BerkMin: a Fast and Robust Sat-Solver

Evgueni Goldberg
Cadence Berkeley Labs(USA),
Email: egold@cadence.com

Yakov Novikov
Academy of Sciences (Belarus),
Email: nov@newman.bas-net.by

Abstract

We describe a SAT-solver, BerkMin, that inherits such features of GRASP, SATO, and Chaff as clause recording, fast BCP, restarts, and conflict clause "aging". At the same time BerkMin introduces a new decision making procedure and a new method of clause database management. We experimentally compare BerkMin with Chaff, the leader among SAT-solvers used in the EDA domain. Experiments show that our solver is more robust than Chaff. BerkMin solved all the instances we used in experiments including very large CNFs from a microprocessor verification benchmark suite. On the other hand, Chaff was not able to complete some instances even with the timeout limit of 16 hours.

1. Introduction

Given a conjunctive normal form (CNF) F specified on a set of variables $\{x_1,\ldots,x_n\}$, the satisfiability problem is to satisfy (set to 1) all the disjunctions of F by some assignment of values to variables from $\{x_1,\ldots,x_n\}$. A disjunction of F is also called a clause of F. Many problems such as ATPG [16], logic synthesis [5], equivalence checking [6,9], and model checking [4] reduce to the satisfiability problem.

In the last decade substantial progress has been made in the development of practical SAT algorithms [1,2,8,11,12,13,15,18]. All of them are search algorithms that aim at finding a satisfying assignment by variable splitting. Search algorithms of that kind are descendants of the DPLL-algorithm [7].

DPLL-algorithm can be considered as a special case of general resolution which is called tree-like resolution. It was shown in [3] that there is exponential gap between the performance of tree-like resolution and that of general resolution.

Modern SAT-solvers have made at least two steps towards general resolution trying to eliminate the drawbacks and limitations of pure tree-like resolution. First, they record so called conflict clauses [15], which are implicates of the original CNF. Adding conflict clauses allows one to prune many of the branches of the search tree that are yet to be examined [1,13,15,18]. The deduced implicates are just added to the current CNF which we will also refer to as (clause) database.

Second, some of the state-of-the-art SAT-solvers use the strategy of restarts when the SAT-solver abandons the current search tree (without completing it) and starts a new one. So instead of one complete search tree the SAT-solver constructs a set of incomplete (except the last one) trees. In [1,10] the usefulness of restarts was proven experimentally. Restarts are effectively used in Chaff [13].

We introduce a new SAT solver called BerkMin (BerkMin stands for Berkeley-Minsk, the cities where the authors live). BerkMin can be considered as the next representative of the family of SAT-solvers that includes GRASP [15], SATO [18], Chaff [13]. BerkMin uses the procedures of conflict analysis and non-chronological backtracking introduced in GRASP, fast BCP suggested in SATO, and Chaff's idea of reducing the contribution of "aged" conflict clauses into decision making. Besides, BerkMin uses restarts.

At the same time BerkMin introduces many new features into decision making and clause database management. First, the set of conflict clauses is organized as a chronologically ordered stack (the top clause is the one deduced the last). If in the current node of the search tree there are unsatisfied conflict clauses, the next branching variable is chosen among the free variables, whose literals are in the top unsatisfied conflict clause. Second, we introduce a heuristic to pick which out of two possible assignments to the chosen branching variable should be examined first. In the case of using a single search tree, spending time on the selection of the branch to be examined first makes sense only for satisfiable CNFs. (For unsatisfiable CNFs both branches are "symmetric" i.e. the search tree size is not affected by whichever branch is examined first.) However, when using restarts the symmetry between the two alternative branches is broken even for unsatisfiable CNFs.

Third, our procedure of computing the activity of variables is different from that of Chaff. The activity of variables in conflict making is used by Chaff to single out good candidates for branching variables. For computing the activity of a variable x Chaff counts the number of occurrences of x in conflict clauses. This may lead to overlooking some variables that do not appear in conflict clauses while actively contributing to conflicts (e.g. if these variables are deduced). BerkMin solves this problem by taking into account a wider set of clauses involved in conflict making. Fourth, we use a new procedure of clause database management performed after the current search tree is abandoned. The novelty of the procedure is that the decision whether a database clause should be removed is based not only on its size (the number of literals). It is also based on the "activity" of this clause in conflict making and its "age".

