Checking Satisfiability of a Conjunction of *BDD*s

Robert Damiano
Advanced Technology Group
Synopsys, Inc.
Hillsboro, OR
robertd@synopsys.com

James Kukula
Advanced Technology Group
Synopsys, Inc.
Hillsboro, OR
kukula@synopsys.com

ABSTRACT

Procedures for Boolean satisfiability most commonly work with Conjunctive Normal Form. Powerful SAT techniques based on implications and conflicts can be retained when the usual CNF clauses are replaced with BDDs. BDDs provide more powerful implication analysis, which can reduce the computational effort required to determine satisfiability.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms

Algorithms, verification

Keywords

Satisfiability, BDD

1. INTRODUCTION

Determining whether a Boolean formula has a satisfying assignment is a fundamental problem with many applications in CAD. Automatic test pattern generation and formal verification, both combinational equivalence checking and sequential model checking, are primary applications. Boolean formulas can be directly derived from circuit designs at register transfer or gate levels. The formula to be analyzed may also include logic derived from other parts of the problem specification, such as the temporal logic formula to be checked, etc. Whatever the application, a Boolean formula can be created in some standard form and given to a satisfiability engine. A more powerful satisfiability engine can thereby benefit a variety of applications. Engines in industrial use are based on binary decision diagrams (BDDs) [3], conjunctive normal form (CNF), and logic circuits, as well as various hybrids of these. We introduce here a new approach, where powerful non-chronological backtracking techniques developed for use with CNF can be applied to an implicit conjunction of BDDs.

Non-chronological backtracking techniques have a long history (see [1] for a sketch of the history), but broke into CAD usage primarily with the introduction of GRASP [7]. Further refinements were incorporated in CHAFF [8]. Our new approach reproduces the basic mechanisms of GRASP in a different context. We will first sketch how GRASP works with CNF, and then show how we have used a similar approach with BDDs.

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

2. PARTIAL ASSIGNMENT SEARCH

A boolean formula in CNF consists of a set of clauses, where each clause is a set of literals, and each literal is an instance of a variable or of its complement. A clause represents the disjunction of its literals, and a CNF formula the conjunction of the clauses. It is in general difficult to find an assignment of values to variables for which a CNF formula evaluates to true - indeed, this is a canonical instance of the NP-complete class of problems. To find such a satisfying solution, GRASP (and many other SAT solvers) searches the space of partial assignments.

In a partial assignment, some variables are assigned a 1 value, some are assigned 0, and the remaining variables are left unassigned. Each clause of a CNF formula can then be evaluated, yielding an evaluation for the entire formula. If the partial assignment gives a 1 value to any literal in a clause, the clause has value 1. If all the literals have value 0, then the clause has value 0. If no literal has value 1 but some literal has an unassigned value, then the clause itself has unassigned value. The entire CNF formula then has value 0 if any clause has value 0, 1 if all clauses are 1, and unassigned otherwise.

By evaluating partial assignments in this way, GRASP searches for an assignment for which the CNF formula evaluates to 1. The basic operations in the search either extend a partial assignment by assigning more variables to 0 or 1, or contract it by unassigning some variables.

When a partial assignment leaves the value of the CNF unassigned, GRASP extends the partial assignment. First, clausal implications are propagated. If a partial assignment leaves any clause with only one unassigned literal and all other literals with value 0, then for the clause (and CNF) to be true, that last literal must be assigned the value 1. The partial assignment can therefore be extended to include this implied value. Each such extension may trigger further implications. When no further implications can be drawn, the partial assignment may have been extended enough that the

CNF formula itself has reached an assigned value. If not, a second mechanism, called decision, is invoked to further extend the assignment: an unassigned variable is heuristically selected and assigned a value. This new assignment may then trigger further implications. Interleaving these two mechanisms, the partial assignment is extended until the CNF formula has reached an assigned value.

