Using Gate Recognition and Random Simulation for Under-Approximation and Optimized Branching in SAT Solvers

Markus Iser, Felix Kutzner, Carsten Sinz
Karlsruhe Institute of Technology (KIT), Germany
{markus.iser, felix.kutzner, carsten.sinz}@kit.edu

***Abstract*—We extract structure from CNF problems using a gate recognition algorithm and perform random simulation on that structure to generate conjectures about literal equivalences and backbone variables. We use these conjectures in two approaches to optimize CDCL SAT solving. In the first approach, we perform under-approximation by adding the conjectures to the original problem, while performing subsequent corrections in a refinement loop. In the second approach, we modify the branching heuristic such that it exploits the conjectures in order to stimulate clause learning. Experimental results show improvements, especially on unsatisfiable circuit-equivalence checking problems.**

I. Introduction

Automatic determination of the satisfiability of propositional formulas is a technology that over the past decades found its way into numerous application domains, including hard- and software verification [1], planning [2] and scheduling [3], as well as hardware layout [4].

Problem instances of application domains like these are highly structured, as they usually originate from problems that are formulated in more expressive logics (e.g. relational logic [5]). They can even be projections from higher order logics that are grounded and transformed to conjunctive normal form (CNF) using Tseitin encoding [6] or related procedures ([7], [8], [9]).

State of the art SAT solvers based on the conflict driven clause learning (CDCL) method are used to solve huge problem instances from application domains containing millions of variables. The most successful heuristics in CDCL SAT solvers [10] are conjectured to implicitly make use of the structure of the underlying problem.

Thiffault et al. [11] successfully implemented DPLL search for non-clausal propositional formulas, and show how direct exploitation of formula structure leads to improved solver performance. Järvisalo et al. show that some CNF (pre-)processing techniques simulate circuit level optimization [12]. The concept of blocked clauses [13] is crucial to clause elimination [14] and formula decomposition [15] and has also been used for the recovery of gate structure [16]. In [17], we present an algorithm that locally uses the properties of blocked clauses to recover *gate structure* from CNF more effeciently and use it to improve SAT solver performance by applying explicit circuit-level simplification as a preprocessing step.

Given the circuit representation of a SAT problem, *random simulation* has been successfully applied in hardware verification with Binary Decision Diagrams (BDD) [18] and Bounded Model Checking (BMC) with And-Inverter Graphs (AIG) [19]. In [15], Heule and Biere use random simulation to apply SAT sweeping techniques to CNF formulas. They use an approach that is called Blocked Clause Decomposition (BCD) to compute a large satisfiable subset of a given CNF formula, on which they apply a sweeping algorithm to generate a set of conjectured literal equivalences which they verify afterwards. With their approach, they can solve 10 more of those instances for which their tool-chain was able to extract a maximal blocked set within 100 seconds.

It has been shown that the efficiency of random simulation can be improved by using nonuniform probability distributions to randomize the circuit inputs. For instance, Knuth [20] has observed that in some instances, the quality of random simulation can be improved by assigning the value 1 to input variables with high probability (compare also [21]). Lu et al. [22] successfully used random simulation in their implicit learning approach which they employed in a circuit SAT solver.

Abstraction is often used as a method of problem simplification (e.g. size reduction) and can lead to performance improvements in many applications of formal verification. For example, the approach of *Counter-Example Guided Abstraction-Refinement* (CEGAR) ([23], [24]) is the foundation of the widely-used DPLL(T)-based SMT-solvers [25].

Silva et al. present a formalization of abstraction [26] that embeds CDCL in over- and under-approximating fixed-point iteration, while at the same time lifting CDCL to other problem domains (similar to constructions like DPLL(T) or SMT). They also present a language for abstraction-based algorithms that use CDCL and combine it with other theories [27].

Few approaches deal directly with the CNF problem and implement the abstraction refinement loop inside a SAT solver. Dantsin and Wolpert [28] effectively apply clause-shortening-based under-approximation in combination with local search and successive refinement to efficiently find models for SAT problems. In Cube and Conquer [29], Heule et al. use a lookahead solver to examine a problem first and later propose unit-literals to the CDCL solver. Nadel et al. [30] observed that

2375-0197/17/31.00 ©2017 IEEE
DOI 10.1109/ICTAI.2017.00158

adding assumption-variables as unit-clauses to the problem increases the effectiveness of preprocessing. This approach also shows the similarity of under-approximation and branching in the context of CDCL. While with under-approximation we explicitly modify the formula (in case of [30] by addition of unit-clauses), in branching we basically simulate that behavior by keeping track of an assignment trail.

In this paper we use gate recognition on CNF formulas as a basis for circuit-level random simulation. Based on a huge number of partial assignments, that are traversed in a random simulation step, conjectures about literal equivalences and backbone variables are deduced. Conjectures are a set of constraints that not necessarily hold, but that we might assume to hold. We implemented our ideas on top of the well-known Glucose solver [31]. In our first approach we use these conjectures for under-approximation in an abstraction refinement loop, where we rely on incremental solving with assumptions ([32], [30]). In our second approach we use the conjectures to direct a novel branching heuristic. The methods presented in this paper have been studied extensively by Felix Kutzner in his diploma thesis [33]. We observe performance improvements, especially on circuit equivalence checking problems.

