Propelling SAT and SAT-based BMC using Careset

Malay K. Ganai
NEC Laboratories America, Princeton, NJ, USA

***Abstract*—We introduce the notion of careset, a subset of variables in a Boolean formula that must be assigned in any satisfying assignment. We propose a restricted branching technique in a CDCL solver (i.e., DPLL-based SAT solver with clause learning) such that every decision path is prefixed with decisions on such a careset. Although finding a non-trivial careset may not be tractable in general, we demonstrate that for a SAT-based bounded model checking (BMC) application we can derive it automatically from the sequential behaviors of programs. Our proposed branching technique significantly reduces the search effort of a CDCL solver, and leads to a performance improvement of 1-2 orders of magnitude over well-known heuristics, and over top-ranked solvers of SAT2009 competition, that do not exploit system-level information. We also discuss the proof complexity of such a restricted CDCL solver.**

I. Introduction

In application domains such as bounded model checking (BMC) of software and hardware [1], the analysis engine has to explore paths of bounded length to validate the reachability property. The problem instances are typically derived from transition relation capturing the sequential behaviors of an underlying system using suitable transformation. These problem instances are typically encoded into Boolean formulas (e.g., CNF DIMACS format). The core of the analysis engine uses a DPLL-based [2] SAT solver to search through a Boolean formula. As paths get longer, the number of possible paths, and hence the search space, increases exponentially.

The state-of-the-art SAT solvers use various techniques to prune the search space faster. Some of the important ones are frequent restarts [3], [4], intelligent branching heuristics [5], [6], and learning conflict-driven resolution clauses [7] and binary clauses [8]. These solvers are also well-engineered using techniques such as two-literal watch scheme [6], efficient preprocessing [9], hybrid representation [10], and many others (e.g. [11], [12]). In spite of these improvements, the "loss" of high-level information during encoding can significantly degrade their performance. By loss, we imply that system-level structure and behavior cannot be inferred from a Boolean formula without knowing the actual transformation steps.

- *Structure of the transition relation*: During logic synthesis (i.e., bit-blasting of the transition relation), there are substantial losses of structural information such as types of arithmetic and logical modules, connectivity among such modules (i.e., their dependencies), and independent (i.e., controlling) variables.
- *System level behavior*: The constraints and sequential behaviors get lost during behavioral-level synthesis (i.e., during modeling of a system).

Previous experimental studies [10], [13]–[18] have shown some success in exploiting structural information in a propositional formula to improve CDCL solvers (i.e., DPLL-based solver using Conflict-Driven Clause Learning). Some of these include: (I) branching restriction on dominating input variables [13]–[15], backdoors variables [19], justification gates [10], [18], fanout gates, and variables in dependency graphs [14], [20]; (II) learning non-trivial circuit clauses corresponding to symmetry [21], special gates such as XOR, XNOR, and ITE gates [16], [17]; and (III) re-coding CNF using circuit observability don't cares (Cir-ODC) [22]. However, these techniques do not exploit system-level information.

A. Overview of our approach

Although it has been proved [23] that CDCL is exponentially stronger (i.e., the search tree can be exponentially shorter) than DPLL [2], the size of the search tree of CDCL can still be very large as it is sensitive to a branching order. Choosing the right variables and their order to shorten the search tree are the primary focus of this paper.

It is a well known fact that not all variables need to be assigned while determining a satisfiable result. With that in mind, we formalize the notion of *careset*, a subset of variables that must be assigned in any satisfying assignment. We extend the definition to an unsatisfiable instance, by defining careset on maximal satisfiable subsets. We propose a restricted branching technique in a CDCL solver such that every decision path is prefixed with a sequence of decisions on such a careset. We refer to such a sequence as a *branching prefix sequence*. Even though finding such a non-trivial set and such a sequence may not be tractable in general, we demonstrate that for a software verification application we can derive them automatically from the sequential behaviors of programs.

We compare formally the proof complexity [24] of restricted CDCL vis-a-vis unrestricted CDCL in terms of the size of the shortest proofs, measured in the number of decisions, they can produce. For a given careset c, and its size $|c|$, we show that the shortest proof (π') (and its size $|\pi'|$) obtained in restricted CDCL cannot be greater than the shortest proof (π) (and its size $|\pi|$) obtained in unrestricted CDCL by more than a factor of $f(c)$ i.e., $|\pi'| \leq f(c) \cdot |\pi|$, where $f(c) = 2^{|c|}$ in general. However, for the software model checking application $f(c)$ can be much smaller than $2^{|c|}$.

For such an application, we demonstrate that our branching technique significantly reduces the search effort of our CDCL solver (based on [10]) by helping it learn shorter and useful clauses earlier during the search process. We observe that the length of clauses learnt are reduced by an order-of-magnitude on average. This leads to a performance gain of 1-2 orders of magnitude over the well known heuristics such as VSIDS [6] and circuit-based [10], [14], [18], [22]. Even though we have not yet included the latest and greatest improvements in our solver, we demonstrate an order of magnitude improved performance of such a restricted CDCL solver over the well-engineered top-ranked solvers of SAT2009 competition [25].

For generality reasons, these advanced solvers do not intend to exploit system-level information. However, without such

information, the performance penalty incurred by these solvers is in orders of magnitude as observed in our experiments. Our goal is to draw attention to the SAT community of substantial progresses that are still possible in branching techniques as they play decisive role in the SAT performance.

B. Related Work

In [13]–[15], [26], problem structure was exploited to restrict the branching only to a smaller set of variables, referred to as an independent variables set (IVS). These variables correspond to non-deterministic initial state variables and primary input variables for circuit applications [14], action variables in planning applications [13], and task variables in task sequencing problems [23]. By definition, these variables dominate others variables that are not in the set i.e., dependent variables. A total assignment on IVS uniquely determines the values of the dependent variables. While such restrictions help in specific applications, they can degrade the performance of CDCL exponentially worse when compared to DPLL on some other application [27].

In [19], a notion of backdoor variables was introduced, where the branching was restricted only to such variables. The idea is that once all of these variables have values, the reduced formula can be solved by a polynomial-time solver. For a constraint Boolean circuit, an IVS is a backdoor set. It was demonstrated [28] that there is a strong correlation between the size of a backdoor set and the hardness of the corresponding Boolean formula. In general, finding a backdoor set from a given Boolean formula, is intractable [28]. Researches have also studied both theoretically and empirically [29] with the notion of backbone set [30]. A backbone set of a satisfiable Boolean formula is a set of literals which are assigned unique common values in every satisfying assignment. It has been shown that finding such a set is also intractable [28].

