A Pearl on SAT Solving in Prolog

Jacob M. Howe[1] and Andy King[2]

[1] Department of Computing, City University London, EC1V 0HB, UK
[2] School of Computing, University of Kent, CT2 7NF, UK

Abstract. A succinct SAT solver is presented that exploits the control provided by delay declarations to implement watched literals and unit propagation. Despite its brevity the solver is surprisingly powerful and its elegant use of Prolog constructs is presented as a programming pearl.

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 [5].

Recently, it has been demonstrated how a dedicated external SAT solver coded in C can be integrated with Prolog [2] and this has been utilised for a number of applications. This work was published as a pearl owing to its elegant use of Prolog to transform logical formulae to Conjunctive Normal Form (CNF). This work begs the question of the suitability of Prolog as a medium for coding a SAT solver. In this short paper 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 [11] that lie at the heart of the paradigm. This pearl was discovered when implementing an efficient groundness analyser [8], naturally emerging from the representation of Boolean functions using logical variables; the solver has not been previously described.

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 [15]. 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, though their centrality in the paradigm has only recently been formally established [10]. This paper demonstrates just how good the match between Prolog and SAT is, when implementing the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [3] with watched literals [14]. Watched literals are one of the most powerful features in speeding up SAT solvers. The resulting solver is elegant and concise, coded in twenty lines of Prolog, it is self-contained and it will be argued that it is efficient

```
(1)   function DPLL(f: CNF formula, θ : truth assignment)
(2)   begin
(3)       θ1 := θ ∪ unit-propagation(f, θ);
(4)       if (is-satisfied(f, θ1)) then
(5)           return θ1;
(6)       else if (is-conflicting(f, θ1)) then
(7)           return ⊥;
(8)       endif
(9)       x := choose-free-variable(f, θ1);
(10)      θ2 := DPLL(f, θ1 ∪ {x ↦ true});
(11)      if (θ2 ≠ ⊥) then
(12)          return θ2;
(13)      else
(14)          return DPLL(f, θ1 ∪ {x ↦ false});
(15)      endif
(16)  end
```

Fig. 1. Recursive formulation of the DPLL algorithm

enough for solving some interesting, albeit modest, SAT instances [8, 9]. The solver can be further developed in a number of ways, a few of which are discussed here, and provides an easy entry into SAT solving for the Prolog programmer. The rest of the paper contains a short summary of relevant background on SAT solving, gives the code for the solver and comments upon it, presents a short empirical evaluation to demonstrate its power, discusses extensions to the solver and concludes with a discussion of the limitations of the solver and its approach.

2 Background

This section briefly outlines the SAT problem and the DPLL algorithm with watched literals [14] 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 under which the formula evaluates to true. Most recent Boolean satisfiability solvers have been been based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [3]. Figure 1 presents a recursive formulation of the algorithm adapted from that given in [16]. The first argument of the function DPLL is a formula, f, defined over a set of propositional variables X. As usual f is assumed to be in CNF. The second argument, θ, is a partial (truth) function over $X \rightarrow \{true, false\}$. The call DPLL($f$, $\emptyset$) decides the satisfiability of f where $\emptyset$ denotes the empty truth function. If the call returns the special symbol $\perp$ then f is unsatisfiable, otherwise the call returns a truth function θ that satisfies f.