II. Theoretical Background

We use a Boolean algebra with variables V and the common operators $\wedge, \vee, \neg$ with their usual semantics. A literal l is either a variable v or its complement $\neg v$. The complement of a literal l is $\bar{l} = \neg v$ if $l = v$, and $\bar{l} = v$ if $l = \neg v$. Given a Boolean formula F, $\mathsf{vars}(F)$ denotes the set of variables in F and $\mathsf{lits}(F)$ the set of literals that can be constructed from $\mathsf{vars}(F)$. A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of disjunctions of literals. A CNF formula F is represented by a set of clauses, where each clause is a set of literals.

Given a CNF formula F an assignment is a set of literals $a \subseteq \mathsf{lits}(F)$ such that $l \in a \implies \bar{l} \notin a$. An assignment a is complete iff $|a| = |\mathsf{vars}(F)|$. Given an assignment a, the interpretation function I_a is defined for literals, clauses and CNF formulas, and maps them to the values true ($\top$), false ($\bot$), or undefined ($\mathbf{U}$). For $l \in \mathsf{lits}(F)$, $I_a(l) = \top$ if $l \in a$, $I_a(l) = \bot$ if $\bar{l} \in a$ and $I_a(l) = \mathbf{U}$ otherwise. Given a clause $c \subseteq \mathsf{lits}(F)$, $I_a(c) = \top$ if $\exists l \in c. I_a(l) = \top$ and $I_a(c) = \bot$ if $\forall l \in c. I_a(l) = \bot$ and $I_a(c) = \mathbf{U}$ otherwise. Given a CNF formula $F = \{c_i | c_i \subseteq \mathsf{lits}(F)\}$, $I_a(F) = \top$ if $\forall c \in F. I_a(c) = \top$ and $I_a(F) = \bot$ if $\exists c \in F. I_a(c) = \bot$ and $I_a(F) = \mathbf{U}$ otherwise.

Given two clauses c_1 and c_2 and a literal l, such that $l \in c_1$ and $\bar{l} \in c_2$, the *resolvent* $c_1 \otimes_l c_2$ is the clause $(c_1 \cup c_2) \setminus \{l, \bar{l}\}$. The resolvent $c_1 \otimes_l c_2$ is a consequence of c_1 and c_2, i.e., $c_1, c_2 \models c_1 \otimes_l c_2$ holds. For sets of clauses C_1 and C_2, the set $C_1 \otimes_l C_2$ denotes the set of all resolvents between clauses in C_1 and C_2 on l.

Given a CNF C, a clause $c \in C$ is *blocked* in C if there exists a literal $l \in c$ such that for every clause $d \in \{c \in C \mid l \in c\}$ the resolvent $c \otimes_l d$ is tautological. The literal l is also called the *blocking literal* of c. A set of clauses $D \subseteq C$ is blocked iff each clause D is blocked in C.

A. Gate Recognition

For an n-ary Boolean function $g : \mathbb{B}^n \to \mathbb{B}$, a *gate* G is a triple $G = (g, o, P)$, where o is a Boolean variable for the output of the function, and $P = (p_1, \ldots, p_n)$ is a tuple of n Boolean variables for the inputs of function g. For every gate $G = (g, o, P)$, a *gate encoding* Γ_G is a formula, such that Γ_G is equivalent to $o \leftrightarrow g(p_1, \ldots, p_n)$. A *partial gate encoding* for a gate G is a Boolean formula F, such that $\Gamma_G \models F$. For example $F = o \rightarrow g(p_1, \ldots, p_n)$ is a partial gate encoding for gate $G = (g, o, P)$. If the formula F of a (partial) gate encoding is in CNF, we call it a *(partial) CNF encoding* of gate G. Given a (partial) CNF encoding F with $\mathsf{vars}(F) = \{o, p_1, \ldots, p_n\}$, then F is a blocked set with blocking variable o (proof in [17]).

Often the gates we consider are for simple Boolean functions, such as binary AND, OR, or XOR.

A *gate structure* $S = (\mathbb{G}, N)$ is a set of gates $\mathbb{G} = \{G_1, \ldots, G_m\}$ with $G_i = (g_i, o_i, P_i)$, distinct output variables (i.e. $o_i \neq o_j$ for $i \neq j$), and a nesting relation $N : G \times G$ that establishes a strict partial order ($\leq$) on the gates. A gate $G_1 = (g_1, o_1, P_1)$ is *directly nested* in a gate $G_2 = (g_2, o_2, P_2)$, i.e. $(G_1, G_2) \in N$, or $G_1 \leq G_2$, iff $o_1 \in P_2$. G_1 is *nested* in G_2 if (G_1, G_2) is in the transitive closure of N. A variable p is input to the structure if there is no o_i with $o_i = p$. A gate $G = (g, o, P)$ is a *root gate* in a gate structure if it is not nested in any other gate of the structure.

A gate $G = (g, o, P)$ is nested monotonic in a gate structure $\mathbb{G}$, if it is either a root gate in $\mathbb{G}$ or if for all $G' \in \mathbb{G}$ with $(G', G) \in N$, G' is nested monotonic and the output o is a monotonic argument to the corresponding function g' of G'. Otherwise, G is nested non-monotonic.