We experimentally compare the performance of BerkMin with that of Chaff that is currently considered as the best SAT-solver used in the EDA domain. Experiments clearly show that BerkMin is more robust than Chaff. By greater robustness of BerkMin we mean that it is able to solve more instances than Chaff in a reasonable amount of time. Though Chaff is faster on some instances, BerkMin does not have problems solving them. On the other hand, we give examples of CNFs that are relatively easy for BerkMin and that cannot be solved by Chaff even with the timeout limit of 16 hours.

In particular, BerkMin solved all the instances of the benchmark suite fvp-unsat2.0 [17] in less than two hours. This class consists of large unsatisfiable CNFs (except one instance) describing verification of pipelined microprocessors. The proof of unsatisfiability of the instance pipe6.cnf from this suite by the best SAT-solver takes 60 hours of CPU time on a SUN computer with clock frequency 336MHz [17]. BerkMin solved this instance in less than 20 minutes on a SUN computer with clock frequency 450 MHz. A more complex instance pipe7.cnf that no SAT-solver has been able to complete was solved by BerkMin in 1 hour.

2. Basic notions

Given a CNF F, a DPLL-algorithm based SAT-solver looks for a satisfying assignment that is also called a solution to the satisfiability problem. Search is organized as a binary tree. If no solution is found after completing search tree examination, then F is unsatisfiable. At each node of the search tree the following three steps are performed: a) choosing the next variable to split on and selecting the value, 0 or 1, to be first assigned to the chosen variable; b) running the ***Boolean Constraint Propagation*** (BCP) procedure; c) performing conflict analysis and backtracking if a conflict is encountered.

The choice of the next branching (splitting) variable is usually based on a heuristic. Different heuristics are used for different classes of CNFs (see, for example, [1,8,11,12,13,15,18]). The heuristics used by BerkMin are described in Section 4.

After choosing a branching variable (say variable y) and its first value, the ***BCP procedure*** is initiated. It performs as follows. Suppose that in the first branch value 1 is assigned to variable y. Then all the clauses having the positive literal of y are satisfied and so removed from the current CNF. From all the clauses having the negative literal of y, this literal is removed. This may result in producing ***unit clauses*** i.e. clauses having only one literal left. Suppose that $\sim x$ is such a unit clause. This clause can be satisfied only by the assignment x=0. This assignment is called ***deduced*** from clause $\sim x$. After making the deduced assignment (in our case x=0) the procedure of discarding satisfied clauses and removing literals that are set to 0 (in our case positive literals of x) is performed again. This may lead to producing new unit clauses. The BCP procedure is performed until 1) a solution if found or 2) all clauses of the current CNF contain at least two literals or 3) an empty clause is produced (this situation is called a conflict). Before producing an empty clause two unit clauses having the opposite literals of the same variable must appear, for example z and ~z. Deducing the value of z from either clause makes the other clause unsatisfiable. The efficiency of the BCP procedure can be improved by using the technique suggested in SATO [18] that was also incorporated into Chaff [13]. We omit the description of this technique. In BerkMin, our own version of SATO's BCP procedure is implemented.

Suppose that BCP has been completed and no solution is found in the current node. If no conflict is encountered then the algorithm moves away from the root to the next node. If there is a conflict, the algorithm backtracks. Suppose that in the last chosen assignment leading to a conflict, x was equal to 0. The simplest backtracking is chronological: assignment x=0 is undone and all deduced assignments obtained by the BCP procedure, initiated by x=0, are undone as well. The aim of backtracking is to restore the CNF as it was before making the assignment x=0. If assignment x=1 has not been examined yet then it is made now and the BCP procedure is initiated. Otherwise, the algorithm backtracks to the first node of the search tree where the alternative assignment has not been examined yet. However, conflict analysis sometimes allows to backtrack non-chronologically, skipping a set of nodes of the search tree. The technique of conflict analysis and non-chronological backtracking is given in detail in [15]. In Section 3 we give a short description of this technique.

3. Non-chronological backtracking and conflict analysis

The notion of conflict analysis is crucial for understanding non-chronological backtracking. Let F be a CNF consisting of the set of clauses of the initial CNF F^* and implicates of F^* deduced during search. Let C be a clause of F. Suppose that a conflict is caused by clause $C=\sim a \vee x \vee \sim c$ being unsatisfiable. That is, variables a,x,c have been assigned a=1, x=0, c=1 which resulted in removing all literals from clause C. A set R of value assignments to variables of F is called an assignment leading to a conflict on C (or a ***conflict assignment*** for short) if after making the assignments from R and running the BCP procedure, clause C becomes unsatisfiable.

