A Hybrid BDD and SAT Finite Domain Constraint Solver

Peter Hawkins and Peter J. Stuckey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{hawkinsp,pjs}@cs.mu.oz.au

Abstract. Finite-domain constraint solvers based on Binary Decision Diagrams (BDDs) are a powerful technique for solving constraint problems over finite set and integer variables represented as Boolean formulæ. Boolean Satisfiability (SAT) solvers are another form of constraint solver that operate on constraints on Boolean variables expressed in clausal form. Modern SAT solvers have highly optimized propagation mechanisms and also incorporate efficient conflict-clause learning algorithms and effective search heuristics based on variable activity, but these techniques have not been widely used in finite-domain solvers. In this paper we show how to construct a hybrid BDD and SAT solver which inherits the advantages of both solvers simultaneously. The hybrid solver makes use of an efficient algorithm for capturing the inferences of a finite-domain constraint solver in clausal form, allowing us to automatically and transparently construct a SAT model of a finite-domain constraint problem. Finally, we present experimental results demonstrating that the hybrid solver can outperform both SAT and finite-domain solvers by a substantial margin.

1 Introduction

Finite-domain constraint satisfaction problems (CSPs) are an important class of problems with a wide variety of practical applications. There are many competing approaches for solving such problems, including propagation-based constraint solvers and Boolean Satisfiability (SAT) solvers.

We have previously shown how to represent many finite-domain constraint problems using Binary Decision Diagrams (BDDs) by modeling problems in terms of Boolean variables and representing both variable domains and constraints as formulæ over these variables [10]. The BDD representation of these formulæ allows us to "package" together groups of Boolean variables, where each group represents a set, multiset, or integer variable, and to make inferences on the sets of values that each group of variables can take *simultaneously*. This allows us to describe bounds, domain, and other types of propagation using BDD operations.

Another important class of finite-domain CSPs is the class of Boolean Satisfiability problems. While there is a variety of algorithms for solving SAT problems,

some of the most successful complete SAT solvers are based on variants of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [6]. The basic algorithm has existed for over forty years, and a great deal of effort has gone into producing robust and efficient implementations. Three key elements of a modern SAT solver are a suitable branching heuristic, an efficient implementation of propagation, and the use of conflict-directed learning [23].

Most SAT solvers operate on problems expressed as a set of clauses, (although recently there has been some interest in non-clausal representations for SAT problems [20]). This uniform representation allows the use of highly efficient data structures for performing unit propagation, and the generation of conflict clauses in order to avoid repeating the same subsearch, as well to drive heuristics that lead the search to new parts of the search space.

Modern SAT solvers are very effective on some kinds of problems, and practical SAT solvers such as MiniSAT [8] have been successfully applied to a wide range of problems including electronic circuit fault detection and software verification. The main disadvantage of SAT solvers is that some kinds of constraints are hard to model efficiently using clauses—for example the set constraint $|S| = k$ requires $\binom{n}{k-1} + \binom{n}{k+1}$ clauses to express, and the resulting propagation is weak.

Although both BDD and SAT solvers are Boolean solvers, they represent different tradeoffs in the general propagation-search paradigm: BDDs are expensive to manipulate but produce powerful propagation and minimize search, while SAT propagation is quick and weak, leading to more search but hopefully requiring less time overall. One of the unique advantages of the SAT solver comes from the use of nogood learning, which allows substantial search space reductions for structured problems. To a certain extent the strengths of each solver are complementary, and in this paper we show how to create a hybrid solver that inherits from both.

In this paper, we present a novel approach to combining a BDD-based finite-domain constraint solver and a SAT solver into an efficient hybrid constraint solver. While dual modeling is not new, the key contribution of this paper is an efficient algorithm for capturing the inferences of a finite-domain solver in clausal form. Not only does this allow us to use the conflict-directed learning and backjumping algorithms of a SAT solver in a finite-domain constraint solver, we can also use this algorithm to lazily construct a SAT model from a finite-domain constraint problem represented in BDD form, giving us some of the speed advantages of a SAT solver without the need for an explicit clausal model of a problem.

The contributions of this paper are:

- We show how we can construct a hybrid constraint solver by pairing finite-domain variables with a dual Boolean representation.
- We show how we can efficiently convert the inferences performed by a BDD based solver into clausal form. These inferences can be used to lazily construct a SAT model of a problem from a finite-domain model, and allow us to apply conflict-directed learning techniques to the inferences of a finite-domain solver.

– We demonstrate experimentally that combining BDD and SAT solvers can substantially improve performance on some set benchmarks.

2 Propagation-based Constraint Solving, BDDs and SAT

In this section we introduce definitions and notation for the rest of the paper. Most of these definitions are standard (see e.g. [15]).

We consider a typed set of variables $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_S$ made up of *integer* variables $\mathcal{V}_I$, for which we use lower case letters such as x and y, and *sets of integers* variables $\mathcal{V}_S$, for which we use upper case letters such as S and T.

A *domain* D is a complete mapping from a fixed (countable) set of variables $\mathcal{V}$ to finite sets of integers (for the integer variables in $\mathcal{V}_I$) and to finite sets of finite sets of integers (for the set variables in $\mathcal{V}_S$). A domain D_1 is said to be *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$.