Example:

$$\mathbb{G} = \{G_1, G_2, G_3\}, G_1 \leq G_2 \leq G_3$$
$$G_1 = (\wedge, o_1, \{o_2, o_3\})$$
$$G_2 = (\oplus, o_2, \{o_3, p_1\})$$
$$G_3 = (\vee, o_3, \{p_2, p_3\})$$

In the example we have three gates $G_1, \ldots G_3$ with Boolean functions AND ($\wedge$), XOR ($\oplus$), and OR ($\vee$), respectively. G_2 and G_3 are both nested in G_1 whereas G_2 is directly nested in G_1 and G_3 is directly nested in G_2. G_2 is nested monotonic in G_1 as g_1 is a monotonic function, and G_1 is root gate of the structure. G_3 is nested non-monotonic as g_2 (logical xor) is a non-monotonic function.

B. Random Simulation

Applying random simulation to a gate structure means repeatedly assigning random input vectors to its input variables and subsequently propagating the assignments to all outputs in the gate structure. From the resulting sequence of gate output assignments, conjectures about equivalent or constant gate outputs can be deduced efficiently. Random Simulation

generates a set of conjectures C that can be used to speedup the subsequent search.

C. Abstraction via Under-Approximation

Abstraction is a general method to simplify a problem (in our case a Boolean formula) in order to reduce the problem's size and thus diminish the state space explosion problem. Abstraction can be used to remove details or possibly irrelevant components from a formula. Solving the abstraction of a problem might however produce results not holding for the original problem. Consequently, it is necessary to maintain knowledge about how the abstraction is related to the original problem. We focus here on an abstraction method known as *under-approximation*.

Definition 1 (Under-Approximation): Given a Boolean formula F and its abstraction F', then F' is an under-approximation of F iff $F' \models F$. Note that the solution space of F' is smaller or equal than that of F ($\#F' \leq \#F$).

Example: Methods to derive an under-approximation F' from a given formula F include the addition of further clauses or the removal of literals from given clauses. Given the formula $F = \{\{a, b, c\}, \{a, \neg b, \neg c\}, \{d\}\}$, then both $F' = \{\{a, b, c\}, \{a, \neg b, \neg c\}, \{d\}, \{\neg a, c\}\}$ as well as $F' = \{\{a, b, c\}, \{a, \neg b\}, \{d\}\}$ are both under-approximations of F.

If F' is derived to be unsatisfiable, additional checks have to be executed in order to verify that the result also holds for F. If this is not the case, then the abstraction has to be refined (e.g. by removing added constraints) and the search has to be restarted with the updated F'. In our scenario, an under-approximation F' of F is formed by addition of clauses C such that $F' = F \wedge C$. We start with an initial abstraction $F' = C_0 \wedge F$ and in each refinement step $i > 0$, a weaker constraint C_i with $C_i \models C_{i-1}$ is created such that the refined abstraction is $F' = C_i \wedge F$. A fixed point is reached when we solve the original problem with $C_i = \emptyset$ such that $F' = F$.

The abstraction-search-refinement loop of this abstraction scheme is similar to the branch-propagate-backtrack loop of CDCL, where branching can be regarded as a way of under-approximating the problem by adding a unit-clause that can be easily removed in order to facilitate backtracking.

III. Approach

In both approaches we start with problem analysis (see section III-A1) in order to generate conjectures which we later use in different ways. To this end, we use a variant of the gate extraction algorithm we described in [17]. The extracted gates are then used by a random simulation procedure that produces a set of conjectures about equivalent and backbone literals.

In our first approach (section III-B), such conjectures are used in an abstraction-refinement loop. In our second approach (section III-C) the conjectures are used in a modified branching heuristic in order to stimulate clause learning.

A. Problem Analysis

Given a SAT problem F to solve, we start with problem analysis consisting of a gate extraction phase and a random simulation phase. These initial phases are used to generate a set of problem-specific conjectures about literal equivalence and backbone literals.

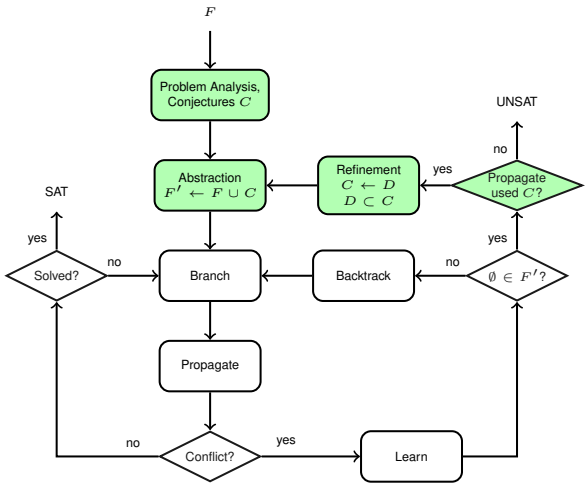

Fig. 1. Model of an under-approximating SAT solver

1) Gate Extraction: Our heuristic gate extraction algorithm described in [17] can detect gate structure in CNF problems of numerous applications as it is tailored specifically to the widely used Tseitin and related encodings. It uses the property that each encoding is a blocked set and extracts gate structure starting from a heuristically determined root (usually a unit clause) and subsequently detects nested gates based on a combination of checks for locally blocked sets and clausal patterns (see [17] for details).