A trivial assignment leading to a conflict on C=~a∨x∨~c is the set {a=1,x=0,c=1} since after making these value assignments all literals of C are removed. However, this conflict assignment is of no interest because it does not provide any new information. Suppose, that we found out that R={x=0, y=1, z=1} is also an assignment leading to a conflict on C. Then clause x∨~y∨~z specified by set R is an implicate of the current CNF F and can be added to F. Clause x∨~y∨~z recording conflict assignment R={x=0, y=1, z=1} is called a ***conflict clause***.

Suppose that assignment x=0 of R={x=0, y=1, z=1} is chosen (or deduced) in the current node. Let us assume that values y=1 and z=1 are deduced at nodes v' and v'' respectively. Suppose also that v > v' > v'' where $n_1 > n_2$ means that node n_2 is closer to the root than n_1. After adding conflict clause x∨~y∨~z to the current CNF we can back-jump from node v right to node v' (skipping the nodes that may lie between v, v'). The point is that current CNF $F_{v'}$ at node v' is different from the one obtained at node v' before, because clause x∨~y∨~z has been added to the database. Since variables y and z are assigned value 1 at nodes v' and v'', then at node v' clause x∨~y∨~z of CNF $F_{v'}$ is just equal to x and so assignment x=1 can be deduced from it. Before adding clause x∨~y∨~z this deduction was not possible. With this clause in the current CNF we can back-jump to node v' to make this deduction.

Conflict analysis and conflict clause derivation are described in [15]. We sketch here the main idea. Suppose that at the current node v clause ~a∨x∨~c becomes unsatisfiable. We need to find a conflict assignment R that includes *only one* assignment "initiated" at node v. (This is either an assignment made to the branching variable of node v or an assignment deduced in the BCP procedure performed at node v.) The rest of the assignments of R are made at nodes located above node v (i.e. closer to the root).

Such a conflict assignment can be constructed from the trivial conflict assignment R_1={a=1, x=0, c=1} by performing the BCP procedure "backwards". For example, suppose that all the value assignments of R_1 were made at node v and assignment a=1 was deduced from clause a∨x∨~z (due to assignments x=0, z=1 made before). Now R_1 can be replaced with a non-trivial conflict assignment R_2={x=0, c=1, z=1}. (This conflict assignment is obtained from R_1 by replacing assignment a=1 with x=0, z=1.) Suppose that assignment z=1 was deduced at node v' located above v. Then conflict assignment R_2 contains only two assignments (x=0 and c=1) that were made at node v. Proceeding with the "reverse" BCP procedure we will eventually produce a conflict assignment containing only one assignment made at node v. Suppose, for example, that assignment c=1 was deduced from clause c∨~y∨~z and assignment y=1 was made at node v'' located above v. By replacing c=1 with y=1, z=1 we obtain from R_2 conflict assignment R_3={x=0, y=1, z=1}. R_3 satisfies our requirement because only one assignment of R_3, namely x=0, was made at node v. The corresponding conflict clause is x∨~y∨~z.

Formally, the deduction of clause x∨~y∨~z can be described by the following chain of resolutions: (~a∨x∨~c) ∧ (a∨x∨~z) ⇒ x∨~c∨~z , (x∨~c∨~z) ∧ (c∨~y∨~z) ⇒ x∨~y∨~z. Only the final result of this chain (in our case clause x∨~y∨~z) is added to the current CNF. The intermediate conflict assignments (like clause x∨~c∨~z) are thrown away. The clauses of the current CNF that are used in the deduction of the final conflict clause will be called ***clauses responsible for the conflict***. In our example, clauses ~a∨x∨~c , a∨x∨~z , c∨~y∨~z are responsible for the conflict. Clauses responsible for a conflict are identified during the reverse BCP procedure used to construct the conflict clause to be added to the database.

4. BerkMin's decision making

When designing the decision making strategy of BerkMin we tried to take into account the following two facts. 1) Current SAT-solvers can quickly solve fairly large "real-life" CNFs. This suggests that in such CNFs short implicates can be deduced from a small subset of clauses by branching on a small subset of variables. 2) The set of variables responsible for conflicts may change very quickly. Suppose, for example, that a variable x of the CNF to be tested for satisfiability describes the output of a gate feeding many AND gates. Then in the branch x=0 the variables corresponding to the rest of the inputs of these AND gates are not involved in conflict making. However, in the branch x=1 all these variables immediately become "active" again.

