Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1655-1692
submitted: 15/10/03, accepted: 14/8/04, appeared: 28/12/04 © J.UCS

Function-Complete Lookahead in Support of Efficient SAT Search Heuristics

John Franco
(University of Cincinnati, Cincinnati, Ohio, U.S.A.
franco@gauss.ececs.uc.edu)
Michal Kouril
(University of Cincinnati, Cincinnati, Ohio, U.S.A.
mkouril@ececs.uc.edu)
John Schlipf
(University of Cincinnati, Cincinnati, Ohio, U.S.A.
schlipf@ececs.uc.edu)
Sean Weaver
(University of Cincinnati, Cincinnati, Ohio, U.S.A.
fett@gauss.ececs.uc.edu)
Michael Dransfield
(National Security Agency, U.S.A.
mdransfi@restarea.ncsc.mil)
W. Mark Vanfleet
(National Security Agency, U.S.A.
wvanflee@restarea.ncsc.mil)

Abstract: Recent work has shown the value of using propositional SAT solvers, as opposed to pure BDD solvers, for solving many real-world Boolean Satisfiability problems including Bounded Model Checking problems (BMC). We propose a SAT solver paradigm which combines the use of BDDs and search methods to support efficient implementation of complex search heuristics and effective use of early (preprocessor) learning. We implement many of these ideas in software called SBSAT. We show that SBSAT solves many of the benchmarks tested competitively or substantially faster than state-of-the-art SAT solvers.

SBSAT differs from standard propositional SAT solvers by working directly with non-CNF propositional input; its input format is BDDs. This allows some BDD-style processing to be used as a preprocessing tool. After preprocessing, the BDDs are transformed into state machines (different state machines than the ones used in the original model checking problem) and a good deal of lookahead information is precomputed and memoized. This provides for fast implementation of a new form of lookahead, called local-function-complete lookahead (contrasting with the depth-first lookahead of zChaff [Moskewicz et al. 01] and the breadth-first lookahead of Prover [Stålmarck 94]). SBSAT provides a choice of search heuristics, allowing users to exploit domain-specific experience. We describe SBSAT in this paper.

We use SBSAT in conjunction with the tool `bmc` from Carnegie Mellon to translate a bounded model checking problem to classical propositional logic and then use SBSAT to solve the `bmc` output. We show this approach is faster than the now traditional approach of translating the `bmc` output to CNF clauses and using a CNF-based SAT solver, such as zChaff. The work continues that of [Franco et al. 01] and [Franco et al. 04].

Key Words: Binary Decision Diagrams, Bounded Model Checking, DAG, Satisfiability, State Machines, Satisfibility Decision Heuristics
Category: SD B.6.1, SD F.4.1

1 Introduction

Model checking [Clarke et al. 99] is becoming one of the most widely used formal tools for verifying hardware circuits and software protocol design. The design to be verified is modeled as a finite state machine and the specification is formalized by temporal logic formulas. The reachable states of the design are traversed to verify those formulas or to find a counter example. The check is performed as an exhaustive state space search, which is guaranteed to terminate since the model is finite. But both computer time and space requirements can be exponential in the size of the problem, making the computations all too often infeasible.

A partial solution, called Bounded Model Checking (BMC) [Biere et al. 99, Clarke et al. 01], is to bound the state space and admit queries with some limit on the number of time steps. Now the verification problem can be expressed with formulas of (classical, non-temporal) propositional logic. These can now be translated to BDDs and solved using traditional BDD methods [Bryant 86]. BDD methods depend on using operators which are highly efficient as a function of the sizes of the BDDs, but in all too many cases the BDDs grow exponentially large, making BDD methods often infeasible.

So in Bounded Model Checking, the propositional formulas are typically translated to Conjunctive Normal Form (CNF) and solved using off-the-shelf SAT solvers. For example, GRASP [Marques-Silva and Sakallah 96], SATO [Zhang 97], BerkMin [Goldberg and Novikov 02], Siege [Ryan 03], and variants of zChaff [Moskewicz et al. 01] have been applied with great success.

However, in this translation to CNF, some of the domain-specific information is hidden or garbled. A simple example is the following: if one simply translates a formula to CNF, the size of the formula may grow exponentially. By adding new variables, called *system variables*, one may achieve only linear growth in formula size [Tseitin 68, Schöning 89]. Often, during the search for a solution, it is not efficient to branch on system variables, more commonly known as dependent variables. But the dependent variables can be more difficult for a standard CNF-based solver to find; to do so amounts to recovering domain-specific information from the CNF translation. We do not say this cannot be done but it does require work and may be only partially successful. By staying in the original domain one can branch on independent or dependent variables first, as needed by the input problem. We have found orders of magnitude difference in search time by choosing independent variables first or dependents first. Therefore, separating these two groups appears to be very important. Our thesis is that speed can generally be gained by not obscuring such information in the first place.

We present a search-based solver, based on a new state-based architecture, for solving many propositional problems and study its efficiency on several problems including some well-known BMC benchmarks. To avoid confusion, we emphasize that the state machines of the proposed architecture, technically Mealy machines, are separate from state machines that are normally given when specifying an original BMC problem. The solver, which we call SBSAT, adapts tools of state-of-the-art SAT solvers without requiring translation to CNF. It has an adaptable form of memoization of domain-specific and lookahead information that may be used to implement search heuristics that would otherwise be too expensive in computation time to be practical. The architecture trades space for speed: this is now reasonable due to significant and steady advancements in RAM technology, both in size and in speed.

SBSAT generalizes CNF-based SAT solvers. It takes as input a set of constraints: not a set of clauses, but a set of BDDs. (Some other inputs, such as CNF, are accepted, but they are converted initially to BDDs.) Like CNF-based SAT solvers, SBSAT searches for a variable assignment that simultaneously satisfies all its constraints. SBSAT examines the constraints for domain-specific and lookahead information, some of it highly complex, which it memoizes during preprocessing, in a data structure that affords table-lookup during search.

As is standard with complete SAT solvers, SBSAT implements a Davis-Putnam-Loveland-Logemann (DPLL) [Davis et al. 62] backtracking procedure to search for a satisfying truth assignment, starting with no assignments to any variable and assigning values to variables one-by-one until a solution is found or it is determined that no satisfying truth assignment exists. The standard DPLL algorithm is shown in Figure 1. In this algorithm DPLL is called with the empty set as argument.

In Figure 1 and elsewhere in the text we use the following terms. A *literal* is a variable or its complement. The complement of variable x is written $\neg x$. A *partial interpretation* I is a subset of literals with the meaning that every variable x such that $x \in I$ has value `True`, every variable x such that $\neg x \in I$ has value `False`, and all other variables are unassigned. A *contradiction* means some variable is either assigned or implied to have both `True` and `False` values.

The meaning of "easy inferences" in Figure 1 varies from solver to solver. Minimally, it means unit clause propagation. Under a partial interpretation I, a unit clause is such that the complements of all but one of its literals is in I and neither the only other literal nor its complement is in I (that is, its associated variable is unassigned). The single non-falsified literal of a unit clause will be refered to as a *unit literal* below. In unit clause propagation, while there exists a unit clause c under I, add the unit literal of c to I, thereby assigning it the value that makes c `True`. As variables are added to I, one or more non-satisfied clauses

```
Given: a set $B$ of boolean formulas.

boolean DPLL(partial interpretation $I$) {
  Add to $I$ all "easy" inferences from $B \cup I$
  If $I$ contains a contradiction, return False
  If $I$ is total (every literal or its negation is in $I$ ),
    output $I$ and return True
  Pick a literal $x$ to branch on (via some heuristic)
  If DPLL($I \cup \{x\}$)
    return True
  Else
    return DPLL($I \cup \{\neg x\}$)
}

if (not DPLL($\emptyset$)) output "unsatisfiable."
```

Figure 1: Sketch of the Davis-Putnam-Loveland-Logemann algorithm.

may become unit clauses. This effect may propagate for several iterations. Some solvers do much more ambitious (and more expensive!) "unit lookaheads." For example, for each unassigned variable, x check whether unit propagation gives a contradiction from $I \cup x$ (resp., $I \cup \neg x$); if so, add $\neg x$ (resp., x) to I. SBSAT uses a different form of lookahead, called local-function-complete-lookahead, which is described in Section 3.
The following steps are an overview of the operation of SBSAT:

1. Simplify the input BDDs, using both standard and tailored BDD operations. These are described in Section 2.

2. Compute and memoize information about the resultant BDDs in state machines, called Smurfs. Smurfs are described in Section 3.

3. Do a DPLL search for a satisfying truth assignment. The search uses the Smurfs to keep track of its current state information, to speed up inference during branching, and (normally) to guide the search heuristic. Information about failed computation paths is memoized in *constraint clauses*, also called *lemmas*, in the style of many modern SAT-solvers. The search heuristic is described in Section 4 and lemmas are described in Section 5.

A solver's *search heuristic* picks the variable to branch on next depending on the current partial interpretation I. One possibility is to pick an *open* variable, that is a variable not assigned a value in I, such that the variable appears more often than any other open variable in non-satisfied constraints (constraints are clauses in CNF formulas). Many modern CNF solvers use a variant (e.g., the Chaff family [Zhang 01]): these solvers "learn" extra clauses, called *conflict clauses* (lemmas), as the search progresses; an oversimplification is that Chaff picks the literal occurring in most clauses, both among non-satisfied clauses and the conflict clauses. Some solvers using unit lookaheads branch on the open literal which, when looked ahead on, gives the most inferences (for example, van Maaren's MARCHL solvers [Heule 04]). BerkMin [Goldberg and Novikov 02] picks an open variable out of the most recently inferred conflict clause, presumably increasing relevance, and among such variables, picks one similarly to Chaff.

One design goal of SBSAT is to use the best features of modern backtrack CNF-based SAT solvers: conflict-clauses, or lemmas, are used to turn a tree search into a DAG search (see Section 5); non-chronological backtracking, including backjumping, is employed (see Section 5); and advanced data structures (*c.f.*, the watched literals of [Lynce and Marques-Silva 02, Moskewicz et al. 01]) are implemented (specifically type WL in [Lynce and Marques-Silva 02]) to allow fast access to the lemmas. Heuristics have been developed to choose, keep, and discard the lemmas appropriately. Such features have been used in CNF-based SAT solvers in recent years, with the result that many CNF problems considered very difficult just two or three years ago are now considered trivial. A new feature of SBSAT (discussed below) is to precompute and memoize *local-function-complete lookahead* information to support efficient implementation of complex search heuristics.
The important features of SBSAT are the following:

1. It does as much BDD-type preprocessing as is feasible. The goals of preprocessing are to 1) simplify the collection of BDDs; 2) uncover inferences; and 3) collect and memoize information useful for a search heuristic. Each search heuristic will have its own memoization requirements. Preprocessing results in smaller search spaces which are traversed faster. We created a new BDD operation, called *strengthening*, for revealing inferences and simplifying a collection of BDDs while avoiding size explosion (see Section 2).

   SBSAT does not attempt to reorder the variables. Rather, using the input variable ordering, it does as much preprocessing as it deems useful, and then depends upon DPLL search to finish the satisfiability check.

2. Preprocessing results are recorded in special automata called SMURFs which consume a lot of space but which allow replacement of computations during search with table lookups (see Section 3).

3. We propose a new search heuristic which is made feasible due to information memoized during preprocessing. Since BDDs encode more complex interrelationships than do CNF formulas, we can precompute complete lookahead information on individual BDDs, considering all partial truth assignments for a single input BDD. We can then combine precomputed lookahead information from all BDDs during the search. We call this *local-function-complete* lookahead (see Section 3). This is critical because, since we do not break inputs into CNF, there is more information in a *single* BDD than in a *single* clause.

Several previous approaches (for example, [Puri and Gu 96], [Kalla et al. 00], [Gupta et al. 03], [Giunchiglia and Sebastiani 99], [Paruthi and Kuehlmann 00], [Damiano and Kukula 03], [Cimatti et al. 01], [Gupta and Ashar 98]) have used hybrid algorithms, combining BDD tools and DPLL style search, but their methods and/or goals are quite different from ours. We distinguish between a few here.

The work of [Damiano and Kukula 03] proposes a solver resembling GRASP but with BDDs instead of clauses as input. During branching they choose a direction on a variable, look at all the BDDs involving that variable and if any BDD leads to 0, then they build a normal conflict clause and backtrack. They mark edges in a BDD to help speed up the branching process: this is done more efficiently with Smurfs (see Section 3).

Learning is moved to BDDs in the proposal of [Gupta et al. 03] which targets Bounded Model Checking applications. Learned and original CNF clauses are clustered into (small) BDDs around certain “seed” nodes, identified during or before search, of a circuit graph. That is, learned clauses are not necessarily entered into a “lemma” database at the point they are or would be “discovered” during search. Clauses falsifying learned BDDs are passed to a normal SAT solver as lemmas. Success of this approach depends on the heuristics used to choose the “seed” nodes. The similarities to our work are that significant learning is done in pre-processing as well as during search. However, our approach additionally attempts to minimize the time needed per step for a search heuristic to compute weights with Smurf structures that are insensitive to BDD variable orderings. Incidentally we considered combining all lemmas into large BDDs but found that impractical as did the authors of [Gupta et al. 03].