2) Random Simulation: We deterministically generate pseudo-random bit-vectors (one bit for each input variable of the previously recovered gate structure) such that each bit holds the value of an input variable. From these random input variable assignments, we determine complete assignments of gate outputs by bit-parallel propagation within the gate structure. Our propagation algorithm is an optimized adaption of the `GetMultipleSolutions` algorithm in [15]. We repeat this process for a finite amount of steps. The amount of simulation steps is determined by a given maximum. Furthermore we quit random simulation if the amount of changes to conjectures falls below a given threshold. The bit-vectors are generated with a circularly changing bias towards 0 rsp. 1.

During random simulation, conjectures are maintained by iteratively computing hash values of each output variable's sequence of assignments. For gate outputs having the same value in each step we generate a backbone conjecture. For gate outputs that are equal (not equal) in each step we generate an equivalence (xor) conjecture.

We describe the set of conjectures by a set of non-empty sets of literals $C = \{c_1, c_2, \ldots, c_k\}$. A conjecture c_i is a backbone conjecture if $|c_i| = 1$ and an equivalence conjecture otherwise. A backbone conjecture $c = \{l\}$ is satisfied

Algorithm 1: RSAR

```
   Data: F : CNF formula
   Result: R : SAT or UNSAT
1  gates ← extractGates(F)
2  conjectures ← randomSimulation(gates)
3  i ← 0
4  conjectures ← conjectures \ select_0(conjectures)
5  (E, A) ← encode(conjectures)
6  F ← F ∪ E
7  while TRUE do
8      (R, usedConjectures) ← search(F, A)
9      if R = UNSAT then
10         if usedConjectures ≠ ∅ then
11             i ← i + 1
12             remove ← usedConjectures ∪ select_i(conjectures)
13             (E, A) ← refine(conjectures, remove)
14             conjectures ← conjectures \ remove
15             F ← F ∪ E
16         else
17             return UNSAT
18     else
19         return SAT
```

under an assignment a if evaluation function $I_a(l) = \top$. An equivalence conjecture $c = \{l_1, \ldots, l_n\}$ is satisfied under an assignment a if it either hold that $\forall l_i \in c, I_a(l_i) = \top$ or that $\forall l_i \in c, I_a(l_i) = \bot$.

B. Abstraction Refinement

Figure 1 shows a general model of a SAT solver using abstraction to under-approximate a problem with conjectures. The conjectures are generated in an initial analysis step, in which we perform *gate extraction* and *random simulation*. Then, we create an abstraction of the original problem by adding CNF encoded conjectures to the problem and starting CDCL search. If a solution is found, this solution also holds for the original problem and the problem is satisfiable. If the abstraction is unsatisfiable, we check if one of the added conjectures was used in the proof of unsatisfiability and refine the set of conjectures if necessary.

1) Conjecture Encoding: In order to under-approximate a given CNF formula F, we have to encode each conjecture $c \in C$ in CNF. We add a unique assumption literal to each clause of a conjecture's encoding, such that we can enable and disable conjectures by appropriately setting the values of its assumption variables. We also use the assumption literals to check if conjectures have been used in derivation of UNSAT by checking the clauses that were reason to the final propagations (compare [34]).

For backbone conjectures $c = \{l\}$, the CNF encoding is given by $\mathsf{enc}(c) = \{\{l, a\}\}$ with a unique assumption variable a. For equivalence conjectures $c = \{l_1, l_2 \ldots, l_m\}$ we encode the set of implications $l_1 \rightarrow l_2 \rightarrow \cdots \rightarrow l_m \rightarrow l_1$ using $\mathsf{enc}(c) = \{\{\bar{l}_1, l_2, a_1\}, \{\bar{l}_2, l_3, a_2\}, \ldots \{\bar{l}_{m-1}, l_m, a_{m-1}\}, \{\bar{l}_m, l_1, a_m\}\}$ with unique assumption literals $a_1 \ldots a_m$.

During refinement we disable conjectures for a set of variables $S \subset \mathsf{vars}(F)$. For all $v \in S$ we update the encoding of conjectures c with $v \in c$ or $\neg v \in c$. If c is a backbone conjecture or an equivalence conjecture with $|c| = 2$, we just disable all clauses in $\mathsf{enc}(c)$. If c is an equivalence conjecture with $|c| > 2$, we only disable the clauses that constrain v and add an additional clause with a new assumption literal such that the encoding of the conjecture about the remaining literals is preserved.

Example: Given the conjecture $c = \{v_1, v_2, v_3\}$, its encoding $\{\{\neg v_1, v_2, a_1\}, \{\neg v_2, v_3, a_2\}, \{\neg v_3, v_1, a_3\}\}$ with assumptions $A = \{\neg a_1, \neg a_2, \neg a_3\}$ and the set of variables to deactivate $D = \{v_2\}$. Then refinement generates a new clause $\{\neg v_1, v_3, a_4\}$ and a new set of assumptions $A = \{a_1, a_2, \neg a_3, \neg a_4\}$.

2) Random-Simulation-based Abstraction Refinement (RSAR): Algorithm 1 describes the procedure with more details. In line 1, the gate structure is determined and in line 2, the conjectures are generated by random simulation. In line 4, we remove conjectures from the initial set by heuristic selection (described in the follow-up section) that becomes more rigorous in each refinement step (lines 11 and 12), which is why we introduced the counter in line 3. In line 5, the conjectures are encoded as clauses E, and a set of assumptions A is returned that initally activates all the conjectures encoded in E. In line 8, the CDCL solver is executed on the modified formula F, using assumptions A.

