Probing-Based Preprocessing Techniques for Propositional Satisfiability

Inês Lynce and João Marques-Silva
IST/INESC-ID, Technical University of Lisbon, Portugal
{ines,jpms}@sat.inesc.pt

Abstract

Preprocessing is an often used approach for solving hard instances of propositional satisfiability (SAT). Preprocessing can be used for reducing the number of variables and for drastically modifying the set of clauses, either by eliminating irrelevant clauses or by inferring new clauses. Over the years, a large number of formula manipulation techniques has been proposed, that in some situations have allowed solving instances not otherwise solvable with state-of-the-art SAT solvers. This paper proposes probing-based preprocessing, an integrated approach for preprocessing propositional formulas, that for the first time integrates in a single algorithm most of the existing formula manipulation techniques. Moreover, the new unified framework can be used to develop new techniques. Preliminary experimental results illustrate that probing-based preprocessing can be effectively used as a preprocessing tool in state-of-the-art SAT solvers.

1. Introduction

Propositional Satisfiability (SAT) is a well-known NP-complete problem, with extensive applications in many fields of Computer Science and Engineering. SAT has been the subject of intensive research in recent years, with significant theoretical and practical contributions. From a practical perspective, several competing solution strategies for SAT have been proposed. Local search algorithms have allowed solving extremely large satisfiable instances of SAT, and in particular have been shown to be very effective in randomly generated instances of SAT. On the other hand, several improvements to the backtrack search Davis-Putnam-Logemann-Loveland algorithm have been introduced, thus allowing to solve either satisfiable and unsatisfiable instances. These new backtrack search algorithms utilize advanced conflict analysis procedures, that record the causes of failures and that can therefore backtrack non-chronologically.

In addition, there have been significant contributions in terms of formula manipulation techniques which can in some cases yield competitive approaches [2, 3, 4, 7, 8]. It is generally accepted that the ability to reduce either the number of variables or clauses in instances of SAT impacts the expected computational effort of solving a given instance. This ability can actually be essential for specific and hard classes of instances. Interestingly, the ability to infer *new* clauses *may* also impact the expected computational effort of SAT solvers. Observe that these new clauses can be useful for reducing the number of variables (and consequently the number of clauses).

This paper proposes the utilization of *probing-based techniques* for manipulating propositional satisfiability formulas. Probing allows the formulation of hypothetical scenarios, obtained by assigning a value to a variable, and then applying unit propagation. Furthermore, probing-based techniques build upon a *very simple* idea: a table of *triggering assignments*, which registers the result of applying probing to every variable in the propositional formula.

The new probing-based approach not only provides a generic framework for applying different SAT preprocessing techniques (by establishing conditions on the entries of the table of assignments), but also can be used to implement most existing formula manipulation techniques. Moreover, and to our best knowledge, represents the first approach to *jointly* apply variable and clause probing.

The paper is organized as follows. The next section describes the proposed framework, which allows the integration of existing formula manipulation techniques. Next, experimental results are presented and analyzed. Finally, Section 4 overviews related work and section 5 concludes the paper.

2. Integrating Probing-Based Techniques

In this section we propose an integrated approach for implementing probing-based techniques. We start by describing the table of assignments, which records the prob-

ing results. In addition, we establish reasoning conditions for identifying necessary assignments and inferring new clauses. Finally, we present *ProbIt*: a probing-based preprocessor for propositional satisfiability.

2.1. Preliminaries

In what follows we analyze conditions relating sets of assignments. Given an assignment α [1] and a formula φ, the result of applying Boolean Constraint Propagation (the iterative appliance of unit propagation) to φ given α is denoted by $\mathrm{BCP}(\varphi, \alpha)$. When clear from the context, we use the notation $\mathrm{BCP}(\alpha)$, and the existence of the CNF formula φ is implicit. Without assignment, $\mathrm{BCP}(\varphi)$ (or $\mathrm{BCP}()$) denotes plain unit propagation, given the existence of unit clauses. Assignment α is referred to as the *triggering assignment* of the assignments in $\mathrm{BCP}(\alpha)$. We may also use the notation $\mathrm{BCP}(A)$ to denote the result of applying unit propagation as the result of *all* assignments in the set of assignments A.

Reasoning conditions are analyzed based on a tabular representation of triggering assignments, i.e. the *table of assignments*, where each row represents a triggering assignment, and each column represents a possible implied assignment [2]. In this table, each 1-valued entry (α_t, α_i) denotes an implied assignment α_i given a triggering assignment α_t. Hence the 1-valued entries of a given row α_t denote the elements of set $\mathrm{BCP}(\alpha_t)$.