The work of [Novikov 03] proposes improved preprocessing techniques for CNF SAT solvers. The basic idea is to analyze lookahead information on small groups of variables (about 5). This somewhat resembles our notion of function-complete lookahead (see Section 3) except that we rely on domain-specific, semantic connections between variables to decide where the lookaheads are to occur, we look for inference information as well as additional information that will help support an efficient implementation of complex search heuristics, and we retain the results of preprocessing in Smurfs for faster use by those heuristics.

In [Gopalakrishnan et al. 03] it is proposed to integrate SAT and BDD techniques to help mitigate the problem of BDD explosion. Search is begun using CNF techniques, then switched to BDD techniques when it appears that the residual expression is small enough to be solved by BDDs. If not, CNF search is continued and so on. This is quite different from what is proposed in this manuscript.

It has been observed in [Kuehlmann et al. 01] that many CAD tasks operate on circuits containing a significant number of redundancies. The approach taken in that paper is to interleave structural transformations, CNF search, and BDD sweeping on a common graph representing a given circuit. In our approach, all BDD operations end with preprocessing to keep backtracking costs down. In other words, BDD operations are performed on the input set of BDDs, then search commences and BDD operations are no longer performed.

In [Hong et al. 97] BBD minimization operations that are guaranteed not to increase BBD size are described. Such operations are similar to those we present in this paper. Our motivation for developing those operations is the same: to prevent BDD explosion while revealing inferences during preprocessing. We point out that the reduction illustrated in Fig. 7 of [Hong et al. 97] is not pertinent to our solver since we look for equivalences and single variable inferences and would have found all those inferences just by looking at the first BDD (a), never needing to restrict or prune it with anything.

The work of [Reda et al. 02] is specific to the problem of checking the equivalence of functions. For such problems, the authors prove a relationship between the search space of a simple variant of DPLL and the size of a BDD representing the function the variant is applied to. This relationship is then exploited by a proposed search heuristic for the variant. Although this is a step in the right direction, the result depends strongly on the nature of the equivalence problem and the DPLL variant studied is lacking the features that have caused search techniques to become competitive in recent years, so the applicability of the ideas expressed in this paper is uncertain.

In [Cabodi et al. 03], as in our work, BDDs are used in a preprocessing "learning" phase to help prune a following backtrack search. The method of approximate reachability is used to construct a smaller, and simpler, although inaccurate BDD representation. The resulting information is dumped to CNF form in one of several ways, differing mainly in how the BDD is to be cut through the introduction of extra variables. The dumped CNF expression is larger than the original by varying amounts (4–220% increase in clauses from reported results).

The HypBinRes algorithm[Bacchus and Winter 03], and the algorithm `NiVER` and `LiVER`[Subbarayan and Pradhan 04], are preprocessors for CNF search engines, sharing an emphasis upon preprocessing with SBSAT. They simplify CNF input by doing different forms of limited resolution. HypBinRes also aggressively determines variable equivalences, partly by doing unit lookaheads on all open variables during preprocessing. By contrast SBSAT also searches for equivalent variables in preprocessing, but only with its function-complete lookaheads. SBSAT's technique of looking for equivalences function by function generally gets fewer inferences than unit lookahead does, although it may pick up some extra ones because of the full classical analysis on each individual function. SBSAT's function-by-function lookahead is also cheaper.

The `CirCUs` solver[Jin and Somenzi 04] allows both CNF and BDD input, and during preprocessing it combines clauses input into BDDs. It then does BDD-type preprocessing to simplify the BDDs ([Jin and Somenzi 04] does not list enough details for us to compare it carefully to SBSAT's preprocessing). `CirCUs` converts those BDDs to CNF for its search.

Recently, the March family of CNF solvers has shown a good deal of success on BMC problems[Heule and van Maaren 04]. They differ from Chaff, Berkmin, and other solvers most notably in that (i) they do not store lemmas, (ii) they aggressively try to show literals to be equivalent to each other and use the equivalences to guide their searches, and (iii) at decision points they do *global* lookaheads on each open variable, keeping track of inferences and which variables lead to contradictions. In comparison, SBSAT does function-by-function lookaheads, which are generally noticeably less powerful but far less expensive. (SBSAT's strengthening operation (see Section 2) captures some of these inferences without the cost of global lookaheads.) Rather like SBSAT, March uses the lookaheads to guide branching: roughly, branch on literals forcing many inferences. We expect to compare March to SBSAT in future work.

In what follows, Sections 2 to 5 explain the operation of SBSAT including two preprocessing phases, Smurfs, the search heuristic, and lemmas. Section 6 explains the use of the `bmc` tool in producing benchmarks suitable for testing with SBSAT: the two forms of output presented by `bmc` help to test whether SBSAT takes advantage of domain-specific information. Section 7 presents experimental results. Section 8 presents conclusions and Section 9 presents thoughts on where the research is going.

2 Preprocessing, Phase 1: Simplifying BDDs

SBSAT takes as input a Boolean formula that is a conjunction of functions, which are represented in the preprocessing phase as BDDs. In case the input is a CNF formula, a clustering algorithm is applied to construct a reduced set of BDDs, each representing more than one and usually several clauses. At the moment, clustering is restricted to matching commonly occurring clause patterns such as those representing an equation of the form

$$x = and(x_1, x_2, ..., x_k).$$

To economize on space, such BDDs represent functions of no more than 17 variables (this was a decision based on current practical considerations and may be increased if more RAM is available, using more variables should give better performance). BDD operations are applied to the collection to simplify and reveal inferences.

Because our state machine (described in Section 3) for an n-variable boolean function may contain (at worst) close to 3^n states (there is potentially a state for every partial interpretation up to n variables and there are 3^n partial interpretations on n variables), we currently apply only BDD simplifications that do not increase the number of variables per BDD. The BDD-type operations that SBSAT applies include:

Primitive Inference: An individual BDD may force a literal to be `True` or two literals to be equivalent this is unlike the case of an individual (non-unit) clause. The set of BDDs is simplified for all such inferences found (e.g., if $x_{17} \oplus x_{297}$ is inferred, x_{297} is replaced with $\neg x_{17}$ throughout). Whenever other simplifications are made in preprocessing, primitive inferences are again found and applied. This step is fairly trivial but yet sometimes results in significant reductions of the number and size of input BDDs. Observe that the inference $x_{17} \oplus x_{297}$ would be hidden if the input were converted to CNF.

Existential Quantification: This is also a standard tool in BDD software packages. Suppose variable x_i appears in only one BDD, say b_1. Then it may be "eliminated":

$$\exists x_i(b_1 \wedge \cdots \wedge b_m) \text{ is logically equivalent to } \exists x_i(b_1) \wedge b_2 \wedge \cdots \wedge b_m.$$

SBSAT searches for a satisfying assignment for the second formula; since it has fewer variables than the original set of BDDs, search is *usually* faster. If the simplification is unsatisfiable, so is the original problem. If the simplification is satisfiable, the assignment t returned can be expanded to satisfy the original BDD set by choosing any value of x_i satisfying b_1.

Strengthening: Function f strengthened by g is defined as

$$f \wedge \exists v_1, v_2, ..., v_k(g)$$

where $v_1, v_2, ..., v_k$ are the variables appearing in g but not f. So, every variable occuring in f strengthened by g also occurs in f. When strengthening is applied to BDDs b_i, b_j, form b'_j by existentially quantifying out of b_j all variables not occurring in b_i, and then replacing b_i with $b_i \wedge b'_j$. Similarly replace b_j but using the new b_i. The principal value of strengthening in SBSAT is that it may reveal additional inferences via primitive inference. An example is shown in Fig. 2.

Strengthening is quite similar to the NPAnd (non-polluting and) of the CUDD package [CUDD 04]. Like `strengthen`, `NPAnd` quantifies out variables not occurring in the second BDD, but it similarly quantifies out variables as it recurses down the BDD. Since the goal of strengthening is only to limit the number of variables, strengthening does not quantify out additional variables on subBDDs.

Restrict ([Coudert and Madre 90, Coudert and Madre 91])**:** This is also a standard BDD operation. *restrict*(f, c) removes from BDD f all branches contradicting BDD c. Restrict is similar to an operation called *generalized cofactor* (gcf) or *constrain* [Brace et al. 90, Coudert and Madre 91]; we do not use gcf since it may increase the number of variables in a BDD. Both *restrict* and gcf are highly dependent upon variable ordering. As noted, SBSAT makes no attempt to reorder variables.

For SBSAT there are four benefits to restricting. (1) The BDDs produced tend to be smaller. (2) Smaller BDDs result in smaller memoized structures (described in the next section). This is a great benefit since those structures tend to grow exponentially with the number of variables. (3) The local-function-complete lookahead heuristic (described in Section 4) seems to make the right variable and value choices often. (4) Inferences are sometimes revealed before search commences.

The preprocessing phase ends with a collection of BDDs but usually significantly reduced in number and in size, and in which some variables have been eliminated. Since most BDD-type preprocessing is sometimes useful and sometimes not, SBSAT gives the experienced user the option of adjusting the amount of preprocessing; for example, if complete domain-specific information is available, strengthening may not be worth the time. The following is an overview of preprocessing, Phase 1.
Given a collection of BDD's, SBSAT's default is to:

1. Simplify using primitive inference until a fixed point is reached.
2. Simplify using both existential quantification and primitive inference until a fixed point is reached.
3. Simplify using both strengthening and primitive inference until a fixed point is reached.

4 Simplify using both restrict and primitive inference until a fixed point is reached.

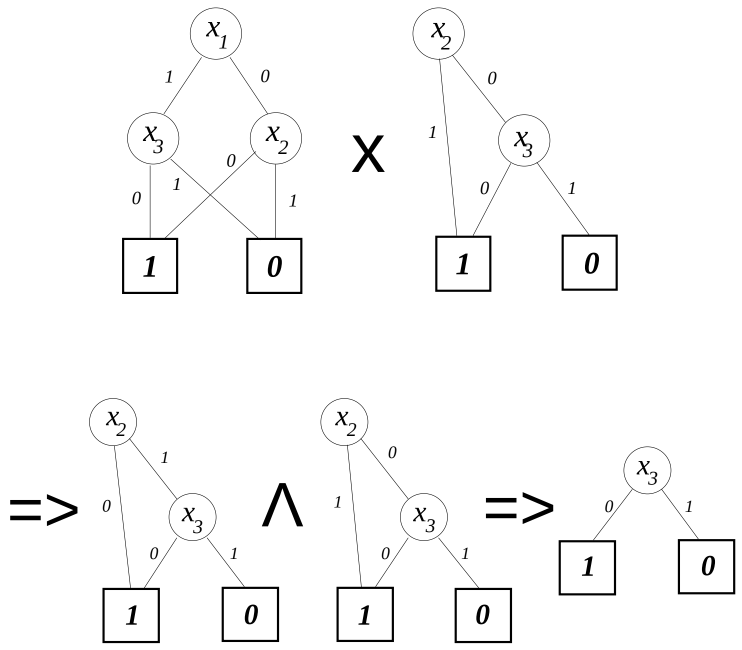

Figure 2: **Strengthening example**: The operation of strengthening is applied to the BDDs at the top of the figure and reveals an inference. The BDD at the upper left corresponds to b_j in the text and the BDD at the upper right corresponds to b_i in the text. To strengthen, first existentially quantify away x_1 from b_j. This results in b'_j which is shown at the bottom left of the figure. The strengthen operation is completed by conjoining b'_j and b_i which are the two leftmost BDDs on the bottom. The result is the BDD at bottom right. This BDD reveals the inference $x_3 = 0$.

Options are provided in SBSAT to allow an experienced user to demand a different order. Observe that since the above process continues until a fixed point is reached, preprocessing times can vary greatly depending on the input.

3 Preprocessing, Phase 2: Memoization

After the BDDs are simplified as above, full classical-logic information about possible partial interpretations is computed and memoized for each individual BDD. As we shall note below, this structure (i) allows unit-time inference of literals forced by any single BDD; (ii) holds information that is useful for search heuristics; and (iii) holds lemma information. We describe here what is currently implemented in SBSAT, but we note that it would be relatively easy to modify

SBSAT to store additional information needed for a different search heuristic, so long as that information pertains to only one BDD at a time.

The structure is a collection of state machines, one for each of the BDDs remaining after the simplification described above. More precisely, each machine is an acyclic Mealy machine: a transition outputs a (frequently empty) set of literals. Each state machine is called a SMURF (for State Machine Used to Represent Functions). The SMURF represents *all* the BDDs for a function f, i.e., for all orderings of the variables.

Here we define the SMURF for a BDD b as the end result of a series of constructions. Our actual algorithm for constructing SMURFs is optimized in two ways: 1) operations are not performed in exactly the sequence described below, in particular the algorithm groups some of the passes together; 2) as in constructing BDDs, we memoize extensively, in particular the SMURF state for a specific residual function is created only once, even though that state may arise in several input functions. We illustrate the SMURF for BDD $ite(x_1, x_2 \wedge (x_3 \oplus x_4), x_4 \wedge (x_2 \oplus x_3))$ in Fig. 3.