The result R can be either SAT or UNSAT. If R is UNSAT, the set of conjectures used in the proof is stored in usedConjectures, which is otherwise empty. In line 12, a set of conjectures to remove is determined such that it contains usedConjectures as well as the conjectures determined by heuristic selection (see section III-B3). The conjectures are reencoded in line 13 such that the conjectures to remove are deactivated by a new set of assumptions A and the remaining conjectures are preserved by clauses E. The procedure continues until a solution is found (line 19) or no conjecture was needed for determining that F is unsatisfiable (line 17).

3) Heuristic Conjecture Removal: Among the conjectures that are calculated by random simulation, subsequently those have to be removed during problem refinement, which are not implied by the original problem (false positives). In this paragraph, the number of input variables that is used by random simulation to derive a conjecture c is denoted by $\mathsf{n}(c)$. A priori, the likelihood that a given conjecture c is false positive (under a fixed amount of simulation steps) correlates positively with $\mathsf{n}(c)$. The function select_i (see algorithm 1, lines 4 and 12), which is configured by an array of integers $(a_0, \ldots, a_n)$, determines additional conjectures to be removed in each refinement step. Each integer a_i specifies the maximum $\mathsf{n}(c)$ for each refinement step i, such that each conjecture c with $\mathsf{n}(c) < a_i$ must be removed in refinement step i. Note that step select_0 is already applied in an initalization step (algorithm 1, line 4). We experimented with several configurations of this refinement heuristic (see section IV).

C. Implicit Learning

In our implicit learning approach, we use the conjectures in a specialized branching heuristic for CDCL solvers. For

Algorithm 2: RSIL Branching

```
Data: F : CNF formula, C : Conjectures, T : Newest Partial Assignment
Result: l : Branching Literal
// p is globally initialized with 1
1  if randomNumber(0, 1) ≤ p then
       // i is globally initialized with 0
2      i ← i + 1
       // bound is initialized by global heuristic
          configuration
3      if i > bound then
4          bound ← bound + bound/2
5          p ← p/2
6      for c ∈ C do
7          d ← vars(c) ∩ vars(T)
8          if d ≠ ∅ ∧ |d| < |c| then
               // Violate conjecture c
9              v ← pickOne(vars(c) \ d)
10             r ← pickOne(d)
11             if (r ∈ c ≡ r ∈ T) then
12                 return v ∈ C ? ¬v : v
13             else
14                 return v ∈ C ? v : ¬v
15 else
16     return branchDefault
```

each branching decision, if a variable that is constrained by a conjecture c_i has been assigned in the current decision level, we prefer to pick the value of an unassigned variable $v \in c_i$ such that c_i is violated. We do this to provoke a conflict, hoping to guide the SAT solver to quickly learn the conjecture through clause learning in case the conjecture holds.

Random-Simulation-based Implicit Learning (RSIL): Algorithm 2 describes the branching algorithm of our implicit learning approach. In line 1, we deterministically calculate a random number in order to determine if the solver should use the modified branching in the current execution of the branching method. Otherwise, it falls back to default branching of the solver (line 16). We keep track of the number of calls (line 2) and if that count reaches a pre-configured bound the call propability is decreased and the bound increased (lines 3 to 5).

In the modified branching, we iterate over all equivalence conjectures (line 6) and check if any variable of the conjecture was assigned by the given partial assignment T (lines 7 and 8). If this is the case and there are still variables of the conjecture unassigned (line 8), we select an unassigned variable v and an assigned variable r of the conjecture c (lines 9 and 10). In lines 11 to 14, we pick a value for v such that it violates the conjectured value determined by r.

IV. Experimental Results

We evaluated the methods we presented in sections III-B and III-C using the application track benchmark set of SAT Competition 2014 (`Comp2014`) and a set of unsatisfiable circuit equivalence checking problems (`Miter`, all unsatisfiable) by Armin Biere [35]. For our experiments, we used a computer cluster with each node equipped with 2 Intel Xeon E5430 CPUs running at 2.66 GHz and 32 GB of RAM. The cluster's operating system is OpenSuSE 11.1 Linux 64 bit. We ran each process with a CPU time limit of 5000 seconds and a memory limit of 8 GB and executed at most two SAT solver processes per compute node in parallel.

TABLE I
Benchmark problem sets

Name	Amount of problems	
	Satisfiable	Unsatisfiable
`Comp2014`	59	124
`miter`	0	341

We implemented our algorithms on top of the `Glucose 3` SAT solver and use the runtime of `Glucose` as a baseline for comparison to our approach. In both benchmark sets `Comp2014` and `Miter`, we include only those problems where gate recognition could extract at least 10 gates. The sizes of the benchmark sets we thereby obtained are shown in Table I.

