TLP **8** (1): 121–128, 2008. © 2007 Cambridge University Press
doi:10.1017/S1471068407003146 First published online 25 May 2007 Printed in the United Kingdom

PROGRAMMING PEARL
Logic programming with satisfiability

MICHAEL CODISH
Department of Computer Science, Ben-Gurion University, Be'er sheva, Israel
(*e-mail:* mcodish@cs.bgu.ac.il)

VITALY LAGOON
Department of Computer Science & Software Engineering, University of Melbourne, Melbourne, Australia
(*e-mail:* vitaly.lagoon@gmail.com)

PETER J. STUCKEY
NICTA Victoria Laboratory and Department of Computer Science & Software Engineering, University of Melbourne, Melbourne, Australia
(*e-mail:* pjs@csse.unimelb.edu.au)

submitted 10 May 2006; revised 30 January 2007; accepted 12 February 2007

Abstract

This paper presents a Prolog interface to the MiniSat satisfiability solver. Logic programming with satisfiability combines the strengths of the two paradigms: logic programming for encoding search problems into satisfiability on the one hand and efficient SAT solving on the other. This synergy between these two exposes a programming paradigm that we propose here as a logic programming pearl. To illustrate logic programming with SAT solving, we give an example Prolog program that solves instances of Partial MAXSAT.

KEYWORDS: Propositional logic, Boolean satisfiability, SAT solvers

1 Introduction

The use of SAT solvers in a wide range of applications is a great success of recent years and is quickly growing more popularly. Contributing to this success are two main factors: (a) SAT solvers are continuously becoming more powerful and (b) encoding techniques are emerging to represent a wide range of search problems as propositional formulae such that each satisfying assignment of the encoding represents a solution of the problem.

This paper presents a Prolog interface to the MiniSat SAT solver (Eén and Sörensson, 2004; MiniSAT, 2006). MiniSat is a small (≈1200 lines of C-code) and efficient open-source SAT solver that is designed to enable easy integration with other tools and languages. Application of this interface facilitates the best of both worlds: Prolog programming for encoding a search problem into a propositional formula on the one hand, and application of powerful SAT solving tools on the other.

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

In recent work (Codish *et al.*, 2006b), we apply SAT encodings to decide LPO termination (Dershowitz, 1982) of a given term rewrite system τ. Here LPO termination of τ is encoded as a propositional formula φ_τ and any assignment satisfying φ_τ indicates a precedence on the symbols in τ such that its corresponding lexicographic path order orients the rules of τ. Namely, for each rule $l \to r$ in τ, $l \succ_{\text{lpo}} r$ holds. Prolog is well suited for the tasks of parsing the given system τ and constructing the encoding φ_τ. Searching for an assignment that satisfies φ_τ is conveniently delegated to the SAT solver.

In other recent works, Codish *et al.* apply SAT encodings to determine termination of rewrite systems, using dependency pairs (Arts and Giesl, 2000) and using recursive path orders (RPO) Dershowitz (1982). A contribution of these works is the propositional encodings for so-called *argument filterings* and *usable rules*, in Codish *et al.* (2006c), and the encodings for multiset orders and lexicographic path orders modulo permutation of arguments, in Annov *et al.* (2006).

All of the above-mentioned results have been developed on top of the described Prolog MiniSat interface. We have also had positive experience in using the interface as an educational tool for teaching logic programming concepts as well as the basics of SAT encodings and SAT solving.

2 Preliminaries

Most SAT solvers assume as input a propositional formula in conjunctive normal form (CNF). That is a conjunction of disjunctions of literals, or equivalently a conjunction of clauses. Each literal is a propositional variable p or its negation $\neg p$. A truth assignment is a mapping from propositional variables into $\{0, 1\}$.

