Theoretical Computer Science 435 (2012) 43–55

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A pearl on SAT and SMT solving in Prolog☆

Jacob M. Howe [a], Andy King [b,*]

[a] Department of Computing, City University London, EC1V 0HB, United Kingdom
[b] School of Computing, University of Kent, Canterbury, CT2 7NF, United Kingdom

ARTICLE INFO

Keywords:
SAT solving
SMT solving
Prolog
Constraint logic programming

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. Furthermore, the SAT solver can be integrated into an SMT framework which exploits the constraint solvers that are available in many Prolog systems.

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 [9]. Propositional solvers are either applied to pure SAT instances, or increasingly are combined with constraint solvers in the SAT modulo theories, SMT [27], approach.

Recently, it has been demonstrated how a dedicated external SAT solver coded in C can be integrated with Prolog [5] 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 propositional formulae to Conjunctive Normal Form (CNF). Likewise SMT problems are posed as Boolean formulae combining atomic constraints. The work of [5] 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 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 [20] that lie at the heart of the logic programming paradigm. This pearl was discovered when implementing an efficient groundness analyser [12], naturally emerging from the representation of Boolean functions using logical variables; the solver has not been described prior to [14].

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 [25]. 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 [19]. This paper demonstrates just how good the match between Prolog and SAT is, when implementing the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [6] with watched literals [24]. Watched literals are one of the most powerful features in speeding up SAT solvers. The resulting solver is elegant and

☆ The research 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, Pinner, HA5 2EX, UK.

* Corresponding author.
E-mail addresses: jacob@soi.city.ac.uk (J.M. Howe), a.m.king@kent.ac.uk (A. King).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.02.024

```
(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)       else
(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)      endif
(17)  end
```

Fig. 1. Recursive formulation of the DPLL algorithm.

concise, coded in twenty-two lines of Prolog, it is self-contained and it will be argued that it is efficient enough for solving some interesting, albeit modest, SAT instances [12,13].

The solver can be developed in a number of ways, a few of which are discussed here, and provides an easy entry into SAT and SMT solving for the Prolog programmer. For instance, the solver can be enhanced with a technique based on a so-called black pearl [3] to avoid replicating search when the solver is applied incrementally in conjunction with, say, learning. This dovetails with the lazy-basic instance of SMT [21,27] which, when applied with a technique for finding an unsatisfiable core of a system of unsatisfiable constraints [17], provides a neat way of realising an SMT solver. Developing [5], it is argued that Prolog also aids the translation of formulae over theory literals that involve constraints into the SMT equivalent of CNF.

The rest of the paper contains a short summary of relevant background on SAT and SMT solving, gives the code for the solver and comments upon it, discusses extensions to the solver and concludes with a discussion of the limitations of the solver and its approach.

2. Background

2.1. SAT solving

This section briefly outlines the SAT problem and the DPLL algorithm [6] with watched literals [24] 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 [6]. Fig. 1 presents a recursive formulation of the algorithm adapted from that given in [29]. The first argument of the function DPLL is a propositional formula, f, defined over a set of propositional variables X. As usual f is assumed to be in CNF. The second argument, $\theta : X \to \{true, false\}$, is a partial (truth) function. The call 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 θ that satisfies f.

2.1.1. 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, 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 satisfiability 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.

2.1.2. Watched literals

Information can only be derived from a clause if it does not contain two unknowns. This is the observation behind watched literals [24], 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. Note that no more than these monitors are required.

When the initial empty θ is augmented with $x \mapsto \mathit{true}$, a new monitor for the third clause is not available and unit propagation immediately applies to infer $z \mapsto \mathit{true}$. 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 $y \mapsto \mathit{false}$, 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 $w \mapsto \mathit{false}$. 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 needlessly.

2.1.3. 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$(f, \theta_1) = \mathit{false}$ 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 $f = (\neg x) \wedge (x \vee y) \wedge (\neg y)$ and $\theta = \emptyset$. From the first and third clauses it follows that $\theta_1 = \{x \mapsto \mathit{false}, y \mapsto \mathit{false}\}$. 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 $\bot$.

2.1.4. 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 $x \mapsto \mathit{true}$. If satisfiability cannot be detected with this choice, DPLL is subsequently invoked with θ_1 augmented with $x \mapsto \mathit{false}$. Termination is assured because the number of unassigned variables strictly reduces on each recursive call.

2.2. SMT solving

This section briefly outlines the SMT scheme [27] and the SMT algorithm that the solver implements. The examples assume that the theory is quantifier-free linear real arithmetic where the constants are numbers, the functors are interpreted as addition and subtraction, and the predicates include equality, disequality and both strict and non-strict inequalities.

SMT gives a general scheme for determining the satisfiability of problems consisting of a formula over atomic constraints in some theory T, whose set of literals is denoted Σ. The scheme separates the propositional skeleton, that is the logical structure of combinations of theory literals, and the meaning of the literals. A bijective encoder mapping $e : \Sigma \to X$ associates each literal with a unique propositional variable. Then the encoder mapping e is lifted to theory formulae, using $e(\phi)$ to denote the propositional skeleton of a theory formula ϕ. To illustrate, consider the problem of checking the entailment $(a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \models (1 \leq a + b)$. The entailment check amounts to determining that the theory formula $\phi = (a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \wedge \neg(1 \leq a + b)$ is not satisfiable. For this problem, the set of literals is $\Sigma = \{a < b, \ldots, 1 \leq a + b\}$. Suppose, in addition, that the encoder mapping is defined as follows:

$$e(a < b) = x, \qquad e(a = 0) = y, \qquad e(a = 1) = z, \qquad e(b = 0) = u, \qquad e(b = 1) = v, \qquad e(1 \leq a + b) = w.$$