We frequently use *set range* notation: $[L \mathrel{..} U]$ denotes the set of sets of integers $\{A \mid L \subseteq A \subseteq U\}$ when L and U are sets of integers. A set is said to be *convex* if it can be expressed as a range. The *convex closure* of a set S is the smallest range that includes S, and is written $conv(S)$. For example $conv(\{\{1,3\},\{1,4,5\},\{1,4,6\}\}) = [\{1\} \mathrel{..} \{1,3,4,5,6\}]$. We lift the concepts of convex and convex closure to domains in the natural way.

A *valuation* θ is a mapping of integer and set variables to correspondingly typed values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, S_1 \mapsto A_1, \ldots, S_m \mapsto A_m\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v_i) \in D(v_i)$ for all $v_i \in vars(\theta)$.

A constraint is a restriction placed on the allowable values for a set of variables. We shall be interested in constraints over integer and set variables. We define the *solutions* of a constraint c to be the set of valuations θ that make that constraint true, i.e. $solns(c) = \{\theta \mid (vars(\theta) = vars(c)) \wedge (\models \theta(c))\}$

We associate a *propagator* with every constraint. A propagator f is a monotonically decreasing function from domains to domains, so $D_1 \sqsubseteq D_2$ implies that $f(D_1) \sqsubseteq f(D_2)$, and $f(D) \sqsubseteq D$. A propagator f is *correct* for a constraint c if and only if for all domains D: $\{\theta \mid \theta \in D\} \cap solns(c) = \{\theta \mid \theta \in f(D)\} \cap solns(c)$ This is a weak restriction since, for example, the identity propagator is correct for any constraints. We assume that all propagators are correct.

A *set bounds propagator* f for constraint c is a propagator that maps convex domains to convex domains. For set problems typically set bounds propagators are employed.

A *propagation solver* for a set of propagators F and a domain D repeatedly applies the propagators in F starting from the domain D until a fixpoint is reached.

2.1 Binary Decision Diagrams

A *Reduced Ordered Binary Decision Diagram* (ROBDD) is canonical representation of a propositional expression (up to reordering on the propositions), which permits an efficient implementation of many Boolean function operations, including conjunction ($\wedge$), disjunction ($\vee$), existential quantification ($\exists$). In an ROBDD each node is either 1 (true) or 0 (false) or of the form $n(v, t, e)$ where v is a Boolean variable, and t and e are ROBDDs. For more details the reader is referred to the work of Bryant [3]. The modeling of constraint problems using ROBDDs is discussed extensively in [10].

2.2 SAT and Unit Propagation

A *proposition* $p \in \mathcal{P}$ is a Boolean variable, where $\mathcal{P}$ denotes the universe of Boolean variables. A *literal* l is either a proposition p or its negation $\neg p$. The *complement* of a literal l, $\neg l$ is $\neg p$ if $l = p$ or p if $l = \neg p$. A *clause* C is a disjunction of literals. An *assignment* is a set of literals A such that $\forall p \in \mathcal{P}. \{p, \neg p\} \not\subseteq A$. An assignment A *satisfies* a clause C if one of the literals in C appears in A.

A SAT solver takes a conjunction (or set) of clauses and determines if there is an assignment that simultaneously satisfies all the clauses. Complete SAT solvers typically involve some form of the DPLL algorithm which combines search and propagation by recursively fixing the value of a proposition to either 1 (true) or 0 (false) and using unit propagation to determine the logical consequences of each decision made so far. The unit propagation algorithm finds all clauses $p_1 \vee p_2 \vee \ldots \vee p_k$ where at least $k - 1$ of the literals are known to be false, and asserts the remaining literal to be true (since it is the only possible way for the clause to be satisfied). If all k literals are known to be false, then we have discovered a conflict in the set of assignments made so far and we must backtrack. Unit propagation can be performed very efficiently by using watched literal techniques [16].

Modern SAT solvers make use of *nogood learning* in order to reduce the search space, and guide search away from unprofitable areas. Nogood learning relies on building an *implication graph* for values derived by unit propagation (although the graph is usually represented implicitly). The implication graph is a directed acyclic graph where the nodes $l@t$ are pairs of literal l and timestamp t indicating the time the literal became known.

Unit propagation on a clause $l_1 \vee \cdots \vee l_n$ from nodes $\neg l_1@t_1, \ldots, \neg l_{i-1}@t_{i-1}$, $\neg l_{i+1}@t_{i+1}, \ldots, \neg l_n@t_n$, $1 \leq i \leq n$ adds a new node $l_i@t_i$ where t_i is the maximum timestamp $t_i = \max\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}$ as well as arcs $\neg l_j@t_j \rightarrow l_i@t_i, 1 \leq j \neq i \leq n$. If we discover a conflict from a clause $l_1 \vee \cdots \vee l_n$ using literals $\neg l_1@t_1, \ldots, \neg l_n@t_n$, we add a node $\bot@t$ where $t = max(t_1, \ldots t_n)$ and arcs $\neg l_i@t_i \rightarrow \bot@t, 1 \leq i \leq n$.

When we derive a contradiction then any cut across the graph that leaves the contradiction on one side (the conflict side), and all the decisions (nodes without incoming arcs) on the other side (the reason side) defines a *nogood*. Nogoods can

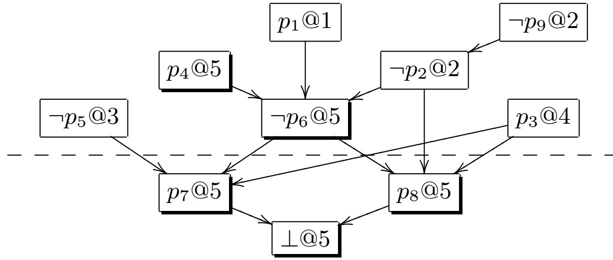

Fig. 1. An example implication graph, showing a possible set of implications on the set of clauses $(p_9 \vee \neg p_2) \wedge (\neg p_1 \vee p_2 \vee \neg p_4 \vee \neg p_6) \wedge (\neg p_3 \vee p_5 \vee p_6 \vee p_7) \wedge (p_2 \vee \neg p_3 \vee p_6 \vee p_8) \wedge (\neg p_7 \vee \neg p_8)$.

be added to the solver's store of learnt clauses in order to assist with future decisions.