1. **Initialization:** We use the term *partial truth assignment* to mean a function mapping *some* (maybe none, maybe all) variables to $\{$`True`, `False`$\}$. Create one state s for each partial truth assignment I to the variables of b. For each I, form the *residual function* r of b for I by "plugging in" the values of I into b and simplifying b. We refer to the state here as $s = (I, r)$.

   The *start state* of the SMURF is $s_0 = (\emptyset, b)$. Any state $s = (I, \texttt{True})$ is *final.*

   For each state $s = (I, r)$, for each variable x_i appearing in r, there are (i) a transition labeled x_i out of s to the state with partial truth assignment $I \cup \{x_i\}$, and (ii) a transition out of s labeled $\neg x_i$ to the state with partial truth assignment $I \cup \{\neg x_i\}$. Additionally, all transitions are labeled with *inference lists* that are initially empty. When construction is complete, literals in a transition's inference list are those that must be forced to value `True` when extending a partial assignment corresponding to the transition's outgoing state by the literal labeling the transition during search. The machine is acyclic since partial assignments associated with states get larger as paths through the machine get longer.

2. **State Elimination:** Recall that, in the earlier BDD simplification phase, SBSAT identifies any literals implied by any single BDDs (primitive inference), infers them, and simplifies the remaining BDDs. So b, the residual function of s_0, does not imply any literals.

   For *any* state $s = (I, r)$ where r implies literals $\lambda_1, \ldots, \lambda_h$, (i) change the target of the transition to the state s' with partial truth assignment $I \cup \{\lambda_1, \ldots, \lambda_h\}$, and (ii) add $\lambda_1, \ldots, \lambda_h$ to the transition's inference list. Continue this way until no states' residual functions imply any literals.

Observe that any state $s' = (I', \texttt{False})$, representing a partial truth assignment which falsifies b, is inaccessible from s_0: if there was originally a transition into it from some state $s_\lambda = (I', b')$ on literal λ, then b' implies $\neg\lambda$, so each transition into s^λ is moved to some other state with partial assignment (some superset of) $I' \cup \{\neg\lambda\}$.

Finally, remove all states inaccessible from s_0.

3. **State Merging:** Across all SMURFs, states with the same residual function are merged (as with reducing BDDs). Each state is now identified just by a residual function r and no longer by a partial truth assignment.

Observe that any state represents a "vantage point" from which the entire future of any search is visible, with respect to b alone, with a partial assignment implied by the path to a state. Therefore, at any point during search, complete information about possible outcomes with respect to b is available to a search heuristic. This is what we mean by "local-function-complete-lookahead." The lookahead information is memoized in the states and transitions of the SMURFs. Exactly what information is memoized depends on the search heuristic. In the next section an example is given.

Complete future information for single functions is more than what is provided by CNF solvers but not enough to guarantee the smallest search space. Memoized future information from all SMURFs must be combined by a search heuristic to make the best possible guess at the next unassigned variable that should be assigned a value.

For a BDD b with k variables, the SMURF can have, in the worst case, nearly 3^k states (corresponding to the 3^k partial truth assignments on k variables). However several optimizations are possible. The most useful and important are due to some frequently occurring *special functions* in circuit design: those involving conjunctions or disjunctions of many variables or functions. For example:

$$x = and(x_1, x_2, \ldots, x_k)$$

which evaluates to 1 if and only if x is assigned a value that is identical to the value of $and(x_1, x_2, \ldots, x_k)$. In such cases counters are used to simulate states (the count being the number of unassigned variables in an equation that is not yet satisfied) but some flexibility in memoizing search future information is lost. To support the heuristic described in the next section, it is sufficient, however, to memoize a function of the counter value. This will be described in the next section.

4 Search: The LSGB Heuristic

The search phase begins after preprocessing is completed. Search consists of extending partial assignments until either all SMURFs reach their final state, in

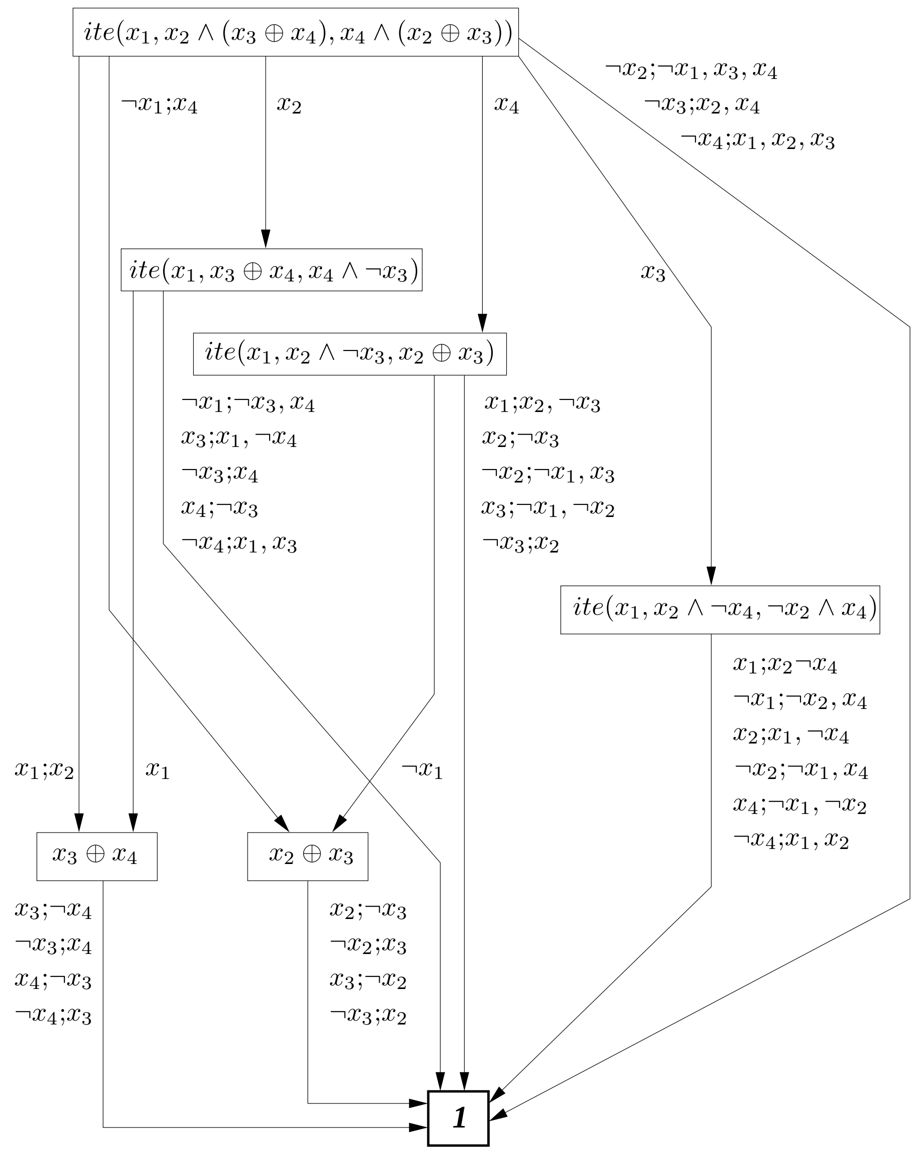

Figure 3: The SMURF representing $ite(x_1, x_2 \wedge (x_3 \oplus x_4), x_4 \wedge (x_2 \oplus x_3))$. Rectangles are states, labeled with their residual functions. The start state is at the top; the final state, at the bottom, is labeled ***1***. Transition labels consist of a literal followed by a semi-colon followed by a comma-separated list of literals with the meaning that setting the leftmost literal to `True` forces each of the rightmost literals to have value `True`. For example, Label $\neg x_2; \neg x_1, x_3, x_4$ on a transition out of the start state means that setting x_2 to `False` forces x_1 to `False` and x_3 and x_4 to `True`. Multiple transitions from a state are shown as one line with multiple labels. Not shown are lemmas.

which case the original input expression is satisfied (see Page 12 for the meaning of final state), or until some contradiction arises among the inferences generated by Smurf transitions. In the former case, a solution is returned. In the latter case, a backtrack occurs: that is, some of the variables in the partial assignment become unassigned, at least one is reassigned, and some previously unassigned variables are assigned values and the search proceeds by trying to extend the new (current) partial assignment. To minimize the number of backtracks, it is important to use a well-tailored search heuristic for choosing variables and values when extending partial assignments and to use information learned during the course of the search process. In this section an implemented search heuristic, called LSGB for *locally skewed, globally balanced*, is described.

The LSGB search heuristic is developed by combining the effects of two valuable notions. First, search effort can likely be reduced significantly if variables are chosen and given values so as to force the value of many literals (that is, make inferences). The Smurf structure allows variables to be chosen and assigned values by efficiently taking into account inferences that will be made immediately from the assignment, as well as after a second variable is assigned a value, as well as after a third variable and so on.

A summary of all this looking ahead can be precomputed and memoized in the Smurf structure as a number for instant access by the search heuristic. It is our intuition that the impact of immediate inferences on search size is usually greater than the impact of inferences that can be made after another variable is assigned a value, and the impact of those is greater than the impact of inferences that can be made after two variables are assigned values and so on. Therefore, to account for the effect of inferences activated at varying depths from the current state, we assign a weight to each Smurf transition. The weight counts the number of literals forced by taking that transition, plus the expected weight of literals forced below that state, where the weight of a forced literal after t additional variable selections is $1/K^t$. We note that K is a parameter that has been discovered to significantly affect the performance of the heuristic but which depends on the type of input. We have found a value of 3 to generally give the best results. The ratio $1/K^t$ reflects the notion that inferences become geometrically less valuable with increasing free variable assignments needed to generate them. This is seen clearly for the case that Smurfs correspond to disjunctions, or CNF clauses, where the value of an inference decreases by a factor of 2 for each free variable assignment needed to generate it under the assumption that clauses are randomly generated (this is the Johnson or Jeroslow-Wang heuristic [Johnson 74, Jeroslow and Wang 90]). For random CNF expressions the Johnson heuristic proves powerful: instead of choosing variables to greedily force the maximum number of inferences, the geometric weighting causes variables to be chosen to maximize the expected number of satisfying assignments to unassigned variables, thereby improving the chance of finding a satisfying assignment, if one exists. We merely generalized this idea for more complex functions.

Formally, the weight $w(s)$ of a state s is computed as follows. Let $\mathcal{N}_i$ denote the number of inferences on the transition from state s to state s_i. If state s has p successors $\{s_1, \cdots, s_p\}$ then

$$w(s) = \frac{\sum_{i=1}^{p} (w(s_i) + \mathcal{N}_i)}{K \cdot p}$$

and the weight on a transition out of state s to state s_i is the number of inferences on that transition plus $w(s_i)$. The weight of the final state is 0.

Thus, in Fig. 3, the transition out of the start state on $\neg x_1$ has weight

$$1 \; + \; \frac{1}{K}.$$

The 1 on the left is due to the single inference on the transition, the $1/K$ term equals $4/4K$. The 4 in the numerator of this term is the number of inferences on transitions out of the state $x_2 \oplus x_3$, and 4 in the denominator is the number of branches out of the transition. Since all transitions go to the final state, which has weight 0, there is no contribution from state weights in the numerator. The transition out of the start state on x_4 has weight

$$0 \; + \; \frac{8 + \frac{1}{K}}{K \cdot 6}.$$

This weight is obtained as follows. Since the final state has weight 0 and since the number of successors to the state labeled $x_2 \oplus x_3$ is 4, the weight of the state labeled $x_2 \oplus x_3$ is the number of inferences, namely 4, to the final state divided by $4K$ which is $1/K$. The number of transitions out of the state labeled $ite(x_1, x_2 \vee \neg x_3, x_2 \oplus x_3)$ is six, but five of the transitions go to the final state; hence only inferences along those transitions contribute to the weight of the state. The total number of inferences on those transitions is 8. The weight of the state $ite(x_1, x_2 \vee \neg x_3, x_2 \oplus x_3)$ is this weight plus the weight of state $x_2 \oplus x_3$, the quantity divided by $6K$ since the number of successors is 6. Thus, the weight of the *ite* state is $(8 + 1/K)/(6K)$. The weight of the transition to that state from the root is that weight plus the number of inferences on the transition, i.e., 0.

The second valuable notion that is part of LSGB is that balancing the size of two subordinate search spaces generated from the two values the selected variable may take tends to reduce search size, at least on random problems (see [Freeman 95] for a discussion). To account for the desired balancing of subordinate search spaces, LSGB computes the sum S_i^+ of the *weight*s of transitions on x_i out of all current SMURF states and the sum S_i^- of the *weight*s of transitions on $\neg x_i$. LSGB selects x_i such that $S_i^+ \cdot S_i^-$ is maximum (this idea is borrowed from [Freeman 95]). The first value to assign x_i is determined by the larger of S_i^+ and S_i^-.

This is a highly complex heuristic that requires a lot of information to compute. We show how the SMURF structures can store information which makes the use of LSGB little more expensive than some table lookups. All the weights are memoized during preprocessing. During search, the inference weight of choosing a variable and its value is obtained by looking up those numbers on the SMURF transitions. This is easily done since there is always a pointer set to the current state of each SMURF and the numbers of interest are on transitions out of the current states. Exactly how these numbers are used is described below.