Then the propositional skeleton of ϕ, given e, is $e(\phi) = x \wedge (y \vee z) \wedge (u \vee v) \wedge \neg w$. A SAT solver gives a truth assignment θ satisfying the propositional skeleton. From this, a conjunction of theory literals, $\hat{Th}(\theta, e)$ is constructed. A conjunct is the literal l if $\theta(e(l)) = \mathit{true}$ and $\neg l$ if $\theta(e(l)) = \mathit{false}$. This problem is passed to a specialised solver for the theory that can determine satisfiability of conjunctions of constraints. Either satisfiability or unsatisfiability is determined, in the latter case the SAT solver is asked for further truth assignments. Fig. 2 gives a recursive reformulation of Algorithm 11.2.1 from [21]. The first argument of the function LAZY-BASIC is a Boolean formula, f, and the second an encoder mapping, e. In the initial call, f is the conversion to CNF of $e(f)$. The call LAZY-BASIC(f,e) returns the symbol $\bot$ if ϕ is not satisfiable, and returns $\top$ otherwise.

2.2.1. Truth assignments from a SAT solver

In line (3), a call to the DPLL algorithm is made to find a truth assignment satisfying the propositional formula f which is initially the propositional skeleton (converted to CNF) of the problem ϕ, and in further recursive calls will have been strengthened with blocking clauses describing truth assignments which do not correspond to a satisfying assignment to ϕ. If no such model exists, then ϕ is unsatisfiable. In the example, the initial truth assignment found by DPLL$(e(\phi), \emptyset)$ will be $\theta' = \{x \mapsto \mathit{true}, y \mapsto \mathit{true}, z \mapsto \mathit{true}, u \mapsto \mathit{true}, v \mapsto \mathit{true}, w \mapsto \mathit{false}\}$.

```
(1)   function LAZY-BASIC(f : CNF formula, e : Σ → X)
(2)   begin
(3)       θ := DPLL(f,∅);
(4)       if (θ = ⊥) then
(5)           return ⊥;
(6)       else
(7)           t := deduction(T̂h(θ, e));
(8)           if (t = ⊤) then
(9)               return ⊤;
(10)          else
(11)              return LAZY-BASIC(f ∧ e(t), e);
(12)          endif
(13)      endif
(14)  end
```

Fig. 2. Recursive formulation of the SMT scheme.

2.2.2. Deduction

Of course, a truth assignment satisfying the propositional skeleton does not guarantee that the theory problem ϕ is satisfiable. First, a model θ of f is used to construct a conjunction of literals in the theory, $\hat{Th}(\theta, e)$. In the example, this gives $\hat{Th}(\theta', e) = (a < b \wedge a = 0 \wedge a = 1 \wedge b = 0 \wedge b = 1 \wedge 1 > a + b)$. Then the procedure deduction uses a theory specific decision procedure to determine whether or not $\hat{Th}(\theta, e)$ is satisfiable. If it is, then the initial problem ϕ is satisfiable and $\top$ is returned, if not, deduction returns the negation of a conjunction of literals in the theory that are not satisfiable. In the example, deduction will determine that $\hat{Th}(\theta', e)$ is unsatisfiable and might return $\neg(a = 0 \wedge a = 1) = \neg(a = 0) \vee \neg(a = 1)$.

2.2.3. Search and the recursive call

The value returned by deduction is mapped to a new clause, a blocking clause, which is added to the Boolean formula. LAZY-BASIC is then called recursively with the updated formula. In the example, the clause $(\neg y \vee \neg z)$ is added to the formula $e(\phi)$ and DPLL$(e(\phi) \wedge (\neg y \vee \neg z), \emptyset)$ gives $\theta'' = \{x \mapsto \mathit{true}, y \mapsto \mathit{false}, z \mapsto \mathit{true}, u \mapsto \mathit{true}, v \mapsto \mathit{true}, w \mapsto \mathit{false}\}$. Again, $\hat{Th}(\theta'', e)$ is unsatisfiable, this time leading to $(\neg x \vee \neg z \vee \neg u)$ being added to the Boolean formula. Continuing this, either $\top$ will be returned or all possible Boolean truth assignment will have been explored and $\bot$ will be returned (this is the case when running the example to completion). Note that since the new clause blocks the previous model from being returned, a new model is always found and the algorithm clearly terminates, assuming deduction terminates.

2.2.4. Theories

The theory in the SMT scheme can be instantiated by any theory that comes with a decision procedure for conjunctions of theory literals. Many theories have been considered, but this paper concentrates on quantifier-free linear real arithmetic. That is, on solving conjunctions of arithmetic constraints consisting of strict or non-strict linear inequalities, equalities and disequalities over the reals. This decision problem has been extensively studied [15] and in particular the decision procedure that underpins the CLP($\mathcal{R}$) scheme [16] as implemented in [11] decides this problem. The authors have also considered the theory of equality logic over uninterpreted functions.

3. The SAT solver

The code for the solver is given in Fig. 3. 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 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 `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

```
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).
```

Fig. 3. Code for SAT solver.