We have a choice of which cut of a conflict graph to take. The *decision cut* simply keeps all the decisions (nodes without parents) (which corresponds to the cut commonly chosen by Conflict-Directed Backjumping schemes in a CSP context). A cut scheme that has been shown to be more effective experimentally in the SAT context, called 1-UIP [22], is to choose a cut that places only nodes with the same timestamp as the contradiction on the conflict side; of these, only nodes between the Unique Intersection Point (a node which dominates all nodes between the conflict and itself) closest to the conflict and the conflict itself are placed on the conflict side.

Example 1. Consider the implication graph shown in Figure 1. All the nodes with the same timestamp as the contradiction are shown with shadow. The 1-UIP cut is shown as the dashed line, since $\neg p_6$ is the closest node to the conflict which is included on all paths from the decision at time 5 to the confict. The nogood generated is $p_2 \vee \neg p_3 \vee p_5 \vee p_6$. The decision cut generates $\neg p_1 \vee \neg p_3 \vee \neg p_4 \vee p_5 \vee p_9$.

3 A Hybrid SAT and Finite-Domain Constraint Solver

Since the powerful inference abilities of a finite-domain constraint solver and the efficient propagation and conflict-directed learning of a SAT solver are to some extent complementary, we would like to construct a hybrid solver that combines the advantages of both. Such a hybrid solver can be created through a process of dual modeling, where the same constraint problem is modeled in multiple cooperating solvers.

There are several important points we must consider when creating a dual model of a constraint problem:

- How should the problem variables and their domains be modeled in each solver?
- How should deductions be communicated between the two solvers during the search process?

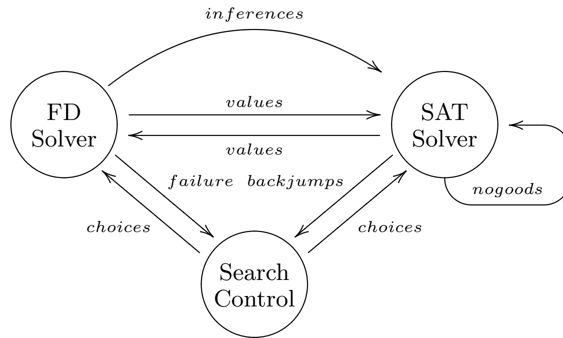

Fig. 2. Interactions between FD and SAT solvers and search.

– Which of the problem constraints be modeled in each solver, and how should they be modeled?
– How should the execution of the search procedure and the two solvers be scheduled?

In the next three sections we outline our approach to these problems. An illustration of the interaction of the two solvers is shown in Figure 2.

3.1 Boolean Modeling of Constraint Variables and Domains

As a first step in constructing a hybrid model of a constraint problem, we need to establish how we will represent the constrained variables and their domains in each solver. Since finite-domain solvers are more expressive than Boolean models, it is reasonable to assume that we already have a finite-domain model for a problem, and we need only consider the how to represent finite-domain variables and domains in a propositional manner suitable for a SAT solver

Note that in the specific case of our hybrid solver the finite-domain solver is based on BDDs and also operates using a very similar Boolean model [10], but the discussion here is completely general and independent of the structure of the finite-domain solver.

Set variables are natural to model in terms of propositional logic. Suppose S is a set variable over a domain $[\emptyset \mathrel{..} \{1, \ldots, n\}]$. We can model S as a set of propositions $P(S) = \{S_1, \ldots, S_n\}$, where $S_i \leftrightarrow i \in S$. Constraints then map to propositional formulæ over these variables. For example, the expression $S = A$ for some fixed set A in $[\emptyset \mathrel{..} \{1, \ldots, n\}]$ is equivalent to the propositional formula $B(S = A) = \bigwedge_{i=1}^{n} S_i \leftrightarrow (i \in A)$.

Finite domain integer variables do not have such a natural propositional representation. Many encodings are possible — we outline here direct and log encodings (for more details, see, e.g. [21]).

In the *direct* or unary encoding, we model a finite domain integer variable x over a domain $\{0, \ldots, n\}$ by the propositions $P(x) = \{x_0, \ldots, x_n\}$ where x_i holds if and only if $x = i$. Since this representation would allow x to take on multiple

values at once, we must also add constraints stipulating that x must take exactly one value: $(x_0 \vee \cdots \vee x_n) \wedge \bigwedge_{i=0}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \vee \neg x_j$. The expression $x = i$ for a fixed value i is modeled by the propositional formula $B(x = i) = x_i \wedge \bigwedge_{j=0, j \neq i}^{n} \neg x_j$. Note that we can think of the direct encoding as encoding integers as singleton sets.