2.1 Unit propagation

At line (3) the function extends the truth assignment θ to θ_1 by applying so-called unit propagation on f and θ. For instance, suppose $f = (\neg x \vee z) \wedge (u \vee \neg v \vee w) \wedge (\neg w \vee y \vee \neg z)$ so that $X = \{u, v, w, x, y, z\}$ and θ is the partial function $\theta = \{x \mapsto true, y \mapsto false\}$. Unit propagation examines each clause in f to deduce a truth assignment θ_1 that extends θ and necessarily holds for f to be satisfiable. For example, for the clause $(\neg x \vee z)$ to be satisfiable, and hence f as a whole, it is necessary that $z \mapsto true$. Moreover, for $(\neg w \vee y \vee \neg z)$ to be satisfiable, it follows that $w \mapsto false$. The satisfability of $(u \vee \neg v \vee w)$ depends on two unknowns, u and v, hence no further information can be deduced from this clause. The function unit-propagation(f, θ) encapsulates this reasoning returning the bindings $\{w \mapsto false, z \mapsto true\}$. Extending θ with these necessary bindings gives θ_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 [14], 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 $(u \vee \neg v \vee w)$ are the unknowns u and v, suitable monitors for $(\neg w \vee y \vee \neg z)$ are w and z and $(\neg x \vee z)$ must have monitors x and z. When the initial empty θ is augmented with $x \mapsto true$, a new monitor for the third clause is not available and unit propagation immediately applies to infer $z \mapsto true$. The new binding on z is detected by the monitors on the second clause, which are then updated to be w and y. If θ is further augmented with $y \mapsto false$, the change in y is again detected by the monitors on $(\neg w \vee y \vee \neg z)$. This time there are no remaining unbound variables to monitor and unit propagation applies, giving the binding $w \mapsto false$. Now notice that the first clause, $(u \vee \neg v \vee w)$, 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 be 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, θ_1) = false since $(u \vee \neg v \vee w)$ is not satisfied under θ_1 because u and v are unknown whereas w is bound to false. If is-satisfied(f, θ_1) were satisfied, then θ_1 could be returned to demonstrate the existence of a satisfying assignment.

Conversely, a conflict can be observed when inspecting f and θ_1, from which it follows that f is unsatisfiable. To illustrate, suppose $f = (\neg x) \wedge (x \vee y) \wedge (\neg y)$

```
sat(Clauses, Vars) :-
        problem_setup(Clauses), elim_var(Vars).

elim_var([]).
elim_var([Var | Vars]) :-
        elim_var(Vars), (Var = true; Var = 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).
```

Fig. 2. Code for SAT solver

and $\theta = \emptyset$. From the first and third clauses it follows that $\theta_1 = \{x \mapsto false, y \mapsto false\}$. The predicate is-conflicting(f, θ) detects whether f contains a clause in which every literal is unsatisfiable. The clause $(x \vee y)$ satisfies this criteria under θ_1, therefore it follows that f is unsatisfiable, which is indicated by returning $\bot$.

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 θ_1 augmented with the new binding $x \mapsto true$. If satisfability cannot be detected with this choice, DPLL is subsequently invoked with θ_1 augmented with $x \mapsto false$. 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 give in Figure 2. It consists of just twenty 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 in the figure.

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 $\neg x \vee (y \wedge \neg z)$ would thus be represented in CNF as $(\neg x \vee y) \wedge (\neg x \vee \neg z)$ and presented to the solver as the list `L = [[false-X, true-Y], [false-X, false-Z]]` where `X`, `Y` and `Z` are logical variables. The second argument is a list of the variables occurring in the problem. Thus the query `sat(L, [X, Y, Z])` will succeed and bind the variables to a solution, for example, `X = false`, `Y = true`, `Z = true`. As a by-product, `L` will be instantiated to `[[false-false, true-true], [false-false, false-true]]`. This illustrates that the interpretation of `true` and `false` in `L` 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 `L` is unsatisfiable then `sat(L, 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 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 `(Var = true; Var = false)` sub-goals of `elim_vars(Vars)` coroutine 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 eminate from a single binding in `elim_vars(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 backtracking can enumerate all the satisfying assignments, unlike most SAT solvers (therefore also [2]). For example, the query `sat(L, [X, Y, Z])` will give the solutions:

```
X = false, Y = true, Z = true;   X = false, Y = false, Z = true;
X = true,  Y = true, Z = false;  X = false, Y = true,  Z = false;
```

and `X = false, Y = false, Z = false`.

4 Extensions

The development of SAT solvers over the last decade has resulted in numerous heuristics that dramatically improve the performance of general purpose solvers. This section outlines how a number of these refinements might be incorporated into the solver presented above. However, discussion of the popular learning heuristic [13] is left until section 6 as its integration into the solver is more problematic.