Once the CNF formula has a value, if that value is 1 then a satisfying value has been reached and the procedure is complete. If a partial assignment gives the CNF formula a value of 0, then GRASP engages in a process of conflict analysis. One or more sub-assignments are found for which pure implication propagation would still give a 0 value to the CNF. For each of these conflicting sub-assignments, a new clause, called a conflict clause, is added to the CNF. This conflict clause is constructed to contain literals for just the variables assigned in the sub-assignment, and to evaluate to false for just the variable values of the sub-assignment. Once the conflict clauses are constructed, the partial assignment is contracted to return the CNF to an unassigned value, and the search procedure returns back to the extension mechanisms. Conflict analysis determines how much contraction is required to return to the narrowed space of possibly satisfying assignments. GRASP is capable of *non-chronological* backtracking, contracting back through multiple decision layers. See [7] for details.

With conflict clauses added to the CNF, the implication mechanism will steer the search process away from partial assignments already found to yield 0 values for the CNF. As the procedure iterates between extension and contraction, more and more conflict clauses are added to the CNF. Eventually either the search will be steered to a satisfying assignment, or enough conflict clauses will be added that pure implication yields a conflict when starting from the empty assignment, in which case unsatisfiability of the CNF can be concluded.

The various mechanisms of GRASP can be broken into two layers: a formula level that extends and contracts partial assignments and performs conflict analysis; and a clause level that determines whether, for a given clause, a partial assignment induces a conflict, i.e. a 0 value for the clause, or an implication, i.e. further variable assignments required for the clause to have value 1. The CHAFF [8] SAT solver maintained this structure while improving on details at both levels. One of the key advantages of CHAFF is a clause level *watcher* mechanism for detecting implications. At the formula level, CHAFF provided new mechanisms such as random restart.

Our new approach preserves the formula level mechanisms of GRASP intact, but extends the clause level to support a heterogeneous set of clauses.

3. BDDS AS CLAUSES

Our basic approach is to extend CNF so that the set of clauses includes types of clauses beyond just simple disjunctions of literals. Conceptually, a clause could be any boolean formula at all. The entire boolean formula whose satisfiability is to be determined is then a conjunction of such various types of clauses. To fit within the GRASP framework, efficient mechanisms are required for each type of clause to determine whether a given partial assignment results in a conflict or implication for that clause.

Several advantages can potentially be gained from thus extending CNF. Under some circumstances, part of a larger Boolean formula might be hidden in a way that makes representation in CNF difficult, for example part of a design might be provided as an actual physical device. Some functions might be more compactly represented in an alternative form. Generally the introduction of auxiliary variables gives CNF the power to express functions as compactly as most other representations, but these auxiliary variables can interfere with the detection of conflicts and implications. Alternative representations may support more powerful conflict and implication detection.

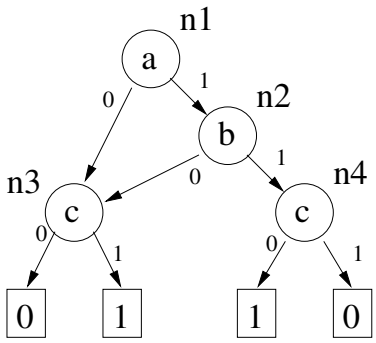

Figure 1: A simple BDD

Note that the conflict clauses that are constructed during the search process continue to be normal CNF disjunctive clauses. Even if the original problem is expressed homogeneously using clauses of some other type, as soon as the search process begins, disjunctive clauses will be introduced and the set of clauses will become heterogeneous.

Here we will explore in particular the use of BDDs for clauses. Before delving into the details of conflict and implication mechanisms for BDDs, we first address the potential advantages we might anticipate for BDD clauses. One can always convert a BDD to equivalent CNF; the advantages of the BDD approach must be compared to some CNF equivalent. We consider two approaches to converting a BDD to CNF.