2.2. Motivating Examples

This section analyzes a few examples, that motivate the techniques upon which our framework is based, and which allow the identification of necessary assignments and the inference of new clauses. Consider a CNF formula φ having the following clauses:

$\omega_1 = (a \vee b)$ $\quad$ $\omega_5 = (\neg a \vee \neg c \vee d)$
$\omega_2 = (a \vee \neg b \vee d)$ $\quad$ $\omega_6 = (\neg a \vee \neg d \vee e)$
$\omega_3 = (a \vee \neg c \vee e)$ $\quad$ $\omega_7 = (c \vee e)$
$\omega_4 = (\neg a \vee c)$

This formula has the the table of assignments shown in Table 1. Each line in the table corresponds to the result of applying BCP, given a triggering assignment. For example, given the assignment a=0 (denoted as $\langle a, 0\rangle$), we can conclude from the table that BCP($\langle a, 0\rangle$) = $\{\langle a, 0\rangle, \langle b, 1\rangle, \langle d, 1\rangle\}$.

2.2.1. Necessary Assignments Observe that for the two possible assignments to variable a, we always obtain the implied assignment $\langle d, 1\rangle$. Since variable a must assume one of the two possible assignments, then the assignment $\langle d, 1\rangle$ is deemed necessary. This conclusion is represented as $\mathrm{BCP}(\langle a, 0\rangle) \cap \mathrm{BCP}(\langle a, 1\rangle) = \langle d, 1\rangle$ [3]

	$\bar{a}$	a	$\bar{b}$	b	$\bar{c}$	c	$\bar{d}$	d	$\bar{e}$	e
$\bar{a}$	1			1				1		
a		1				1		1		1
$\bar{b}$		1	1			1		1		1
b				1						
$\bar{c}$	1			1	1			1		1
c						1				
$\bar{d}$							1			
d								1		
$\bar{e}$		1				1	1	1	1	
e										1

Table 1. Table of assignments

The same conclusion could be achieved by considering clause $\omega_2 = (a \vee \neg b \vee d)$. Any assignment to the variables that satisfies the formula must also satisfy this clause, and so at least one of the assignments that satisfies the clause must hold. Given that in this example the three assignments that satisfy the clause also imply the assignment $\langle d, 1\rangle$, then this assignment is part of *any* assignment that satisfies the CNF formula, and so it is a necessary assignment. This conclusion is represented as $\mathrm{BCP}(\langle a, 1\rangle) \cap \mathrm{BCP}(\langle b, 0\rangle) \cap \mathrm{BCP}(\langle d, 1\rangle) = \langle d, 1\rangle$ [4]

The two previous examples concern necessary assignments conditions for *formula satisfiability*. Next we address necessary assignments conditions for preventing *formula unsatisfiability*.

First, note that the triggering assignment $\langle e, 0\rangle$ implies both $\langle d, 0\rangle$ and $\langle d, 1\rangle$, and hence a conflict is necessarily declared. As a result, the assignment $\langle e, 1\rangle$ is deemed necessary.

Another explanation for the same assignment comes from considering clause $\omega_6 = (\neg a \vee \neg d \vee e)$. The assignment $\langle e, 0\rangle$ makes this clause unsatisfied. Hence, a conflict is declared, and the assignment $\langle e, 1\rangle$ is deemed necessary. Observe that the necessary assignments obtained from unsatisfiability conditions correspond to the well-known *failed-literal rule* [5].

2.2.2. Inferred Clauses Besides the identification of necessary assignments, the table of assignments can also be used for inferring new clauses.

1 An assignment α is a pair $\langle l, v\rangle$ that denotes assigning value v to literal l.

2 In practice the table of assignments is represented as a sparse matrix and so the memory requirements are never significant.

3 This condition is the inference rule used in the Stålmarck's method [12].

4 This rule is utilized in [4] for deriving shared implications.

Let us consider the triggering assignment $\langle a, 1\rangle$ and the respective implied assignment $\langle e, 1\rangle$. Hence, the clause $(\neg a \vee e)$ can be inferred. Clearly, for each entry in the table of assignments a new binary clause can be created. In practice our goal is to be selective with which entries to utilize for inferring new clauses.

Consider clause $\omega_4 = (\neg a \vee c)$. Each assignment that satisfies clause ω_4 either implies the set of assignments $\{\langle a, 0\rangle, \langle b, 1\rangle, \langle d, 1\rangle\}$ or $\{\langle c, 1\rangle\}$. Hence, because at least one of the assignments that satisfies ω_4 must hold, the clause $(b \vee c)$ can be inferred.