For the special counter-based SMURFs mentioned in the previous section, the above calculation is simulated. Consider a SMURF state s representing a clause $x_1 \vee \cdots \vee x_k$ with $2 \leq k' \leq k$ still unassigned variables. We show by induction that, for x_i an unassigned variable, (i) the weight of transition x_i is 0, (ii) the weight of transition $\neg x_i$ is $1/(2K)^{k'-2}$, and (iii) $w(s) = 1/(2K)^{k'-1}$:
Part (i) is trivial: setting x_i to `True` satisfies the clause, so there are no inferences and the transition goes to the final state, which which has weight 0.
[Base case: $k' = 2$:] (ii) Let x_g, x_h be the unassigned variables. Transition $\neg x_g$ forces x_h and goes to the final state (which has weight 0), for a total weight of $1 = 1/(2K)^{2-2}$. (iii) $w(s) = (0 + 1 + 0 + 1)/(4K) = 1/(2K)$.
[Inductive case: $k' > 2$:] (ii) Transition $\neg x_i$ forces no literals and goes to a state with $k' - 1$ unassigned variables, which has weight $1/(2K)^{k'-2}$ by inductive hypothesis. So the weight of the transition is also $1/(2K)^{k'-2}$. (iii) Out of s there are $2k'$ transitions. Half have weight 0 and half $1/(2K)^{k'-2}$, so

$$w(s) = (k' \cdot 0 + k'/(2K)^{k'-2})/(K \cdot 2k') = 1/(2K)^{k'-1}.$$

These values are stored in the simulated "counted SMURF." Weights for functions $x_1 \oplus \cdots \oplus x_k$ are be computed similarly. To handle functions $x = x_1 \vee \cdots \vee x_k$, we coded the recurrence relation to solve for the weights; SBSAT computes however many values it will need and stores them in a look up table during preprocessing.

LSGB is similar to the well-known "Johnson heuristic" (*a.k.a.* the Jeroslow-Wang heuristic) [Johnson 74, Jeroslow and Wang 90], which has been applied to CNF formulas, if K is set to 2 and input functions are clauses.

There are circumstances where other search heuristics are known to work well. LSGB was intended for applications where not much is known about, or easily determined about, a given problem. We show it performs well in that case. If a problem is known to have a lot of exploitable structure, it may be better to specify a different heuristic. We allow the experienced user some choice. The SMURF structure supports a multitude of heuristics: on a simple heuristic, it may not be needed, but (except for preprocessing time) it is not a hindrance either. Additional work is needed on hybrid heuristics.

5 Learning: Lemmas

SBSAT makes extensive use of backjumping, recent advanced data structures, and lemmas. SBSAT creates clause lemmas, not BDD lemmas. A clause lemma, or lemma, is an assignment of values to a subset of variables which is known to be sufficient to cause the input formula to evaluate to `False` no matter what values are given to the other variables. These could be managed as a BDD or a collection of BDDs but we chose not to do this for reasons of efficiency. Lemmas are used to prune the search space: when a partial assignment is such that some cached lemma contains a single unassigned literal x and no `True` literals then x is inferred `True` and the opposite branch ($x =$ `False`) is skipped.

SBSAT creates lemmas lazily: during branching rather than precomputation. SBSAT is not able to memoize lemmas in each state because lemmas are consequences of the path taken to a certain state, not a consequence of the state itself. However, lemmas could be memoized in each state if SBSAT did not share states between Smurfs. A lemma is created for an inference given by a Smurf by traversing the path from the start state of the Smurf to the current state and inserting into the lemma the negation of all transition literals and the literal of the inference for which the lemma is being built.

The rules for lemma creation (that is, conflict analysis) are as follows. A lemma is created as the result of a conflict that arises when a variable is assigned a value. A conflict arises when the value of some variable is inferred both `True` and `False`, due to the traversal of SMURF transitions or analysis of lemmas currently stored in the lemma cache. The two lemmas associated with the conflicting inferences are resolved to create a new, temporary, *backtracking lemma*. At this point, a backtrack to a previous node in the search space occurs. What happens at this node depends on its type. Nodes can represent choicepoints (that is, points at which a voluntary assignment of a value to a variable occurs), or inferencepoints (that is, points at which a variable's value is inferred). If the node is an inferencepoint, there are two cases: (i) if the corresponding inference exists as a literal in the backtracking lemma, the lemma associated with that inference is resolved with the backtracking lemma; the result is a new backtracking lemma; (ii) if the variable is not in the backtracking lemma, it is ignored and backtracking continues to another search space node a level higher. If the node is a choicepoint there are two cases: (i) if it occurs in the backtracking lemma, backtracking stops and branching commences by switching the value of the branch variable and turning the resulting assignment into an inference inferred by the current backtracking lemma (this will prevent another value switch at this point in case of a backtrack up to it); (ii) if it does not occur in the backtracking lemma a backjump over the node occurs with the result that a branch on its opposite value never occurs at this node. Backtracking lemmas that are UIP (Unique Implication Point) lemmas are added to the lemma cache. All lemmas representing choicepoints are UIP lemmas. Backtracking lemmas that are not UIP lemmas are deleted before forward search resumes.

Lemmas are treated like CNF clauses. During search, if a partial assignment negates all but one literal of a lemma, that last literal is inferred `True`. Discovery of such a situation is achieved using a modified zChaff-type data structure, based on watched literals of type WL in [Lynce and Marques-Silva 02].

Our approach to backtracking is different than the approach taken by zChaff in the following way. When zChaff is done collecting UIP lemmas (meaning zChaff has backtracked to a choicepoint), it backjumps to one of the watched literals contained in one of the new UIP lemmas which is highest in the search tree. At this point, at least one of the new UIP lemmas will infer at least one literal. SBSAT collects lemmas the same way zChaff does. But, where zChaff backjumps to the highest watched literal in all new UIP lemmas, SBSAT backjumps to the lowest watched literal in all new UIP lemmas, where most often the lowest watched literal is in fact the current choicepoint. SBSAT stores these UIP lemmas in a special structure allowing their inferences to be applied and reapplied at each necessary level of the backtrack tree. A UIP lemma is removed from this structure and added into the lemma cache when both of it's watched literals become unassigned.

The special lemma structure is described as follows. Associated with each choicepoint in a line to the root of the search tree is a linked list, possibly empty, of UIP lemmas. UIP lemmas are added to the lemma list of a choicepoint when searching below it. When backtracking through the choicepoint, all the inferences of all UIP lemmas in the list are applied. Some lemmas will then be removed from the list, because both watched literals are unset. When backtracking to this choicepoint the second time, its UIP lemma list is pushed up to the next highest choicepoint. This action allows the possibility that more than one variable is reassigned on a backtrack: namely, those inferences that are due to lemmas in the list.

All UIP lemmas are cached (see the section below on caching policy for details). SBSAT currently does not restart and its lemma deletion policy is rather simple: essentially delete the lemma least recently used. More work is needed here to determine an optimum lemma caching heuristic.

Every solver using lemmas uses a lemma cache capable of storing a limited number of them. Too many lemmas stored increases overhead prohibitively. Thus, some lemmas must occasionally be thrown out of the cache. The question is how to decide which. For illustration, a lemma heuristic might be based on the following idea: small lemmas and lemmas that made an impact in the past should be kept and long lemmas and lemmas that had never been used should be thrown out. The three level cache attempts to support this idea. In the first level we give a lemma a chance to be useful. All lemmas start at this level. In the second level we have lemmas that either were useful in the past or are small. In the third level there are lemmas that were used but not recently and we opted to keep them around for a little bit longer rather then throw them away, just in case they may prove useful in the not too distant future. Each of these levels has a preset size in terms of the number of lemmas. Lemmas can be turned off from the command line: this is useful for some problem sets, particularly some hand-made problems, which do not seem to benefit from lemmas.

The specific lemma caching policy used currently by SBSAT is the following:

1. There are level 1, 2, and 3 lemma caches. About 1/10 of the entire lemma cache is occupied by the level 1 cache. The level 2 cache is about 1/2 of the lemma cache and the remainder of the lemma cache is used for the level 3 cache.
2. There is an order, from front to back, of the lemmas in each cache.
3. All lemmas are created and initially cached to the front of the level 1 cache.
4. If a lemma is used during backtracking (that is, caused an inference), the lemma is moved to the front of the level 2 cache, regardless of which cache the lemma had been in.
5. If a newly created lemma finds the level 1 cache full, it is still placed in the level 1 cache. However, a lemma already existing in the cache will be moved or eliminated to make room for it. This is decided as follows: if the last lemma of the level 1 cache has less than 7 literals, then the last lemma of the lemma 1 cache is moved to the front of the level 2 cache; otherwise the last lemma of the lemma 1 cache is dropped.
6. If any lemma is moved to the front of the level 2 cache and the level 2 cache is full, the last lemma of the level 2 cache is moved to the front of the level 3 cache but only if it has less than 7 literals, otherwise it is dropped.
7. If the level 3 cache is full the last lemma of the level 3 cache is dropped.
8. The lemma cache is currently set to 10000.

6 Translating `bmc` output to BDDs

Among the experiments we have run, those inputs relating specifically to bounded model checking benchmarks have been obtained from the output of the `bmc` program of [Biere et al. 99]. That program inputs a model checking problem and a number of time steps and outputs a propositional logic formula representing the BMC problem in three formats: a large propositional logic formula, three-address code representing the parse tree for that formula, and a CNF translation of the formula. Program `bmc` internally represents all formulas recursively as

```
<function> = <variable>;
<function> = ¬<variable>;
<function> = <function> op <function>;
```

where **op** is one of $\vee$, $\wedge$, $\rightarrow$, $\equiv$. The binary tree associated with such a recursion is stored as a tree of pointers. Each node of the tree is represented as a triple of pointers: to the left descendent, the right descendent, and the parent. A pointer to the root of such a tree represents the output formula in three-address code. Further processing inside `bmc` converts this to a CNF expression which is also available as output. As an example, we use `bmc` to generate the three-address code problems for queue benchmarks (see next section) as follows:

```
genqueue # > queue#
bmc -k # queue# -prove
```

where `genqueue` is part of the `bmc` suite and `#` is replaced by a number representing problem complexity. The CNF versions are created by replacing the last line above with this:

```
bmc -k # queue# -dimacs
```

We use `bmc` to generate three-address and CNF benchmarks directly, instead of taking already generated CNF formulas "off the shelf" so we have equivalent three-address and CNF data. Thus, times we report for zChaff, BerkMin, and Siege may differ from published times. But these times do not include time used by `bmc` to construct the CNF formulas.

The largest propositional logic formula output by `bmc` is a conjunction of smaller formulas, so the obvious course for SBSAT is to read in each of those smaller formulas as a BDD. Nevertheless, for some of the `bmc` outputs, those propositional logic formulas were much too large even to store as BDDs. Of course, we also did not want to use the three-address code or the CNF representation directly, since that would negate the benefits of SMURFs which are to retain potentially exploitable domain-specific relationships. Our current approach is successful in spite of being amazingly simplistic:

1. We read in the three-address code and recreate the large propositional formula so as not to lose domain-specific information. Starting at the bottom of this formula we start building a BDD. We use a greedy algorithm: when the BDD gets too large (10-18 variables) we insert a new variable to represent the BDD so far, include a BDD asserting that is what the new variable represents, replace the part we have translated with the new variable, and continue the process. This particular translation goes against our intention of staying in the original domain, however, this simple process still proves useful. In future research we hope to find a better algorithm.

2. To break each resultant BDD f down to a 10-variable maximum (so that the SMURFs remain suitably small), we do the following:

(a) Compute all projections f_i of the BDD onto 10-variable subsets of its variable set.

(b) Simplify the f_i's against each other as in Section 2 and delete resultant f_i's which become `True`. Below we call the final simplified f_i's $f_1, \ldots, f_k$.

Note that f logically implies each f_i; we can think of them as "approximations" to f, in the sense that each is `False` on some, but probably not all, of the truth assignments on which f is `False`.

(c) Recall that the goal is to replace f with a set of smaller BDD's (which will be treated, by the solver, as the conjunction of those smaller BDD's). Note (e.g., by truth table) that if f logically implies g, f is logically equivalent to $g \wedge (g \rightarrow f)$. Let

$$f^{\star} = (f_1 \wedge f_2 \wedge \cdots \wedge f_k) \rightarrow f.$$

Then f is logically equivalent to (the conjunction of) $\{f_1, f_2, \ldots, f_k, f^{\star}\}$.

If $f^{\star}$ has ≤ 10 variables, we replace f with. $\{f_1, f_2, \ldots, f_k, f^{\star}\}$. Otherwise, we replace f with $\{f_1, f_2, \ldots, f_k\}$ plus the translation of $f^{\star}$ into CNF. (Typically, $f^{\star}$ is satisfied in most truth assignments, so the CNF translation is fairly short.)

Again, this procedure is simplistic. We hope in the future to find a better algorithm.