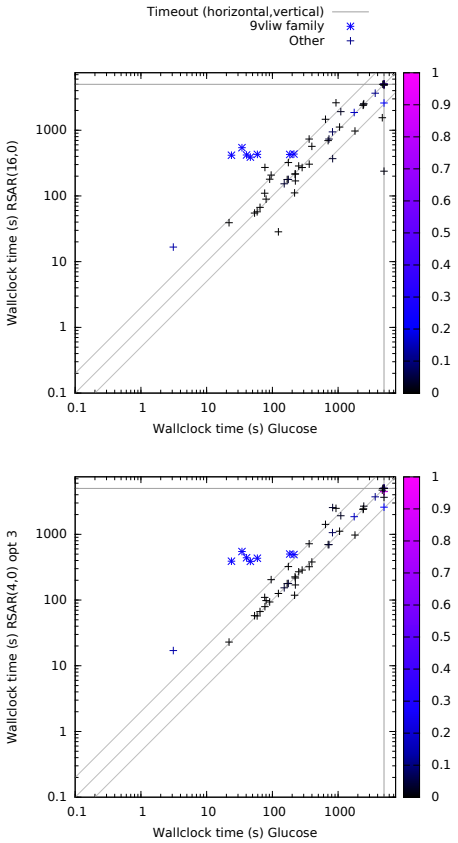

Fig. 2. Runtime scatter plots of RSAR Variant A with select_i-configuration $(16, 0)$ (top) and Variant B with select_i-configuration $(4, 0)$ (bottom), both on satisfiable instances in `Comp2014` (colors indicate the fraction of gate outputs relative to the total number of variables)

A. Results for the RSAR Algorithm

In our experiments with the RSAR algorithm, we used a maximum number of random simulation rounds of 2^{20}. We used a harsh refinement heuristics for select_i which removes

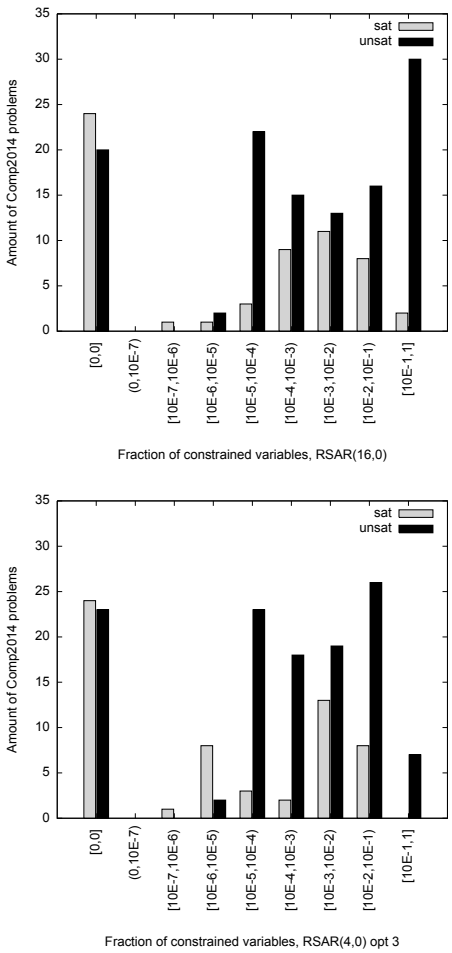

Fig. 3. Histograms showing the distribution of the amounts of variables constrained by conjectures used by RSAR to construct the first approximation, relative to the total amount of variables occurring in the respective problem. The data is shown for RSAR Variant A with select_i-configuration $(16, 0)$ (top) and Variant B with select_i-configuration $(4, 0)$ (bottom)

all conjectures in the first refinement step. In variant A we use this heuristic in conjunction without further restriction on the conjectures and in variant B we only use conjectures of size smaller than 4 (denoted by "`opt 3`" in figures 2 and 3). The reason for the additional size constraint in variant B is our assumption that small conjectures are more likely to hold.

Figure 3 shows the amounts of `Comp2014` problems grouped by the fraction of variables additionally constrained by the first approximation computed by the RSAR algorithm. The figure gives a comparison of variant A employed with the $\mathtt{select}_i$-configuration $(16, 0)$ with variant B employed with the $\mathtt{select}_i$-configuration $(4, 0)$. The data shows that using harsher refinement heuristics may cause significantly fewer variables to be constrained due to the conjectures used by RSAR.

Table II shows the results for the instances of the `Comp2014` benchmark set. In the first column, the heuristic configuration of select_i indicates the number of maximum input dependencies allowed in each refinement step (0 means removal of all conjectures). The second and third column specify the number of solved instances for each variant. While Glucose executed in incremental mode without abstraction solves 47 of the given satisfiable instances, using RSAR it could solve up to 3 more satisfiable instances.

However, RSAR did not work particularly well on unsatisfiable instances in `Comp2014`. The bad runtimes that RSAR produced here could not be mitigated by the harsh heuristic configurations. For example, in variant B with $\mathtt{select}_i$-configuration $(4, 0)$ the approach solved only 86 unsatisfiable instances, while Glucose solved 92 instances within the runtime and memory limit.

Figure 2 shows the runtime scatter plots for two heuristic configurations of the RSAR algorithm. Although we observe less timeouts with RSAR, the runtime distribution is mostly noisy. We highlighted the deteriorating runtimes on the `9vliw` family of problems contained in `Comp2014`. Curiously, we could observe that pattern for `9vliw` in all the experiments that we conducted (see also section IV-B).