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 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_vars(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` 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_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 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 = false,  Y = true,   Z = true;       X = false,  Y = false,  Z = true;
X = true,   Y = true,   Z = false;      X = false,  Y = true,   Z = false;
X = false,  Y = false,  Z = false.
```

4. Interlude: saving and restoring search state

In this section it is demonstrated how search in the SAT solver may be initialised from a given (partial) truth assignment. Accompanying this with a mechanism to save a previous assignment gives an efficiency optimisation to the solver presented in Section 5 implementing the SMT scheme in Fig. 2. The scheme involves repeated calls to the SAT solver with the initial propositional skeleton augmented by blocking clauses resulting from deduction. When using the SAT solver from Section 3, finding the nth model will involve repeating all search involved in finding the $(n-1)$th model. The state restoration mechanism presented here saves this repeated search.

The approach uses extra-logical features of Prolog and is akin to the technique used for backjumping in search described as a black pearl in [3]. The new version of the SAT solver is given in Fig. 4 (the remaining predicates are as in Fig. 3). The solver uses the extra-logical blackboard where data can be stored away with `bb_put/2` and retrieved with `bb_get/2` to maintain a state to be restored when `sat/2` is called. This target state (henceforth referred to as the history) might have resulted from a previous call to `sat/2` or have been directly set using `initialise/1`. (Clearly if the solver is not initialised, it will fail at the first call to `bb_get`.)

Storing the history is simple — after a (complete) satisfying assignment has been found it is placed on the blackboard (it is reversed owing to the structure of `elim_vars`). For example, consider the SAT instance `[[true-X],[true-Y,true-Z],[true-U,true-V],[false-W]]`. With the variables in the second clause of `sat` ordered `[X,Y,Z,U,V,W]` this will place the list of truth values `[false,true,true,true,true,true]` on the blackboard.

Restoring state from the history is not quite as straightforward since the solver needs to be directed to an assignment without search, after which point search, including backtracking past the restored assignments, needs to continue. This is dealt with by replacing the `assign/1` facts of Fig. 3 with calls to the assignment predicates `assign_true/2` and `assign_false/2`. There are three cases to consider: first when the history is empty, that is, when state is not being restored and search is proceeding as normal; second, when state is being restored and the restoration step is successful; third, when state is being restored and the restoration step fails. This third case is expected when blocking clauses are added to the problem; the conflict indicates the point where further search starts. The key point to note is that when an `assign` decision point is revisited on backtracking the history is read from the blackboard again, and it might well be different from when the first branch was explored, in particular it may be empty.

The first case is straightforward. The history is empty and `assign_true` and `assign_false` unify `Var` with `true` or `false` respectively.

In the second case, the history is not empty and the head of the history is (successfully) unified with `Var`. If `Var` was non-ground then it is has been assigned the value it had in the previous iteration. The history is then updated. Observe that search is avoided since the history value sets a variable immediately, rather than exploring a range of unsuccessful assignments first. Notice also that if search returns to this decision, the history will be empty and backtracking possible. For example, suppose that the SAT instance above has been augmented with `[false-Y,false-Z]` and in subsequent search a new assignment is found and the new history `[false,true,true,true,false,true]` has been placed on the blackboard. Starting search with this history and the problem further augmented with the clause `[false-X,false-Z,false-U]` the first step will be to assign `W`. The history says `W` should be assigned `false` (the head of the history) and this is achieved in `assign_false` and the tail of the history is posted back to the blackboard. The history now says that `V` should be assigned

```
:- module(sat_solver, [sat/2, initialise/1]).
:- use_module(library(lists)).

initialise(State) :- bb_put(history, State).

sat(Clauses, Vars) :-
    problem_setup(Clauses), elim_var(Vars),
    reverse(Vars, Rev), bb_put(history, Rev).

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

assign(Var) :- bb_get(history, Hs), assign_true(Hs, Var).
assign(Var) :- bb_get(history, Hs), assign_false(Hs, Var).

assign_true([], true).
assign_true([true | Hs], Var) :-
    (Var = true ->
        bb_put(history, Hs)
    ;
        bb_put(history, []), fail
    ).

assign_false([], false).
assign_false([false | Hs], Var):-
    (Var = false ->
        bb_put(history, Hs)
    ;
        bb_put(history, []), fail
    ).
```

Fig. 4. Code for a SAT solver with state restoration.

`true` and again this is achieved in `assign_true`. If search returned to this decision with `[]` as the history, search can backtrack to explore `assign_false([], V)`.

In the third case, unification with the head of the history fails. This ends the restoration process. Note that assignment in the SAT solver is ordered, with `true` being the first value assigned. The solver needs to ensure that after state restoration regions of the search space visited in previous iterations are not reexplored, and that no region of the (new) search space is omitted. If the conflict arises when the history value is `true`, search can continue: the history is not needed, hence the empty history is posted to the blackboard and the explicit `fail` drives search into the false branch. This is the next possible assignment, hence no part of the search space has been omitted. If the conflict arises when the history value is `false`, search should fail and return to a previous decision, this is done by updating the history to empty and an explicit `fail`.

Continuing the example above, after unifying `U` with `true` attempting to unify `Z` with `true` leads to conflict, hence the history is emptied and the fail leads to the search backtracking to the last call of `assign`, it then continues with the `assign_false` branch (which is possible since now the history contains `[]`); this leads to the next solution `Vars=[true,true,false,true,true,false]`.

5. The SMT solver

The code for the SMT solver is given in Fig. 5. The solver needs to be coupled with a theory solver given as a module `theory` and exporting `post_all/1` and `unsat_core/3`. Code for one theory – quantifier-free linear real arithmetic – is given in Fig. 6.

5.1. Invoking the solver

It is assumed that the solver is called with the theory formula having been preprocessed into its propositional skeleton (converted into CNF) coupled with an association list mapping the logical variables of the skeleton to the theory literals of the input problem (plus any Tseitin variables, introduced in CNF conversion [28], that are mapped to a trivial term, `triv`). The solver is called with `smt(Clauses, Vars, ConsMap)` where `Clauses` is the propositional skeleton of the theory formula presented in CNF, `Vars` is a list of the variables in the skeleton, and the `ConsMap` is an association list that represents the