7 Experimental Results

We tested our ideas, using our implementation called SBSAT, on several popular benchmark suites. We also ran current versions of SATZoo (v. 1) [SatZoo 03], zChaff (v. 2003.10.9) [zChaff 2003], BerkMin (v. 561) [Berkmin 561], and Siege (v. 4) [Ryan 03] on these benchmarks for comparison. In addition, we concocted a class of random problems, called *sliders*, which resemble BMC problems in that copies of the same function, each differing only in the input variables it depends on, are conjoined. Making those functions random, in some sense, makes sliders hard. Specifically, sliders are defined as follows:

Choose m to be an even number. This will be the number of variables and the number of functions minus 2. Let the variable set be $\{x_1, x_2, ..., x_m\}$. Choose integers $k > 2$ and $l > 2$ to be small relative to m and choose integers $1 < i_1 < i_2 < ... < i_{k-2} < m/2$ and $1 < j_1 < j_2 < ... < j_{l-2} < m/2$. Parameters k and l will be the number of inputs each function takes and $i_1 \ldots$ and $j_1 \ldots$ are the base of indices of input variables that signify which input variables are inputs to the functions. Choose f, a Boolean function of k variables and g, a Boolean function of l variables. Finally, let O be a $m/2$ dimensional 1-0 vector and call its i^{th} component o_i. Corresponding to the choices above, a *slider* is the expression

$$\begin{aligned}&\left(\bigwedge_{0\leq h\leq m/2} f(x_{1+h}, x_{i_1+h}, ..., x_{i_{k-2}+h}, x_{(m/2)+h})\right)\\ \wedge\quad &\left(\bigwedge_{0\leq h\leq qm/2}(g(x_{1+h}, x_{j_1+h}, ..., x_{j_{l-2}+h}, x_{(m/2)+h}) = o_h)\right).\end{aligned}$$

In what follows, each o_h is independently and uniformly chosen from $\{0, 1\}$. We find sliders appealing because they resemble some real-world problem domains and because f and g can be designed to force inferences to occur only when nearly all inputs of f and g are assigned values. This fact makes conflict analysis useless, and is challenging to a search heuristic which is looking for information contained in groups of variables.

At this stage of our SBSAT implementation, lemmas are handled in a rather primitive manner so we observe an unusually low number of backtracks per second. All experiments were run on a single processor Pentium 4, 2 GHz, with 2 GB RAM.

7.1 Time consumption

Our first set of results, shown in Tab. 1, is for the problem of verifying a long multiplier. The circuit definition is available from CMU at [Biere 99]. All benchmarks of this set are unsatisfiable. The left column of the table shows the number of time steps involved in the verification of each benchmark (see Section 6). Experiments were run from 4 time steps to 70 time steps. The next three columns present the observed performance of SBSAT on three-address inputs in total number of choice points, total time, and search time. The next three columns present the same information except when translated CNF formulas are input (see Section 6). The next two columns present the performance of zChaff in choice points and total time. We use the terms *branching time* and *search time* interchangeably and total time is always the preprocessing time plus the search time. The last three columns present the results of Siege, BerkMin, and SATZoo on the CNF versions we generated.

Observe that SBSAT working in the user domain on three-address code shows a slight advantage to working with the CNF translation. For example, the benchmark for which the bottom row of Tab. 1 is dedicated is solved in 455.52 seconds using three-address inputs versus 532.48 seconds using the CNF translation of the benchmark. It is interesting that in the case of CNF inputs, more preprocessing seems to result in less searching. Continuing the example, the number of choice points taken is 24942 for the three-address input versus 10878 for the CNF translation. The fact that preprocessing varies so much from benchmark to benchmark may reflect the imprecision of guesses made when trying to recreate domain-specific information from given CNF formulas. Such preprocessing fluctuations are not as pronounced when three-address codes are input to SBSAT.

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
#time steps	number choices	branch (sec)	total (sec)	number choices	branch (sec)	total (sec)	number choices	total (sec)	total (sec)	total (sec)	total (sec)
4	791	0.17	1.80	614	0.71	1.38	1041	0.45	0.2	0.27	6.34
8	10844	7.50	11.97	11742	32.06	33.84	33272	50.37	12.73	18.9	73.86
12	22151	32.48	40.15	32639	157.10	160.72	122522	357.1	71.61	96.9	288.0
16	18315	46.56	57.77	34098	233.31	239.70	125026	366.7	177.4	200.6	408.0
20	12515	48.55	63.70	35206	316.93	327.35	164373	585.9	165.2	178.8	391.1
24	15355	88.06	107.65	24225	260.63	276.17	214263	790.3	542.8	312.2	640.5
28	16738	108.12	132.27	24142	309.66	331.36	220045	888.2	805.4	255.0	1028
32	15571	114.67	144.18	24902	368.68	397.73	216916	882.8	1035	334.6	1719
36	16250	128.18	161.42	22282	372.07	409.54	269856	1055	576.8	420.4	1622
40	18692	166.96	206.44	12155	238.04	285.64	289687	1103	845.3	442.6	1597
50	13751	138.80	194.84	11337	286.32	361.48	472053	2032	1552	466.9	3609
60	20920	255.97	332.17	11072	319.98	431.70	461867	2183	3340	709.2	3779
70	24942	356.42	455.52	10878	379.46	532.48	850942	5875	2860	844.7	7015

Table 1: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Long Multiplier benchmarks

Observe that zChaff, Siege, and SATZoo cannot compete with SBSAT on long multiplier benchmarks. The problem seems to be due to encountering many more choicepoints during search. BerkMin visits only about an order of magnitude more choicepoints than SBSAT on CNF benchmarks but the slower implementation of lemmas in SBSAT enables BerkMin to be only a fraction slower than SBSAT, in general. The difference in choicepoints suggests the success in this case is due to the complex search heuristic used natively in SBSAT.

Table Tab. 2 shows timings for the well-known set of barrel benchmarks, a part of the `bmc` suite. The three-address code equivalents were generated by applying the `bmc` tool to the output of the `genbarrel` utility in the `bmc` suite. All benchmarks are unsatisfiable. Runs were cut off prematurely if not completed before 3600 seconds. This is reflected as a line (—) through a table entry. In this and following tables no run of SBSAT failed to complete due to preprocessing: that is, for SBSAT, a line — in a table entry means SBSAT ran out of time while searching.

Observe that in all cases, SBSAT solved the problems constructed from the three-address code without any search. This is probably because barrel problems have many variables related by equivalences which are caught by SBSAT in preprocessing. This raises the question of whether a BDD tool might also do as well. This appears not to be the case, since we build a collection of BDDs of about 10 variables each and then strengthen them against each other. The inferences resulting from this process are enough to generate a contradiction before search is applied. We suppose a BDD tool would either have attempted to build a single BDD from the three-address code, in which case it would have been forced to give up due to unmanageable sizes, or it would have used the conjoin operation instead of the strengthening operation to combine the BDDs, probably again taking too much space. We note that the work of [Jin and Somenzi 04] was

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
benchmark	numb choice	branch (sec)	total (sec)	numb choice	branch (sec)	total (sec)	numb choice	total (sec)	total (sec)	total (sec)	total (sec)
barrel2	0	0.00	0.02	3	0.00	0.05	3	0.00	0.01	0.0	0.0
barrel3	0	0.00	0.08	13	0.00	0.08	48	0.00	0.01	0.0	0.02
barrel4	0	0.00	0.13	33	0.01	0.15	201	0.02	0.01	0.01	0.17
barrel5	0	0.00	0.47	354	0.21	0.66	8856	0.58	0.67	0.65	1.07
barrel6	0	0.00	0.82	1205	1.96	2.89	28110	2.81	5.97	5.56	4.19
barrel7	0	0.00	1.33	2848	8.51	11.10	66959	11.37	21.19	29.96	19.8
barrel8	0	0.00	2.02	4304	18.71	25.15	116858	31.98	136.7	298.3	58.4
barrel9	0	0.00	18.90	—	—	—	649532	254.6	41.24	89.27	216.0
barrel10	0	0.00	26.84	—	—	—	1801476	1191	86.34	184.0	304.8
barrel11	0	0.00	36.92	—	—	—	—	—	134.7	238.3	639.0
barrel12	0	0.00	48.54	—	—	—	—	—	927.1	999.3	1298
barrel13	0	0.00	62.79	—	—	—	—	—	629.9	1049	1817
barrel14	0	0.00	82.22	—	—	—	—	—	2122	3389	2970
barrel15	0	0.00	107.94	—	—	—	—	—	—	—	—
barrel16	0	0.00	135.66	—	—	—	—	—	—	—	—

Table 2: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Barrel benchmarks

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
benchmark	number choices	branch (sec)	total (sec)	number choices	branch (sec)	total (sec)	number choices	total (sec)	total (sec)	total (sec)	total (sec)
queue4	41	0.0	0.1	19	0.00	0.11	32	0.00	0.01	0.0	0.02
queue8	651	0.07	3.04	291	0.10	0.49	561	0.05	0.04	0.05	1.86
queue12	4351	1.02	5.53	3875	4.38	5.52	11752	3.09	1.04	0.96	6.36
queue16	30835	14.7	22.3	41029	104	107	73407	62.22	30.27	32.38	16.86
queue20	311127	227	265	565559	2412	2420	698914	1874	400.4	401.0	1926
queue22	1052750	798	843	2016859	9356	9367	—	—	1886	1050	—
queue24	3262464	2613	2666	—	—	—	—	—	—	2724	—

Table 3: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Queue benchmarks

pointed out as a possible way to achieve this for BDDs alone but we have not experimented with this approach at this time.

Although the time taken by SBSAT in preprocessing is considerable, it is shown to be well-spent as SBSAT, zChaff, Siege, BerkMin, and SATZoo all have difficulty with the larger CNF versions of the barrel benchmarks. Thus, it appears staying closer to the user-domain and preprocessing to reveal inferences early has paid off on these benchmarks.

Tables Tab. 3 and Tab. 4 show timings for a set of queue benchmarks and permute benchmarks generated by `genqueue` and `genpermute`, respectively, from the `bmc` suite. Cutoff of runs was set at 3600 seconds for the queue benchmarks and 60000 seconds for the permute benchmarks. All benchmarks are unsatisfiable. The pattern observed is similar to the previous sets of runs. When SBSAT works with three-address code timings are much better than when equivalent CNF inputs are used. Working in three-address code gets results faster than other solvers on equivalent CNF inputs.

The story changes on the queue invariant benchmarks of Tab. 5. In this case,

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
benchmark	number choices	branch (sec)	total (sec)	number choices	branch (sec)	total (sec)	number choices	total (sec)	total (sec)	total (sec)	total (sec)
permute2	0	0.00	0.01	1	0.00	0.05	1	0.00	0.01	0.00	0.00
permute3	5	0.00	0.04	14	0.00	0.07	11	0.00	0.01	0.00	0.00
permute4	68	0.00	0.65	47	0.00	0.11	52	0.00	0.01	0.01	0.02
permute5	174	0.01	10.1	304	0.10	0.27	199	0.02	0.02	0.03	0.54
permute6	893	0.09	11.46	1655	1.15	1.44	2021	0.28	0.17	0.16	1.41
permute7	5537	0.81	23.24	8551	8.77	9.21	16485	9.51	2.88	1.12	2.67
permute8	64607	70.21	71.16	58051	243.2	244.1	110492	172.93	21.6	15.7	27.73
permute9	454726	685.0	686.6	471422	2573	2575	361422	1018	315	228	234
permute10	1311291	2062	2064	—	—	—	2118409	12101	3003	3891	4497
permute11	20462503	39257	39260	—	—	—	—	—	—	—	—

Table 4: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Permute benchmarks

	SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
benchmark	number choices	branch (sec)	total (sec)	number choices	total (sec)	total (sec)	total (sec)	total (sec)
queueinv4	74	0.01	0.06	136	0.00	0.01	0.01	0.09
queueinv8	332	0.07	0.22	1122	0.04	0.06	0.06	1.01
queueinv12	1115	0.56	1.20	4368	0.22	0.31	0.12	2.35
queueinv16	1846	0.77	1.13	7721	0.27	0.53	0.24	3.52
queueinv20	4964	5.86	9.27	16258	1.63	0.73	0.81	9.76
queueinv24	8197	13.10	19.85	26995	2.96	1.89	1.90	13.97
queueinv28	14205	29.80	42.23	38145	5.69	3.88	3.40	23.31
queueinv32	11601	12.94	15.12	68641	3.20	3.74	4.20	13.59
queueinv36	26663	104.06	179.52	103281	23.58	9.59	10.33	56.42
queueinv40	35963	179.44	305.07	145691	38.08	17.62	16.46	74.84
queueinv44	48158	314.02	523.46	166634	46.42	57.38	25.16	105.1
queueinv48	56113	460.72	786.65	217615	79.95	62.00	43.61	152.6
queueinv52	62753	719.28	1183.19	297830	179.2	155.50	55.93	191.8
queueinv56	67894	943.76	1590.32	397142	239.1	514.90	82.13	239.8

Table 5: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Queue Invariant benchmarks

SBSAT experienced memory problems. In order to fit the resulting SMURFs into memory, the BDDs upon which they were based were required to be so small we were forced to choose a rather small maximum size for BDDs. The result was dismal. We did not feel it was worthwhile reporting them. Although SBSAT did solve the CNF versions of these problems, the other solvers performed better than in previous benchmark sets.