TABLE II
AMOUNT OF PROBLEMS SOLVED BY RSAR ON THE INSTANCES OF `Comp2014`

	Variant A		Variant B	
select_i	SAT	UNSAT	SAT	UNSAT
$(2, 0)$	48 (+1)	86 (-6)	49 (+2)	87 (-5)
$(4, 0)$	48 (+1)	86 (-6)	**50** (+3)	86 (-6)
$(8, 0)$	48 (+1)	81 (-11)	49 (+2)	86 (-6)
$(16, 0)$	49 (+2)	79 (-13)	49 (+2)	88 (-4)
$(32, 0)$	49 (+2)	72 (-20)	49 (+2)	83 (-9)
$(64, 0)$	49 (+2)	74 (-18)	49 (+2)	76 (-16)

B. Results for the RSIL Algorithm

In our experiments with the RSIL algorithm, we used a maximum number of random simulation rounds of 2^{17}. We also only use conjectures with a maximum size of 3. Table III shows the number of solved instances for several bounds of heuristic configuration of RSIL. Best results could be achieved with a bound of 10^6 where RSIL solves 1 more unsatisfiable instance in the `Comp2014` benchmark set and 7 more instances in the `Miter` benchmark set.

Figure 4 shows two runtime scatter plots for bound 10^6 and 10^7 on the `Comp2014` benchmark set. It shows particularly good runtimes on circuit equivalence problems. However, the performance on the `9vliw` family of problems included in `Comp2014` deteriorated in any scenario we used in our experimentation (see also section IV-A).

TABLE III
AMOUNT OF PROBLEMS SOLVED BY RSIL ON INSTANCES OF `Comp2014` AND `Miter`

bound	`Comp2014/sat`	`Comp2014/unsat`	`Miter`
10^3	49 (0)	92 (−1)	316 (0)
10^4	49 (0)	93 (0)	316 (0)
10^5	49 (0)	93 (0)	320 (+4)
10^6	49 (0)	**94** (+1)	**323** (+7)
10^7	49 (0)	91 (−2)	**323** (+7)

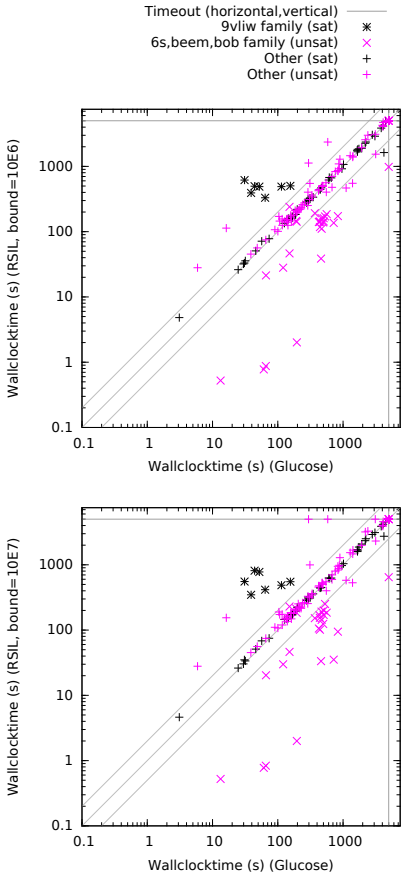

Fig. 4. Runtime scatter plots of RSIL with bound 10^6 (top) and with bound 10^7 (bottom), both on all instances in `Comp2014`

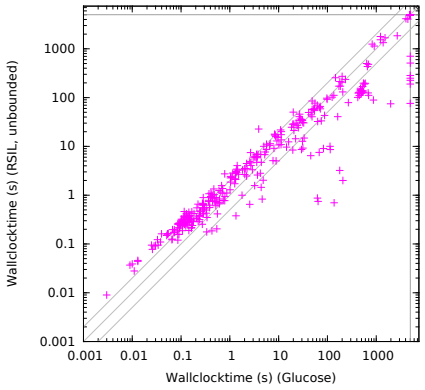

Fig. 5. Runtime scatter plot of RSIL without bound on all instances in `Miter`

Figure 5 shows a runtime scatter plot for unbounded RSIL on the `Miter` benchmark set, where the method produces exceptionally good results. For the lower runtimes (a few seconds and less) the overhead incurred through the problem analysis step is clearly noticable in the plot. However, the advantage of the new branching method RSIL becomes clear-cut in the long running benchmarks.

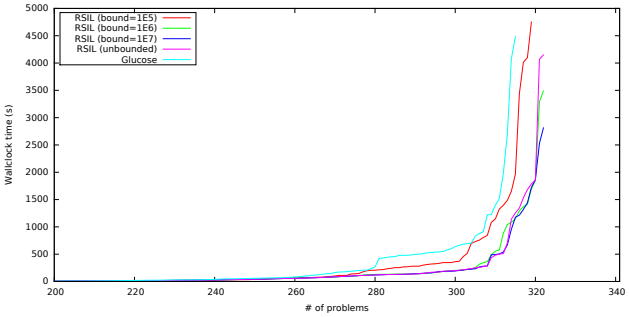

Fig. 6. Runtime cactus plot of Glucose and RSIL with several bounds on all instances in `Miter`

The cactus plot in figure 6 summarizes the runtimes of RSIL configured with several bounds on `Miter`. Clearly, no mitigation through bounding of the method is needed for the `Miter` set of instances.

V. Conclusion and Future Work

We showed that explicit exploitation of recovered gate structure in SAT problems can be used effectively to improve the runtime of SAT solvers on many application instances. In contrast to previous approaches that use gate extraction and an external tool-chain for reencoding the CNF ([16], [17]), we include the cost of the preprocessing in the runtime evaluation. Our solver includes all the structure-recovery and exploitation and does not rely on external circuit simplification tools.