Syntax: In the logic programming setting, we represent literals as terms of the form `X` or $-$`X`, where `X` is a logic variable; clauses as lists of literals; and conjunctions of clauses as lists of clauses. For propositional formulae we use terms involving the symbols `0/0`, `1/0`, `-/1`, `*/2`, `+/2`, `==/2`, `xor/2` for false, true, negation, conjunction, disjunction, bi-implication, and xor. The syntax supports also `(X→Y;Z)`, which is equivalent to `X*Y+(-X)*Z`. In the logic programming setting, a truth assignment is a substitution of the variables to the constants $\{0, 1\}$. We also use a list of propositions to represent nonnegative integers in two different ways. The unary representation is given by the sum of the bits. For example, [0,1,0,1] represents 2. The binary representation is as usual, but giving the least significant bit first. For example, [0,1,0,1] represents $0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 10$.

We now recall the satisfiability and maximum satisfiability problems. Partial MAXSAT is a generalization of SAT and MAXSAT introduced by Miyazaki *et al.* (1996). See also Cha *et al.* (1997).

SAT: Given a propositional logic formula φ in conjunctive normal form, is there an assignment of truth values to the propositional variables that makes φ true?

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

MAXSAT: Given a propositional logic formula φ in conjunctive normal form, find an assignment of the logical variables that maximizes the number of clauses in φ that are true.

Partial MAXSAT: Given propositional logic formulae φ and ψ in conjunctive normal form, the problem is to find an assignment that satisfies all clauses in φ and maximizes the number of clauses in ψ that are true.

3 Interfacing Prolog with MiniSat

The library implementation is written primarily in SWI-Prolog (Wielemaker, 2003; SWI-Prolog, 2005) and interfaces the MiniSat solver (Eén and Sörensson, 2004; MiniSAT, 2006) for solving SAT instances. We have integrated MiniSat and SWI-Prolog through ≈190 lines of C-code and ≈140 lines of Prolog code. The C-code handles the low-level interface and conversion between the internal data representations of SWI-Prolog and MiniSat. The Prolog code provides a high-level interface for using the SAT solver in Prolog applications. The SAT solver is deterministic. It does not, and is not intended to, provide alternative satisfying assignments over backtracking. The MiniSat library for Prolog can be downloaded from Codish *et al.* (2006a). It includes three modules for the user: `cnf.pl`, `adder.pl` and `minisat.pl`.

The module `cnf.pl` exports the predicate `cnf/2`. A call to `cnf(F, Cnf)` transforms a propositional formula `F` to a conjunctive normal form `Cnf`. The implementation of `cnf(F,Cnf)` applies a Tseitin transformation Tseitin (1968) to ensure that the size of the conjunctive normal form `Cnf` is linear in that of the input formula `F`. The basic idea is to associate fresh variables with the subformulae of `F`. So, for example, if `F` is of the form `G+H` and variables `A, B, C` are associated respectively with subformulae `F, G, H`, then the clauses for the bi-implication `A` ↔ `(B` ∨ `C)` are introduced to the `Cnf` together with the clauses for the transformations of `B` ↔ `G` and `C` ↔ `H`. This process establishes the conjunctive normal form of `A` ↔ `F`. Adding the singleton clause `[A]` then provides a CNF equisatisfiable to F. The variables `A, B, C` are sometimes referred to as "Tseitin variables."

A technique proposed by Plaisted and Greenbaum (1986) is applied to reduce the size of the transformation. If it can be determined that we are interested only in the truth of a Tseitin variable, we can simplify the transformation only including the implication. In the above example, since we are interested only in the truth of Tseitin variable `A` (for `F`), the CNF created for `A` ↔ `(B` ∨ `C)`, will be `[[-A,B,C]]` rather than `[[-A,B,C],[A,-B],[A,-C]]`. Similarly when we are interested only in falsity of a Tseitin variable.