The two facts above imply that the choice of branching variables for "real-life" CNFs should be very dynamic. It must quickly adapt to all the changes of the set of relevant variables caused by new value assignments. Chaff's idea of conflict clause "aging" is a significant contribution to creating such a dynamic decision making strategy.

Here is how this idea is implemented in BerkMin. Each variable is assigned a counter (denote it by $ac(z)$) that stores the number of clauses, responsible for at least one conflict, that have a literal of z. The value of $ac(z)$ is updated during the reverse BCP procedure. As soon as a new clause responsible for the current conflict is encountered, the counters of the variables, whose literals are in this clause, are incremented by 1. The values of all counters are periodically divided by a small constant greater than 1. (This constant is equal to 2 for Chaff and 4 for BerkMin.) This way the influence of "aged" clauses is decreased and preference is given to recently deduced clauses.

In Chaff, only literals of the conflict clauses added to the current CNF are taken into account when computing the "activity" of a variable *z*. (So if a literal of *z* occurs in a clause responsible for the current conflict but *z* is not in the conflict clause, Chaff does not change the value of the activity of *z*.) Taking into account a wider set of clauses responsible for conflicts allows BerkMin to get a more accurate estimate of the activity of variables. If a variable *z* is frequently assigned through deduction it may drive many conflicts without appearing in conflict clauses. So if, when computing the activity of *z*, one takes into account only conflict clauses, this activity will be underestimated.

Currently, BerkMin has two decision making procedures. The first procedure is to pick the variable *z* with the maximum value of *ac*(*z*) as the next branching variable. However, it is BerkMin's second choice. The main decision making procedure (that was used in all experiments) is based on chronological conflict clause ordering.

The reason for using such an ordering is that we believe that Chaff's strategy is not "dynamic" enough. In particular, it cannot adjust quickly to changes of the set of relevant variables mentioned in the beginning of the section. Consider the following example. Suppose that Chaff divides counters by 2 , say, every 100 conflicts. Suppose that immediately after dividing the counters, a change of relevant variables occurred. Then for the next 20-30 conflicts the choice of branching variables will be dominated by an "obsolete" set of active variables.

Here is how the problem is solved in BerkMin. The clause database is organized as a stack. The clauses of the initial CNF are located at the bottom of the stack and each new conflict clause is added to the top of the stack. In the process of assigning values some clauses of the stack get satisfied. However there is always an unsatisfied clause that is the closest to the top of the stack. We will call this clause the ***current top clause***. If the current top clause is a conflict clause then it is just the most recently deduced conflict clause that is not satisfied yet.

Let *C* be the current top clause. The idea is to choose the next branching variable among free (unassigned yet) variables of *C*, if it is a conflict clause. Among the variables of *C*, the variable *z* with the largest value of *ac*(*z*) is selected. Note, that variable *z* may look very "passive" in the set of all the currently free variables. However, the activity of these variables may be the result of a very different set of conflicts that happened before deducing *C* or would occur after the deduction of *C*. These conflicts may involve quite different sets of variables. Choosing *z* we take into account the fact that these active variables may be irrelevant to the conflict which had led to the deduction of *C* and to "similar" conflicts i.e. the ones involving sets of variables that are close to that of *C*.

Let ~*z* be the literal of variable *z* that is in the top clause *C*. Suppose that *z* is chosen as the next branching variable. BerkMin does not try to satisfy *C* immediately by assigning *z*=0. It has a separate procedure for choosing the branch to be examined first. This procedure is further on. However, it is not hard to see that clause *C* will become satisfied after no more than *n*-1 assignments to branching variables where *n* is the number of literals of *C*. Indeed, if *C* is not satisfied by the chosen assignment to *z* and no conflict is produced during the following BCP procedure, *C* remains the current top clause. Then a new free variable is selected from the ones whose literals are in *C*. So eventually either *C* will be satisfied by an assignment to a branching variable (or by deduced assignment) or after removing from *C* all literals but 1 it will be satisfied during the BCP procedure.

If the current top clause is a clause of the original CNF then all the unsatisfied clauses in the stack are also clauses of the original CNF. In this case the most active free variable of the current CNF (i.e. free variable *z* with the greatest value of *ac*(*z*)) is selected.