For completeness, we include results on the dlx suite available from [Velev 00] in Tab. 6. Some benchmarks are satisfiable and some are unsatisfiable. We applied SBSAT to two variations: namely Trace and CNF formats (both available). All problems in this suite are easy for all the solvers and that is about all that can be said about them. We did not include results of dlx9 benchmarks because SBSAT had some memory problems.

Finally, Tab. 7 shows the result of applying all the solvers to a family of slider problems, some satisfiable and some unsatisiable, based on the following:

	SBSAT on Trace			SBSAT on CNF			zChaff on CNF		Siege	BerkMin	SATZoo
Name	number choices	branch (sec)	total (sec)	number choices	branch (sec)	total (sec)	number choices	total (sec)	total (sec)	total (sec)	total (sec)
dlx1_c	525	0.02	0.12	592	0.03	0.12	1082	0.02	0.01	0.01	0.21
dlx2_aa	1755	0.06	0.22	2062	0.08	0.26	5224	0.10	0.06	0.02	1.83
dlx2_ca	7247	1.00	1.49	6861	0.91	1.60	9800	0.30	0.17	0.12	7.02
dlx2_cc	9655	2.03	2.60	9631	1.97	2.83	17825	0.95	0.36	0.26	8.58
dlx2_cl	9375	1.56	2.14	8872	0.57	2.33	25390	1.50	0.71	0.29	7.29
dlx2_cs	8489	1.31	1.84	7916	1.37	2.15	16310	0.77	0.20	0.23	7.90
dlx2_la	6233	0.64	1.06	6814	0.84	1.41	9246	0.26	0.11	0.10	5.32
dlx2_sa	2938	0.16	0.35	2168	0.15	0.38	5563	0.14	0.08	0.03	0.83
dlx2_cc_bug01	6603	1.20	1.77	6448	1.25	2.11	14471	0.84	0.18	0.28	0.71
dlx2_cc_bug02	6584	1.22	1.80	6432	1.25	2.09	13717	0.79	0.48	0.26	0.70
dlx2_cc_bug03	6861	1.23	1.81	6628	1.23	2.09	22776	1.05	0.01	0.10	1.04
dlx2_cc_bug04	6932	1.33	1.92	6699	1.28	1.12	12860	0.52	0.08	0.08	0.70
dlx2_cc_bug05	3743	0.65	1.24	3413	0.62	1.47	376	0.01	0.22	0.13	0.83
dlx2_cc_bug06	3630	0.60	1.19	3581	0.67	1.52	374	0.01	0.01	0.10	0.82
dlx2_cc_bug07	4601	0.77	1.36	3567	0.65	1.50	316	0.01	0.03	0.05	0.78
dlx2_cc_bug08	5964	1.06	1.65	5353	0.92	1.75	747	0.02	0.01	0.04	0.78
dlx2_cc_bug09	2549	0.42	0.92	2693	0.33	1.18	321	0.01	0.01	0.02	0.66
dlx2_cc_bug10	3423	0.55	1.15	3564	0.56	1.41	259	0.00	0.02	0.02	0.68
dlx2_cc_bug11	6037	1.03	1.60	6886	1.35	2.20	10528	0.43	0.02	0.06	0.76
dlx2_cc_bug12	7099	1.43	2.00	5702	1.05	1.91	11099	0.44	0.07	0.10	0.75
dlx2_cc_bug13	5998	1.12	1.69	6133	1.08	1.91	12049	0.50	0.03	0.02	0.62
dlx2_cc_bug14	253	0.01	0.59	298	0.01	0.87	234	0.01	0.12	0.02	0.63
dlx2_cc_bug15	4405	1.27	1.93	3756	0.99	1.99	296	0.01	0.01	0.06	0.94
dlx2_cc_bug16	252	0.01	0.58	297	0.01	0.86	233	0.01	0.13	0.01	0.63
dlx2_cc_bug17	504	0.06	1.16	4453	1.01	2.97	5806	0.40	0.01	0.01	2.23
dlx2_cc_bug18	1066	0.10	1.06	3236	0.78	2.51	337	0.01	0.01	0.02	2.28
dlx2_cc_bug19	269	0.02	0.63	302	0.02	0.89	4452	0.15	0.01	0.00	0.64
dlx2_cc_bug20	703	0.03	0.60	777	0.50	0.89	521	0.01	0.01	0.02	1.89
dlx2_cc_bug21	331	0.02	0.59	360	0.02	0.85	458	0.01	0.01	0.01	0.62
dlx2_cc_bug22	744	0.40	0.62	865	0.05	0.91	4456	0.19	0.01	0.04	0.70
dlx2_cc_bug23	620	0.03	0.60	323	0.02	0.86	4726	0.14	0.01	0.10	0.76
dlx2_cc_bug24	270	0.02	0.59	313	0.02	0.86	4034	0.14	0.01	0.04	0.63
dlx2_cc_bug25	3931	0.75	1.32	3233	0.59	1.44	4406	0.14	0.01	0.02	0.67
dlx2_cc_bug26	4200	0.83	1.42	3687	0.48	1.58	543	0.02	0.02	0.02	0.66
dlx2_cc_bug27	591	0.02	0.52	2979	0.46	1.16	293	0.01	0.03	0.00	1.49
dlx2_cc_bug28	2205	0.22	0.88	5275	1.09	2.05	339	0.01	0.08	0.01	0.96
dlx2_cc_bug29	324	0.01	0.58	334	0.02	0.87	243	0.01	0.19	0.03	0.64
dlx2_cc_bug30	311	0.02	0.60	267	0.02	0.89	323	0.01	0.30	0.02	1.83
dlx2_cc_bug31	294	0.02	0.58	325	0.02	0.88	247	0.00	0.24	0.02	0.65
dlx2_cc_bug32	278	0.02	0.59	317	0.02	0.86	242	0.00	0.02	0.01	0.67
dlx2_cc_bug33	299	0.02	0.58	305	0.02	0.88	272	0.01	0.19	0.06	0.67
dlx2_cc_bug34	329	0.02	0.60	506	0.03	0.86	298	0.01	0.30	0.02	1.81
dlx2_cc_bug35	282	0.02	0.59	328	0.02	0.89	318	0.01	0.32	0.03	0.59
dlx2_cc_bug36	279	0.02	0.61	325	0.02	0.86	316	0.01	0.08	0.07	0.66
dlx2_cc_bug37	3643	0.71	1.28	3214	0.60	1.45	329	0.01	0.05	0.01	0.63
dlx2_cc_bug38	6249	0.43	1.70	5854	1.09	1.93	9500	0.36	0.44	0.07	1.31
dlx2_cc_bug39	3307	0.54	1.07	6058	1.04	1.88	12314	0.50	0.04	0.40	1.83
dlx2_cc_bug40	8046	1.64	2.21	6748	1.40	2.26	9972	0.41	0.12	0.02	0.63

Table 6: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the DLX benchmarks. Benchmarks above the line are satisfiable the rest are unsatisfiable.

`slider`m`_sat`:

$$f = (x_1 \oplus (\neg x_{i_3} \wedge x_{i_1}) \oplus \neg(x_{m/2} \wedge x_{i_4})) \equiv ite(x_{i_2}, x_{i_1} \vee \neg x_{m/2}, \neg x_{i_1})$$
$$g = \neg x_1 \oplus (x_{j_2} \oplus (\neg x_{j_3} \wedge x_{j_4}) \oplus x_{j_3}) \oplus (x_{m/2} \equiv x_{j_1})$$

where the input indices for f and g are different for each slider and are given by the following tables:

	SBSAT			zChaff		Siege		BerkMin	SATZoo	
Name	number choices	branch (sec)	total (sec)	number choices	total (sec)	number choices	total (sec)	total (sec)	number choices	total (sec)
slider60_sat	1152	0.04	0.14	534	0.02	2900	0.16	0.09	60746	2.2
slider70_sat	1265	0.05	0.22	1511	0.07	329	0.01	0.01	58962	1.85
slider80_sat	111575	5.12	5.22	149153	52.8	38044	6.20	73.5	25	0.27
slider90_sat	3576	0.18	0.30	66152	14.3	47180	9.56	8.63	119680	5.54
slider100_sat	51994	2.69	2.83	104054	85.5	70693	35.8	48.2	710943	40.5
slider110_sat	282213	15.8	16.0	280126	173.3	576670	437.4	801.4	237571	12.41
slider120_sat	1539977	86.1	86.3	—	—	—	—	—	214258	11.42
slider60_unsat	10004	0.37	0.46	27414	3.4	19505	2.63	4.37	86216	4.28
slider70_unsat	9373	0.39	0.50	18157	1.93	17735	2.21	2.17	85962	3.4
slider80_unsat	190177	8.57	8.67	245112	116.6	215436	104.4	—	584733	35
slider90_unsat	626812	29.7	29.8	685026	513.4	501539	302.5	—	1428857	87.3
slider100_unsat	2403878	124.1	124.2	1495633	3094	2482913	6540	—	3995389	284
slider110_unsat	10256075	564.5	564.7	—	—	—	—	—	16966800	1285

Table 7: SBSAT, zChaff, Siege, BerkMin, SATZoo times on the Slider benchmarks

f:

m	i_1	i_2	i_3	i_4
60	13	15	17	24
70	12	15	17	24
80	15	17	33	24
90	15	17	24	33
100	15	17	24	43
110	15	17	24	43
120	15	24	43	57

g:

m	j_1	j_2	j_3	j_4
60	12	16	18	27
70	12	15	19	27
80	12	16	18	27
90	12	16	18	27
100	18	26	27	42
110	20	26	27	42
120	6	18	27	42

and `slider`m`_unsat`:

$$f = (x_1 \oplus (\neg x_{i_3} \wedge x_{i_1}) \oplus \neg(x_{m/2} \wedge x_{i_4})) \equiv ite(x_{i_2}, x_{i_1} \vee \neg x_{m/2}, \neg x_{i_1})$$
$$g = (x_{m/2} \equiv (\neg x_1 \oplus (x_{j_2} \oplus (\neg x_{j_3} \wedge x_{j_4}) \oplus x_{j_3}) \oplus (x_{j_5} \equiv x_{j_1})))$$

If "unsat" is in the name of the benchmark, then it is unsatisfiable, otherwise it is satisfiable. The number in the name of each benchmark refers to the value of m. The value of k for all benchmarks is fixed at 6 and the value of l is 6 or 7 (see the beginning of this section for an explanation of this family of benchmarks and the meaning of m, k and l). The two functions were chosen to yield somewhat balanced BDDs, requiring nearly all inputs to have a value before an inference could be established.

These are designed to be hard problems and no solver does well for even "small" problems involving just a couple of hundred variables. The usefulness of lemmas is rather limited on these benchmarks so we turned the lemmas off on sbsat in Tab. 7. Preprocessing also did not accomplish much and the default preprocessing took little time. We did not know how to turn lemmas off on the other solvers. Both SBSAT and SATZoo did much better than the rest. This is interesting because both are designed to take a "more general view" of what SAT

	SBSAT with Lemmas			SBSAT without Lemmas		
Name	number choices	branch (sec)	total (sec)	number choices	branch (sec)	total (sec)
slider60_sat	1051	0.10	0.25	1152	0.04	0.14
slider70_sat	622	0.06	0.27	1265	0.05	0.22
slider80_sat	79884	39.2	39.4	111575	5.12	5.22
slider90_sat	2765	0.44	0.64	3576	0.18	0.30
slider100_sat	36761	15.4	15.9	51994	2.69	2.83
slider110_sat	171163	113.2	113.4	282213	15.8	16.0
slider120_sat	—	—	—	1539977	86.1	86.3
slider60_unsat	9227	1.27	1.49	10004	0.37	0.46
slider70_unsat	7957	1.29	1.46	9373	0.39	0.50
slider80_unsat	148242	78.6	78.8	190177	8.57	8.67
slider90_unsat	429468	263.0	263.4	626812	29.7	29.8
slider100_unsat	1600514	1065	1066	2403878	124.1	124.2
slider110_unsat	—	—	—	10256075	564.5	564.7

Table 8: SBSAT times and choice points on the Slider benchmarks, with and without Lemmas.

by exploiting domain-specific information where possible and available. The gap between these solvers and the rest is enormous.

Tab. 8 compares the running times of SBSAT on sliders with lemmas turned off and on. The number of choice points generated did not change very much, showing that learning from conflict analysis does not make much of a difference in this case and, therefore, suggesting that these are hard problems. Observe that SBSAT running time changes by an order of magnitude with lemmas turned on. This clearly points to adjustments that must be made to lemma handling. Despite this, SBSAT does better than all the solvers except SATZoo with lemmas turned on.

Sample BDD statistics are shown in Tab. 9 and Tab. 10 for the `queueinv` and `longmult` benchmarks. The meaning of each column is as follows: `orig. benchmark, number vars` is the number of variables in the original benchmark; `orig. benchmark, number BDDs` is the number of BDDs in the original benchmark; `after preprocessing, number vars` is the number of variables in BDDs after SBSAT preprocessing; `after preprocessing, number BDDs` is the number of BDDs internal to SBSAT after preprocessing; `SBSAT Times, preproc` is the number of seconds spent by SBSAT in preprocessing; `SBSAT Times, w/preproc` is the time spent during search if the preprocessor is not run; `SBSAT Times, w/o preproc` is the time spent during search if the preprocessor is run.