To demonstrate the use of the module, consider the following queries in which `T`, `T1`, and `T2` are the Tseitin variables.

```
?- cnf(X==Y,Cnf).
  Cnf = [[T], [-X, Y, -T], [X, -Y, -T]]
  ?- cnf((X*Y)+(-X*Z),Cnf).
  Cnf = [[T], [-T, T1, T2], [-T2, -X],
[-T2, Z], [-T1, X], [-T1, Y]]
```

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

```
:- module(adder,[sum/3]).

% sum(+,-,-).
sum([B],[S],(S==B)).
sum([B1,B2|Bs],Sum,F1*F2*F3) :-
        split([B1,B2|Bs],Xs,Ys),
        sum(Xs,S1,F1), sum(Ys,S2,F2), add(S1,S2,Sum,F3).

% split(+,-,-).
split([],[],[]).
split([X],[X],[0]).
split([X,Y|XYs],[X|Xs],[Y|Ys]) :- split(XYs,Xs,Ys).

% add(+,+,-,-).
add([X|Xs],[Y|Ys],[Z|Zs],(CXY==CarryXY)*(Z==SumXY)*Sum) :-
        halfadder(X,Y,SumXY,CarryXY),
        adder(Xs,Ys,CXY,Zs,Sum).

% adder(+,+,-,-).
adder([],[],Carry,[Z],Z==Carry).
adder([X|Xs],[Y|Ys],Carry,[Z|Zs],(CXY==CarryXY)*(Z==SumXY)*Rest) :-
        fulladder(X,Y,Carry,SumXY,CarryXY),
        adder(Xs,Ys,CXY,Zs,Rest).

% fulladder(+,+,+,-,-).
fulladder(X, Y, C, (X xor Y xor C), (C -> X+Y ; X*Y) ).
% halfadder(+,+,-,-).
halfadder(X, Y,    (X xor Y),       X*Y              ).
```

Fig. 1. A ripple-carry adder circuit.

Figure 1 illustrates the module `adder.pl`, which includes a textbook (Cormen *et al.*, 1990) ripple-carry circuit for binary addition. The module exports the predicate `sum/3`. A call to `sum(Unary, Binary, Psi)` constructs a Boolean circuit `Psi`. The arguments `Unary` and `Binary` are lists of Boolean formulae, the truth values of which encode unary and binary numbers respectively. These are the inputs and outputs of the circuit that can be seen as computing the binary sum of its unary inputs. The formula `Psi` is a propositional statement capturing the correspondence between unary and binary representations of nonnegative integers. The formula `Psi` is satisfiable exactly when `Unary` and `Binary` encode the same number. In the recursive call of the code, the formula `Psi=F1*F2*F3` is constructed by splitting the inputs into two equal halves (padding with a zero if necessary). From the recursive calls, the subformulae `F1` and `F2` relate the two halves of the inputs (unary) to the (binary) numbers `S1` and `S2`. The call to `add(S1,S2,Sum,F3)` constructs the formula `F3` for the binary `Sum` of `S1` and `S2`. Observe that the length of `Binary` is $\lceil \log_2 |\texttt{Unary}| \rceil$.

To demonstrate the use of the module, consider the following query constructing the circuit `Psi` for the inputs `Unary = [X+Y,X*Y,X==Y,X xor Y]` and outputs `Binary`.

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

X	*Y*	Unary	Binary
0	0	[0, 0, 1, 0]	[1, 0, 0]
0	1	[1, 0, 0, 1]	[0, 1, 0]
1	0	[1, 0, 0, 1]	[0, 1, 0]
1	1	[1, 1, 1, 0]	[1, 1, 0]

Table 1. *Truth values for the circuit summing* `Unary=[X+Y,X*Y,X==Y,X xor Y]`

```
% minimize(+,+).
minimize([],CNF) :- sat(CNF).
minimize([B|Bs],CNF) :- minimize(Bs,CNF), ( (B=0, sat(CNF)) -> true ; B=1 ).

% maximize(+,+).
maximize([],CNF) :- sat(CNF).
maximize([B|Bs],CNF) :- maximize(Bs,CNF), ( (B=1, sat(CNF)) -> true ; B=0 ).
```

Fig. 2. Minimization and maximization in `minisat.pl`.

```
?- sum([X+Y,X*Y,X==Y,X xor Y],Binary,Psi).
Binary = [S1, S2, S3]
Psi = (T1==X+Y)*(T2==(X==Y))*(T3==T1 xor T2)*(T4==T1*T2)*(T5==X*Y)*
   (T6==X xor Y)*(T7==T5 xor T6)* (T8==T5*T6)*(S1==T3 xor T7)*
   (S2==T4 xor T8 xor (T3*T7))* (S3==(T3*T7->T4+T8;T4*T8))
```

Table 1 illustrates the declarative meaning of the predicate depicting the truth values for `Unary` and `Binary` determined by `Psi` for the four truth assignments of `X` and `Y`.