- One can convert a BDD to CNF without introducing any auxiliary variables. However, this can result in exponential space explosion for common functions such as exclusive-or. Thus BDDs can provide a much more compact representation than CNF without auxiliary variables.

- By introducing auxiliary variables, a BDD can easily be converted to CNF whose size is proportional to that of the BDD. A BDD can be viewed as a circuit of multiplexors. An auxiliary variable can be introduced for each BDD node, and for each node a collection of clauses defining a multiplexor added to the CNF. Thus CNF can be as compact as BDDs. However, BDDs can provide more powerful conflict and implication mechanisms than the corresponding CNF clauses.

To see how auxiliary variables can interfere with detection of implications, examine Fig. 1. Introducing auxiliary variables n_1, n_2, n_3, n_4 for the nodes, one can convert this BDD

to the CNF

$$(a + n_3)(\overline{a} + n_2)$$
$$(\overline{n_2} + b + n_3)(n_2 + b + \overline{n_3})(\overline{n_2} + \overline{b} + n_4)(n_2 + \overline{b} + \overline{n_4})$$
$$(\overline{n_3} + c)(n_3 + \overline{c})(n_4 + c)(\overline{n_4} + \overline{c})$$

Consider the partial assignment $b = 0$. Every clause here will then be either satisfied or will still have at least two unassigned variables, therefore no implications are possible. We will show below, however, how the implication $c = 1$ can be detected using the BDD in Fig. 1. Thus BDDs can support more powerful implication mechanisms than simply derived equivalent CNF. Of course, CNF can in general be supplemented by additional clauses that do not change the function represented but which support more direct implications - this is exactly what conflict clauses do. Eventually enough clauses could be accumulated to support the same implications as a BDD can. The advantage of a BDD is simply that it supports detection of conflicts and implications directly, without supplementary accumulation of information.

It should be no surprise that a BDD can support more powerful detection of conflicts and implications than equivalent CNF generally can. If one converts CNF into an equivalent BDD, then satisfiability can be checked in constant time, and satisfying solutions (if they exist) constructed in linear time. The difficulty lies in constructing the BDD, which runs into space explosion problems often enough to demand alternative methods. CNF is a flexible representation that can efficiently encode a wide range of problems, but which supports only weak mechanisms for detecting conflicts and implications. A BDD is a canonical representation that supports very powerful conflict and implication detection mechanisms, but which too often demands infeasible resources to construct. By extending CNF to include BDDs as clauses, we can take advantage of the power of BDDs while avoiding their potential for space explosion.

3.1 Implication and Conflict Detection

The set of satisfying assignments of a boolean function maps onto the set of paths from the root node to the 1 terminal of the BDD representing that function. Given some arbitrary partial assignment, there will be some subset, possibly empty, of the satisfying assignments compatible with that partial assignment, and correspondingly some subset of the paths in the BDD. Our mechanism for detecting conflicts and implications in BDDs works by computing the subset of paths compatible with a partial assignment.

- A conflict is detected if no path to the 1 terminal exists that is compatible with the partial assignment.

- A given partial assignment may potentially imply values for any subset of the unassigned variables. Let v denote some unassigned variable. If every path to the 1 terminal compatible with the partial assignment passes through a BDD node labeled by v and along the $v = 0$ edge of that node, then the value $v = 0$ is implied by that partial assignment. Similarly, if every compatible path traverses a $v = 1$ edge, then $v = 1$ is implied. If, however, any compatible path does not include a v node, or if some compatible paths traverse $v = 0$ edges while others traverse $v = 1$ edges, then the partial assignment does not imply any value for v.

Detecting implications is the more comprehensive of these two mechanisms, so we focus on that. A BDD node is on a compatible path from the root to the 1 terminal if there is a compatible path from the node to the root and also a compatible path from the node to the 1 terminal. A simple algorithm for marking which nodes and edges are on compatible paths from root to 1 terminal is therefore:

1. Traverse the BDD in topological order from root to terminals, marking every node that can be reached by a compatible path from the root. More concretely, let n be some node labeled by variable v, and n_0, n_1 be the nodes referenced by the outgoing edges of n. Initially mark the root node. Then when a node n is reached during traversal, if n is marked, mark either both n_0 and n_1 (when v is unassigned), or n_a (when v has been assigned the value a).