Among the presented approaches, RSIL showed the best overall results regarding runtime improvements (specifically on unsatisfiable `Miter` instances), while the results obtained for RSAR only show positive results on satisfiable problem instances. Regarding branching as a specialized form of abstraction is fundamental to a new approach that combines implicit learning and under-approximation in a complementary way. The ongoing tighter integration of the presented methods into a new SAT solver and further experiments with the presented algorithmic framework should lead to better overall results.

In the presented work, runtime deteriorations have generally been mitigated with more restricive configurations of conjecture removal and refinement heuristics. In future work we investigate distinctive properties of those CNF problems where solver performance benefits from less restrictive heuristic configurations.

We also plan to increase the usage of conjectures about problem structures such that more expressive conjectures (e.g. cardinality constraints [36]) can be derived in a preceding problem analysis step, which can then be exploited by specialized propagators (and might become subject to refinement).

References

[1] S. Falke, F. Merz, and C. Sinz, "LLBMC: Improved bounded model checking of C programs using LLVM - (competition contribution)," in *TACAS*, 2013, pp. 623–626.

[2] J. Rintanen, "Engineering efficient planners with SAT," in *ECAI*, 2012, pp. 684–689.

[3] R. Alves, F. Alvelos, and S. D. Sousa, "Resource constrained project scheduling with general precedence relations optimized with SAT," in *Progress in Artificial Intelligence*. Springer, 2013, pp. 199–210.

[4] A. Mihal and S. Teig, "A constraint satisfaction approach for programmable logic detailed placement," in *SAT*, 2013, pp. 208–223.

[5] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, 2007, pp. 632–647.

[6] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic*, A. O. Slisenko, Ed., 1970, pp. 115–125.

[7] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation." *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.

[8] T. Boy de la Tour, "An optimality result for clause form translation," *J. Symb. Comput.*, vol. 14, no. 4, pp. 283–301, Oct. 1992.

[9] P. Jackson and D. Sheridan, "Clause form conversions for boolean circuits," in *7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004, pp. 183–198.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 530–535.

[11] C. Thiffault, F. Bacchus, and T. Walsh, "Solving non-clausal formulas with DPLL search," in *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, ser. CP'04. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 663–678.

[12] M. Järvisalo, A. Biere, and M. J. Heule, "Simulating circuit-level simplifications on CNF," *Journal of Automated Reasoning*, vol. 49, no. 4, pp. 583–619, 2012.

[13] O. Kullmann, "On a generalization of extended resolution," *Discrete Applied Mathematics*, vol. 96, pp. 149 – 176, 1999.

[14] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, 2010, pp. 129–144.

[15] M. Heule and A. Biere, "Blocked clause decomposition," in *Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, K. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer Berlin Heidelberg, 2013, pp. 423–438.

[16] T. Balyo, A. Fröhlich, M. Heule, and A. Biere, "Everything you always wanted to know about blocked sets (but were afraid to ask)," in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, pp. 317–332.

[17] M. Iser, N. Manthey, and C. Sinz, "Recognition of nested gates in CNF formulas," in *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, 2015, pp. 255–271.

[18] F. Krohm, A. Kuchlmann, and A. Mets, "The use of random simulation in formal verification," in *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, ser. ICCD '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 371–.

[19] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 50–57.

[20] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Boston: Addison-Wesley Professional, 2015.

[21] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[22] F. Lu, L.-C. Wang, J. Moondanos, and Z. Hanna, "A signal correlation guided circuit SAT solver," vol. 10, no. 12, pp. 1629–1654, 2004.

[23] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of the 12th International Conference on Computer Aided Verification*, ser. CAV '00. London, UK, UK: Springer-Verlag, 2000, pp. 154–169.

[24] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[25] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *CAV*, 2004, pp. 175–188.

[26] V. D'Silva, L. Haller, and D. Kroening, "Abstract conflict driven learning," *SIGPLAN Not.*, vol. 48, no. 1, pp. 143–154, Jan. 2013.

[27] ——, "Abstract satisfaction," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 139–150.

[28] E. Dantsin and A. Wolpert, "A faster clause-shortening algorithm for SAT with no restriction on clause length," *JSAT*, vol. 1, no. 1, pp. 49–60, 2006.

[29] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, ser. HVC'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 50–65.

[30] A. Nadel and V. Ryvchin, "Efficient SAT solving under assumptions," in *SAT*, 2012, pp. 242–255.

[31] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[32] G. Audemard, J.-M. Lagniez, and L. Simon, "Improving glucose for incremental sat solving with assumptions: Application to mus extraction," in *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 309–317.

[33] F. Kutzner, "Exploiting gate-structure to direct CDCL search via variable selection and approximation," Diplomarbeit, Karlsruhe Institute of Technology, June 2016.

[34] J. Lagniez and A. Biere, "Factoring out assumptions to speed up MUS extraction," in *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, 2013, pp. 276–292.

[35] H. Biere and M. Järvisalo, "Equivalence checking of HWMCC 2012 circuits," in *Proceedings of SAT Competition 2013*, vol. B-2013-1. University of Helsinki: Department of Computer Science, 2013, p. 104.

[36] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, "Detecting cardinality constraints in CNF," in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer International Publishing, 2014, pp. 285–301. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09284-3_22