The module `minisat.pl` exports four predicates:

- `solve(Cnf)` succeeds if and only if the formula `Cnf` in conjunctive normal form is satisfiable, binding its variables to truth values 0 (false) and 1 (true) that satisfy `Cnf`.
- `sat(Cnf)` succeeds if and only if the formula `Cnf` in conjunctive normal form is satisfiable. It is similar to `solve(Cnf)` but does not bind any variables.
- `minimize(Vec,Cnf)` is similar to `sat(Cnf)`. The additional argument `Vec` is a list of variables (e.g., occurring in `Cnf`). The variables of `Vec` are assigned the truth value that minimizes the binary number represented by `Vec` for all solutions of `Cnf`. If `Cnf` has no solutions (i.e., is unsatisfiable), it fails.
- `maximize(Vec,Cnf)` is similar to `minimize(Vec,Cnf)` but the assignment returned maximizes the value of the nonnegative integer represented by `Vec`.

The predicates `minimize/2` and `maximize/2` are illustrated in Figure 2. Consider the query `?- maximize(Xs, Cnf)`, where `Xs` is a list of k variables and `Cnf` a formula in conjunctive normal form. To solve the query, the basic idea is to pose k questions, one for each variable of `Xs`, to the SAT solver to determine the maximum value of `Xs`. Each question determines the satisfiability of `Cnf` when setting the next bit in `Xs` to its maximal value. Observe, in the code, that these questions are posed using `sat` so as not to bind variables in the formula. Note also that each of these questions is of size $O(|\texttt{Cnf}|)$.

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

```
% partialMaxSat(+,+).
partialMaxSat(Phi,Psi) :-
      sum(Psi,Max,SumPsi), cnf(Phi*SumPsi,Cnf),
      maximize(Max,Cnf),   solve(Cnf).
```

Fig. 3. Solving partial MAXSAT.

To demonstrate the use of the module, consider the following queries:

- `solve/1`. The call succeeds binding `X` and `Y` to truth values.

```
?- cnf(X==Y,Cnf), solve(Cnf).
X=0, Y=0
```

- `sat/1`. The call succeeds without binding `X` and `Y`.

```
?- cnf(X==Y,Cnf), sat(Cnf).
Yes
```

- `sum/3` with `solve/1`. The first call succeeds binding `X=0` and `Y=0`. In this case the circuit output is 1 (only one of the inputs is true under this assignment). The second call indicates that it is possible to satisfy two of the inputs under the assignment `X=0` and `Y=1`.

```
?- sum([X+Y,X*Y,X==Y,X xor Y],Sum,F), cnf(F,Cnf), solve(Cnf).
X = 0, Y = 0
Sum = [1, 0, 0]
?- sum([X+Y,X*Y,X==Y,X xor Y],[0,1,0],F), cnf(F,Cnf), solve(Cnf).
X = 0, Y = 1
```

- `sum/3` with `maximize/2`. The answer `Sum=[1,1,0]` indicates that it is possible to satisfy at most three of the four formulae. The call `maximize(Sum,Cnf)` binds only the elements of `Sum`. To obtain the maximizing assignment, `X=1`, `Y=1`, the call to `maximize` must be followed by a call to `solve(Cnf)`.

```
?- sum([X+Y,X*Y,X==Y,X xor Y],Sum,F), cnf(F,Cnf),
   maximize(Sum,Cnf), solve(Cnf).
 Sum=[1,1,0]
 X=1, Y=1
```

4 Solving Partial MAX-SAT

To solve a Partial MAXSAT problem for CNF formula φ and ψ, we seek an assignment that satisfies φ, and maximizes the number of clauses of ψ, which are satisfied. We solve the more general problem with φ, an arbitrary propositional formula, and ψ, a list of n propositional formulae.