```
:- use_module(theory).
:- use_module(sat_solver).
:- use_module(library(assoc)).

smt(Clauses, Vars, ConsMap) :-
    initialise([]),
    smt_call(Clauses, Vars, ConsMap).

smt_call(Clauses, Vars, ConsMap) :-
    copy_term(Clauses-Vars, CopyClauses-CopyVars),
    sat(CopyClauses, CopyVars), !,
    zip(CopyVars, Vars, ZipVars),
    smt_proceed(ZipVars, Clauses, Vars, ConsMap).

smt_proceed(ZipVars, _Clauses, _Vars, ConsMap) :-
    satisfiable(ZipVars, [], ConsMap), !.
smt_proceed(ZipVars, Clauses, Vars, ConsMap) :-
    unsat_core(ZipVars, ConsMap, Min),
    new_clause(Min, Vars, NewClause),
    smt_call([NewClause | Clauses], Vars, ConsMap).

satisfiable([], Cons, _) :- post_all(Cons).
satisfiable([Val-Var | Vals], Acc, ConsMap) :-
    get_assoc(Var, ConsMap, Con),
    satisfiable(Vals, [Val-Con | Acc], ConsMap).

zip([], [], []).
zip([X | Xs], [Y | Ys], [X-Y | Zs]) :- zip(Xs, Ys, Zs).

new_clause([], _, []).
new_clause([Val | Vals], Vars, Rest) :-
    new_clause(Val, Vals, Vars, Rest).

new_clause(true, Vals, [Var | Vars], [false-Var | Rest]) :-
    new_clause(Vals, Vars, Rest).
new_clause(false, Vals, [Var | Vars], [true-Var | Rest]) :-
    new_clause(Vals, Vars, Rest).
new_clause(na, Vals, [_ | Vars], Rest) :-
    new_clause(Vals, Vars, Rest).
```

Fig. 5. Code for SMT solver.

encoder mapping. For instance, to solve the example given in Section 2.2, `Clauses = [[true-X], [true-Y, true-Z], [true-U, true-V], [false-W]]`, `Vars = [X, Y, Z, U, V, W]`, and `ConsMap` is an association list created through a series of calls such as `empty_assoc(ConsMap0)`, `put_assoc(X, ConsMap0, A < B, ConsMap1)`, `put_assoc(Y, ConsMap1, A = 0, ConsMap2)`, etc and finally assigning `ConsMap = ConsMap6`. The goal `smt(Clauses, Vars, ConsMap)` succeeds then if the problem is satisfiable and fails otherwise. Note that the predicate `smt/3` will also initialise the history in the SAT solver.

5.2. Finding a truth assignment

A truth assignment satisfying the propositional skeleton is found with a call to the SAT solver from Section 4 (or 3). Note that the arguments are copies of the clauses and variables and the solution is afterwards paired up with the original uninstantiated variables – this results from the recursive formulation of the SMT solver with its repeated calls to the SAT solver, without backtracking.

5.3. Deduction: finding a countermodel

The truth assignment given by the SAT solver is a candidate model for satisfying the theory problem. The predicate `satisfiable/3` tests whether this is the case; theory literals are paired with Boolean values from the truth assignment

```
:- module(theory, [post_all/1, unsat_core/3]).
:- use_module(library(clpr)).
:- use_module(library(assoc)).

post_all([]).
post_all([Val-Con | Cons]) :- post_con(Val, Con), post_all(Cons).

post_con(true, Con) :- post_true(Con).
post_con(false, Con) :- post_false(Con).

post_true(triv).
post_true(X=<Y) :- {X=<Y}.
post_true(X<Y) :- {X<Y}.
post_true(X=Y) :- {X=Y}.

post_false(triv).
post_false(X=<Y) :- {X>Y}.
post_false(X<Y) :- {X>=Y}.
post_false(X=Y) :- {X=\=Y}.

unsat_core(VarMap, ConsMap, Min) :-
    assoc_to_vals(VarMap, ConsMap, [], Cons),
    remove_redundant(Cons, [], [], Min).

assoc_to_vals([], _, Cons, Cons).
assoc_to_vals([Val-Var | VarMap], ConsMap, Acc, Vs) :-
    get_assoc(Var, ConsMap, Con),
    assoc_to_vals(VarMap, ConsMap, [Val-Con | Acc], Vs).

check_redundant(Val-Con, Cons, TestedCons, Core, Min) :-
    append(Cons, TestedCons, AllCons),
    copy_term(AllCons, CopyCons),
    post_all(CopyCons), !,
    remove_redundant(Cons, [Val-Con | TestedCons], [Val | Core], Min).
check_redundant(_, Cons, TestedCs, Core, Min) :-
    remove_redundant(Cons, TestedCs, [na | Core],  Min).