Before describing how the branch to be examined first is selected, we would like to make a few comments. It is obvious that if the initial CNF is satisfiable, there may be asymmetry between the two alternative branches regardless of whether or not the used algorithm has restarts. However, if the initial CNF is unsatisfiable and the algorithm builds a single search tree, the two alternative branches are symmetric. That is, the search tree size is not affected by the choice of the branch to be examined first.

Indeed, suppose that *x* is the next branching variable and branch *x*=0 is examined first. The size of the subtree built in the alternative branch *x*=1 is, in general, affected by conflict clauses deduced in branch *x*=0. However these conflict clauses cannot contain literal *x* because a clause containing literal *x* is satisfied by *x*=1. So the only kind of conflict clauses that affect computations in branch *x*=1 are the ones that do not contain a literal of *x*. But the conflicts described by such clauses are symmetric in *x* and can be deduced in either branch. So branches *x*=0 and *x*=1 "interact" only through conflict clauses that are symmetric in *x*. Then the size of subtrees constructed in branches *x*=0 and *x*=1 does not depend on which one is examined first. However, in case of using restarts, asymmetry between the alternative branches exists even for unsatisfiable CNFs. The reason is that, when branching on a variable *x*, the algorithm does not commit to examining the alternative branch if no solution was found in the branch examined first.

Let us describe now the procedure used for branch selection. First we consider the case when the current top clause is a conflict one. In this case the choice of the first assignment to a variable x is aimed at having uniform distribution of positive and negative literals in conflict clauses of the database. For this purpose, for each literal l cost function $lit\_ac(l)$ is computed. $lit\_ac(l)$ gives the number of conflict clauses generated so far that contain literal l. Initially $lit\_ac(l)$ is equal to 0. As soon as a conflict clause C is added to the database, for each literal l of C the value of $lit\_ac(l)$ is incremented by 1. The value of $lit\_ac(l)$ is not divided by a constant (as it is done for counters $ac(z)$). Besides, the value of $lit\_ac(l)$ is not recomputed after some conflict clauses are removed from the database according to the rules described in section 5.

If x is the next branching variable, the literal l, $l \in \{x,\sim x\}$ with the largest value of $lit\_ac(l)$ is selected. If $lit\_ac(x) = lit\_ac(\sim x)$, literal l, $l \in \{x,\sim x\}$ is selected at random. Then x is assigned the value setting the chosen literal l to 1. So all clauses having literal l become satisfied and no conflict clause can contain l. On the other hand, the opposite literal (i.e. $\sim l$) is set to 0 and so conflict clauses may include $\sim l$. So, setting the literal with the largest value of $lit\_ac(l)$ to 1, we reduce the gap between the numbers of occurrences of x and $\sim x$ in conflict clauses of the database.

Now we describe how the branch to be examined first is selected when the current top clause is a clause of the original CNF. We will call a clause ***binary*** if it contains only two literals. For each literal l, a cost function $nb\_two(l)$ is computed that approximates the number of binary clauses in the "neighborhood" of literal l. Function $nb\_two(l)$ is computed as follows. First, the number of all binary clauses containing literal l is calculated. Then for each binary clause C containing literal l, the number of binary clauses containing literal $\sim v$ is computed where v is the other literal of C. The sum of all computed numbers gives the value of $nb\_two(l)$. This cost function can be considered as an estimate of the power of BCP performed after setting l to 0. The greater the value of $nb\_two(l)$ is the more assignments will be deduced from the binary clauses containing literal $\sim v$ after setting literal l to 0 in clause C. This is the reason why, given the next branching variable x, the literal l, $l \in \{x,\sim x\}$, with the greatest value of $nb\_two\ (l)$ is selected. If $nb\_two\ (x) = nb\_two\ (\sim x)$ then $l \in \{x,\sim x\}$ is chosen at random. Then x is assigned the value setting literal l to 0. To reduce the amount of time spent on computing $nb\_two\ (l)$ we use a threshold value (in our experiments it was equal to 100). As soon as the value of $nb\_two\ (l)$ exceeds the threshold its computation is stopped. It should be noticed that the idea of taking into account binary clauses in decision making is not new. For example, it is successfully used in SATZ [12] (though cost functions of SATZ are different from ours).