Another representation of finite domain integer variables is a *log* or binary encoding, which uses propositions $P(x) = \{x_0, x_1, \ldots, x_k\}$ that correspond to a binary representation $x = x_0 + 2x_1 + \cdots + 2^k x_k$, where $k = \lfloor \log_2 n \rfloor$. The expression $x = i$ for a fixed value i is modeled as the propositional formula $B(x = i) = \bigwedge_{j=0}^{k} (x_j \leftrightarrow i_j)$ where i_j, $0 \leq j \leq k$ is the jth bit of the binary encoding of the number i. The log encoding does not require any auxiliary constraints to ensure that each variable takes a single value. The log encoding is known to produce weaker unit propagation than the direct encoding [21] but produces smaller models in some cases and allows operations such as addition to be defined more compactly.

Since we can map set or integer values to propositional formulæ, we can easily map the domains of constrained variables as well. Given a variable v, we can define the propositional representation $B(D(v))$ of the domain $D(v)$ as $B(D(v)) = \bigvee_{d \in D(v)} B(v = d)$. Note that $B(D(v))$ is equivalent to a conjunction of propositions if v is a set variable and $D(v)$ is a convex set, or if v is an arbitrary integer variable in the direct encoding. However, it is important to note that with any of these encodings we can in theory represent arbitrary domains, not necessarily just those defined by conjunctions of propositions. However, SAT solvers are limited to domains represented as conjunctions of propositions, and hence we confine our attention to conjunctive domains in this paper.

Example 2. Consider the set variable S which ranges over $[\emptyset \mathinner{..} \{1, 2, 3\}]$, then $P(S) = \{S_1, S_2, S_3\}$. Then for the domain $D(S) = [\{1\} \mathinner{..} \{1, 2\}]$ we have $B(D(S)) = S_1 \wedge \neg S_3$. For the non convex domain $D'(S) = \{\{1\}, \{2, 3\}\}$ then $B(D'(S)) = (S_1 \wedge \neg S_2 \wedge \neg S_3) \vee (\neg S_1 \wedge S_2 \wedge S_3)$.

For an integer x ranging over $\{0, 1, 2, 3\}$ then the domain $D(x) = \{1, 2\}$ in the direct encoding is simply $B(D(x)) = \neg x_0 \wedge \neg x_3$, while in the log encoding it is $B(D(x)) = (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1)$.

3.2 Channeling Constraints and Clause Generators

The most interesting part of constructing a dual solver is the channeling of information between the finite-domain and SAT solvers. The standard approach to channeling information between solvers is to create channeling constraints, which ensure that the domains of the corresponding variables in each solver are equal. While we could use such an approach when coupling a SAT solver and a finite-domain solver, simply communicating information about the values in a domain is insufficient to allow the SAT solver to build an inference graph and perform nogood computations. In order for the SAT solver to compute meaningful nogoods we also need to communicate the *reasons* for any deductions made by finite-domain constraint propagation.

Our basic strategy is for each proposition p inferred through finite-domain propagation we will derive an *inference clause*. An inference clause is a SAT clause that is a logical consequence of a finite-domain propagator, and that would have derived p through unit propagation from the variables that were fixed at the time that the finite-domain propagator deduced p. This new clause effectively encapsulates an inference of a finite-domain propagator in a form that the SAT solver can understand.

By adding inference clauses to the clause set of the SAT solver, we can perform nogood learning and conflict-directed backjumping on the inferences of the finite-domain solver. We can either add inference clauses explicitly to the SAT solver's store of learnt clauses, or extend the SAT solver so that inference clauses are used implicitly for performing nogood calculations. In the former case, we are effectively using finite-domain propagators as *generators* for SAT clauses, thus removing the need for a SAT model of the problem. As the search progresses and more inference clauses are added, effectively a SAT model of each finite domain constraint is constructed.

While it would be possible to augment any finite-domain solver to generate inference clauses by extending the propagator implementations on a case-by-case basis, since our FD solver is based on BDDs we can derive these clauses efficiently and automatically.

Let f be a propagator for a constraint c and D be a domain, and suppose $f(D) = D'$ where $D' \neq D$. Let $vars(c) = \{v_1, \ldots, v_n\}$. Clearly we have that $(c \wedge \bigwedge_{i=1}^{n} \bigvee_{x \in D(v_i)} v_i = x) \rightarrow \bigwedge_{i=1}^{n} \bigvee_{x \in D'(v_i)} v_i = x$. In the Boolean formalism this is equivalent to $(B(c) \wedge \bigwedge_{i=1}^{n} B(D(v_i))) \rightarrow \bigwedge_{i=1}^{n} B(D'(v_i))$, where $B(c)$ is a formula representing constraint c.

Now suppose $B(D)$ and $B(D')$ are conjunctions of propositions. This assumption holds if we restrict ourselves to use set bounds propagators [10]. We can treat these conjunctions as sets. Let $p \in B(D'(v_i)) \setminus B(D(v_i))$ be a *newly inferred proposition*. Then in any context where the constraint c holds, the clause $(\bigwedge_{i=1}^{n} B(D(v_i))) \rightarrow p$ or equivalently $(\bigvee_{i=1}^{n} \neg B(D(v_i))) \vee p$ also holds, and p would have been derived from this clause through unit propagation. We call this clause the *simple inference* of p, which we can add as a new (redundant) constraint.

Example 3. Consider the constraint $c \equiv |S| = x$ where S ranges over $[\emptyset \mathinner{..} \{1, \ldots, 5\}]$ and x ranges over $\{0, 1, 2, 3, 4, 5\}$. Let $D(S) = [\{1, 2\}, \{1, 2, 4\}]$ and $D(x) = \{3, 4, 5\}$. Then the strongest set bounds propagator f for c is such that $f(D) = D'$ where $D'(S) = [\{1, 2, 4\}, \{1, 2, 4\}]$, $D'(x) = \{3\}$.