   If the 1 terminal remains unmarked at the end of this first traversal, then a conflict has been detected.

2. Traverse the BDD in the reverse topological order, from the terminals to the root. Start by unmarking the 0 terminal. When a marked node n is reached, if neither n_0 nor n_1 are marked (when v is unassigned), or if n_a is not marked (when v has been assigned the value a), then unmark n.

During the second traversal one can accumulate counts for each variable v of how many 0 edges, 1 edges, and skipping edges are on compatible paths, and thus whether a value is implied for v. This algorithm is derived from the *weight* and *walk* procedures of Yuan et al. [11]

We can improve the efficiency of the overall procedure by exploiting the fact that during the search for a satisfying solution to the full extended CNF formula, the partial assignment changes incrementally. Thus an event-driven approach can eliminate redundant computation. Our event-driven conflict and implication detection mechanism adds the following data to each BDD node:

- List of incoming edges.
- Count of how many incoming edges are on compatible paths to the root.
- Count of how many outgoing edges are on compatible paths to the 1 terminal.

A node is considered marked, i.e. on a compatible path from root to 1 terminal, if both the incoming and the outgoing counts are positive.

For each variable, we maintain a list of nodes labeled by that variable together with counts for how many 0 edges and how many 1 edges are currently on compatible paths between the root node and the 1 terminal. When a edge on a compatible path skips over a variable v, going from a node with a label less than v in the BDD order to a node with a label greater than v, we count that as both a 0 edge and a 1 edge for v, thus blocking the detection of any implication for v.

We can incrementally update this data as the partial assignment evolves. The update operations required are those to assign a variable, extending the partial assignment, and to unassign a variable, contracting the partial assignment. Assigning a variable can decrement the compatible outgoing edge counts for each node n labeled by that variable,

as well as potentially its incoming nodes recursively up to the root. Assigning a variable can also decrement the compatible incoming edge counts for the outgoing nodes of n recursively down to the 1 terminal. Conversely, unassigning a variable can increment the outgoing edge counts recursively up to the root, and the incoming edge counts down to the 1 terminal. As these edge counts are incremented and decremented, the corresponding 0 and 1 edge counts for each variable are maintained. An implication is detected whenever an unassigned variable has only one of its edge counts positive.

3.2 Conflict Analysis

As the search for a satisfying solution progresses, conflict clauses of the normal CNF disjunctive form are constructed to record where the possibility of such solutions has been eliminated. Conflict clauses steer the search process toward satisfying assignments by blocking off those infeasible parts of the search space. The smaller the conflict clause, the larger the subspace blocked off. A primary objective of conflict analysis is to generate conflict clauses as small as possible.

Conflict clauses are derived from an implication graph whose elements are the conflicts and implications detected at the clause level. When a partial assignment causes an implication to be detected at a clause, all of the assigned variables in that clause could be considered to have caused the implication. With normal disjunctive clauses no smaller set of assignments could have caused that implication, since implication only occurs for the last unassigned variable. With BDD clauses, however, implications are possible with arbitrary subsets of the variables assigned. If a smaller set of assignments can be recognized to have caused a conflict, the resulting conflict clause will be smaller. We use a simple greedy approach to find a minimal partial assignment causing an implication when performing conflict analysis and constructing conflict clauses.