remove_redundant([], _, Min, Min).
remove_redundant([Con | Cons], Tested, Core, Min) :-
    check_redundant(Con, Cons, Tested, Core, Min).
```

Fig. 6. Linear real arithmetic theory using CLP($\mathcal{R}$).

before using the theory predicate `post_all` to determine whether or not they are satisfiable. If `post_all`, hence `satisfiable`, succeeds then the theory problem has been solved.

Otherwise, it is enough to note that the current model is unsatisfiable. However, the deduction step aims to make a better diagnosis of why the conjunction of theory literals is unsatisfiable. Therefore, the second clause of `smt_proceed` uses the theory predicate `unsat_core/3` to find an inconsistent core, that is a subset of the current model that is still unsatisfiable. The final argument of `unsat_core` is unified with a list of values, each corresponding to whether in the inconsistent core the literal is posted positively (`true`), posted negatively (`false`) or is not included (`na`). That is, `Min` describes the inconsistent core and `na` corresponds to a theory literal not in this core. Referring to the example in 2.2.2, when `unsat_core` is called with the first argument `[true-X,true-Y,true-Z,true-U,true-V,false-W]` the third argument will be unified with `[na,true,true,na,na,na]` indicates that the literals associated with `Y` and `Z` are inconsistent.

5.4. Recursion and adding clauses

This minimised model is negated and added to `Clauses` as a blocking clause in `new_clause` and `smt_call` is called recursively. As discussed in Section 4, the SAT solver returns truth assignments one by one. If a call to the SAT solver results in failure then there are no further models to consider and the theory problem is unsatisfiable. Note that when using the state restoration solver from Section 4, only the original propositional skeleton and the new blocking clause are required.

5.5. Theory: linear real arithmetic

SMT solving is illustrated in this section with the theory of quantifier-free linear real arithmetic. This example has been chosen as Prolog systems often come with the CLP($\mathcal{R}$) constraints package which will determine the consistency of conjunctions of linear arithmetic constraints. Fig. 6 presents code to realise the theory in such a way as to be used by the SMT solver.

It is assumed that the input problem has been normalised so that all the constraint predicates are either =, =< or <. The predicate `post_all` posts to the store a series of constraints according to their polarity. One of the main functions of the CLP($\mathcal{R}$) package is to determine the consistency of its constraint store – exactly what is required.

The implementation of `unsat_core` given here flattens the association list and finds an unsatisfiable core of the set of constraints by omitting from the current set of inconsistent constraints a single constraint at a time and testing the remainder for consistency. If the system is still inconsistent, then the omitted constraint is not required for inconsistency. For example, when `unsat_core` is called with first argument `[true-X, true-Y, true-Z, true-U, true-V, false-W]`, the predicate `remove_redundant` is called with its first argument `[false-(1=<A+B), true-(B=1), true-(B=0), true-(A=1), true-(A=0), true-(A<B)]`. Omitting each of the first three constraints still leaves an unsatisfiable system and the constraints are discarded from the core, but omitting the fourth (and fifth) constraint from those remaining leads to a satisfiable system. Omitting the final constraint still leaves an unsatisfiable system and `remove_redundant` succeeds with its fourth argument unified with `[na, true, true, na, na, na]` indicating (note the order in which the list is constructed) that the constraint sub-system comprising of just `A=0` and `A=1` is unsatisfiable. The approach used to find an unsatisfiable core requires n calls to `post_all` where n is the length of the list that represents the model initially passed to `unsat_core`. (The method is thus similar in spirit to serial constraint deletion in the calculation of interpolants [17, section 5]). Finally, `Y` and `Z` are the corresponding variables to these constraints and the clause `[false-Y, false-Z]` is constructed by `new_clause` and added to the skeleton.

5.6. Theory: equality logic with uninterpreted functions

If a Prolog system does not come equipped with an appropriate constraint library, there is no reason why a decision procedure cannot be coded in Prolog itself. Indeed, the declarative features of the paradigm make it eminently suitable for such proposes. To illustrate, consider the theory of equality logic with uninterpreted functions [18,26] that is widely applied in verification [21]. This theory satisfies the congruence axiom [21]: if $x_i = y_i$ for all $i \in \{1, \ldots, n\}$ then $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$, though the converse does not hold. For example, it follows that $(a = b \wedge b = g(c)) \vee (a = g(b) \wedge b = c) \models a = g(c)$. This can be demonstrated by checking that the theory formula $\phi = ((a = b \wedge b = g(c)) \vee (a = g(b) \wedge b = c)) \wedge \neg(a = g(c))$ is unsatisfiable. Using the encoder

$$e(a = b) = z, \quad e(b = g(c)) = y, \quad e(a = g(b)) = x, \quad e(b = c) = w, \quad e(a = g(c)) = v$$

the skeleton $e(\phi)$ can be converted into the following CNF formula, denoted f:

$$f = \begin{cases} (\neg v) & \wedge & (x \vee \neg t_2) & \wedge & (w \vee \neg t_2) & \wedge & (\neg x \vee \neg w \vee t_2) & \wedge \\ (t_1 \vee t_2) & \wedge & (z \vee \neg t_1) & \wedge & (y \vee \neg t_1) & \wedge & (\neg z \vee \neg y \vee t_1) & \end{cases}$$

where t_1 and t_2 are fresh Tseitin variables.