The solution is illustrated in Figure 3. The arguments `Phi` and `Psi` correspond to φ and ψ, respectively. The key idea is to construct the formula `SumPsi` representing the sum of the n formulae in `Psi`. This results in a vector `Max` with $\lceil \log_2 n \rceil$ bits. We then aim to satisfy the conjunction `Phi*SumPsi`, while maximizing the number

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

represented by `Max`. Solving the query `maximize(Max,Cnf)` involves $O(\log n)$ calls to the SAT solver.

For example, the following query provides an assignment that satisfies $\varphi = X + Y$ and four of the seven formula in the second argument.

```
?- partialMaxSat(X+Y,[X*Y,X==Y,X xor Y,-X+Y, -X, -Y, X]).
X = 1, Y = 1
```

5 Conclusion

We define a Prolog library for solving SAT instances through an interface to the MiniSat solver. The combination of Prolog for manipulating Boolean formulae, and powerful SAT tools for solving them, is compelling. We can straightforwardly build solutions to difficult problems, with small and clean, pearlish code. We illustrate this in the paper by encoding a reduction from Partial MAXSAT to SAT. We have successfully used the library to solve many such problems (Annov *et al.*, 2006; Codish *et al.*, 2006b,c).

References

Annov, E., Codish, M., Giesl, J., Schneider-Kamp, P. and Thiemann, R. 2006. A sat-based implementation for RPO termination. URL: `http://www.lix.polytechnique.fr/~hermann/LPAR2006/short.html`. Short Paper at LPAR. Access date December 2006.

Arts, T. and Giesl, J. 2000. Termination of term rewriting using dependency pairs. *Theoretical Computer Science 236* (1–2), 133–178.

Cha, B., Iwama, K., Kambayashi, Y. and Miyazaki, S. 1997. Local search algorithms for partial MAXSAT. In *AAAI/IAAI*. 263–268.

Codish, M., Lagoon, V. and Stuckey, P. J. 2006a. Minisat library for Prolog. `URL:http://www.cs.bgu.ac.il/~mcodish/Software/pl-minisat.tgz`. Access date December 2006.

Codish, M., Lagoon, V. and Stuckey, P. J. 2006b. Solving partial order constraints for LPO termination. In *Term Rewriting and Applications (RTA)*. Lecture Notes in Computer Science, vol. 4098. Springer, Seattle, WA, 4–18.

Codish, M., Schneider-Kamp, P., Lagoon, V., Thiemann, R. and Giesl, J. 2006*c*. Sat solving for argument filterings. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Lecture Notes in Artificial Intelligence, vol. 4246. Springer, 30–44.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press, Chapter 29.

Dershowitz, N. 1982. Orderings for term-rewriting systems. *Theoretical Computer Science 17*, 279–301.

Eén, N. and Sörensson, N. 2004. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003 (Selected Revised Papers)*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science, vol. 2919. Springer, 502–518.

MiniSat 2006, MiniSat solver. Accessed December 2005. URL: `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`.

Miyazaki, S., Iwama, K. and Kambayashi1, Y. 1996. Database queries as combinatorial optimization problems. In *CODAS*. 477–483.

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press

Plaisted, D. and Greenbaum, S. 1986. A structure preserving clause form translation. *Journal of Symbolic Computation 2*, 293–304.

SWI-Prolog 2005. SWI-prolog. Accessed December 2005. URL: `http://http://www.swi-prolog.org/`.

Tseitin, G. 1968. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*. 115–125. Reprinted in J. Siekmann and G. Wrightson (eds), 1983. *Automation of Reasoning*, vol. 2. Springer-Verlag, Berlin, 466–483.

Wielemaker, J. 2003. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, F. Mesnard and A. Serebenik, Eds. Katholieke Universiteit Leuven, Heverlee, Belgium, 1–16. CW 371.

https://doi.org/10.1017/S1471068407003146 Published online by Cambridge University Press