These results, plus those shown in Tab. 2, illustrate that the effect of preprocessing varies considerably from problem to problem. Preprocessing did little in the case of `queueinv` problems, helped considerably in the case of `longmult` benchmarks, and was enough to solve the problem in the case of `barrel` benchmarks.

	orig. benchmark		after preprocessing		SBSAT times (seconds)		
benchmark	number vars	number BDDs	number vars	number BDDs	preproc	search w/ preproc	search w/o preproc
queueinv4	258	955	226	244	0.1	0.01	0.02
queueinv8	514	2273	464	522	0.2	0.07	0.19
queueinv12	1114	7335	1049	2489	0.6	0.61	1.56
queueinv16	1170	6496	1090	1286	0.3	0.80	1.88
queueinv20	2437	20671	2343	7142	3.5	6.06	11.80
queueinv24	3241	28214	3135	9948	6.6	13.51	21.73
queueinv28	4173	37006	4055	13250	11.4	30.75	38.39
queueinv32	3202	22468	3068	3751	2.1	13.13	23.49
queueinv36	6584	69972	6437	24438	72.6	105.36	178.63
queueinv40	7900	84615	7741	29832	125.9	182.08	284.83
queueinv44	9344	100699	9173	35786	207.8	332.60	433.52
queueinv48	10916	118222	10733	42300	327.3	462.11	794.42
queueinv52	12616	137187	12421	49374	466.2	736.97	958.61
queueinv56	14444	157590	14237	57008	655.0	943.40	1310.02

Table 9: Sample preprocessing statistics: the `queueinv` benchmark set.

	orig. benchmark		after preprocessing		SBSAT times (seconds)		
# time steps	number vars	number BDDs	number vars	number BDDs	preproc	search w/ preproc	search w/o preproc
4	1343	1531	140	288	0.50	0.17	0.32
8	2474	2738	344	722	1.24	7.59	19.35
12	3605	3945	548	1161	2.50	32.70	64.69
16	4735	5151	752	1592	3.34	46.55	50.10
20	5866	6358	956	2026	4.89	64.13	128.75
24	6997	7565	1160	2465	6.74	87.76	185.40
28	8127	8771	1364	2896	8.89	108.18	229.03
32	9258	9978	1568	3330	11.54	115.13	137.96
36	10389	11185	1772	3769	14.49	126.77	278.14
40	11519	12391	1976	4200	18.22	167.46	440.67
50	14346	15408	2486	5286	27.72	138.57	520.70
60	17173	18425	2996	6377	39.49	256.16	611.12
70	19999	21441	3506	7460	52.91	355.51	729.46

Table 10: Sample preprocessing statistics: the `longmult` benchmark set.

7.2 Space consumption

As noted, SBSAT trades time for space. Our primary concern is with solver time, but space is certainly an important issue. Accordingly, we have run a limited space comparison for CNF solvers and SBSAT. The results are shown in Tables Tab. 11, Tab. 12, Tab. 13, and Tab. 14. The meaning of the columns of these tables is as follows: `BMC vars` shows the number of variables in the classical logic formula produced by `bmc` for the given benchmark; `3 addr vars` shows the number of variables in the 3-address code translation of the logic formula (it is approximately the sum of the number of BMC vars and the number of temporary variables created to reflect sharing in the 3-address code); `CNF number vars` shows the number of variables in the CNF output of `bmc` for the given benchmark; `CNF number clauses` shows the number of clauses of that output; `CNF number literals` shows the total number of literals of that output; `SBSAT # vars bef preproc` shows the number of variables in BDDs of SBSAT before preprocess-

	BMC		CNF			SBSAT		
#time steps	BMC vars	3 addr vars	number vars	number clauses	number literals	# vars bef preproc	# vars aft preproc	number states
4	435	4851	3694	11316	26896	1343	365	86747
8	783	9055	6858	21188	50452	2474	822	256481
12	1131	13259	10022	31060	74008	3605	1280	426230
16	1479	17463	13186	40932	97564	4735	1737	592727
20	1827	21667	16350	50804	121120	5866	2194	762461
24	2175	25871	19514	60676	144676	6997	2652	932210
28	2523	30075	22678	70548	168232	8127	3109	1098707
32	2871	34279	25842	80420	191788	9258	3566	1268441
36	3219	38483	29006	90292	215344	10389	4024	1438190
40	3567	42687	32170	100164	238900	11519	4481	1604687
50	4437	53197	40080	124844	297790	14346	5624	2027411
60	5307	63707	47990	149524	356680	17173	6768	2450150
70	6177	74217	55900	174204	415570	19999	7911	2869637

Table 11: Space requirements for the `longmult` problem set

	BMC		CNF			SBSAT		
benchmark	BMC vars	3 addr vars	number vars	number clauses	number literals	# vars bef preproc	# vars aft preproc	number states
queue4	89	1069	707	2655	6684	437	76	4104
queue8	260	3575	2246	9371	24008	1333	290	76155
queue12	519	7886	4713	21576	56071	2767	513	252389
queue16	815	12341	7461	34536	89703	4439	863	422474
queue20	1238	20633	11752	59282	155933	7118	1856	688994
queue22	1448	23995	13754	69656	183207	8352	2204	853701
queue24	1674	27596	15908	80822	212553	9682	2579	1023972
queue28	2174	35520	20672	105646	277809	12630	3411	1419415
queue32	2738	44307	26012	133382	350649	15898	4355	1885587

Table 12: Space requirements for the `queue` problem set

ing; `SBSAT # vars aft preproc` shows the number of variables in BDDs of SBSAT after preprocessing; `SBSAT number states` shows the number of states produced by SBSAT (many of these are shared by individual Smurfs). The numbers in columns `3 addr vars` and `SBSAT vars bef preproc` differ because SBSAT might pick temporary variables at different points than the 3-address code did. The 3-address code contains temporary variables for construction of the 3-address code itself (breaking the big formula into 3-address code pieces). SBSAT reconstructs a large formula by eliminating the 3-address code temporary variables but SBSAT has to introduce its own temporary variables because large formulas may not be able to be handled.

Comparing the total number of variables in the CNF translation to the number of variables in the SBSAT translation, before and after preprocessing, gives a simplistic comparison of search space sizes. Also relevant is the amount of space needed for lemmas, since information processed into the Smurf states by SBSAT must be gathered in lemmas in a CNF solver. This is difficult to measure meaningfully. Instead we measure the size of the memory heap used during execution. The results are shown, measures in maximum bytes used, in Tab. 15, for most of the `bmc` benchmarks.

	BMC		CNF			SBSAT		
benchmark	BMC vars	3 addr vars	number vars	number clauses	number literals	number vars	number vars	number states
queueinv4	34	416	256	955	2421	140	69	16697
queueinv8	56	886	512	2273	5878	282	135	54284
queueinv12	78	2509	1112	7335	18671	1067	411	178222
queueinv16	94	2140	1168	6496	17172	822	369	227389
queueinv20	116	6128	2435	20671	53074	3041	1142	773577
queueinv24	132	7912	3239	28214	72245	3915	1513	826804
queueinv28	148	9952	4171	37006	94571	4894	1935	1524322
queueinv32	164	6114	3200	22468	60653	2726	1293	618146
queueinv36	186	17923	6582	69972	180696	12016	5444	2330404
queueinv40	202	20987	7898	84615	218183	14117	6523	3050507
queueinv44	218	24307	9342	100699	259337	16371	7695	3793711

Table 13: Space requirements for the `queueinv` problem set

	BMC		CNF			SBSAT		
benchmark	BMC vars	3 addr vars	number vars	number clauses	number literals	number vars	number vars	number states
permute2	21	170	93	285	679	39	0	–
permute3	40	371	220	717	1758	93	0	–
permute4	85	858	474	1782	4533	223	29	12540
permute5	126	1419	797	3160	8128	375	51	66189
permute6	168	2076	1182	4749	12320	549	79	135377
permute7	216	2901	1671	6770	17684	747	115	199472
permute8	351	4956	2752	11885	31520	1106	221	328843
permute9	440	6725	3758	16758	44551	1479	272	255349
permute10	528	8608	4849	21633	57713	1861	352	241232
permute11	624	10807	6130	27342	73167	2283	434	278564
permute12	728	13334	7616	33948	91090	2768	512	359966
permute13	840	16249	9322	41514	111659	3317	610	388994
permute14	960	19540	11263	50103	135051	3918	717	438321
permute15	1088	23243	13454	59778	161443	4595	847	796712
permute16	1513	33418	18782	86452	235177	7799	1047	280507
permute17	1710	39755	22371	105149	285905	9626	1189	129502
permute18	1900	46068	26008	121971	332070	11092	1322	148540
permute19	2100	53013	30017	140469	382883	12696	1481	163518

Table 14: Space requirements for the `permute` problem set

benchmark	heap size	benchmark	heap size	benchmark	heap size	benchmark	heap size
barrel2	57347699	longmult4	177624072	queueinv4	75966624	queue4	71342822
barrel3	65738360	longmult8	238379638	queueinv8	164327032	queue8	172283234
barrel4	65898366	longmult12	375604642	queueinv12	212748644	queue12	234560404
barrel5	145132600	longmult16	508778176	queueinv16	225558180	queue16	297358828
barrel6	147410020	longmult20	569725386	queueinv20	611524038	queue20	551008538
barrel7	150566373	longmult24	705680618	queueinv24	718164568	queue24	751931588
barrel8	154689059	longmult28	765645384	queueinv28	1070479936	queue28	974986692
barrel9	374259779	longmult32	922572570	queueinv32	448472276	queue32	1226077516
barrel10	379225611	longmult36	1058907826	queueinv36	1469731282		
barrel11	385121937	longmult40	1035171360	queueinv40	1917585056		
barrel12	467617070	longmult50	1260972326	permute2	57358331		
barrel13	476897812	longmult60	1564560974	permute3	58144832		
barrel14	485636446	longmult70	1790079068	permute4	74235196		
barrel15	572138081			permute5	169642700		
barrel16	584031096			permute6	193695400		
barrel17	1849457431			permute7	217373212		
				permute8	465487268		
				permute9	312690458		

Table 15: Heap size for all `bmc` problem sets

8 Conclusions

We showed there is merit in staying close to the user-domain, particularly on BMC problems where we were able to explore performance for three-address and translated CNF versions of the same inputs. When translating to CNF, a general, lengthy expansion of the input formula is difficult to avoid. By not translating we avoid dealing with large numbers of CNF clauses and instead work with relatively few complex functions; those functions likely contain exploitable, ungarbled domain-specific information. We have introduced and studied structures for maintaining and using that information efficiently in complex search heuristics.

Our method of translating `bmc` output to BDDs (see Section 6) is still quite primitive: since the `bmc` output tends to be too large to read in directly as BDDs, we use a simple greedy algorithm to split the formula. Nevertheless, SBSAT's times in most of the experiments reported in this manuscript are significantly better than zChaff's. We anticipate that, as we refine our splitting strategy, as well as search heuristics, SBSAT will be substantially improved.

We have four particular reasons to explain why SBSAT should outperform CNF-based solvers on Bounded Model Checking problems:

1. SBSAT's translation clusters a good deal of information about adjacent time steps in the same BDDs. Accordingly, SBSAT will make some inferences that initially go undetected in other solvers. A good example of this is illustrated in Tab. 2. In that class of problems many variables are related by equivalences and such related variables are usually in the same function describing a particular time step. As long as these functions can be represented by relatively small BDDs, SBSAT will catch these equivalances in preprocessing.

2. The zChaff search heuristic seems to us to be a combination of identifying (learning) "key literals" ("back-door variables" [Kautz et al. 97]) and branching on them eagerly. For a BMC problem with a very large number of time steps, there may be a great deal of similarity between one time step and the next. We conjecture that, as a result, there will tend to be many literals with very similar heuristic weights. SBSAT's LSGB heuristic, with its local lookaheads, might be better able to make effective choices in this case.

3. Input is utilized *before* translation to CNF. Therefore, information relating variables, available in the original problem, is not garbled by a translation to CNF which can greatly increase the size of the input as well add variables that do not exist in the original formulation. Because the domain-specific

information remains intact, it is more likely to be exploited by specially designed search heuristics.

4. SBSAT trades space for time. A lot more space is utilized by SBSAT than by a typical SAT solver on a CNF translation of an input due to the construction of SMURFs. However, this space is used to memoize data that can be used over and over again by search heuristics. Therefore, the extra space used is compensated for by the savings in time achieved through table lookups of memoized data during search.

We do not have the data to confirm any of these conjectures, but our data shows that the approach of SBSAT speeds up the solution of many bounded model checking and other problems.

9 Future Work

There are numerous improvements that can be made and are some things that must be studied due to the unconventional nature of the paradigm. Probably the most significant is the effect of caching on performance. Since SBSAT attempts to use very large blocks of memory, the question of how much time is spent moving blocks dedicated to SMURFs between cache levels is important. A similar problem arises in maintaining a large lemma base (the dlx problems, for example, benefit greatly from a database of 100,000 lemmas). A related question is to determine whether there is an easily implemented optimal way of storing SMURFs which minimizes delays due to caching. Finally we mention the possibility of changing lemma cache size dynamically (more precisely, considering varying portions of the lemma cache to be "alive"). Until now these questions have not been considered: our main thrust has been to see whether the idea of function-complete lookahead on non-translated conjunctions of Boolean functions to support efficient implementations of complex search heuristics is feasible.