4. MEASURING EFFECTIVENESS

To see how well BDD clauses work, we chose 22 SAT problems expressed in normal CNF with purely disjunctive clauses. We then translated these to extended CNF that includes BDD clauses. We could then compare a conventional SAT solver working with the normal CNF against our new SAT solver working with BDD clauses. Our translation scheme involves three steps:

1. Each disjunctive clause is converted to a simple BDD.
2. The set of resulting simple BDDs is partitioned. A single more complex BDD is built for each block of this partition, by conjoining all of the simple BDDs in the block.
3. Whenever a variable occurs in only a single complex BDD, that variable can be existentially quantified from the BDD and removed from the problem, preserving the (un-)satisfiability of the problem.

Our translator actually interleaves steps 2 and 3, greedily trying to eliminate as many variables as possible. This approach is an adaptation of the *VarScore* method of Chauhan et al. [5]. The coarseness of the clause partitioning is controlled by a simple BDD size threshold parameter. This lets us easily explore the tradeoff between static BDD computation and dynamic clause-based search.

		Original CNF		BDD Thrshd 100	
name	res	vars	clauses	vars	clauses
X1	U	9118	26484	1028	1108
X2	U	4390	12741	725	951
X3	U	4179	13667	809	1667
X4	U	2583	7410	540	702
X5	U	1784	5269	331	243
X6	S	11828	33928	3013	3827
X7	S	4566	13987	957	1545
X8	U	4672	12810	815	960
X9	U	5765	17886	1324	2166
X10	U	19665	65263	4916	9160
X11	U	3331	9296	605	893
X12	U	106	496	49	104
ssa7552-038	S	1493	3567	73	107
ii32e4	S	387	7106	219	1699
hanoi4	S	718	4934	358	2112
par16-1	S	1015	3310	174	119
bf0432-007	U	1028	3656	396	693
bf2670-001	U	1387	3428	98	54
bmc-ibm-1	S	9685	55855	3589	23175
bw_large.a	S	459	4675	382	3612
3blocks	S	283	9690	273	9392
hole8	U	72	297	60	181

Table 1: Problem Characteristics

Table 1 shows characteristics of the problems we used for our measurements.

name The problems with names with an initial "X" are derived from a variety of industrial problems. The other problems are standard benchmark problems.

res "U" denotes unsatisfiable problems, "S" denotes satisfiable.

Original CNF vars All these problems are initially specified as CNF. This column gives the number of variables in the original CNF.

Original CNF clauses This column gives the number of clauses in the original CNF.

BDD Thrshd 100 vars As discussed above, after blocks of CNF clauses have been conjoined, some variables can be eliminated through existential quantification. This column shows the number of variables remaining when the clause partitioning is controlled by a BDD size threshold of 100. Sometimes a large fraction of variables could be eliminated, e.g. in ssa7552-038, while in other cases only a small fraction, e.g. 3blocks.

BDD Thrshd 100 clauses This column shows the number of BDDs in the final formula used for search in the threshold 100 case, i.e. the number of blocks in the partition of the original CNF clauses. The reduction in the number of clauses tend to track with the reduction in the number of variables.

Table 2 shows cpu times for each problem for each of three different procedures. These times were all measured on an Intel XEON processor with clock speed 2.2GHz, running the LINUX operating system.

name	limmat	BDD	thrshd 1	thrshd 100	
			sat	a-s	sat
X1	119	8	59	20	18
X2	10	2	47	5	32
X3	1674	2	2140	4	16
X4	0	1	2	1	2
X5	58	1	79	1	57
X6	1049	12	29	38	2
X7	0	2	2	5	0
X8	1	2	7	4	3
X9	5	4	15	9	33
X10	>3600	34	332	118	467
X11	0	2	2	2	0
X12	38	0	96	0	37
ssa7552-038	0	0	0	0	0
ii32e4	0	2	1	0	0
hanoi4	17	0	16	0	32
par16-1	3	0	21	0	10
bf0432-007	0	0	1	0	0
bf2670-001	0	0	0	0	0
bmc-ibm-1	2	15	45	33	14
bw_large.a	0	0	0	0	0
3blocks	0	1	3	0	1
hole8	1	0	0	0	22
Totals	> 6577	88	2897	240	746

Table 2: Run times in cpu seconds

limmat We used Biere's limmat [2] SAT solver on each problem, as a basis for comparison of our new method against current state of the art technology. This column shows the run time required for limmat to solve each problem. We aborted one run because of excessive cpu time.