In the Boolean representation $B(D(S)) = S_1 \wedge S_2 \wedge \neg S_3 \wedge \neg S_5$, $B(D(x)) = \neg x_0 \wedge \neg x_1 \wedge \neg x_2$, $B(D'(S)) = S_1 \wedge S_2 \wedge \neg S_3 \wedge S_4 \wedge \neg S_5$, and $B(D'(x)) = \neg x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge \neg x_5$. The newly inferred propositions are S_4, $\neg x_4$, $\neg x_5$, and x_3. Considering $\neg x_4$, we can see that the clause $(S_1 \wedge S_2 \wedge \neg S_3 \wedge \neg S_5 \wedge \neg x_0 \wedge \neg x_1 \wedge \neg x_2) \rightarrow \neg x_4$ or equivalently $\neg S_1 \vee \neg S_2 \vee S_3 \vee S_5 \vee x_0 \vee x_1 \vee x_2 \vee \neg x_4$ is valid and captures the deduction on x_4 made by the constraint propagation. Similarly we can produce one clause for each of the other propositions deduced by propagation.

```
mininf(p,c,P)
    let M := P
    let {p_1,...,p_m} := P
    for i := 1..m
        if ¬ sat(B(c) ∧ ¬p ∧ (M \ {p_i}))
            M := M \ {p_i}
    return M
```

Fig. 3. A generic algorithm for finding minimal reasons for inferences

The simple clauses generated above, while logically valid, are not minimal, and hence are unlikely to generate much useful propagation since they will only produce inferences when all but one of the Boolean variables in the clause are fixed. Since the decision variables of all of the variables involved in the constraint are included in the clause, we are unlikely to revisit this particular combination of variable assignments again as the search progresses. In order to maximize the chance of generating useful inferences, we need to generate minimal reasons for each inference.

We define a *minimal inference* of p as follows. Let $P = \bigcup_{i=1}^{n} B(D(v_i)))$ be the set of candidate propositions for the minimal inference of p and let $M = \{m_1, \ldots, m_k\} \subseteq P$. We say M is a sufficient set of reasons for p if $(B(c) \wedge \bigwedge_{j=1}^{k} m_j) \rightarrow p$. We say M is a minimal set of reasons for p if M is a sufficient set of reasons for p and for any sufficient set $N \subseteq M$ we have $M = N$.

Example 4. For the propagation in Example 3 some minimal reasons define the inference clauses:

$$
\begin{aligned}
(S_1 \wedge S_2 \wedge \neg S_3 \wedge \neg S_5 \wedge \neg x_0 \wedge \neg x_1 \wedge \neg x_2) &\rightarrow S_4 \\
(\neg S_3 \wedge \neg S_5) &\rightarrow \neg x_4 \\
\neg S_3 &\rightarrow \neg x_5 \\
(S_1 \wedge S_2 \wedge \neg S_3 \wedge \neg S_5 \wedge \neg x_2) &\rightarrow x_3
\end{aligned}
$$

Note that other minimal reasons exist, e.g. $\neg S_5 \rightarrow \neg x_5$.

In general a minimal set of reasons is not unique. It is also possible that the same proposition would be inferred by multiple propagators, each propagator producing different minimal sets of reasons, and hence the reasons deduced for the value of a proposition depend on the order of execution of the propagators.

A generic algorithm for computing minimal reasons for inferences is shown in Figure 3. This algorithm needs $O(n)$ satisfiability checks where n is the number of candidate propositions. Such an algorithm would be prohibitive to use in general, although a divide-and-conquer version similar to the QUICKXPLAIN algorithm of [11] would perform better.

However, we can specialize this algorithm for use in a BDD context, leading to an efficient algorithm for inferring a minimal set of reasons for a deduction p. First we generate the BDD for $G = \bar{\exists}_{vars(P)} B(c) \wedge \neg p$, that is the constraint c with the information that p is false, restricted to the propositions P of interest

```
BDDmininf(G,P,M)
    if G = 0 return M
    let G = n(v, G_v, G_¬v)
    let P = n(v_P, P_{v_P}, P_{¬v_P})
    if P_{v_P} = 0 then l := ¬v_P else l := v_P
    if v = v_P then
        if BDDsatconj(G_¬l, P_l) then return BDDmininf(G_l, P_l, M ∪ {l})
        else return BDDmininf(G_l ∨ G_¬l, P_l, M)
    else
        return BDDmininf(G, P_l, M)
```

Fig. 4. A BDD-based algorithm for finding minimal reasons for inferences

in determining the minimal set M. We then recursively visit the BDD G determining whether each proposition in order is required for a contradiction. The resulting algorithm is shown in Figure 4.

The algorithm works as follows. The initial call is BDDmininf(G, P, $\emptyset$). The first argument is the remaining BDD, the second the remaining set of possible reasons (represented as a conjunction BDD) and the last is the set of propositions in the minimal inference so far (which in practice is also represented by a BDD). The algorithm maintains the invariant that $G \wedge P$ is unsatisfiable.