In addition, observe that the triggering assignments $\langle a, 0\rangle$ and $\langle a, 1\rangle$ imply the assignments $\langle b, 1\rangle$ and $\langle c, 1\rangle$, respectively (besides other triggering assignments). Since a must be subject to one of the two possible assignments, then one of the assignments in $\{\langle b, 1\rangle, \langle c, 1\rangle\}$ must also hold, and so the clause $(b \vee c)$ can be inferred.

The previous examples illustrate how to infer clauses from *formula satisfiability* requirements. Next, we illustrate the inference of clauses from necessary conditions for preventing *formula unsatisfiability*.

First, observe that the set of assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ unsatisfy clause ω_5. As a result, the assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ must not hold simultaneously, and so the clause $(\neg a \vee d)$ can be inferred.

Alternatively, observe that the assignments in $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ imply the assignments in $\{\langle a, 1\rangle, \langle c, 1\rangle, \langle d, 0\rangle, \langle d, 1\rangle, \langle e, 1\rangle\}$, that denote an inconsistent assignment due to variable d. Hence, the assignments $\{\langle a, 1\rangle, \langle d, 0\rangle\}$ must not hold simultaneously, and so the clause $(\neg a \vee d)$ can be inferred.

2.3. Reasoning with Probing-Based Conditions

The examples of the previous section illustrate the forms of reasoning that can be performed given information regarding the assignments implied by each triggering assignment. These forms of reasoning include identification of necessary assignments and inference of new clauses. In this section we formalize different conditions, both for identifying necessary assignments and for inferring new clauses. All proposed reasoning conditions result from analyzing the consequences of assignments made to variables and of propagating those assignments with BCP.

2.3.1. Satisfiability-Based Necessary Assignments The purpose of this section is to describe the identification of necessary assignments based on formula satisfiability conditions. The first condition identifies common implied assignments given the two possible triggering assignments that can be assigned to a variable.

Theorem 2.1 *Given a CNF formula φ, for any variable x of the formula, the assignments defined by $\mathrm{BCP}(\langle x, 0\rangle) \cap \mathrm{BCP}(\langle x, 1\rangle)$ are necessary assignments.*

Any complete set of assignments to the variables that satisfies the CNF formula must assign either value 0 or 1 to each variable x. If for both assignments to x, some other variable y is implied to the same value v, then the assignment $\langle y, v\rangle$ is deemed necessary.

The second condition identifies common implied assignments given required conditions for satisfying each clause.

Theorem 2.2 *Given a CNF formula φ, for any clause ω of the formula, the assignments defined by $\bigcap_{l \in \omega} \mathrm{BCP}(\langle l, 1\rangle)$ are necessary assignments.*

Any complete set of assignments that satisfies the CNF formula must satisfy all clauses. Hence, assignments that are common to all assignments that satisfy a given clause must be deemed necessary assignments.

2.3.2. Unsatisfiability-Based Necessary Assignments We now proceed describing the identification of necessary assignments based on formula unsatisfiability conditions. As mentioned earlier, these conditions correspond to the failed-literal rule [5].

Theorem 2.3 *Given a CNF formula φ, if $\mathrm{BCP}(\langle x, v\rangle)$ yields a conflict, then the assignment $\langle x, \neg v\rangle$ is deemed necessary.*

The previous theorem includes the conditions regarding both the identification of inconsistent assignments to a variable and the identification of unsatisfied clauses. (Observe that most BCP algorithms do not distinguish between these two situations, being a conflict declared in both cases.)

2.3.3. Implication-Based Inferred Clauses As illustrated earlier, probing can also be used for inferring new clauses. One simple approach for inferring new clauses is to use each entry in the table of assignments.

Theorem 2.4 *Given a CNF formula φ, if $\langle l_2, 1\rangle \in \mathrm{BCP}(\langle l_1, 1\rangle)$, then the clause $(\neg l_1 \vee l_2)$ is an implicate of φ.*

Clearly, this result can yield many irrelevant binary clauses. Hence, as described in Section 2.4, the objective is to be selective with which clauses to actually consider.

2.3.4. Satisfiability-Based Inferred Clauses This section describes the inference of clauses based on formula satisfiability conditions.

Theorem 2.5 *Given a CNF formula φ, for every pair of literals l_1 and l_2 for which there exists a variable x such that, $\langle l_1, 1\rangle \in \mathrm{BCP}(\langle x, 0\rangle) \wedge \langle l_2, 1\rangle \in \mathrm{BCP}(\langle x, 1\rangle)$ then the clause $(l_1 \vee l_2)$ is an implicate of φ.*

Clearly, the two possible truth assignments on x_i either imply $\langle l_1, 1\rangle$ or $\langle l_2, 1\rangle$, then one of these two assignments must hold.

