propagation examines each clause in \( f \) to deduce a truth assignment \( \theta_1 \) that extends \( \theta \) and necessarily holds for \( f \) to be satisfiable. For example, for the clause \( (\neg x \lor z) \) to be satisfiable, and hence \( f \) as a whole, it is necessary that \( z \mapsto \text{true} \). Moreover, for \( (\neg w \lor y \lor \neg z) \) to be satisfiable, it follows that \( w \mapsto \text{false} \). The satisfiability of \( (u \lor \neg v \lor w) \) depends on two unknowns, \( u \) and \( v \), hence no further information can be deduced from this clause. The function \( \text{unit-propagation}(f, \theta) \) encapsulates this reasoning returning the bindings \( \{w \mapsto \text{false}, z \mapsto \text{true}\} \). Extending \( \theta \) with these necessary bindings gives \( \theta_1 \).

### 2.2 Watched literals

Information can only be derived from a clause if it does not contain two unknowns. This is the observation behind watched literals [6], which is an implementation technique for realising unit propagation. The idea is to keep watch on a clause by monitoring only two of its unknowns. Returning to the previous example, before any variable assignment is made suitable monitors for the clause \((u \lor \neg v \lor w)\) are the unknowns \( u \) and \( v \), suitable monitors for \((\neg w \lor y \lor \neg z)\) are \( w \) and \( y \) and \((\neg x \lor z)\) must have monitors \( x \) and \( z \). Note that no more than these monitors are required.

When the initial empty \( \theta \) is augmented with \( x \mapsto \text{true} \), a new monitor for the third clause is not available and unit propagation immediately applies to infer \( z \mapsto \text{true} \). The new binding on \( z \) is detected by the monitors on the second clause, which are then updated to be \( w \) and \( y \). If \( \theta \) is further augmented with \( y \mapsto \text{false} \), the change in \( y \) is again detected by the monitors on \((\neg w \lor y \lor \neg z)\). This time there are no remaining unbound variables to monitor and unit propagation applies, giving the binding \( w \mapsto \text{false} \). Now notice that the first clause, \((u \lor \neg v \lor w)\), is not monitoring.
w, hence no action is taken in response to the binding on w. Therefore, watched literals provide a mechanism for controlling propagation without inspecting clauses unnecessarily.

2.3 Termination and the base cases

Once unit propagation has been completely applied, it remains to detect whether sufficient variables have been bound for f to be satisfiable. This is the role of the predicate is-satisfied(f, θ). This predicate returns true if every clause of f contains at least one literal that is satisfied. For example, is-satisfied(f, θ₁) = false since (u ∨ ¬v ∨ w) is not satisfied under θ₁ because u and v are unknown whereas w is bound to false. If is-satisfied(f, θ₁) were satisfied, then θ₁ could be returned to demonstrate the existence of a satisfying assignment.

Conversely, a conflict can be observed when inspecting f and θ₁, from which it follows that f is unsatisfiable. To illustrate, suppose f = (¬x) ∧ (x ∨ y) ∧ (¬y) and θ = ∅. From the first and third clauses it follows that θ₁ = {x → false, y → false}. The predicate is-conflicting(f, θ) detects whether f contains a clause in which every literal is unsatisfiable. The clause (x ∨ y) satisfies this criteria under θ₁, therefore it follows that f is unsatisfiable, which is indicated by returning ⊥.

2.4 Search and the recursive cases

If neither satisfiability nor unsatisfiability have been detected thus far, a variable x is selected for labelling. The DPLL algorithm is then invoked with θ₁ augmented with the new binding x → true. If satisfiability cannot be detected with this choice, DPLL is subsequently invoked with θ₁ augmented with x → false. Termination is assured because the number of unassigned variables strictly reduces on each recursive call.

3 The SAT Solver

The code for the solver is given in Figure 2. It consists of just twenty-two lines of Prolog. Since a declarative description of assignment and propagation can be fully expressed in Prolog, execution can deal with all aspects of controlling the search, leading to the succinct code given.

3.1 Invoking the solver

The solver is called with two arguments. The first represents a formula in CNF as a list of lists, each constituent list representing a clause. The literals of a clause are represented as pairs, Pol-Var, where Var is a logical variable and Pol is true or false, indicating that the literal has positive or negative polarity. The formula ¬x ∨ (y ∧ ¬z) would thus be represented in CNF as (¬x ∨ y) ∧ (¬x ∨ ¬z) and presented to the solver as the list Clauses = [[false-X, true-Y], [false-X, false-Z]] where X, Y and Z are logical variables. The second argument is the list of the variables occurring in the problem. Thus the query sat(Clauses, [X, Y, Z]) will succeed and bind the variables to a solution, for example, X = false, Y = true, Z = true. As a by-product, Clauses will be instantiated to [[false-false, true-true], [false-false, false-true]]. This illustrates that the interpretation of true and false in Clauses depends on whether they are left or right of the - operator: to the left they denote polarity; to the right they denote truth values. If Clauses is unsatisfiable then sat(Clauses, Vars) will fail. If necessary, the solver can be called under a double negation to check for satisfiability, whilst leaving the variables unbound.