BerkMin uses a relatively complex decision making procedure. However this complexity has been justified in numerous experiments. This fact refutes the claim made in [14] that for SAT-solvers using clause recording the quality of decision making is of no great importance.

5. Clause database management

Before starting the next iteration (i.e. building a new search tree), some clauses are "physically" removed from the database. This allows one to reduce memory used for database allocation. In the process of removing clauses BerkMin's data structures are partially or completely recomputed to fit them into smaller memory blocks.

A fraction of clauses is removed "automatically" due to retaining some value assignments deduced in the last iteration. Namely, all the value assignments, that were deduced from unit conflict clauses (if any) and in the BCP procedure triggered by the assignments deduced from unit conflict clauses, are retained in the new iteration. All the clauses that are satisfied by the retained assignments are removed from the current CNF.

The rest of the clauses to be removed are selected using heuristics described below. Our approach is based on the following hypothesis. More recently deduced clauses are more valuable because it took more time to deduce them from the original set of clauses.

From the view point of clause removal heuristics, the set of conflict clauses is a queue. New conflict clauses are added to the tail of this queue while clauses to be considered for removal are located in its head. In the experiments, from the head part of the queue whose size is 1/16th of the whole queue, all clauses with more than 8 literals were removed. From the rest 15/16 of the queue clauses containing more than 42 literals were removed. However, BerkMin always keeps in the queue the last conflict clause and a fraction of "active" clauses regardless of how many literals they have. The activity of a clause C is measured by the number of conflicts for which C has been responsible so far. In the head part of the queue only clauses with activity value greater than 60 are considered as active. In the rest of the queue active clauses are the ones whose activity is greater than 7. The threshold value of the activity for the clauses of the head part of the queue is increased every 1024 nodes (the starting value is 60). So large clauses that are not used in conflicts any more will be soon removed from the database.

The current search tree is abandoned after generating 550 conflict clauses (that are added to the tail of the queue). Before starting a new search tree BerkMin starts removing clauses trying to get rid of at least 1/16 of the conflict clauses forming the queue. If after applying the rules described above, the number of removed clauses was less than 1/16 of the queue, the threshold on the size of clauses removed from the head part decreased by one literal. (However, after the threshold value reaches 4, it does not decrease any more.) In our experiments it was almost always possible to remove 1/16 of the queue, so the clause database was kept small. It should be noted that the described procedure of clause removal makes BerkMin incomplete because there is a possibility of looping. Indeed, it is possible that the algorithm removes and then deduces the same set of clauses. For example, if all the clauses deduced in the current iteration have more than 42 literals and their activity is less than 7, all of them will be removed from the database. Then after restarting the algorithm, the same set of clauses will be deduced and then removed again and so on.

A simple way of eliminating the possibility of looping is as follows. One conflict clause deduced since the last restart is marked and forever forbidden to be removed from the database (unless it is satisfied by an assignment retained from the previous iteration). Then we guarantee that the number of marked clauses in the database grows monotonically and so no looping is possible. The number of marked clauses to be kept can be reduced *n* times by marking a clause only after performing *n* iterations (restarts). This technique is not hard to implement. However, we have not done that because BerkMin never looped in our experiments.

6. Experimental results

BerkMin was written from scratch in Microsoft's Visual C++ under Windows-95. Then it was ported to Solaris using GNU's C++ compiler gcc. In the experiments, we compared BerkMin with Chaff [13]. The binary of Chaff was downloaded from [19]. Both programs were run on the same SUNW, Ultra-80 system with clock frequency 450MHz and 4Gbytes of memory.

The results of the experiments are shown in Tables 1,2,3. Table 1,2 are mostly self-explanatory. Table 1 contains instances for which BerkMin's and Chaff's performances are comparable. Table 1 gives results on two classes of instances from the Dimacs benchmark suite (hole and par16) that can be downloaded from [20]. All the other Dimacs instances that were used in [13] to estimate Chaff's performance are too easy for both programs. Besides, Table 1 contains results on the class blocksworld, that is also available at [20], and the "easy" classes of CNFs from [17] encoding verification of pipelined microprocessors. The result of the program that had the best runtime on a class of instances is shown in bold. It is worth mentioning that in class fvp-unsat1.0 Chaff is faster only on one instance (9vliw_bp_mc.cnf). All the 100 CNFs of the class vliw-sat1.0 are obtained by modification of 9vliw_bp_mc.cnf. So Chaff has better performance on classes fvp-unsat1.0 and vliw-sat1.0 due to one instance.