Our proposed notion of careset is different from the notion of backdoor set or IVS. As we shall see later, a careset is a necessary set while a backdoor set (or IVS) is a sufficient set for a satisfiable formula. A careset is also different from a backbone set, as careset variables need not have a unique common assignment in every satisfying assignment.

In [10], [18], [22] circuit observability don't cares (Cir-ODC) were used to restrict the branching to justification gates only, and avoid branching on the unobservable gates. In general, such a branching is oblivious to system-level information. In [31], functional information such as arithmetic types were used to guide the decision engine. In our previous work [32], we bias the decision choice on variables corresponding to control state predicates, and thereby, use sequential behaviors to guide the search. In this work, we provide a formal justification for such biasing, and further improve the decision process using branching prefix sequences.

Outline: The rest of the paper is organized as follows: With some background in Section II, we formalize the notion of careset, and introduce our branching method in Section III. In Section IV, we give an overview of software model checking. For that application, we present a method to generate careset variables automatically, and describe our branching technique in Section V. This is followed by a formal exposition on proof complexity of the method in Section VI, and its detailed experimental evaluation in Section VII. We give our conclusions and future directions in Section VIII.

II. Preliminaries

`CNF`. A CNF formula F is defined as a conjunctive set, i.e., $AND\ (\cdot)$ of clauses where each *clause* is a disjunctive set, i.e., $OR\ (+)$ of literals. A *literal* is a variable v (positive) or its negation $\bar{v}$ (negative). Let $vars(F)$ and $clauses(F)$ represent the set of all variables and clauses in F, respectively. An *assignment* for F is a Boolean function $\alpha : V \mapsto \{0,1\}$, where $V \subseteq vars(F)$. We use $v \in \alpha$ to denote that v is assigned under α. We say an assignment α is *total* if $V = vars(F)$, otherwise, it is *partial*. A literal l is *false (true)* under α if $\alpha(l) = 0(1)$. A variable (and literal) is *free* if it is not assigned. A clause is *satisfied* if at least one of its literals is true. A clause is *conflicting* if all its literals are false. An assignment α is *satisfying* if all clauses in F are satisfied by α, and not necessarily all variables be assigned. We use $F|_\alpha$ to denote the simplified formula where the corresponding assigned variables $(\in \alpha)$ are replaced with their assigned values, and false literals and satisfied clauses are removed. A *maximal satisfiable subset* (MSS) of F corresponds to a subset of clauses of F that is maximally satisfiable, i.e., adding any remaining clause would make it unsatisfiable. For a set S, we use $|S|$ to denote its cardinality.

A *P-Solver* solves a Boolean formula F in polynomial time if it accepts F. For example, a 2SAT-Solver that solves 2SAT-CNF (i.e., a set of clauses with at most of 2 literals) but rejects all others, is a P-Solver. A non-empty set of variables S is a *backdoor* [19] in a satisfiable F if for some assignment $\alpha : S \mapsto \{0,1\}$, P-Solver can show $F|_\alpha$ to be satisfiable. Such a set is *strong* if for all such α, P-Solver can solve $F|_\alpha$, i.e., show it to be sat/unsat. A set of variables S is a *backbone* [30] of satisfiable F if there is a unique partial assignment $\alpha : S \mapsto \{0,1\}$ such that $F|_\alpha$ is satisfiable. Note, assigning opposite value to a backbone variable would make $F|_\alpha$ unsatisfiable.

`Circuit`. We consider a Boolean circuit G represented as a DAG where each node represents a circuit gate, i.e, `OR`, `AND`, `XOR`, or `NOT`, and each edge connects a gate to its fanout node. We define an assignment for G as a Boolean function $\alpha : W \mapsto \{0,1\}$, where W is the set of all gate outputs and primary inputs of G. We say a gate is *justified*, when its input values justify its output value. For example, for $g = \texttt{AND}(a,b)$, $g = 0$ can be justified by either $a = 0$ or $b = 0$. Note, a primary input and a gate with no output value are always justified. We say a gate is *totally justified*, if its inputs are also justified transitively; otherwise, it is *partially justified*.

A constraint Boolean circuit is a pair $\langle G, \tau \rangle$ where some gates in G are constrained with an assignment τ. Note, without a constraint τ, a Boolean circuit is always satisfiable. We say $\langle G, \tau \rangle$ is satisfiable if there exists an assignment, referred as *justifying*, which (i) preserves the input/output relation of each gate, and (ii) each constraint gate is totally justified. One can encode a constraint Boolean circuit $\langle G, \tau \rangle$ into an equi-satisfiable CNF formula $cnf(\langle G, \tau \rangle)$ in linear-time using standard "Tseitin translation."

`CDCL`. The basic DPLL procedure [2] has three main steps applied repeatedly: branch on a literal, apply *unit propagation* (UP) rule, i.e., forcing a free literal true when all the other

literals in a clause are false, and backtrack chronologically when a conflict is observed. It stops when either all clauses are satisfied or all branches are explored. Conflict-driven Clause learning [7] (CDCL) improves the basic procedure by learning resolvent clauses after analyzing the causes of a conflict. In the sequel, we use "CDCL" to denote any implementation of the CDCL procedure, and use "a CDCL solver" to denote a specific implementation.

III. CARESET

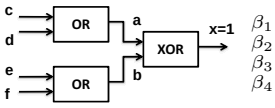
Before we delve into the formal definition of careset, we first define *minimally satisfying assignment* for a satisfiable Boolean formula F for a given P-Solver.

***Definition** 1 (Minimally Satisfying Assignment (MSA)):* We say an assignment α of Boolean formula F is minimally satisfying for a given P-Solver such that (i) UP rule cannot be applied on $F|_\alpha$ further, (ii) $F|_\alpha$ can be shown to be satisfiable by the P-Solver, and (iii) unassigning at least one variable in α would violate the condition (i) or (ii). We use $MSA(F, P)$ to denote the set of all MSAs of F for a given P-Solver.

Example 1: Let F be $(a+\bar{x}+d)(\bar{a}+x+c)(b+\bar{y}+\bar{d})(\bar{b}+y+\bar{c})$. Then, $\alpha = \{x = 0, y = 0\}$ is an MSA w.r.t. a 2SAT-Solver, as $F|_\alpha = (\bar{a} + c)(\bar{b} + \bar{c})$ is a 2SAT-CNF formula.

***Definition** 2 (Minimally Justifying Assignment (MJA)):* For a constraint Boolean circuit $\langle G, \tau\rangle$, we say an assignment β is minimally justifying if un-assigning any $v \in \beta$ would leave some constraint gate partially justified.