BDD This column shows the time to build simple BDDs for each of the clauses.

thrshd 1 sat We ran our solver for each problem directly on the simple BDDs which directly correspond to the original CNF clauses. The clause-level conflict and implication analysis in this case will correspond exactly to a conventional CNF SAT solver. These results show the effectiveness of our formula-level procedures such as conflict clause construction, etc. The total run time for this mode of operation is the sum of this column, labeled "sat", with the column labeled "BDD".

thrshd 100 a-s For each problem we also ran our solver using complex BDDs constructed from a partition of the original CNF clauses using a BDD size threshold of 100. This column gives the time to construct the complex BDDs from the simple BDDs, using the BDD and-smooth operation.

thrshd 100 sat This column gives the time to determine whether the implicit conjunction of complex BDDs has a satisfying assignment. The total run time for each problem is the sum of the times to construct the simple BDDs (column labeled BDD), to combine the simple BDDs to form complex BDDs (column labeled thrshd 100 a-s) and the time recorded in this column.

name	limmat	thrshd 1	thrshd 100
X1	669859	327073	101146
X2	26576	33190	34557
X3	254264	557913	14103
X4	3010	3206	6357
X5	99874	88938	140172
X6	890109	53463	2419
X7	1243	783	210
X8	69838	79246	29018
X9	91180	61450	128172
X10	>1906851	673907	2476695
X11	1892	2676	836
X12	33173	68283	48219
ssa7552-038	153	156	23
ii32e4	37	112	39
hanoi4	29500	19009	35943
par16-1	9309	12135	16613
bf0432-007	1835	1328	1129
bf2670-001	87	102	40
bmc-ibm-1	8564	47076	8776
bw_large.a	49	46	36
3blocks	1662	1036	288
hole8	8151	3564	38472

Table 3: Number of decisions

Our main contribution here is to show how BDDs can support powerful conflict and implication detection mechanisms. A state of the art SAT solver requires additional mechanisms as well as careful engineering of implementation details. The comparison between our tool running with simple BDDs, at threshold 1, and with complex BDDs, at threshold 100, isolates most effectively the effect of using the more powerful detection mechanisms available with BDDs. In 14 of 22 problems, the search process was faster using the complex BDDs than using the simple BDDs which are equivalent to the original disjunctive CNF clauses. In 4 problems their was no measurable difference in run-times, and in 4 problems the complex BDD approach was slower.

Table 3 shows the number of decisions required for each run of each problem. This more precise measurement provides some visibility into behavior of the smaller problems. In 3 of the 4 problems where cpu times could not be distinguished, fewer decisions were required when complex BDDs were used. In several cases complex BDDs took more decisions even though the search CPU time was smaller. In these cases, referring to Table 1 one can see that combining the simple BDDs into complex BDDs allowed many variables and clauses to be eliminated. This simplification of the problem was also reflected in implication counts, another measure of the work required by a SAT solver. For example, par16-1 required 21 million implications with simple BDDs and only 6 million with complex BDDs. For comparison, limmat reported using 15 million implications on this problem.

Table 4 shows the results when the BDD threshold is varied for the single problem X3. Eventually of course if the BDD threshold is set high enough, all the clauses will be merged into a single BDD and the SAT problem solved using BDD operations alone. The disadvantage of a pure BDD approach is that the BDD operations can be quite expensive. This can be seen in the row labeled "and-smooth cpu sec":

threshold	100	1000	10000	100000
variables	809	665	566	529
clauses	1667	1087	882	790
and-smooth cpu sec	4	16	244	885
sat cpu sec	16	16	157	771
decisions	14103	2530	2379	2477

Table 4: Varying Partition Threshold for Problem X3

as the BDD threshold is increased, the time required for BDD operations quickly grows. The numbers of clauses and variables continues to decline as the threshold is increased, but not very quickly. The most interesting thing to observe in this table is that as the BDDs grow more complex, the search cpu time goes up while the number of decisions stays relatively flat. This is apparently caused by the cost of the conflict and implication detection mechanisms, which grows as the BDD sizes grow.