3.2 Watched literals

The solver is based on launching a watch goal for each clause that monitors two literals of that clause. Since the polarity of the literals is known, this amounts to blocking execution until one of the two uninstantiated variables occurring in the clause is bound. The watch predicate thus blocks on its first and third arguments until one of them is instantiated to a truth value. In SICStus Prolog, this requirement is stated by the declaration :- block watch(-, ?, -, ?, ?). If the
sat(Clauses, Vars) :-
    problem_setup(Clauses), elim_var(Vars).

elim_var([]).
elim_var([Var | Vars]) :-
    elim_var(Vars), assign(Var).

assign(true).
assign(false).

problem_setup([]).
problem_setup([Clause | Clauses]) :-
    clause_setup(Clause),
    problem_setup(Clauses).

clause_setup([Pol-Var | Pairs]) :- set_watch(Pairs, Var, Pol).

set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1) :-
    watch(Var1, Pol1, Var2, Pol2, Pairs).

:- block watch(-, ?, -, ?, ?).
watch(Var1, Pol1, Var2, Pol2, Pairs) :-
    nonvar(Var1) ->
        update_watch(Var1, Pol1, Var2, Pol2, Pairs);
        update_watch(Var2, Pol2, Var1, Pol1, Pairs).

update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
    Var1 == Pol1 -> true; set_watch(Pairs, Var2, Pol2).

Figure 2: Code for SAT solver

first argument is bound, then update_watch will diagnose what action, if any, to perform based on the polarity of the bound variable and its binding. If the polarity is positive, and the variable is bound to true, then the clause has been satisfied and no further action is required. Likewise, the clause is satisfied if the variable is false and the polarity is negative. Otherwise, the satisfiability of the clause depends on those variables of the clause which have not yet been inspected. They are considered in the subsequent call to set_watch.

3.3 Unit propagation

The first clause of set_watch handles the case when there are no further variables to watch. If the remaining variable is not bound, then unit propagation occurs, assigning the variable a value that satisfies the clause. If the polarity of the variable is positive, then the variable is assigned true. Conversely, if the polarity is negative, then the variable is assigned false. A single unification is sufficient to handle both cases. If Var and Pol are not unifiable, then the bindings to Vars do not satisfy the clause, hence do not satisfy the whole CNF formula.

Once problem_setup(Clauses) has launched a process for each clause in the list Clauses, elim_var(Vars) is invoked to bind each variable of Vars to a truth value. Control switches to a watch goal as soon as its first or third argument is bound. In effect, the sub-goal assign(Var) of elim_var(Vars) coroutines with the watch sub-goals of problem_setup(Clauses). Thus, for instance, elim_var(Vars) can bind a variable which transfers control to a watch goal that is waiting on that variable. This goal can, in turn, call update_watch and thus invoke set_watch,
the first clause of which is responsible for unit propagation. Unit propagation can instantiate another variable, so that control is passed to another watch goal, thus leading to a sequence of bindings that emanate from a single binding in elim_var(Vars). Control will only return to elim_var(Vars) when unit propagation has been maximally applied.

3.4 Search

In addition to supporting coroutining, Prolog permits a conflicting binding to be undone through backtracking. Suppose a single binding in elim_var(Vars) triggers a sequence of bindings to be made by the watch goals and, in doing so, the watch goals encounter a conflict: the unification $Var = Pol$ in set_watch fails. Then backtracking will undo the original binding made in elim_var(Vars), as well as the subsequent bindings made by the watch goals. The watch goals themselves are also rewound to their point of execution immediately prior to when the original binding was made in elim_var(Vars). The goal elim_var(Vars) will then instantiate Vars to the next combination of truth values, which may itself cause a watch goal to be resumed, and another sequence of bindings to be made. Thus monitoring, propagation and search are seamlessly interwoven.

Note that the sub-goal assign(Var) will attempt to assign Var to true before trying false, which corresponds to the down strategy in finite-domain constraint programming. Moreover, the variables Vars of sat(Clauses, Vars) are instantiated in the left-to-right order. Returning to the initial query where Clauses = [[false-X, true-Y], [false-X, false-Z]], backtracking can enumerate all the satisfying assignments to give:

$$X = \text{false}, \quad Y = \text{true}, \quad Z = \text{false};$$
$$X = \text{true}, \quad Y = \text{true}, \quad Z = \text{false};$$
$$X = \text{false}, \quad Y = \text{false}, \quad Z = \text{true};$$
$$X = \text{false}, \quad Y = \text{true}, \quad Z = \text{false};$$
$$X = \text{false}, \quad Y = \text{false}, \quad Z = \text{false}.$$  

4 Extensions

The full paper [5] develops the solver to provide an easy entry into SAT and SMT solving for the Prolog programmer. For instance, the solver can be enhanced with a technique to avoid replicating search when the solver is applied incrementally in conjunction with, say, learning. This dovetails with the lazy-basic instance of SMT [7] which, when applied with a technique for finding an unsatisfiable core of a system of unsatisfiable constraints, provides a neat way of realising an SMT solver. Developing [1], it is argued that Prolog also aids the translation of formulae over theory literals that involve constraints into the SMT equivalent of CNF. The full paper also discusses further extensions whilst remarking on the limitations of the solver and its approach.

Finally, the solver and other related code is available at www.soi.city.ac.uk/~jacob/solver/.

References