– The first and simplest heuristic is to use a static variable ordering. Variables are ordered by frequency of occurrence in the input problem, with the most frequently occurring assigned first. This wins in two ways: the problem size is quickly reduced by satisfying clauses and the amount of propagation achieved is greater. Both reduce the number of assignments required to reach a satisfying assignment or a conflict. This tactic, of course, can be straightforwardly implemented in Prolog.
– Static variable ordering is the simplest preprocessing tactic aimed at discovering and exploiting structure in a problem. As well as analysing the structure of a problem, another popular tactic is to change the problem by restructuring it using limited applications of resolution steps [4]. Again, these preprocessing steps can clearly be achieved satisfactorily in Prolog.
– Many SAT solvers use non-chronological backtracking, or backjumping, in order to avoid exploration of fruitless branches of the search tree [13]. Backjumping for depth-first search algorithms in Prolog has been explored in [1] and this approach carries over to the solver presented in this paper.
– Dynamically ordering variables during search [14] has also been widely incorporated in SAT solvers. This too can be incorporated into the solver presented in this paper. The approach has some similarities to the backjumping of [1] using the blackboard to hold conflict information which is then used after backtracking to select the next variable for assignment.

5 Experimental Results

In order to illustrate the problems that the solver can tackle, empirical results for a small benchmark suite are included and are tabulated below. In the table, *benchmark* names the SAT instance, whilst *vars* and *clauses* give the number of variables and clauses respectively, *satisfiable* indicates whether or not the instance is satisfiable, *time* gives the time taken to find a first satisfying assignment, or to establish that no such assignment exist, *mini* gives the time for the benchmark using MiniSat and *assigns* gives the total number of variable assignments made in `elim_var`. The MiniSat results have been included for reference and are unsurprisingly considerably faster, owing to its C implementation and use of many heuristics. Also note that the timing granularity of SICStus is different to MiniSat. The implementation is in SICStus 4.0.4 and these experiments were run on a single core of a MacBook with a 2.4GHz Intel Core 2 Duo processor and 4GB of memory. Timeout was set at one minute. Finally, note that these results utilise a static variable ordering as described in section 4 and that this significantly speeds up the solver of these benchmarks.

The first six chat_80 benchmarks are a selection of the largest SAT instances solved in the Pos-based analysis of [8]. It is worth pointing out that these are encodings of entailment checks for stability in fixpoint calculations, therefore are not themselves necessarily positive Boolean formulae (which would be satisfiable by definition). The remaining benchmarks come from [7]. The uf* and uuf* benchmarks are random 3SAT instances at the phase transition boundary and

benchmark	*vars*	*clauses*	*satisfiable*	*time (ms)*	*mini (ms)*	*assigns*
chat_80_1.cnf	13	31	true	0	1	9
chat_80_2.cnf	12	30	true	0	1	5
chat_80_3.cnf	8	14	true	0	1	7
chat_80_4.cnf	7	16	true	0	1	3
chat_80_5.cnf	7	16	true	0	1	4
chat_80_6.cnf	8	14	true	0	1	6
uf20-0903.cnf	20	91	true	0	1	8
uf50-0429.cnf	50	218	true	10	1	89
uf100-0658.cnf	100	430	true	20	1	176
uf150-046.cnf	150	645	true	290	15	3002
uf250-091.cnf	250	1065	true	2850	171	13920
uuf50-0168.cnf	50	218	false	0	1	79
uuf100-0592.cnf	100	430	false	50	6	535
uuf150-089.cnf	150	645	false	770	18	8394
uuf250-016.cnf	250	1065	false	t/o	1970	
2bitcomp_5.cnf	125	310	true	130	1	7617
flat200-90.cnf	600	2237	true	380	12	1811

Fig. 3. Experimental evaluation of the SAT solver

are included to illustrate behaviour on problems likely to involve large amounts of search; the individual problems were chosen at random from larger suites. The remaining problems were chosen to illustrate behaviour on structured problems.