Table 2 contains results on more complex classes of CNFs. Class Beijing can be downloaded from [20]. Class Hanoi from the Dimacs benchmark suite consisting of two CNFs hanoi4.cnf and hanoi5.cnf was extended by a more complex example hanoi6.cnf (courtesy of Henry Kautz). Class fvp-unsat2.0, that can be downloaded from [17], consists of large CNFs encoding microprocessor verification. Finally, class Miters consists of CNFs obtained by encoding equivalence checking of artificial combinational circuits (We used artificial circuits because their complexity was easy to control).

Each class of Table 2 contains at least one CNF that Chaff was not able to solve without exceeding the timeout limit (60 000 sec.). In the column "time" describing Chaff's performance, the upper number gives the total time spent on the instances that Chaff finished. The lower number is equal to the upper number plus 60 000 times the number of aborted instances in the class.

In our opinion, the main conclusion that can be drawn from Table 2 is that BerkMin is more robust than Chaff. One more argument substantiating this point of view is the following. The web site [21] gives statistics on the performance of 23 SAT-solvers (including a version of Chaff) on a representative set of instances. This set includes the 16 CNFs of class Beijing which are all satisfiable except one CNF. Each of the 23 SAT-solvers had at least two CNFs of the Beijing class that it was not able to solve in the timeout limit (10 000 sec.) Interestingly, there are no "universally" hard CNFs in this class. That is a CNF that cannot be solved by one SAT-solver can be finished in a few seconds by another, which suggests that these SAT-solvers are not robust. At the same time, BerkMin was able to solve all the 16 CNFs of the Beijing class in about 8 minutes.

Table 3 gives some details of Chaff's and BerkMin's performance on a few instances from classes fvp-unsat1.0, Hanoi, and fvp-unsat2.0. Chaff was aborted on the 3 instances marked with the star symbol. The two columns (Database size) / (Initial CNF size) give the ratio of the total number of generated conflict clauses and clauses of the initial CNF to the number of clauses of the initial CNF. Table 3 shows that BerkMin has better performance because it builds smaller search trees and its clause database is smaller. Besides, column (Largest CNF size)/ (Initial CNF size) shows that the number of clauses BerkMin had to keep in memory at the same time was at most 4 times the number of clauses in the initial CNF. Unfortunately, Chaff does not report the size of the largest intermediate CNF. However judging by the data output by the Unix utility called "top", Chaff took much more memory than our program. BerkMin was developed on a PC with 128 Mbytes of memory. So it was designed to reduce the probability of exceeding that limit. In experiments BerkMin took more than 128 Mbytes of memory only on a few instances of very large CNFs. On the other hand, Chaff often used 1Gbytes of memory and more.

7. Conclusions

This paper describes a new SAT solver, BerkMin, that is more robust than Chaff, currently the best SAT solver in the EDA domain. BerkMin has four new features distinguishing it from the predecessors. Three of them develop Chaff's idea that recently deduced clauses are the most important to satisfy. First, all clauses are chronologically sorted so that BerkMin always tries to satisfy the most recently deduced clause that is left unsatisfied. Second, BerkMin gets a better estimation of the activity of variables by taking into account a set of clauses which is wider than what Chaff takes. Third, BerkMin uses a new clause database management strategy that takes into account not only the size of a clause but its activity in conflict making and its "age". The fourth new feature of BerkMin is its branch selection strategy.

Class of benchmarks	Number of instances	Chaff		BerkMin	
		Time (sec.)	Number of Aborted	Time (sec.)	Number of Aborted
Blocksworld	7	52.5	0	**9.0**	0
Hole	5	**79.9**	0	339	0
Par16	10	33.9	0	**13.6**	0
sss 1.0	48	41.6	0	**13.4**	0
sss 1.0a	8	**17.4**	0	17.9	0
sss-sat 1.0	100	383.9	0	**254.4**	0
fvp-unsat1.0	4	**589.1**	0	1637.4	0
vliw-sat 1.0	100	**2602.9**	0	7305.0	0

Table 1. Benchmarks on which Chaff's and BerkMin's performances are comparable

Class of benchmarks	Number of instances	Chaff		BerkMin	
		Time (sec.)	Number of Aborted	Time (sec.)	Number of Aborted
Beijing	16	468.1 (>120,468.1)	2	**494.0**	0
Miters	5	1515.9 (>121,515.9)	2	**3477.6**	0
Hanoi	3	1320.6 (> 61,320.6)	1	**1401.3**	0
fvp-unsat2.0	22	19679.4 (> 139,679.4)	2	**6869.7**	0