If the BDD G is 0 ($false$) then no further reasons are required and we return the set M. The BDD cannot be 1 ($true$) since it must be unsatisfiable when conjoined with P. We find the literal l in P with the least variable v_P in the BDD variable ordering. If the variable v at the top of the BDD G is not the same then the literal l is irrelevant since it does not appear in G, and so we recurse, looking at the next least literal. Otherwise we check whether $G_{\neg l} \wedge P_l$ is satisfiable which corresponds to if the variable v takes the opposite value from l. If this is satisfiable then l is required to make a minimal inference, since removing it would lead to something satisfiable with the remainder P_l. Hence we add l to M and follow the l choice in the BDD G recursively. Otherwise l is not required, since the remainder of P is sufficient to ensure that both branches of G are unsatisfiable. We project out the variable v from G obtaining $G_v \vee G_{\neg v}$ (this requires building new BDD nodes) and recursively proceed.

Note that BDDsatconj(G, H) checks whether the conjunction H is a satisfying assignment for G, and simply requires following the path H in the BDD G. It does not require constructing new BDD nodes.

3.3 Constraint Modeling, Scheduling and Search

We have a great deal of flexibility in modeling the problem constraints in each solver. However, as discussed in Section 3.2, we do not need to construct a model of a constraint problem for the SAT solver — we can lazily construct it from the finite-domain model. However, it is also possible that in some cases better performance may be obtained by an explicit dual model, although this is not borne out by the experiments in Section 4. Due to space constraints we do not discuss explicit SAT models of constraint problems here.

We also have a great deal of flexibility in deciding how to schedule the propagation of the SAT and BDD solvers. In our solver, we choose to treat the SAT solver as a single "propagator" which is executed at a higher priority than any other propagator. This ensures that the cheap SAT inferences are performed before the relatively expensive finite-domain inferences. Various labeling heuristics can be used, and we present experimental results for a sequential labeling heuristic as well as the Variable-State-Independent-Decaying-Sum (VSIDS) SAT solver heuristic.

4 Experimental Results

We have implemented a hybrid BDD and SAT solver in the Mercury system [19]. The BDD solver makes use of the CUDD BDD package [18] while the SAT solver is an interface to MiniSAT [8], but exporting control of search to Mercury.

The "Social Golfers" problem (problem `prob010` of CSPLib) is problem commonly used as a benchmark for set CSP solvers. The aim of this problem is to arrange $N = g \times s$ golfers into g groups of s players for each of w weeks, such that no two players play together more than once. We can model this problem as a set constraint problem using a $w \times g$ matrix of set variables v_{ij}, where $1 \leq i \leq w$ is the week index and $1 \leq j \leq g$ is the group index. See [10] for the model in detail, although in this paper we have added constraints that allocate the golfers in sequential order to the first week in order to remove symmetries.

All test cases were run on a cluster of 8 identical 2.4Ghz Pentium 4 machines with 1Gb RAM and 2Gb swap space. Each test case was repeated 3 times, and the lowest of the 3 results used. In the result tables: "*" denotes a test case without a solution, "—" denotes failure to complete a test case within 10 minutes, and "×" denotes an out of memory error.

From Table 1, we can see that the best of the hybrid solvers outperforms the BDD bounds and (split) domain solvers that were presented in [10] on almost all of the test cases.[1] Using simple clause learning (B+SB) is not useful, since the overhead of deriving and storing nogoods is not repaid through search space reduction. The most surprising column is perhaps B+M which shows the overhead of minimizing the clauses without reducing the search space. It appears that generating minimized nogoods requires less than double the time taken for the original propagation. Once we make use of the minimal clauses (B+MB) by recording nogoods and performing backjumping we often improve on the bounds solver, but interestingly adding all of the inferred clauses to the SAT solver (B+MA) can lead to substantial further reductions in the search space. The B+MA column corresponds to a hybrid where in some sense we lazily build a CNF model of the problem using only the "useful" clauses found by the BDD model.

Table 2 presents results obtained using a Variable-State-Independent-Decaying-Sum heuristic, which is commonly used by SAT solvers. This table also contains

[1] Note that the BDD bounds solver is substantially faster on these examples than solvers such as Eclipse or Mozart due to better modeling capabilities and a more efficient implementation language [10].

Problem	Domain		Bounds		B+SB		B+M	B+MB		B+MA	
w-g-s	time /s	fails	time /s	fails	time /s	fails	time /s	time /s	fails	time /s	fails
2-5-4	0.1	**0**	0.1	30	<**0.1**	28	0.1	0.1	23	0.1	11
2-6-4	**0.1**	**0**	0.4	2036	0.8	1212	1.2	0.5	499	**0.1**	45
2-7-4	0.3	**0**	1.2	4447	1.9	2087	3.7	0.8	534	**0.2**	90
2-8-5	1.3	**0**	—	—	—	—	—	—	—	**0.8**	472
3-5-4	0.2	**0**	**0.1**	30	**0.1**	28	**0.1**	**0.1**	23	**0.1**	11
3-6-4	1.3	**0**	1.3	2039	1.6	1215	2.5	1.0	502	**0.2**	48
3-7-4	8.0	**0**	3.6	4492	3.7	2131	7.7	1.8	551	**0.5**	99
4-5-4	0.5	**0**	**0.1**	30	0.2	28	0.2	0.2	23	0.2	11
4-6-5	98.0	**0**	19.6	12747	23.1	8600	33.5	9.9	2323	**0.7**	81
4-7-4	—	—	7.0	4498	6.3	2137	12.2	2.9	557	**0.8**	**105**
4-9-4	—	—	**1.5**	71	1.7	69	2.2	2.0	43	1.9	**32**
5-4-3 (*)	29.0	**5165**	87.6	63519	140.8	43402	190.9	52.3	10440	**12.0**	9568
5-5-4	2.9	**41**	5.4	2686	7.1	1661	12.6	9.4	1356	**2.3**	1167
5-7-4	—	—	11.9	4583	9.7	2195	19.7	4.6	608	**1.5**	**159**
5-8-3	7.3	**0**	**0.7**	14	**0.7**	13	0.9	0.9	13	0.9	12
6-4-3 (*)	22.4	2132	130.3	61647	183.8	42986	235.7	12.6	1774	**2.1**	**908**
6-5-3	1.4	**82**	3.0	1455	4.2	967	6.9	2.5	327	**0.9**	282
6-6-3	1.3	**0**	**0.3**	5	**0.3**	5	0.4	0.4	5	0.4	5
7-5-3	—	—	—	—	—	—	—	127.5	11945	**18.2**	**6154**
7-5-5 (*)	<**0.1**	**0**	0.9	131	1.0	131	1.2	1.0	99	0.8	100