Observe that the problems arising from the context where this solver was discovered are all solved quickly, requiring very few variable assignments. On these problems, where there can be thousands of calls to the solver in a single run of the analysis [8], the time to solve the larger SAT instances are beneath the granularity of the clock, thus the solver is clearly fast enough. As expected, on the phase transition problems the amount of search grows sharply with the size of the problem. However, instances with hundreds of clauses are still solved, and this observation is confirmed by the results for the structured problems.

6 Concluding Discussion

Thus far this paper has highlighted the ways in which Prolog provides an easy entry point into SAT solving. This section begins by highlighting the limitations of the approach taken, before concluding with a discussion of the strengths of this implementation technique.

The challenge of SAT solving grows with the size of the problem. This can manifest itself in two ways: the storage of the SAT instance and the growth of the search space. The first of these is perhaps the greatest obstacle to solving really large problems in Prolog – the programmer does not have the fine-grained memory control required to store and access hundreds of thousands of clauses. To address the second issue search heuristics, such as those outlined in section 4,

are needed. One popular kind of heuristic is learning in which clauses are added to the problem that express regions of the search space that do not contain a solution [13]. Unfortunately, it is not clear how to achieve this cleanly in this Prolog solver, as calls to the learnt clauses would be lost on backtracking. One approach would be to store a learnt clause on a blackboard and then add it to the problem at an appropriate point on backtracking, but the approach is both restrictive and unattractive (although it does fit well with random restarts that are used in some solvers). A final point to note is in the implementation of watched literals. The literals being watched change during search and changes made during propagation are undone on backtracking. This makes maintenance of the clauses easy, but loses one advantage that watched literals potentially have, namely that the literals being watched do not need to be changed on backtracking [6].

Owing to the drawbacks outlined above, the solver presented in this paper is not going to be competitive on the large, difficult problems set as challenges presented in the international SAT competitions [12]. However, the solver does provide a declarative description of SAT solving with watched literals in a succinct and self-contained manner, and one which can be extended in a number of ways. In addition it performs well enough to be of use for small and medium-size problems, an example being detecting stability in fixpoint calculations in Pos-based program analysis [8]. In this context, a SAT engine coded in Prolog itself is attractive since it avoids using a foreign language interface (note that [2] hides this interface from the user), simplifies distribution issues, and avoids the overhead of converting a Prolog representation of a SAT instance to the internal C representation used by the external SAT solver.

Finally, the solver is available at `www.soi.city.ac.uk/~jacob/solver/`.

Acknowledgements This work was supported by EPSRC-funded projects EP/E033105/1 and EP/E034519/1. The authors would like to thank the Royal Society and the University of St Andrews for their generous support. They would also like to thank the anonymous referees for their helpful comments.

References

1. M. Bruynooghe. Enhancing a Search Algorithm to Perform Intelligent Backtracking. *Theory and Practice of Logic Programming*, 4(3):371–380, 2004.
2. M. Codish, V. Lagoon, and P. J. Stuckey. Logic Programming with Satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.
3. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.
4. N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *International Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, 2005.
5. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

6. I. P. Gent, C. Jefferson, and I. Miguel. Watched Literals for Constraint Propagation in Minion. In *Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2006.
7. H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT 2000*, pages 283–292. IOS Press, 2000. `http://www.satlib.org`.
8. J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2001.
9. J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.
10. A. King and J. C. Martin. Control Generation by Program Transformation. *Fundamenta Informaticae*, 69(1-2):179–218, 2006.
11. R. A. Kowalski. Algorithm = Logic + Control. *Communication of the ACM*, 22(7):424–436, 1979.
12. D. Le Berre, O. Roussel, and L. Simon. The International SAT Competitions Webpage, 2009. `http://www.satcompetition.org/`.
13. J. P. Marques-Silva and K. A. Sakallah. GRASP – a New Search Algorithm for Satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227. ACM and IEEE Computer Society, 1996.
14. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM Press, 2001.
15. L. Naish. *Negation and Control in Logic Programs*. Springer-Verlag, 1986.
16. L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.