Example 2: All MJAs $\beta_1 - \beta_4$ for the constraint circuit $\langle G, \{(x = 1)\}\rangle$ are shown below: Consider a P-Solver



$\beta_1 = \{x = 1, a = 1, b = 0, e = 0, f = 0, c = 1\}$,
$\beta_2 = \{x = 1, a = 1, b = 0, e = 0, f = 0, d = 1\}$,
$\beta_3 = \{x = 1, a = 0, b = 1, c = 0, d = 0, e = 1\}$,
$\beta_4 = \{x = 1, a = 0, b = 1, c = 0, d = 0, f = 1\}$.

that applies arbitrary values to a set of unassigned primary input variables, and applies UP rule recursively on the circuit clauses. Such a solver, referred as `CktSim`, can always satisfy the gate clauses of an unconstraint Boolean circuit.

***Proposition** 1:* β is an MJA of $\langle G, \tau\rangle$ iff β is an MSA for $cnf(\langle G, \tau\rangle)$ w.r.t. a `CktSim` as P-solver.

One can verify that $\beta_1-\beta_4$ are MSAs for $cnf(\langle G, \tau\rangle)$ w.r.t a `CktSim`. Note that $\alpha = \{x = 1, a = 1, b = 0, e = 0, f = 0\}$ is an MSA w.r.t a 2SAT-Solver, but not w.r.t. a `CktSim`.

In the sequel, we use `CktSim` as the given P-Solver, and use $MSA(F)$ to denote $MSA(F, \texttt{CktSim})$. We now formally introduce the notion of careset for a satisfiable formula F, given `CktSim` as a P-Solver. Let F_{red} denote a reduced formula F after applying the UP rule recursively on F.

***Definition** 3 (Careset):* A non-empty set S of variables ($\subseteq vars(F)$) is a *careset* for a given formula F, such that a S variable is assigned in every MSA of F, i.e., $v \in S \rightarrow \forall \alpha \in MSA(F).\ v \in \alpha$. Such a set S is *maximum* when it includes all such variables, i.e., $S = \{v \mid \forall \alpha \in MSA(F).\ v \in \alpha\}$. We say S is *non-trivial* if $\exists v \in S.\ v \in vars(F_{red})$; otherwise, it is *trivial*.

In the sequel, we use $careset(F)$ to denote a non-trivial careset of F, which may not be maximum unless noted otherwise. Intuitively, a careset is a set of variables that must be assigned to "witness" a satisfying assignment.

Using Proposition 1, we define careset for a Boolean constraint circuit $\langle G, \tau\rangle$ as $careset(F)$ where $F = cnf(\langle G, \tau\rangle)$. For Example 2, non-trivial caresets of $\langle G, (x = 1)\rangle$ are $\{x, a\}$,$\{x, b\}$, and $\{x, a, b\}$ as a, b, c are assigned in all MJAs, i.e., $\beta_1 - \beta_4$. The set $\{x, a, b\}$ is the maximum careset. These caresets are non-trivial as values on a, b cannot be obtained by unit propagation on $x = 1$, while $\{x\}$ is a trivial careset.

We extend the definition of careset to an unsatisfiable formula F by defining it on maximal satisfiable subsets of F. Let $MSS(F)$ denote a set of all MSS of F. Then, $careset(F) := \cup_{F' \in MSS(F)} careset(F')$. Such a careset is maximum, when careset for each F' is maximum. Note, a non-trivial $careset(F')$ for any MSS F' of F is also a non-trivial careset for F.

Comparing Careset, Backdoor, Backbone. In contrast to a backbone set, where variables are necessarily set to unique values, careset variables only need to be assigned, not necessarily to unique values, in any satisfying assignment. Compared to a backdoor set, which is a *sufficient* set, a careset is a *necessary* set for solving a problem satisfiable. Such a necessary set is arguably smaller than a backdoor set, and therefore can help the decision engine prioritize better.

For Example 2, a backbone set is $\{x = 1\}$, a backdoor set is $\{x, a, b, c, e, f\}$ (as `CktSim` returns satisfiable for assignment β_1), a strong backdoor set is $\{c, d, e, f\}$ (as `CktSim` returns SAT/UNSAT for a total assignment on the primary inputs), and a careset is $\{x, a, b\}$.

A. Branching Strategy using Careset

We observe that for a satisfiable instance, a complete assignment on careset variables is a "gateway" to a satisfying solution. Intuitively, for such instances we should branch on careset variables first, before branching on the other variables. Such a branching technique is also a good heuristic for unsatisfiable instances as argued below.

Assume F is unsatisfiable. Let $F' \in MSS(F)$, and $C = clauses(F) \setminus clauses(F')$. Let $vars(\alpha)$ denote the set of variables assigned under $\alpha \in MSA(F')$ and α_S denote values of S variables under assignment α. As F is unsatisfiable, $\exists S \subseteq vars(\alpha)$ such that α_S makes some clause $c \in C$ conflicting. We say α is blocked by c. Any $\beta \in MSA(F')$ is also blocked by $c \in C$, if $\alpha_S = \beta_S$. Since careset variables must be assigned in any MSA of F', branching on them first can lead to early blockage of MSAs, and faster resolution.

We refer to such a branching technique as *branching prefix sequence*. In contrast to a backdoor set where the (ideal) goal is to obtain the smallest set, our (ideal) goal is to obtain the maximum careset. However, obtaining such a set is as hard as finding all MSAs. For practical reasons, we would like to obtain a careset as large as possible, not necessarily maximum. We would like to answer three key questions:

- How can a non-trivial and useful careset be obtained?
- How can such a set be exploited in a CDCL solver?
- How can the strength of such a CDCL solver be accessed?

In Sections IV-V, we answer the first two questions by considering a software model checking application, and using the application-specific knowledge to derive a non-trivial careset and exploit it in CDCL that is restricted with branching prefix sequence. In Section VI, we compare the relative proof complexity of restricted CDCL w.r.t. unrestricted CDCL. In Section VII, we compare experimentally our restricted CDCL solver against the state-of-the-art CDCL solvers that do not exploit such application knowledge.

IV. Application: Model Checking of Software

We briefly discuss our model building step (similar to [32]) from a given C program. We first obtain a simplified control and data flow graph (CDFG) by flattening the structures and arrays into scalar variables of simple finite types (Boolean, 32-bit integer). We handle pointer accesses using direct memory access on a finite heap model, and apply standard slicing and constant propagation. We do not inline non-recursive procedures to avoid blow up, but bound and inline recursive procedures up to a user-defined depth. From the simplified CDFG, we build a deterministic extended FSM (EFSM) where each control state (or block) is identified with a unique *id*. We use a program counter PC to track the control state *id*. For the ease of explanation, we focus on simplified CDFGs that have a unique entry block (Src) and an error block (Err). We are interested in checking reachability properties such as array bounds violations, null pointer dereferencing, and assertion failures; that is, whether there is an execution trace from Src to Err block. We use EFSM and CDFG interchangeably to mean the same structure.