Theorem 2.6 *Given a CNF formula φ, for any clause $\omega = (l_{t_1} \vee \ldots \vee l_{t_k}) \in \varphi$,* all *clauses of the form, $\{l_{i_j} | \langle l_{i_j}, 1\rangle \in \mathrm{BCP}(\langle l_{t_j}, 1\rangle), l_{t_j} \in \omega, j = 1, \ldots, k\}$ are implicates of φ.*

Since the original clause ω must be satisfied, any set of $|\omega|$ assignments, each implied by a different literal in ω, forms an implicate of φ. We should observe that the number of clauses that can be created is upper-bounded by the Cartesian product of each set of assignments that results from applying BCP to each triggering assignment. In addition, observe the previous theorem can yield clauses with duplicate literals. Clearly, simple procedures can be implemented that filter out these duplicate literals.

2.3.5. Unsatisfiability-Based Inferred Clauses Next we describe the inference of clauses based on formula unsatisfiability conditions.

Theorem 2.7 *Given a CNF formula φ, for all pairs l_1 and l_2 for which there exists a variable x such that, $\langle x, 0\rangle \in \mathrm{BCP}(\langle l_1, 0\rangle) \wedge \langle x, 1\rangle \in \mathrm{BCP}(\langle l_2, 0\rangle)$ the clause $(l_1 \vee l_2)$ is an implicate of φ.*

If two assignments imply distinct truth values on a given variable x_i, then the two assignments *must* not hold simultaneously.

One additional condition related with unsatisfiability is the following:

Theorem 2.8 *Given a CNF formula φ, for each set of assignments $A = \{\mathrm{BCP}(\langle l_1, 0\rangle) \cup \ldots \cup \mathrm{BCP}(\langle l_k, 0\rangle)\}$ such that there exists a clause $\omega \in \varphi$, with $w(A) = 0$, then the clause, $(l_1 \vee \ldots \vee l_k)$ is an implicate of φ.*

If the union of sets of assignments resulting from applying BCP to a set of k triggering assignments unsatisfies a given clause, then the simultaneous occurrence of the k assignments must be prevented. Hence a new clause can be created. Observe that a stronger condition can be established if the condition $w_j(\mathrm{BCP}(A)) = 0$ is used, at the cost of additional computational overhead. Moreover, observe that the result of Theorem 2.8 is related with a technique proposed in [6]. For a clause $(l \vee \beta)$, where β is a disjunction of literals, if assigning value 0 to all literals in β yields a conflict, then (β) is an implicate of φ. The two techniques are related since both infer clauses from unsatisfiability requirements. The work of [6] assumes a specific clause and considers BCP of simultaneous sets of assignments[5]. Theorem 2.8 allows any k triggering assignments, but considers the separate application of BCP (which may yield fewer implied assignments).

5 As a result, we refer to this technique as *literal dropping*.

2.4. ProbIt: a Probing-Based SAT Preprocessor

The reasoning conditions described in the previous section were used to implement a SAT preprocessor, ProbIt. This preprocessor is organized as follows:

1. Create the table of assignments by applying BCP to each individual assignment.
2. Apply a restricted set of the reasoning conditions described in the previous sections:
   (a) Identification of necessary assignments, obtained by reasoning conditions from Theorems 2.1, 2.2 and 2.3.
   (b) Identification of equivalent variables, obtained by a *restricted* application of reasoning conditions from Theorem 2.4.
3. Iterate from 1 while more equivalent variables can be identified.

For the current version of ProbIt we opted not to infer new clauses during preprocessing. Existing experimental evidence suggests that the inference of clauses during preprocessing can sometimes result in large numbers of new clauses, which can impact negatively the run times of SAT solvers [7]. The identification of conditions for the selective utilization of clause inference conditions during preprocessing is the subject of future research work.

As a result, the utilization of Theorem 2.4 is restricted to the inference of binary clauses that lead to the identification of equivalent variables. Remember that two-variable equivalence (e.g. $x \leftrightarrow y$) is described by the pair of clauses $(\neg x \vee y) \wedge (x \vee \neg y)$, that can be represented as implications $(x \rightarrow y) \wedge (y \rightarrow x)$ (and also as $(\neg y \rightarrow \neg x) \wedge (\neg x \rightarrow \neg y)$). In ProbIt, rather than inferring new clauses which allow to identify equivalent variables, it is simpler to identify equivalences without having to infer the corresponding clauses. Based on the table of assignments, equivalent variables may be identified as follows:

- If $\langle y, 0\rangle \in \mathrm{BCP}(\langle x, 0\rangle)$ and $\langle y, 1\rangle \in \mathrm{BCP}(\langle x, 1\rangle)$, then $x \leftrightarrow y$.
- If $\langle y, 0\rangle \in \mathrm{BCP}(\langle x, 0\rangle)$ and $\langle x, 0\rangle \in \mathrm{BCP}(\langle y, 0\rangle)$, then $x \leftrightarrow y$.