Solving proceeds in an analogous way to before: the SAT solver finds a truth assignment $\theta' = \{v \mapsto \mathit{false}, w \mapsto \mathit{true}, x \mapsto \mathit{true}, y \mapsto \mathit{true}, z \mapsto \mathit{true}, t_1 \mapsto \mathit{true}, t_2 \mapsto \mathit{true}\}$ for f which is used to construct a conjunction of literals by $\hat{Th}(\theta', e) = (g(c) \neq a \wedge b = c \wedge g(b) = a \wedge g(c) = b \wedge a = b)$ which is unsatisfiable in equality logic. A deduction procedure can be constructed in a similar way to a linear theory to return the unsatisfiable conjunction $g(c) \neq a \wedge b = c \wedge g(b) = a$ from which the blocking clause $(v \vee \neg w \vee \neg x)$ can be derived. Augmenting f with this clause, reapplying the SAT solver discovers the model $\theta'' = \{v \mapsto \mathit{false}, w \mapsto \mathit{false}, x \mapsto \mathit{true}, y \mapsto \mathit{true}, z \mapsto \mathit{true}, t_1 \mapsto \mathit{true}, t_2 \mapsto \mathit{false}\}$ from which $\hat{Th}(\theta'', e) = (g(c) \neq a \wedge b \neq c \wedge g(b) = a \wedge g(c) = b \wedge a = b)$ is constructed; this is also unsatisfiable. Deduction then derives the conjunction $g(c) \neq a \wedge g(c) = b \wedge a = b$ from which the blocking clause $(v \vee \neg y \vee \neg z)$ is inferred. Augmenting f with this additional clause leads to an unsatisfiable SAT instance, hence ϕ is unsatisfiable and the entailment relation follows.

A Prolog implementation of the decision procedure algorithm of [26], which incidentally is both incremental and has been found to be particularly efficient [26], can be realised in less than 200 lines of code. This is because this algorithm relies on symbolic preprocessing and normalisation which can be coded compactly in Prolog, as explained in the following section.

6. Normalisation

The case had already been made [5] that logic programming provides a declarative way of stating satisfiability problems and encoding them in CNF. For example, fresh variables, sometimes known as Tseitin variables [28], are introduced when converting a propositional formula into CNF, a process sometimes referred to as flattening. The idea is to introduce a fresh variable for each sub-formula in the formula. For instance, the formula $(x \vee y) \oplus z$ can be translated to the equisatisfiable

formula $(t \oplus z) \wedge (t \leftrightarrow (x \vee y))$ in which t is fresh and each conjunct involves no more than three variables. The conjuncts $t \oplus z$ and $t \leftrightarrow (x \vee y)$ are then individually translated to CNF, giving a CNF representation for the whole.

Logical variables provide a natural way of generating fresh variables, but their true power is that these placeholders can be unified and applied to decompose a problem into independent steps. This is just what is needed when constructing the SMT equivalent of CNF. Consider the formula $(g(h(i(a), b), c) = d) \wedge (g(h(i(a), b), c) \neq d)$ over the theory of equality logic with uninterpreted functions [21] (which incidentally is unsatisfiable). Decision procedures for such systems [18,26] apply a form of flattening to terms in which a fresh symbol, say t, is introduced to name a non-constant proper sub-term, such as $i(a)$. Then $i(a)$ is replaced everywhere by t, the equation $i(a) = t$ is added to the system, and the process is repeated until all non-constant proper sub-terms have been consistently named. Note that all occurrences of the same sub-term must be replaced with a common symbol, so the problem is not as straightforward as flattening propositional formulae.

The elegance of logical variables is that they can be applied to decompose term flattening into two independent steps. In the first step, fresh symbols are introduced for all proper sub-terms, no matter whether they occur singly or multiply. This is illustrated in the following table, where the right-hand column is a list which records all the substitutions that have been made.

0	$[g(h(i(a), b), c) = d, g(h(i(a), b), c) \neq d]$	$[]$
1	$[g(t_1, c) = d, h(i(a), b) = t_1, g(h(i(a), b), c) \neq d]$	$[h(i(a), b) - t_1]$
2	$[g(t_1, c) = d, h(t_2, b) = t_1, i(a) = t_2, g(h(i(a), b), c) \neq d]$	$[h(i(a), b) - t_1, i(a) - t_2]$
3	$[g(t_1, c) = d, h(t_2, b) = t_1, i(a) = t_2, g(t_3, c) \neq d, h(i(a), b) = t_3]$	$[h(i(a), b) - t_1, i(a) - t_2, h(i(a), b) - t_3]$
4	$[g(t_1, c) = d, h(t_2, b) = t_1, i(a) = t_2, g(t_3, c) \neq d, h(t_4, b) = t_3, i(a) = t_4]$	$[h(i(a), b) - t_1, i(a) - t_2, h(i(a), b) - t_3, i(a) - t_4]$

In the second step, different occurrences of the same sub-term are correlated. This is achieved by key-sorting the list of substitutions $[h(i(a), b) - t_1, i(a) - t_2, h(i(a), b) - t_3, i(a) - t_4]$ to give $[h(i(a), b) - t_1, h(i(a), b) - t_3, i(a) - t_2, i(a) - t_4]$. The sorted list is then scanned in a linear fashion to detect any replicated keys and unify the associated symbols. This unifies the symbols t_1 and t_3 and likewise t_2 and t_4. This, in turn, transforms the flattened system of equations and disequations to $[g(t_1, c) = d, h(t_2, b) = t_1, i(a) = t_2, g(t_1, c) \neq d, h(t_2, b) = t_1, i(a) = t_2]$. This list is then itself sorted to remove duplicates. Therefore logical variables are not only of value when generating fresh symbols, but also enable different symbols to be recoupled via unification.

7. Discussion

Thus far this paper has highlighted the ways in which Prolog provides an easy and elegant entry point into SAT and SMT solving, whilst also making contributions on the preprocessing of SMT inputs and the efficiency of the integration of the SAT solver into the SMT framework. This section discusses what the code presented does not achieve in relation to state-of-the-art SAT and SMT solvers, whilst offering hints as to how the techniques used in these solvers could be realised in Prolog.