Example 3: Consider a low-level C program `foo` as shown in Figure 1, with its EFSM M. The control states, shown as boxes, correspond to control points in the program, as also indicated by the line numbers. Note, each control state is identified with a number in the attached small square box. For example, *Err* block 10 corresponds to the assertion in line 17. Update transitions of data path expressions are shown at each control state. A directed edge (a, b) between control states a, b corresponds to the control flow between the associated control points in the program. Each directed edge is associated with an enabling condition.

Based on such a CDFG, we encode the transition model T of an EFSM symbolically as $T := T_C \wedge T_D$, where T_C encodes (control) transition relation for PC, i.e., the guarded transitions between the control states, and T_D encodes (data) update transition relation for datapath variables based on the expressions assigned to the variables in various control states in the model. We illustrate the translation of T for Example 3. We use v, v' to denote current and next state variable, g_{ij} to denote the guarded transition predicate at a directed edge (i, j), and $B_r := (PC = r)$ to denote the control state predicate. For ease of readability, we use C syntax '?:' to denote *if-then-else* operator, and other standard relation operators. We obtain a Boolean encoding of the update and the guarded transition relations under the assumption of 32-bit integer variables (not shown separately).

Transition relation for PC $[T_C(PC', PC, a, b)]$

$$PC' := B_1 \wedge g_{12} \ ? \ 2 : B_1 \wedge g_{16} \ ? \ 6 : B_2 \wedge g_{23} \ ? \ 3 : B_2 \wedge g_{24} \ ? \ 4 : \cdots : 11$$

where $\forall r \in \{1, \cdots, 11\}\ B_r := (PC = r)$, and $g_{12} := (a \geq b)$, $g_{16} := (a < b)$, $g_{23} := (a < b)$, $g_{24} := (b \leq a)$, and so on.

Update transition relation $[T_D(a', a, b', b, PC)]$

$$a' := B_1 \ ? \ a_0 : B_4 \ ? \ (a - b) : B_7 \ ? \ (a - b) : a$$
$$b' := B_1 \ ? \ b_0 : B_3 \ ? \ (b - a) : B_8 \ ? \ (b - a) : b$$

where a_0, b_0 are initial symbolic state values of a, b, resp., i.e., $1 \leq a_0, b_0 \leq 10$.

Bounded Model Checking. Let s^i denote a state at i^{th} step from some initial state s^0, and $T(s^i, s^{i+1})$ denote the state transition relation. A BMC instance (denoted as BMC^k) comprises checking if an LTL (Linear Temporal Logic) property ϕ can be falsified in *exactly* k steps from s_0, i.e.,

$$BMC^k := I \wedge T^{0,k} \wedge \neg\phi(s^k) \tag{1}$$

where $\phi(s_k)$ denotes the predicate that ϕ holds in state s^k, and I denote the initial state predicate, and $T^{0,k}$ denote the unrolled transition relation $\bigwedge_{0 \leq i < k} T^{i,i+1}$ where $T^{i,i+1} := T(s^i, s^{i+1})$. Given a bound n, a *BMC run* comprises checking the satisfiability of BMC^k iteratively for $0 \leq k \leq n$ using a SAT solver. In the sequel, we focus only on the reachability of block Err from block Src, i.e., $\phi := F(PC = Err)$, where F is the eventually LTL operator, and $I := (PC^0 = Src) \wedge D^0$, where D^0 is the initial state predicate on datapath variables.

A. Control Flow Reachability

We use CFG to denote a CDFG without the enabling condition and update transitions. A *control path* is a sequence of successive control states, denoted as $\gamma^{0,k} = (c_0, \ldots, c_k)$, where (c_i, c_{i+1}) is a directed edge in the CFG. We use $c \in \gamma^{0,k}$ to denote that c belongs to the sequence. An *unrolled CFG* for depth d is a DAG that corresponds to an unfolded CFG where the transitions after depth d is removed, shown as an example in Figure 1 for $d = 7$. A *control state reachability* (CSR) analysis is a breadth-first traversal of the unrolled CFG where a control state b is one step reachable from a *iff* there is a directed edge (a, b). At a given sequential depth d, let $R(d)$ represent the set of control states that can be reached in CFG in one step from the states in $R(d-1)$, with $R(0) = c_0$.

Computing CSR for the unrolled CFG of M (Figure 1), we obtain the set $R(d)$ for $0 \leq d \leq 7$: $R(0) = \{1\}$, $R(1) = \{2, 6\}$, $R(2) = \{3, 4, 7, 8\}$, $R(3) = \{5, 9\}$, $R(4) = \{2, 10, 6, 11\}$, $R(5) = \{3, 4, 7, 8\}$, $R(6) = \{5, 9\}$, $R(7) = \{2, 10, 6, 11\}$.

We use $from(r)$ and $to(r)$ to denote set of blocks reachable from and to r, respectively. The unrolled transition relation $T^{0,k}$ capture the following control flow constraints *implicitly*. We use v^d to denote the unrolled variable v in $T^{d,d+1}$. B_r^d refers to the Boolean control state predicate $(PC^d = r)$, i.e., whether PC at depth d is at control state r. It has been shown that these constraints when added explicitly, improve the search [33].

- Reachable Block Constraint (RBC): At least one block is reachable at d i.e., $\exists r \in R(d)$. (B_r^d).
- Mutual Exclusion Constraint (MEC): At most one block is reachable at d, i.e, $\forall r \neq t$. $(B_r^d \rightarrow \neg B_t^d)$

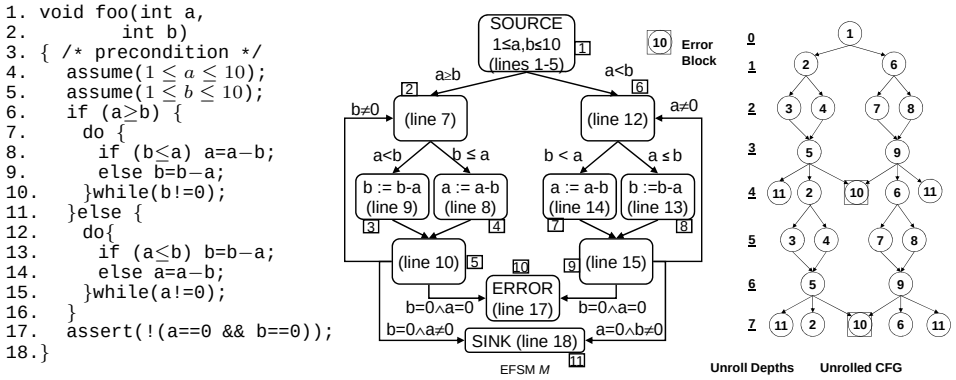


Fig. 1. A sample C code, its EFSM M, and an unrolled CFG for depth 7.