Table 2. Benchmarks on which BerkMin dominates

Instance number	Satis-fiable	Chaff			BerkMin			
		(Database size) / (Initial CNF size)	Number of decisions	Time (sec.)	(Database size) / (Initial CNF size)	(Largest CNF size)/ (Initial CNF size)	Number of decisions	Time (sec.)
9vliw_bp_mc	no	2.42	1,124,797	**567.4**	1.88	1.04	2,384,485	1625.0
hanoi5	yes	31.49	504,463	1313.4	8.68	2.38	194,672	**71.2**
hanoi6*	yes	129.04	5,580,228	49021.1	19.58	4.19	1,948,717	**1328.7**
4pipe	no	3.01	412,358	354.6	1.49	1.08	144,036	**40.9**
5pipe	no	1.65	461,275	333.7	1.09	1.01	213,859	**71.8**
6pipe*	no	15.97	5,580,228	49131.2	1.71	1.05	1,371,445	**1015.6**
7pipe*	no	7.37	11,779,016	48053.1	1.95	1.05	3,357,821	**3673.2**

Table 3. Details of Chaff's and BerkMin's performance on some instances

8. Acknowledgements

We would like to thank the anonymous reviewers for their insightful recommendations for improving this paper.

References

1. L.Baptista, J.P.Marques-Silva. The interplay of randomization and learning on real-world instances of satisfiability // Proceedings of AAAI Workshop on Leveraging Probability and Uncertainty in Computation. - July 2000.
2. R.J.J.Bayardo, R.C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances, in: Proceeding of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence, Rhode Island, 1997, pp. 203-208.
3. E.Ben-Sasson, R. Impagliazzo, A.Wigderson. Near optimal separation of Treelike and General resolution // Proceedings of SAT-2000: Third Workshop on the Satisfiability Problem. - May 2000. -P.14-18.
4. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs // Proceedings of Design Automation Conference, DAC'99. -1999.
5. R.K.Brayton et. al. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, -1984.
6. J.Burch, V.Singhal. Tight Integration of Combinational Verification Methods// Proceedings of International. Conf. on Computer-Aided Design.-1998.
7. M.Davis, G.Longemann, D.Loveland. A Machine program for theorem proving// Communications of the ACM. -1962. -V.5. -P.394-397.
8. O.Dubois, P.Andre, Y. Boufkhad, J.Carlier. SAT versus UNSAT. In: Johnson and Trick, Second DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1996, pp.415-436.
9. E. Goldberg, M.Prasad. Using Sat for combinational equivalence checking// Proceedings of Design, Automation and Test in Europe Conference. –2001. -P.114-121.
10. C.P.Gomes, B. Selman, H.Kautz. Boosting combinational search through randomization // Proceedings of International Conference on Principles and Practice of Constraint Programming. - 1997.
11. J.W. Freeman. Improvements to propositional satisfiability search algorithms // Ph.D. thesis, Department of computer and Information science, University of Pennsylvania, Philadelphia , 1995.
12. C.M.Li. A constrained-based approach to narrow search for Satisfiability // Information processing letters.-1999. -V. 71. -P. 75-80.
13. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In: Proceeding of the 38th Design Automation Conference (DAC'01), 2001.
14. J.P.M.Silva. The impact of branching heuristics in propositional satisfiability algorithms. Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September 1999.
15. J.P.M.Silva, K.A.Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability // IEEE Transactions of Computers. -1999. -V. 48. -P. 506-521.
16. P. Stephan, R.Brayton, A. Sangiovanni-Vencentelli. Combinational Test Generation Using Satisfiability // IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems. -1996.-Vol. 15. -P. 1167-1176.
17. M.Velev. CMU benchmark suite. Available from http://www.ece.cmu.edu/~mvelev.
18. H.Zhang. SATO: An efficient propositional prover// Proceedings of the International Conference on Automated Deduction. -July 1997. -P.272-275.
19. http://www.ee.princeton.edu/~chaff/spelt3/chaff2_20010323_spelt3-bin-solaris.tar
20. http://www.satlib.org/benchm.html
21. http://www.lri.fr/~simon/satex/satex.php3