5. RELATED WORK

Many approaches to combining the strengths of SAT solvers and BDDs have been explored. Most [9, 10] involve performing BDD operations, i.e. creating new BDD nodes, during the search procedure. Our approach certainly requires BDD operations to construct the initial problem, but during the search procedure no nodes are created or destroyed. The search procedure involve only walking existing BDDs and updating data associated with existing nodes and variables.

The work closest to ours appears to be that of Burch and Singhal [4]. A minor difference is that, rather than working with a conjunction of BDDs, they work with a composition of BDDs. The major difference is that their formula level procedure is WALKSAT rather than GRASP, so they are always working with complete rather than partial assignments, and do not detect implications or construct conflict clauses.

Another very closely related paper is that of Gupta and Ashar [6]. They divide their problem up into a SAT-CNF part and a BDD part. They do work with partial assignments and describe an efficient conflict detection mechanism for BDDs. However they do not detect implications from BDDs. Since they only work with a single BDD, they do not provide or need conflict analysis to coordinate the simultaneous search for solutions across multiple BDDs. Working with a single BDD severely limits the benefit that introducing BDDs can provide.

6. FUTURE WORK

Our current plans are focussed on improving the efficiency of the conflict and implication detection mechanism. Just as CHAFF improved upon GRASP by moving from a count-based mechanism to a watcher-based mechanism for detecting implications in disjunctive clauses, we plan to develop a watcher-based mechanism to work with BDDs. This may permit larger BDDs to be used efficiently, thereby providing even more powerful implications.

A secondary direction for research involves refining the initial construction of the BDDs. Currently we take problems expressed in CNF and partition the CNF clauses to form the complex BDDs upon which the search is performed. Our current partitioning mechanism has been crudely adapted from another application. By tuning the partitioner to the present application, we should be able to create complex BDDs that support more efficient search. It may also be advantageous to cluster and conjoin conflict clauses on the fly to form additional complex BDDs. Beyond that, most industrial problems are not initially specified as CNF. Translation directly to an implicit conjunction of BDDs, by-passing CNF, should enable us to leverage more of the natural structure of these problems.

7. CONCLUSION

We have showed how to build a BDD conflict and implication detection mechanism and incorporate it into a GRASP-like SAT engine. Measurements on industrial examples show that the more powerful detection mechanism provided by BDD clauses can improve the efficiency of the search procedure in many cases. We expect that this approach will prove even more advantageous on problems originally posed in a form more conducive to BDD construction than CNF.

8. REFERENCES

[1] A. R. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. PhD thesis, Univ. Oregon, 1995.

[2] A. Biere. www.inf.ethz.ch/personal/biere/projects/limmat/.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 570–576, 1998.

[5] P. Chauhan, E. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith, and D. Wang. Non-linear quantification scheduling in image computation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 293–298, 2001.

[6] A. Gupta and P. Ashar. Integrating a boolean satisfiability checker and bdds for combinational equivalence checking. In *Proc. Int'l Conf. on VLSI Design*, pages 222–225, 1997.

[7] J. P. Marques Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 220–227, 1996.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the Design Automation Conf.*, pages 530–535, 2001.

[9] R. Mukherjee, J. Jain, and D. Pradhan. Functional learning: A new approach to learning in digital circuits. In *Proc. VLSI Test Symposium*, pages 122–127, 1994.

[10] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Proc. Intl. Conf. on Computer Design*, pages 459–464, 2000.

[11] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 584–589, Nov. 1999.