Table 1. Performance results for the Social Golfers problem, using a sequential, smallest-element-in-set labeling heuristic. "Domain" = BDD Domain solver of [10]. "Bounds" = BDD Bounds solver of [10]. "B+SB" = Bounds + simple clause learning + backjumping. "B+M" = Bounds + minimized clause learning, no backjumping. "B+MB" = Bounds + minimized clause learning + backjumping. "B+MA" = Bounds + adding minimized clauses to the SAT solver as learnt clauses

a comparison with the SAT solvers MiniSAT and zChaff, the pseudo-boolean SAT solver MiniSAT+, and a dual BDD and SAT model with all constraints but cardinality constraints duplicated as SAT clauses. It appears that as in the sequential case, the B+MA technique performs the best out of all of the solvers. The SAT solvers are frequently disadvantaged in this comparison because the representation of the cardinality constraints frequently requires a very large number of clauses.

5 Related Work and Conclusion

At present we are unaware of any other BDD based propagation solvers than our own, so in that sense the work is completely novel. But at a feature level there are relationships with much previous work.

Modeling of finite domains as Booleans is well understood and a standard form of dual modeling (see e.g. [4]). There has been interest in encoding CSPs as SAT problems [21]. The novel part of our approach is representing the actions of a finite-domain propagator in terms of clausal inferences. Even though the propagation rules of [4] and membership rules of [1] used to model propagation

Problem *w-g-s*	B+MB time /s	B+MB fails	B+MA time /s	B+MA fails	Dual time /s	Dual fails	MiniSAT time /s	MiniSAT fails	zChaff time /s	zChaff fails	MiniSAT+ time /s
2-5-4	**0.1**	21	**0.1**	22	**0.1**	4	0.2	273	0.2	767	0.2
2-6-4	0.3	83	**0.1**	64	**0.1**	12	0.5	125	0.9	1850	0.3
2-7-4	0.7	161	**0.2**	119	**0.2**	10	1.4	282	2.7	2858	0.6
2-8-5	3.6	437	1.3	622	**0.8**	130	×	×	×	×	1.5
3-5-4	**0.1**	26	**0.1**	24	0.5	215	0.3	534	2.2	7018	0.6
3-6-4	0.5	102	**0.3**	58	1.5	374	0.9	488	2.0	2715	1.2
3-7-4	1.1	128	**0.6**	92	3.5	493	6.9	7517	3.3	2348	2.1
4-5-4	**0.2**	27	0.4	122	2.0	900	0.5	543	3.3	9580	1.2
4-6-5	2.1	186	**1.3**	304	9.7	2135	17.0	763	×	×	4.0
4-7-4	**1.0**	40	**1.0**	98	13.1	1546	53.8	47801	281.7	166710	4.7
4-9-4	**2.0**	35	**2.0**	59	41.3	2161	—	—	×	×	12.9
5-4-3 (*)	66.0	13126	5.6	5876	11.1	10750	**0.4**	4554	1.4	9044	1.2
5-5-4	9.4	667	1.9	581	2.4	785	**1.4**	3291	2.3	7230	2.6
5-7-4	2.2	96	**1.5**	104	29.3	3458	—	—	—	—	8.5
5-8-3	**1.3**	35	1.7	425	10.7	1212	—	—	46.6	110980	7.3
6-4-3 (*)	0.3	74	**0.2**	71	1.3	637	0.3	4307	0.8	5975	1.1
6-5-3	11.1	1062	4.3	2801	8.2	3669	**0.8**	7795	74.4	186858	2.2
6-6-3	**0.4**	16	1.0	275	5.6	1310	2.2	17869	2.6	11666	3.5
7-5-3	127.6	14237	18.0	7018	35.8	118876	66.1	197714	396.8	562386	**6.3**
7-5-5 (*)	86.1	2513	2.0	139	**1.4**	97	8.8	1858	16.9	6910	6.9

Table 2. Performance results for the Social Golfers problem, using a VSIDS labeling heuristic. "B+MB" = Bounds + minimized clause learning + backjumping. "B+MA" = Bounds + adding minimized clauses to the SAT solver as learnt clauses. "Dual" = Bounds + minimized clause learning + backjumping + Dual SAT/BDD model. "MiniSAT" = MiniSAT SAT solver [8]. "zChaff" = zChaff SAT solver [16]. "MiniSAT+" = MiniSAT with Pseudo-boolean extensions [9]

are similar, they only define directional inferences for modeling the behavior of propagators, rather than directly modeling logical inferences.