- Forward Reachable Block Constraint (FRBC): If r is reachable at $d < k$, then $t \in from(r)$ is reachable at $d+1$, i.e., $\exists t \in from(r).\ (B_r^d \rightarrow B_t^{d+1})$.
- Backward Reachable Block Constraint (BRBC): If r is reachable at $d > 0$, then $t \in to(r)$ is reachable at $d-1$, i.e., $\exists t \in to(r).\ (B_r^d \rightarrow B_t^{d-1})$.

V. Generating a Careset for BMC

As per Eqn 1, $BMC^k = B^0_{Src} \wedge D^0 \wedge T^{0,k} \wedge B^k_{Err}$, where D^0 is initial state predicate on datapath variables. Let $\Gamma^{a,b}$ denote a set of all control paths $\gamma^{0,k}$ between control states a and b, i.e., $\{\gamma^{0,k} \mid c_0 = a, c_k = b\}$. We say $c_d \in \Gamma^{0,k}$ iff $c_d \in \gamma^{0,k}$ for some $\gamma^{0,k} \in \Gamma^{0,k}$. The following theorem will provide a basis for generating a non-trivial careset for BMC^k.

***Theorem** 1:* A non-trivial careset for BMC^k is a set of control state predicate variables in all the control paths from Src to Err, i.e., $\{B^d_{c_d} \mid c_d \in \Gamma^{Src,Err}, 0 \leq d \leq k\}$.

Proof. We consider two cases based on whether BMC^k is satisfiable or not.

Case 1: BMC^k is satisfiable. Clearly, $\exists \gamma^{0,k}$ s.t. $\gamma^{0,k}$ witnesses the control reachability of *Err* block from *Src* block. Let α be the corresponding MJA of BMC^k. Then, $\forall c_d \in \gamma^{0,k}.\ \alpha(B^d_{c_d}) = 1$, as otherwise, $B^k_{Err} = true$ would not be totally justified. As per MEC flow constraint, $\forall c_d, c'_d \in R(d).\ B^d_{c_d} \rightarrow \neg B^d_{c'_d}$, i.e., control predicate variables in the control paths other than $\gamma^{0,k}$ are implied false. Thus, the claim holds.

Case 2: BMC^k is unsatisfiable. We construct an MSS F' of BMC^k as follows: Initially, $F' = \emptyset$. We include the constraints $(B^0_{Src} \wedge B^k_{Err})$, and add all the constraints corresponding to the unrolled expressions for PC without the guarded expressions, i.e, we treat g^d_{ij} as free input variables. Note, F' constructed so far captures only the control flow constraints, and is therefore satisfiable. We then add the remaining data path and guarded expressions until F' becomes an MSS of BMC^k. By definition of a careset and using the argument as in Case 1, the claim follows. □

Based on the above theorem, we obtain a $careset(BMC^k)$ by doing forward and backward traversal from Src and Err blocks, resp. on the CFG, and including B^d_r corresponding to a control state $r \in R(d)$ that is visited by traversal in both direction. For Example 3 (Figure 1), $careset(BMC^4)$ is $\{B^0_1, B^1_2, B^2_3, B^2_4, B^3_5, B^1_6, B^2_7, B^2_8, B^3_9, B^4_{10}\}$.

A. Branching Prefix Sequence

We introduce the notion of branching prefix sequence (BPS)[1], a kind of restrictive branching where every decision path (starting at decision level 0) is prefixed with a given ordered sequence of branching literals.

***Definition** 4:* A *branching prefix sequence* (BPS) for a formula F is an ordered sequence $\sigma = (l_1, \cdots l_m)$ of literals of F such that a CDCL solver always picks first free literal l_i in σ (skips the assigned literals), and branches with l_i set to true. If all the literals in σ are assigned, default branching heuristic is applied. During backtracking, some of the literals in σ can become free. At any decision level, the solver always branches on the first free literal in σ, if one exists. However, the literals of σ are neither removed nor reordered. CDCL using a BPS is referred to as CDCL$_{bps}$.

We use the $careset(BMC^k)$ variables to obtain a BPS. We first define an ordering relation based on control distance of a careset variable $B^d_r \in careset(BMC^k)$.

***Definition** 5:* A control distance of a careset variable $B^d_r \in careset(BMC^k)$ is a function $\delta : careset(BMC^k) \mapsto \{0, \cdots, k\}$ such that $\delta(B^d_r) = k - d$.

For example, $B^1_6 \in careset(BMC^4)$ has a control distance $\delta(B^1_6) = 4 - 1 = 3$.

***Definition** 6:* An increasing (decreasing) sequence of literals in $careset(BMC^k)$ is defined as a total order on the careset variables (i.e., positive literals) with respective to a non-decreasing (non-increasing) control distances. Variables with the same control distances are ordered using some heuristic such as literal count. We use IS^k (DS^k) to denote increasing (decreasing) sequence of $careset(BMC^k)$.

An increasing sequence IS^4 for $careset(BMC^4)$ is as follows: $\{B^4_{10}(0)$, $B^3_5(1)$, $B^3_9(1)$, $B^2_3(2)$, $B^2_4(2)$, $B^2_7(2)$, $B^2_8(2)$, $B^1_2(3)$, $B^1_6(3)$, $B^0_1(4)\}$, where the values in the brackets refer to the respective control distances of the variables. Here we broke the tie using the corresponding control state id. In actual implementation, we use the VSIDS [6] scores.

Intuitively, an IS^k used as a BPS helps a CDCL solver to prune the *infeasible local path segments that are closer* to Err block by learning useful clauses with fewer decisions. We observed in our experiments that such an approach reduces the average length of conflict clauses per conflict (denoted as

[1]The notion of BPS differs from branching sequence [23] where a literal is chosen once, and may not be assigned on every decision path.

AvgCL) and search tree size (i.e., number of decisions) by 1-2 orders of magnitude. In Section VII, we provide detailed experimental results supporting our intuition. Now we discuss the proof complexity of such an approach.

VI. Proof Complexity of CDCL$_{bps}$

We use the notion of proof complexity [24] to compare the relative power of proof inference systems P and P' based on the shortest proofs π and π' they can produce, resp. Let $|\pi|$ and $|\pi'|$ denote the respective proof sizes. We say P' *polynomially simulates* P when $|\pi'| \leq 2^{O(logn)}(= poly(n)) \cdot |\pi|$ for all families of formula over n variables; otherwise, P' cannot polynomially simulate P. For example, it was shown [23] that DPLL cannot polynomially simulate CDCL.