The challenge of SAT solving grows with the size of the problem. This can manifest itself in two ways: the growth of the search space and the storage of the SAT instance. The development of SAT solvers over the last decade has resulted in numerous heuristics to reduce the search space that dramatically improve the performance of general purpose solvers. The ways in which a number of these refinements might be incorporated into the solver presented above are now discussed:

- 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 (and was used in the experiments presented in [14]).
- Another tactic is to change the problem by restructuring it using limited applications of resolution [8]. Again, these preprocessing steps can clearly be achieved satisfactorily in Prolog.
- Many SAT solvers use non-chronological backtracking [2], or backjumping, in order to avoid exploration of fruitless branches of the search tree [23]. Backjumping for depth-first search algorithms in Prolog has been explored in [3] and this approach (without learning) carries over to the solver presented in this paper. Note that SAT solvers often realise backjumping by altering the problem with learnt clauses. Here, following [3], backjumping is achieved by coding additional control.
- Another popular heuristic is learning in which clauses are added to the problem that express regions of the search space that do not contain a solution [23]. It is less clear how to achieve this cleanly in this Prolog solver, as calls to the learnt clauses would be lost on backtracking. That said, the SMT solver requires clauses to be added to an SAT problem to produce a new assignment. In Section 4 it was demonstrated how search can be started or resumed at a specified point allowing the incremental problems arising in SMT to be more efficiently solved. The approach will also work in a more general learning context. At appropriate failure points a description of clauses to be learnt can be posted to a blackboard, then the problem restarted with the addition of the learnt clauses followed by state restoration. This approach also fits with the random restarts employed by some solvers. However, it is unclear whether the cost of learning clauses in this way will be fully repaid by reduced search.

- Dynamically reordering variables during search [24] has also been widely incorporated in SAT solvers. This can be incorporated into the solver presented in this paper in conjunction with learning. As above, blocking clauses that will prevent search returning to a previous assignment can be learnt. Then search may be restarted with a new variable ordering (determined by analysing information from the previous search stored on a blackboard).

The extensions to SAT are heuristics attempting to reduce search. Extensions to SMT again aim to reduce the amount of search, in this case by more tightly coupling the SAT solver and the decision procedure for conjunctions of theory literals. There are two possibilities to consider:

- The DPLL(T) scheme [27] ties assignment in the SAT problem to posting constraints in the theory. In the solver presented in this paper a complete variable assignment is found before using this to form the conjunction $\hat{Th}(\theta, e)$ and test it for satisfiability. In DPLL(T) the conjunctive theory problem is incrementally extended by the literal l or $\neg l$ (and tested for consistency) as $e(l)$ is assigned *true* or *false* respectively. This allows unsatisfiability to be detected before a complete assignment has been made, reducing propagation and search. For linear real arithmetic, this scheme could be accomplished using the techniques presented in this paper – a predicate blocked on $e(l)$ would post an appropriate constraint when the variable is instantiated. This would incrementally propagate information from the SAT component to the theory component. Propagating information from the theory to the SAT component, in a fully increment way, is more challenging but might be feasible using systems of reified constraints [4]. For example, there is no reason why the 0/1 variables that indicate whether a reified constraint is entailed or disentailed could not be those propositional variables that are assigned in the SAT component. For theories not exploiting the constraint packages distributed with Prolog systems DPLL(T) requires more effort, since a model of a constraint store needs to be built.
- Theory propagation is where assignment to the propositional variables is made not just in the SAT component of the SMT solver, but also in the deduction procedure. That is, with a partial assignment of the propositional variables, deduction infers that theory satisfiability can only be achieved if theory literal l is satisfied or otherwise. This information is propagated to the SAT problem by setting $e(l)$. For example [21], if $e(x = y) \mapsto \mathit{true}$ and $e(y = z) \mapsto \mathit{true}$ and $x = z$ is also a theory literal, then theory propagation might deduce that $e(x = z) \mapsto \mathit{true}$. As this is a symbolic deduction from a set of constraints, theory propagation could incorporated into a Prolog implementation of DPLL(T), as above.

Returning to the difficulties that arise with large problems identified at the beginning of this section, it is, in fact, the second manifestation that 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. As an example, consider 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 [10].

Owing to the issues outlined above, the solver presented in this paper is not going to be competitive on the large, difficult problems set as challenges in the international SAT [22] and SMT [1] competitions. (Though a reviewer pointed out that larger problems can sometimes be accommodated by consulting rather than compiling the solver.) Nevertheless 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 particular, its incorporation into an SMT scheme using the constraint packages often distributed with Prolog systems gives a straightforward realisation of the theory of linear real arithmetic. Furthermore, a generalisation of constraint logic programming, T logic programming [7], offers the potential to realise new theories and even extend an existing theory, on-the-fly, with axioms gleaned through learning.