There is a substantial body of work on look back methods in constraint satisfaction (see e.g. Dechter [7], chapter 6), but there seems little evidence of success for look back methods that combine with propagation. The most successful combination appears to be Forward Checking with Conflict Directed Backjumping (FC-CBJ) [17]. But other work by Bessière and Regin [2] calls into question whether FC-CBJ should be considered competitive. They showed that maintaining arc consistency (MAC) with an appropriate search strategy is usually better than FC-CBJ, and that conflict directed backjumping did not appear to improve the empirical performance of MAC. We believe our results do not match this conclusion primarily because we are able to use the highly efficient data structures of a SAT solver for maintaining and propagating nogoods, as well as an efficient BDD-based algorithm for calculating dependencies, thus making conflict-directed backjumping a worthwhile investment.

The only propagation based solver we are aware of that incorporates nogoods is the PaLM system [12]. It has been used to investigate new search methods

based principally on dynamic backtracking. Like most previous work on nogoods in CSPs it keeps explanations and derives nogoods based on the constraints and decisions made in the search, rather than a SAT solver which simply records the inferences. Effectively it always uses decision cuts, instead of the more powerful 1-UIP nogoods. The system has been used to show that nogoods can be used constructively inside a propagation based solver [13].

The closest work to our own is that of Katsirelos and Bacchus [14], which showed that one could use nogood technology derived from SAT for storing and managing nogoods in a CSP system using FC-CBJ. Unfortunately no results were presented that combined MAC with nogood recording, which appears to limit the performance of the resulting solver. Another difference is that they don't appear to record the FC inferences as clauses, acting rather like S+MB rather than S+MA. They also reported no success with using SAT-derived labeling heuristics, which does not match our experience. The closest work to our implication detection algorithm is that of Damiano and Kukula [5]. In this work, however, the BDDs are static and not used for finite-domain propagation.

The use of nogoods has lead to a substantial improvement in the ability of SAT solvers to solve practical problems. SAT solvers treat nogoods both more efficiently than traditional CBJ approaches, but also learn better nogoods from a conflict. Our work shows at least in the case of set bounds propagation there is an advantage to using nogoods, because we can quickly determine minimal inferences and make use of the clever SAT technology to both generate and efficiently propagate nogoods.

Although we have hybridized a BDD-based finite domain constraint solver, we could similarly hybridize a more conventional finite domain propagation constraint solver by hard coding the minimal inferences for each primitive constraint supported by the solver. The advantage of BDD-based approach is that it is completely generic, and requires no extra work to support the wide variety of constraints that can be modeled as BDDs and is surprisingly fast.

For future work, we intend to try combining nogood learning with domain propagation, although this is more difficult to achieve, and possibly of less value. We will also try adding a 0-1 Integer Linear Programming solver into the hybrid solver, in the hope of producing a solver with better optimization capabilities.

In conclusion, we have demonstrated that by combining finite-domain constraint propagation and SAT techniques we can produce a highly efficient hybrid solver, which outperforms either of the original solvers on benchmarks. The high performance of this solver is a result of an efficient algorithm for accurately capturing the inferences of a finite-domain constraint solver as SAT clauses.

References

[1] K. Apt and E. Monfroy. Constraint programming viewed as rule-based programming. *TPLP*, 1(6):713–750, 2001.

[2] C. Bessière and J.-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP96*, LNCS. Springer, 1996.

[3] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. ISSN 0360-0300.

[4] C. Choi, J. Lee, and P. J. Stuckey. Propagation redundancy in redundant modelling. In F. Rossi, editor, *Proceedings of CP2003*, volume 2833 of *LNCS*, pages 229–243. Springer-Verlag, 2003.
[5] R. Damiano and J. Kukula. Checking satisfiability of a conjunction of BDDs. In *Proceedings of DAC '03*, pages 818–823, New York, NY, USA, 2003. ACM Press.
[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
[7] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
[8] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of SAT 2003*, volume 2919 of *LNCS*, pages 502–518, May 2003.
[9] N. Eén and N. Sörensson. Minisat+. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat+.html, 2005.
[10] P. Hawkins, V. Lagoon, and P. J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, July 2005.
[11] U. Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, 2001.
[12] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, 2000.
[13] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of CP2000*, volume 1894 of *LNCS*, pages 249–261, Singapore, Sept. 2000. Springer-Verlag.
[14] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of CP2003*, volume 2833 of *LNCS*, pages 873–877. Springer, 2003.
[15] K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction*. The MIT Press, 1998.
[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC 2001)*, June 2001.
[17] P. Prosser. Hybrid algorithms for the constraint satisfaction search. *Computational Intelligence*, 9(3):268–299, 1993.
[18] F. Somenzi. CUDD: Colorado University Decision Diagram package. [Online, accessed 31 May 2004], Feb. 2004. `http://vlsi.colorado.edu/~fabio/CUDD/`.
[19] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
[20] C. Thiffault, F. Bacchu, and T. Walsh. Solving non-clausal formulas with DPLL search. In *Proceedings of CP2004*, volume 3258 of *LNCS*, pages 663–678. Springer, 2004.
[21] T. Walsh. SAT vs CSP. In *Proceedings of CP2000*, volume 1894 of *LNCS*, pages 441–456. Springer, 2000.
[22] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of International Conference on Computer Design (ICCAD)*, pages 279–285, 2001.
[23] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of CAV2002*, volume 2404 of *LNCS*, pages 17–36, July 2002.