In CDCL and CDCL$_{bps}$ proof systems, we measure the size of their shortest proofs in terms of their search tree, i.e., the number of decisions. As CDCL is unrestricted, an identical proof can be obtained in CDCL as in CDCL$_{bps}$ by applying the same decision order in CDCL as applied in CDCL$_{bps}$. Thus, the following holds trivially.

***Proposition** 2:* CDCL polynomially simulates CDCL$_{bps}$.

Unfortunately, we cannot claim in the other direction due to branching restriction in CDCL$_{bps}$. However, we provide a worst case bound on the size of its shortest proof. For any unsatisfiable formula F over n variables, let S be a careset of F, with $|S|$ denoting its size. Let $\pi_{bps}(F)$ ($\pi(F)$) and $|\pi_{bps}(F)|$ ($|\pi(F)|$ denote the shortest proof and its size, resp., obtained in CDCL$_{bps}$ (CDCL).

***Theorem** 2:* The shortest proof obtained in CDCL$_{bps}$ cannot be greater than that obtained in CDCL by more than a factor of $f(S)$, i.e., $|\pi_{bps}(F)| \leq f(S) \cdot |\pi(F)|$, where $f(S) = 2^{|S|}$.

Proof. Consider the search tree of CDCL$_{bps}$. Let $\mathcal{U} = \{\sigma_1 \cdots \sigma_m\}$ represent a set of unique assignments made on careset variables before a proof is generated in CDCL$_{bps}$, i.e, $\forall(i \neq j).\exists v \in \sigma_i, v \in \sigma_j.\ \sigma_i(v) \neq \sigma_j(v)$. Clearly, as F is unsatisfiable, each (partial) assignment on $vars(F)$ that ends in conflict, includes exactly one σ_i for some i. We claim that $|\pi_{bps}(F)| = \Sigma_i^m |\pi_{bps}(F|_{\sigma_i})| = \Sigma_i^m |\pi(F|_{\sigma_i})| \leq \Sigma_i^m |\pi(F)| \leq 2^{|S|} \cdot |\pi(F)|$. The first equality holds as assignments on careset variables are made before the rest in CDCL$_{bps}$. The second equality holds as branching heuristics of CDCL$_{bps}$ and CDCL are the same on non careset variables. The following inequality holds as CDCL is a natural proof system. The last inequality holds as $m \leq 2^{|S|}$. □

In practice, we often see conflict before all the careset variables are assigned. Moreover, all variables in S may not be independent, i.e., some are implied by others in S, in which case $m \ll 2^{|S|}$. Especially, for $F = BMC^k$ (Eqn 1), $S = careset(BMC^k)$ (Theorem 1), and $\Gamma^{Src,Err}$ (denoting a set of control paths from Src to Err), upper bound on $\mathcal{U}$ is determined by the number of control paths, i.e., $|\Gamma^{Src,Err}|$.

***Corollary** 1:* $|\pi_{bps}(F)| \leq |\Gamma^{Src,Err}| \cdot |\pi(F)|$.

VII. Experiments

We experimented with eight sets of benchmarks `E1-E8`, each with 1 to 3 properties. These correspond to software models and properties generated using software verification platform `F-Soft` [32] from real-world C programs such as network protocol and mobile software. Our experiments were conducted on a Linux box with Intel Pentium 4 CPU 3.2GHz, 2GB of RAM. On these models, we used a SAT-based BMC [17], [33].

We used an incremental hybrid SAT solver [10], [17], where a BMC instance is represented in And-Inverter circuit graph (AIG) and the learnt clauses are represented in CNF. It implements Chaff algorithm [6] using 1UIP clause learning scheme [34], and VSIDS [6] for branching. Note, it does not include many recent improvements such as preprocessing (SATeLite [9]), learning binary clauses during BCP [8], smart frequent restarts [4], and others (e.g. [11], [12]). Unlike a pure CNF solver[2], it has direct access to circuit information. Optionally, it uses circuit-based branching heuristics [10], [14], [18] (such as branching on the inputs of currently unjustified gates only). We refer to this branching heuristic as `CKT`.

We also provide such a hybrid solver with a BPS. For each BMC^k instance, we automatically generate sequences IS^k and DS^k (ref. Section V-A). We use `iBPS` (`dBPS`) to denote the branching heuristic where IS^k (DS^k) is used as a BPS.

Combining the above heuristics, we consider following four solvers `B1-B4` for performance comparison.

- `B1:VSIDS` The CDCL solver with `VSIDS` heuristic.
- `B2:CKT+VSIDS` The CDCL solver with `CKT` heuristic. However, when there are many choices at a decision level, the tie is broken with `VSIDS` heursitic.
- `B3:iBPS+CKT+VSIDS` The CDCL solver with `iBPS` heuristic. When there is no free literal in the BPS, it branches as `B2` until the BPS has a free literal.
- `B4:dBPS+CKT+VSIDS` Similar to `B3` but uses `dBPS`.

Experiment Set I. We compare the performance of the solvers `B1-B4` on benchmarks `E1-E8` for each BMC run, comprising solving BMC^k for each $k \geq 0$ until time out. We gave a timeout of 1200s for each BMC run. In each run, we generate and solve BMC instances incrementally at depth k. Other than branching, all other heuristics were kept the same. We show the results in Figure 2.

In `X-axis` we show BMC depths analyzed before timeout occurs, and in `Y-axis` we show the cumulative solve time (in sec) after each unrolled depth. We also labeled a few selected graphs for better readability.

Clearly, `B3` outperforms `B1,B2,B4` by several orders of magnitude. `B4` outperforms `B1` only in 3 cases, i.e., `E1`, `E3`, `E7`. This shows that branching order is equally important in a CDCL solver. We do not see much improvement of `B2` over `B1`. `B3` finds two witnesses (at depths 43 and 63, resp.) in `E7` while `B4` finds only the shorter witness (i.e., at depth 43). `B1` or `B2` finds neither of them. Clearly, circuit information does not provide much of guidance compared to system-level information. We do not show separately the comparison data with the solver used in [32], wherein the control state predicate variables are given higher VSIDS scores initially. We observed that the performance of such a solver is marginally better or comparable to that of `B1` on these BMC runs.