We have built SBSAT with only one search heuristic in mind. Clearly, heuristics need to be tailored to the problem sets they are working on. Heuristics for lemma generation and disposal should also be tailored to the problem domain. In future work the possibility of tailoring heuristics will be studied. This includes support for user-built modules responsible for collecting inferences and memoization in preprocessing as well as support for user-built modules which define search and lemma heuristics based on the memoized information. It also includes attempts at attacking problems which, until now, have shown exponential complexity for all search or BDD methods, for example the `permute` set from BMC. We expect the study of sliders will help greatly in improving performance on many BMC benchmarks due to the combination of the similarity they hold with those benchmarks, their simplicity, and the ability to control their hardness.

We have experimented with a three-level lemma cache (built into SBSAT) and found it to be useful. We need to further explore its potential.

Splitting in BMC problems can be too fine-grained. We need to study how to come up with a better splitter in such cases.

Currently, BDD preprocessing is limited to relatively small BDDs so that reasonably sized SMURFs can be built. There is no reason we can't build larger BDDs during prepreprocessing in order to reveal more inferences and then fall back on a set of smaller BDDs to build the SMURFs from. The extra effort in building larger BDDs may not necessarily be wasted if no extra inferences are revealed. Further research should tell us how best to proceed with the larger BDDs.

SBSAT currently has problems solving some classes of benchmarks due to memory limitations. These problems must be fixed. Some problems will disappear with time due to advancements in computer hardware. But some problems will have to be solved by finding clever ways to build useful SMURFs (that is, from large enough input functions) that are small enough not to cause the memory problems.

We are aware of the existence and use of more sophisticated clustering algorithms. We anticipate such improvements in coming versions. These are currently lacking only because our focus has been on proving the concept.

Work reported here represents the tip of the iceberg, so to speak. The reader who is interested in knowing all the options implemented as well as explanations of when they are effective is referred to the sbsat user manual which is available from the corresponding author.

Acknowledgements

This work was supported in part by DoD contracts MDA-904-99-C-4547 and MDA-904-02-C-1162.

References

[Bacchus and Winter 03] Bacchus, F., and Winter, J.: "Effective Preprocessing with Hyper-Resolution and Equality Reduction"; Proc. 6th Internation Symposium on the Theory and Applications of Satisfiability Testing, Portofino, Italy (2003). Available from `www.cs.toronto.edu/~fbacchus/Papers/BSAT2003.pdf`.

[Berkmin 561] Berkmin homepage: `http://eigold.tripod.com/BerkMin.html`.

[Biere et al. 99] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y.: "Symbolic Model Checking without BDDs"; Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), part of European Conferences on Theory and Practice of Software (ETAPS'99), Amsterdam, Lecture Notes in Computer Science 1579, Springer, New York (1999), 193–207.

[Biere 99] Biere, A.: Bounded model checking page at CMU, `http://www-2.cs.cmu.edu/~modelcheck/bmc.html`.

[Brace et al. 90] Brace, K. S., Rudell, R. L., Bryant, R. E.: "Efficient Implementation of a BDD Package"; Proc. 27th ACM/IEEE Design Automation Conference, ACM, New York (1990), 40–45.

[Bryant 86] Bryant, R. E.: "Graph-Based Algorithms for Boolean Function Manipulation"; IEEE Transactions on Computers, C-35 (1986), 677–691.

[Cabodi et al. 03] Cabodi, G., Nocco, S., Quer, S.: "Improving SAT-Based Bounded Model Checking by Means of BDD-Based Approximate Traversals,"; Design, Automation, and Test in Europe (DATE '03) (2003), 898–903.

[Clarke et al. 01] Clarke, E.M., Biere, A., Raimi, R. E., Zhu, Y.: "Bounded Model Checking Using Satisfiability Solving"; Formal Methods in System Design, 19, (2001) 7–34.

[Clarke et al. 99] Clarke, E. M., Grumberg, O., Peled, D.: "Model Checking"; MIT Press (1999).

[Cimatti et al. 01] Cimatti, A., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: "NuSMV Version 2: BDD-Based + SAT-Based Symbolic Model Checking"; Available from http://sra.itc.it/people/roveri/papers/IJCAR01.ps.gz (2001).

[Coudert and Madre 90] Coudert, O., Madre, J. C.: "A Unified Framework for the Formal Verification of Sequential Circuits"; Proc. International Conference on Computer-Aided Design, ACM, New York (1990), 126–129.

[Coudert and Madre 91] Coudert, O., Madre, J. C.: "Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion"; Proc. International Workshop on Formal Methods in VLSI Design, Miami Florida, USA, ACM-SIGDA, New York (1991).

[CUDD 04] Colorado University Decision Diagram package. Available from http://vlsi.colorado.edu/~fabio/CUDD/.

[Damiano and Kukula 03] Damiano, R., Kukula, J.: "Checking satisfiability of a conjunction of BDDs"; Proc. 40th ACM/IEEE Design Automation Conference (DAC '03), ACM, New York (2003), 818–823.

[Davis et al. 62] Davis, M., Logemann, G., Loveland, D.: "A Machine Program for Theorem Proving"; Communications of the Association of Computing Machinery" 5 (1962), 394–397.

[Franco et al. 01] Franco, J., Dransfield, M., Vanfleet, W. M., Schlipf, J. S.; "State-based Propositional Satisfiability Solver"; US Provisional Patent Application No. 60/296,380 (2001).

[Franco et al. 04] Franco, J., Kouril, M., Schlipf, J. S., Ward, J., Weaver, S., Dransfield, M., Vanfleet, W. M.: "SBSAT: a state-based, BDD-based Satisfiability solver"; Lecture Notes in Computer Science, #2919, Springer, New York (2004), 398–410.

[Freeman 95] Freeman, J.: "Improvements to Propositional Satisfiability Search Algorithms"; Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania (1995).

[Giunchiglia and Sebastiani 99] Giunchiglia, E., and Sebastiani, R.: "Applying the Davis-Putnam Procedure to Non-Clausal Formulas"; Proc. AI*IA '99, Lecture Notes in Artificial Intelligence, #1792, Springer, New York (1999).

[Goldberg and Novikov 02] Goldberg, E., Novikov, Y.: "BerkMin: A Fast and Robust Sat-Solver Design"; Proc. Design, Automation, and Test in Europe (DATE '02), IEEE, New York (2002), 142–149.

[Gopalakrishnan et al. 03] Gopalakrishnan, S., Durairaj, V., Kalla, P.: "Integrating CNF and BDD Based SAT Solvers"; Proc. IEEE International High-Level Design Validation and Test Workshop (HLDVT '03), IEEE, New York (2003), 51–56.

[Gupta and Ashar 98] Gupta, A., Ashar, P.: "Integrating a Boolean Satisfiability Checker and BDDs for Combinational Equivalence Checking"; Proc. 11th IEEE International Conference on VLSI Design: VLSI for Signal Processing, IEEE, New York (1998), 222–225.

[Gupta et al. 03] Gupta, A., M. Ganai, C. Wang, Z. Yang, and P. Ashar.: Learning from BDDs in SAT-based bounded model checking. *Proc. of ACM/IEEE Design Automation Conference*, Anaheim CA, USA, 824–830, June 2003.

[Heule 04] Heule, M.J.H.: "March, towards a lookahead SAT solver for general purposes"; Master's thesis, TU Delft, The Netherlands, March 2004.

[Heule and van Maaren 04] Heule, M.J.H. and van Maaren, H.: "Aligning CNF- and equivalence-reasoning"; Proc. 7th International Symposium on the Theory and Applications of Satisfiability Testing, Vancouver, B.C., Canada (2004). Available from http://www.satisfiability.org/SAT04/programme/72.pdf.

[Hoos 02] Hoos, H.: Satlib. Available from www.satlib.org/Benchmarks/SAT/New/Competition-02/sat-2002-beta.tgz.

[Hong et al. 97] Hong, Y., Beerel, P., Burch, J., McMillan, K.: "Safe BDD Minimization Using Don't Cares"; Proc. 34th ACM/IEEE Design Automation Conference (DAC '97), ACM, New York (1997), 208–213.

[Jeroslow and Wang 90] Jeroslow, R. E., Wang, J.: "Solving propositional satisfiability problems"; Annals of Mathematics and AI, 1 (1990), 167–187.

[Jin and Somenzi 04] Jin, H., Somenzi, F.: "CirCUs: A Hybrid Satisfiability Solver"; Proc. 7th International Symposium on the Theory and Applications of Satisfiability Testing, Vancouver, B.C., Canada (2004). Available from http://www.satisfiability.org/SAT04/programme/24.pdf.

[Johnson 74] Johnson, D. S.: "Approximation Algorithms for Combinatorial Problems"; Journal of Computer and Systems Sciences, 9 (1974), 256–278.

[Kalla et al. 00] Kalla, P., Zeng, Z., Ciesielski, M. J., Huang, C.: "A BDD-based Satisfiability Infrastructure Using the Unate Recursive Paradigm"; Proc. Design, Automation, and Test in Europe (DATE '00), IEEE, New York (2000), 232–236.

[Kautz et al. 97] Kautz, H., McAllester, D., Selman, B.: "Exploiting Variable Dependency in Local Search"; Available from http://www.cs.washington.edu/homes/kautz/papers/dagsat.ps.

[Kuehlmann et al. 01] Kuehlmann, A., Ganai, M. K., Paruthi, V.: "Circuit-Based Boolean Reasoning"; Proc. 38th ACM/IEEE Design Automation Conference (DAC '01), ACM, New York (2001), 232–237.

[Lynce and Marques-Silva 02] Lynce, I., Marques-Silva, J.: "Efficient Data Structures for Backtrack Search SAT Solvers"; Proc. 5th International Symposium on the Theory and Applications of Satisfiability Testing, Cincinnati, Ohio (2002), 308–315. Available from http://gauss.ececs.uc.edu/Conferences/SAT2002/Abstracts/lynce.ps.

[Marques-Silva and Sakallah 96] Marques-Silva, J., Sakallah, K.: "GRASP: a new search algorithm for Satisfiability"; Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD '96), IEEE, New York (1996), 220–227.

[Moskewicz et al. 01] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S.: "Chaff: Engeneering an efficient SAT solver"; Proc. 38th Design Automation Conference (DAC '01), IEEE, New York (2001), 530–535.

[Novikov 03] Novikov, Y.: "Local Search for Boolean Relations on the Basis of Unit Propagation"; Proc. Design, Automation, and Test in Europe (DATE '03), IEEE, New York (2003), 810-815.

[Paruthi and Kuehlmann 00] Paruthi, V., Kuehlmann, A.: "Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation"; Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE, New York (2000), 459–464.

[Puri and Gu 96] Puri, R., Gu, J.: "A BDD SAT Solver for Satisfiability Testing: An Industrial Case Study"; Annals of Mathematics and Artificial Intelligence, 17 (1996), 315–337.

[Reda et al. 02] Reda, S., Drechsler, R., Orailoglu, A.: "On the Relation Between SAT and BDDs for Equivalence Checking," International Symposium on Quality of Electronic Design (ISQED '02) (2002), 394–399.

[Ryan 03] Ryan, L.: Siege (2003). Available from http://www.cs.sfu.ca/~loryan/personal/.

[SatZoo 03] SatZoo homepage: http://www.cs.chalmers.se/ een/Satzoo/.

[Schöning 89] Schöning, U.: "Logic for Computer Scientists", Springer, New York (1980), Page 22.

[Stålmarck 94] Stålmarck, G.: "A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula"; Swedish Patent No. 467,076 (approved 1992), U.S. Patent No. 5,276,897 (1994), European Patent No. 0403,454 (1995).

[Subbarayan and Pradhan 04] Subbarayan, S., and Pradhan, D.: “NiVER: Non increasing variable elimination resolution for preprocessing SAT instances”; Available from
`http://www.itu.dk/people/sathi/niver-SAT2004.pdf`.

[Tseitin 68] Tseitin, S.: “On the Complexity of Derivations in Propositional Calculus”; In *Structures in Constructive Mathematics and Mathematical Logic, Part II*, A.O. Slisenko, ed. (1968), 115–125.

[Velev 00] Velev, M. N.: Superscalar Suite 1.0. Available from
`http://www.ece.cmu.edu/~mvelev`.

[zChaff 2003] zChaff: `http://www.ee.princeton.edu/ chaff/zchaff/index.html`.

[Zhang 97] Zhang, H.: “SATO: an efficient propositional prover”; Proc. International Conference on Automated Deduction (CADE ’97), Lecture Notes in Artificial Intelligence 1249, Springer, New York (1997), 272–275.

[Zhang 01] Zhang, L., Madigan, C., Moskewicz, M. W., Malik, S.: “Efficient Conflict Driven Learning in a Boolean Satisfiability Solver”; Proc. International Conference on Computer Aided Design (ICCAD ’01), (2001), 279–285.