In [14] a brief empirical evaluation of the SAT solver was given that indicated that the solver 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 [12]. In this context, a SAT engine coded in Prolog itself is attractive since it avoids using a foreign language interface (note that [5] hides this interface from the user), simplifies distribution issues, and avoids the overhead of converting a Prolog representation of an 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/. The distribution includes all code from this paper as well as additional code relating to Sections 6 and 7. The distribution also includes Prolog code, kindly donated by a referee, that generates a Sudoku puzzle a solution to which can be found using the SAT solver presented earlier.

Acknowledgements

The authors would like to thank the anonymous referees of this paper and [14] for their helpful and insightful comments.

References

[1] C. Barrett, M. Deters, A. Oliveras, A. Stump, Design and results of the 3rd annual satisfiability modulo theories competition (SMT-Comp 2007), International Journal on Artificial Intelligence Tools 17 (4) (2008) 569–606.
[2] M. Bruynooghe, Intelligent backtracking revisited, in: Computational Logic, Essays in Honor of Alan Robinson, MIT Press, 1991, pp. 166–177.

[3] M. Bruynooghe, Enhancing a search algorithm to perform intelligent backtracking, Theory and Practice of Logic Programming 4 (3) (2004) 371–380.
[4] M. Carlsson, G. Ottosson, B. Carlson, An open-ended finite domain constraint solver, in: PLILP, in: Lecture Notes in Computer Science, vol. 1292, Springer, 1997, pp. 191–206.
[5] M. Codish, V. Lagoon, P.J. Stuckey, Logic programming with satisfiability, Theory and Practice of Logic Programming 8 (1) (2008) 121–128.
[6] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5 (7) (1962) 394–397.
[7] A. Dovier, A. Formisano, A. Policriti, On *T* logic programming, in: International Logic Programming Symposium, MIT Press, 1997, pp. 323–337.
[8] N. Eén, A. Biere, Effective preprocessing in SAT through variable and clause elimination, in: International Conference on Theory and Applications of Satisfiability Testing, in: Lecture Notes in Computer Science, vol. 3569, 2005, pp. 61–75.
[9] N. Eén, S. Sörensson, An extensible SAT-solver, in: Theory and Applications of Satisfiability Testing, in: Lecture Notes in Computer Science, vol. 2919, Springer, 2003, pp. 502–518.
[10] I.P. Gent, C. Jefferson, I. Miguel, Watched literals for constraint propagation in Minion, in: Constraint Programming, in: Lecture Notes in Computer Science, vol. 4204, Springer, 2006, pp. 182–197.
[11] C. Holzbaur, OFAI clp(q,r) Manual, Ed. 1.3.3. Technical report, Austrian Research Institute for Artificial Intelligence, 1995.
[12] J.M. Howe, A. King, Positive boolean functions as multiheaded clauses, in: International Conference on Logic Programming, in: Lecture Notes in Computer Science, vol. 2237, Springer, 2001, pp. 120–134.
[13] J.M. Howe, A. King, Efficient groundness analysis in Prolog, Theory and Practice of Logic Programming 3 (1) (2003) 95–124.
[14] J.M. Howe, A. King, A pearl on SAT solving in Prolog, in: Functional and Logic Programming, in: Lecture Notes in Computer Science, vol. 6009, Springer, 2010, pp. 165–174.
[15] J.-L. Imbert, P. Van Hentenrych, On the handling of disequations in CLP over linear rational arithmetic, in: Constraint Logic Programming, MIT Press, 1993, pp. 49–71.
[16] J. Jaffar, S. Michaylov, P. Stuckey, R. Yap, The CLP($\mathcal{R}$) language and system, ACM Transactions of Programming Languages and Systems 14 (3) (1992) 339–395.
[17] J. Jaffar, A.E. Santosa, R. Voicu, An interpolation method for CLP traversal, in: Principals and Practice of Constraint Programming, in: Lecture Notes in Computer Science, vol. 5732, Springer, 2009, pp. 454–469.
[18] D. Kapur, Shostak's congruence closure as completion, in: Rewriting Techniques and Applications, in: Lecture Notes in Computer Science, vol. 1232, Springer, 1997, pp. 23–37.
[19] A. King, J.C. Martin, Control generation by program transformation, Fundamenta Informaticae 69 (1–2) (2006) 179–218.
[20] R.A. Kowalski, Algorithm = logic + control, Communication of the ACM 22 (7) (1979) 424–436.
[21] D. Kroening, O. Strichman, Decision Procedures, Springer, 2008.
[22] D. Le Berre, O. Roussel, L. Simon, The International SAT Competitions Webpage, 2009. http://www.satcompetition.org/.
[23] J.P. Marques-Silva, K.A. Sakallah, GRASP — a new search algorithm for satisfiability, in: International Conference on Computer-Aided Design, ACM and IEEE Computer Society, 1996, pp. 220–227.
[24] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Design Automation Conference, ACM Press, 2001, pp. 530–535.
[25] L. Naish, Negation and Control in Logic Programs, Springer-Verlag, 1986.
[26] R. Nieuwenhuis, A. Oliveras, Proof-producing congruence closure, in: In Term Rewriting and Applications, in: Lecture Notes in Computer Science, vol. 3467, 2005, pp. 453–468.
[27] R. Nieuwenhuis, A. Oliveras, 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) (2006) 937–977.
[28] G. Tseitin, On the complexity of derivation in propositional calculus, in: A.O. Slisenko (Ed.), Studies in Constructive Mathematics and Mathematical Logic, volume Part II, 1968, pp. 115–125.
[29] L. Zhang, S. Malik, The quest for efficient boolean satisfiability solvers, in: Computer Aided Verification, in: Lecture Notes in Computer Science, vol. 2404, Springer, 2002, pp. 17–36.