We provide detailed comparison results between `B2` and `B3` in Table I. For each benchmark, we obtained a list of BMC^k

[2]In a CNF solver, one can use Cir-ODC CNF encoding [22] to exploit circuit information, albeit with additional overhead compared to [10].

instances that were solved by B2 and B3 in more than 5s but less than 1200s. Note, all these instances are unsatisfiable. After sorting them on k, we selected *min*imum, *med*ian and *max*imum instances from the list as shown in Columns 1-2. The number of variables $(\#V)$ in these instances are about 200k-1.3M, as shown in Column 3. In Columns 4-6, we present the results of B2: the number of decisions (#D), the average conflict-clause length (AvgCL), and the time taken (in sec) (T). In Columns 7-11, we present the results of B3: the size of careset as percentage of #V (#CV), the number of decisions (#D), the number of decisions on careset variables as the percentage of #D (#DCV), the average conflict-clause length AvgCL, and the time taken T(s), resp. We observe that baring a few cases, AvgCL is about an order of magnitude smaller in B3, compared to B2. Clearly, iBPS guides the solver B3 better in learning useful clauses earlier, thereby, reducing the overall solve time significantly. We also find that the number of careset variables is about 1-3% of the number of variables, and in the most unsatisfiable instances, decisions on them are sufficient to solve the instance.

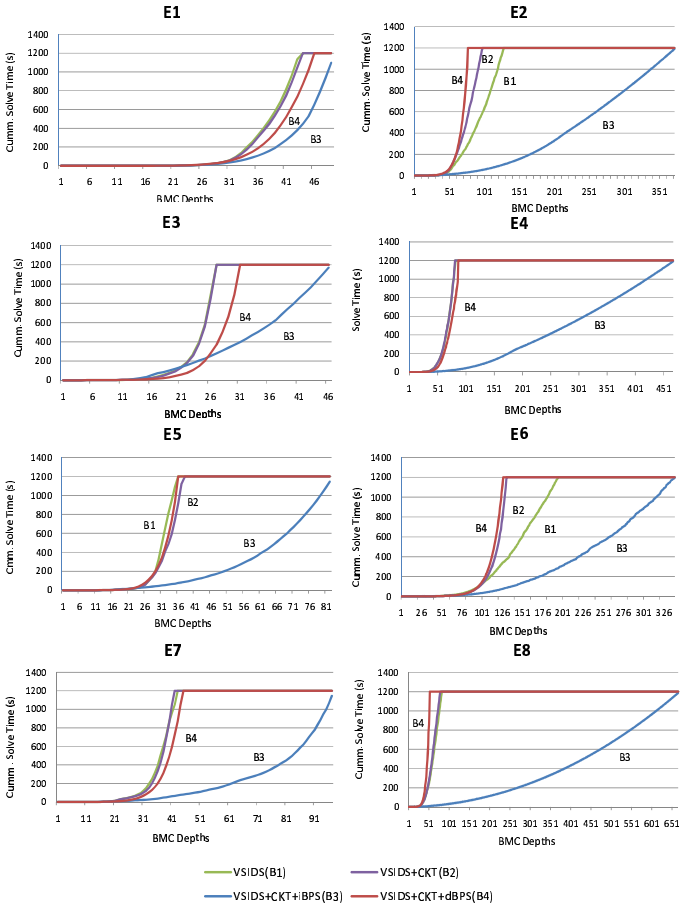


Fig. 2. BMC runs on E1-E8

Experiment Set II. We obtained DIMACS CNF format from the BMC instances at different depths from Experiment Set I. To keep our focus on hard instances, we included those on which B3 solver takes more than 5 sec to solve. We obtained a total of 130 instances; out of which 128 are unsatisfiable and 2 are satisfiable. The number of variables in these instances ranges between 120K-2.2M, and the number of clauses ranges between 350K-6.6M. These benchmarks are also made publicly available [35].

TABLE I
COMPARING B2 AND B3 ON (UNSAT) BMC^k.

BMC^k Instance			B2: CKT+VSIDS			B3: iBPS+CKT+VSIDS				
Ex	Depth k	#V ('000)	#D	AvgCL (#lit)	T (s)	#CV (%#V)	#D	#DCV (%#D)	AvgCL (#lit)	T (s)
E1	min: 28	262	381	247	5	1.1	710	100	27	3
	med: 38	445	1533	691	56	1.1	1977	100	99	21
	max: 43	522	1400	1081	62	1.1	595	100	251	25
E2	min: 46	250	669	275	5	2.3	137	100	27	0
	med: 47	257	421	145	5	2.3	137	100	27	0
	max: 49	271	909	223	9	2.3	124	100	30	0
E3	min: 16	582	242	350	8	0.7	435	100	28	13
	med: 22	1121	1107	2076	61	0.7	557	100	35	17
	max: 24	1323	413	5984	50	0.7	25	100	201	3
E4	min: 42	194	855	75	5	1.3	46	100	30	0
	med: 49	239	1293	85	8	1.3	47	100	63	0
	max: 80	440	2296	682	59	1.2	80	100	158	0
E5	min: 23	442	249	1547	5	0.7	275	100	32	1
	med: 33	721	1844	1262	66	0.7	386	100	286	2
	max: 36	804	1752	1773	65	0.7	198	100	246	1
E6	min: 93	223	397	3980	5	1.4	280	25	499	0
	med: 100	241	550	4062	9	1.4	204	68	537	1
	max: 130	320	1240	8773	88	1.4	320	77	675	2
E7	min: 28	223	243	610	5	1.0	248	100	70	0
	med: 31	257	245	550	8	1.0	141	92	49	0
	max: 40	359	973	1189	56	1.0	348	50	226	1
E8	min: 33	231	766	76	5	1.0	121	100	9	0
	med: 35	250	807	124	6	1.0	109	100	8	0
	max: 38	278	1021	67	8	1.0	109	100	10	0

#V: number of variables in thousands, #D: number of decisions
AvgCL: average conflict-clause length
#CV: careset size as % of #V, #DCV: % of #D on careset variabes)

TABLE II
NECLA SAT VS SAT2009 WINNERS [25].

Solver	SAT Time(s)	UNSAT Time(s)	Total Time(s)
NECLA SAT	24 (2)	2,042 (128)	2,066 (130)
Preco-SAT	468 (2)	18,367 (128)	18,835 (130)
mini-SAT	372 (1)	19,475 (91)	19,847 (92)
			1,601† (NECLA) 12,131† (Preco)
glucose	663 (1)	39,411 (111)	40,074 (112)
			1,789‡ (NECLA) 17,381‡ (Preco)

(.) # solved cases (out of 130) within 1800 sec.
† Time taken on 92 solved cases of miniSAT
‡ Time taken on 112 solved cases of glucose

We used B3 solver without CKT heuristic, and refer it as NECLA SAT solver[3] (For the sake of fair comparison, and to show the benefits of iBPS exclusively, we disabled circuit heuristics.) We compared this solver with PrecoSAT, miniSAT, and glucose, the top-ranked solvers in SAT2009 competition under application category [25]. These solvers use explicit or in-built preprocessor (e.g., SATeLite [9]), and smart frequent restarts [4], while NECLA SAT does not include any of these techniques. We gave a time limit of 1800s per instance. We provide a summary of the results in Table II.

In Column 1, we list the solvers we compared. In Column 2(3), we present the solve time (in sec) for SAT(UNSAT) instances. In Column 4 we present the total time taken (in sec) for the solved instances only. In Columns 2-4, we show the number of instances solved by each solver in brackets. Where the solved instances are less than 130, i.e., for glucose and

[3]NECLA SAT w/o iBPS corresponds to B1 solver.

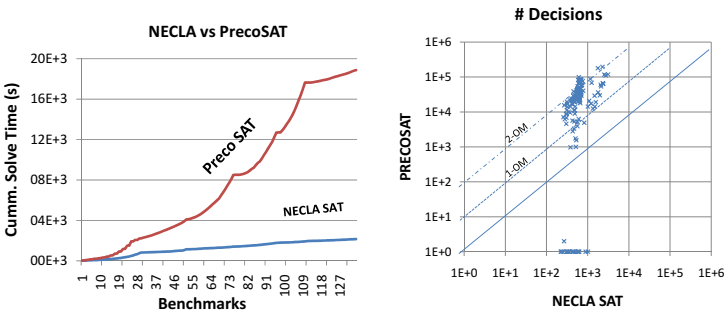


Fig. 3. NECLA SAT vs PrecoSAT: Cumm times, and # decisions

miniSAT, we also provide the total time taken by NECLA SAT and PrecoSAT on those solved instances.

NECLA SAT and PrecoSAT solve all 130 instances, while glucose and miniSAT solve 112 and 92 instances, respectively. Clearly, NECLA SAT solver outperforms the rest solvers by about an order of magnitude.

We also compare NECLA SAT and PrecoSAT in more details as shown in Figure 3. In the left figure, we show the instances solved (along X-axis), and the cumulative time taken in sec (along Y-axis). We observe that NECLA SAT outperforms PrecoSAT consistently by about an order of magnitude. In the right figure, we present a scatter plot comparing the number of decisions between NECLA SAT (along X-axis) and PrecoSAT (along Y-axis) in *logarithmic scale*, where each 'x' mark corresponds to an instance solved. We observe that the number of decisions in NECLA SAT are 1-2 orders of magnitude smaller than that in PrecoSAT, as indicated by clustering of 'x' between dotted lines namely, 1-OM and 2-OM. All the marks on X-axis (i.e., with 0 decision in PrecoSAT) are instances solved by the in-built preprocessor [9] of PrecoSAT. For these instances, NECLA SAT takes about 5-10s, about the same as the preprocessing time. NECLA SAT also requires an order of magnitude fewer backtracks comparatively (not shown separately). Clearly, benefit from using iBPS outweighs many heuristics in PrecoSAT.

VIII. Conclusion and Future Work

Branching plays a crucial role in a CDCL solver. We introduce the notion of careset variables and branching prefix sequence to guide the decision engine. We derive such a careset from software model checking application, and use it to improve the performance of a CDCL solver by an order of magnitude compared to the latest best SAT solvers that do not exploit system-level information. We also compared formally the resolution power of restricted CDCL vis-a-vis unrestricted CDCL. Overall, our results serve as a proof of concept that the analysis of system behaviors can be used to improve a SAT solver performance dramatically.

On the one hand, the current advanced SAT solvers do not intend to take advantage of system-level information for generality reasons, but on the other hand, the performance penalty of not using such information could be in the orders of magnitude, as observed in our software model checking experiments. For a better trade-off, we believe that there is a further scope in improving SAT-formulation where one can generate SAT problems conducive for the state-of-the-art solvers. In future, we would also like to detect careset during runtime, in contrast to its static determination as presented.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 1962.

[3] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of AAAI*, 1998.

[4] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proc. of IJCAI*, 2007.

[5] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *In Proc. of JAR*, 1995.

[6] M. W. Moskewicz and C. F. Madigan and Y. Zhao and L. Zhang and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of DAC*, 2001.

[7] J. Silva and K. Sakallah. GRASP–A New Search Algorithm For Satisfiability. In *Proc. of ICCAD*, Santa Clara, CA, November 1996.

[8] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT*, 2003.

[9] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *Proc. of SAT*, 2005.

[10] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. of DAC*, 2002.

[11] A. Biere. PicoSAT essentials. In *Proc. of JSAT*, 2008.

[12] A. Biere. P{re,i}coSAT@SC'09. In *SAT Competition*, 2009.

[13] E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proc. of AAAI*, pages 948–953, 1998.

[14] O. Strichman. Tuning SAT checkers for bounded model checking. In *Proc. of CAV*, 2000.

[15] E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables in propositional satisfiability. In *Proc. of JELIA*, 2002.

[16] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *CP*, 2002.

[17] M. K. Ganai and A. Gupta. *SAT-based Scalable Formal Verification Solutions*. Springer Science and Business Media, 2007.

[18] M. Järvisalo, T. A. Junttila, and I. Niemelä. Justification-based non-clausal local search for sat. In *ECAI*, 2008.

[19] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proc. of IJCAI*, 2003.

[20] H. Kautz, D. Mcallester, and B. Selman. Exploiting variable dependency in local search. In *In Abstracts of the Poster Sessions of IJCAI-97*, 1997.

[21] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proc. of DAC*, 2002.

[22] Z. Fu, Y. Yu, and S. Malik. Considering circuit observability don't cares in cnf satisfiability. In *Proc. of DATE*, 2005.

[23] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *In Proc. of JAIR*, 2004.

[24] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. In *Journal of Symbolic Logic*, 1977.

[25] SAT competiton 2009. http://www.satcompetition.org/2009/.

[26] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. *Proc. of AAAI*, 1994.

[27] M. Järvisalo and T. A. Junttila. Limitations of restricted branching in clause learning. *Constraints*, 14(3), 2009.

[28] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI*, 2005.

[29] A. J. Parkes. Clustering at the phase transition. In *AAAI/IAAI*, 1997.

[30] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. In *Nature 400:*, 1999.

[31] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC*, 2007.

[32] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. In *Proceedings of ISOLA*, 2004.

[33] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of ICCAD*, 2006.

[34] L. Zhang and C. F. Madigan and M. H. Moskewicz and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of ICCAD*, 2001.

[35] System Analysis and Verification Team. NECLA SAV Benchmarks. *http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php*.