Moreover, the same reasoning can be applied to identify the two-variable equivalence $x \leftrightarrow \neg y$.

3. Experimental Results

In these section we present experimental results to evaluate the usefulness of the new algorithm. First, we analyze the improvements on JQuest2 by integrating ProbIt as a preprocessor. Then, experimental results obtained for ProbIt+JQuest2 are compared with results obtained for other state-of-the-art SAT solvers.

Family	ProbIt	ProbIt+JQ2	JQuest2
barrel(8)	18.36	700.28	1,118.12
longmult(16)	544.18	1,725.17	4,658.67
queueinvar(10)	48.04	70.96	30.00
miters(25)	166.53	248.11	(2)11,175.65
fvp-unsat-1.0(4)	648.57	1,599.44	549.75
quasigroup(22)	61.21	531.43	735.25

Table 2. Improvements on JQuest2

Family	ProbIt+JQ2	2clseq	zChaff
barrel(8)	700.28	1,634.23	487.91
longmult(16)	1,725.17	2,201.37	2,191.08
queueinvar(10)	70.96	83.23	5.52
miters(25)	248.11	170.84	(2)10,537.49
fvp-...-1.0(4)	1,599.44	(2)13,545.74	549.75
quasigroup(22)	531.43	3,726.91	348.07

Table 3. Comparison with other solvers

ProbIt has been integrated on top of JQuest2, a competitive Java SAT solver [6]. JQuest2 is a backtrack search SAT solver, based on efficient data structures, and implementing the most effective backtrack search techniques, namely clause recording and non-chronological backtrack, search restarts, and adaptive branching heuristics. One of the main objectives of JQuest2 is allowing the rapid prototyping of new SAT algorithms. Since ProbIt is still an evolving preliminary implementation, the utilization of JQuest2 facilitates the evaluation and configuration of ProbIt.

Tables 2 and 3 give the CPU time in seconds required for solving for different classes of problem instances, that include some of the hardest instances. For each benchmark suite, the total number of instances is shown. For all experimental results a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The CPU time was limited to 5000 seconds. Consequently, we added 5000 seconds for each instance not solved in the allowed CPU time (the number of aborted instances is indicated in parenthesis).

In Table 2, ProbIt+JQuest2 (JQuest2 with ProbIt integrated) is compared with the original JQuest2. Moreover, the time required for the preprocessor ProbIt is also given. Table 3 compares ProbIt+JQuest2 with other SAT solvers, namely zChaff and 2clseq. zChaff is one of the most competitive SAT solvers. On the other hand, 2clseq is also known a competitive SAT solver, characterized by integrating formula manipulation techniques.

From the obtained results, several conclusions can be drawn:

- ProbIt+JQuest2 comes out as the most robust solver on the set of problem instances considered. Despite being implemented in Java, which necessarily yields a slower implementation, ProbIt+JQuest2 performance is indeed comparable to state-of-the-art SAT solvers.
- The performance of ProbIt+JQuest2 is comparable to 2clseq in instances where formula manipulation helps on solving an instance. This explains why zChaff performance is not competitive for these instances.
- ProbIt+JQuest2 performance is also comparable to zChaff on instances where more sophisticated backtrack search techniques are required. Nonetheless, and when compared to JQuest2 results, ProbIt+JQuest2 may require more time to solve a family of benchmark examples. This can be explained by the time required for applying ProbIt techniques. Clearly this is a drawback when the number of variables in the CNF formula is not reduced.

4. Related Work

The ProbIt algorithm described in the previous section uses probing as the basis for implementing a number of formula manipulation techniques. In this section we relate ProbIt with previous work in probing and formula manipulation techniques.

4.1. Probing-Based Techniques

In the SAT domain, the idea of establishing hypotheses and inferring facts from those hypotheses has been extensively studied in the recent past [3, 4, 5, 6].

The *failed literal rule* is a well-known and extensively used probing-based technique (see for example [5]): if the assignment $x = 0$ yields a conflict (due to BCP), then we must assign $x = 1$. This rule is covered by necessary assignments obtained from unsatisfiability conditions (Theorem 2.3).

Variable probing is a probing-based technique, which consists of applying the *branch-merge rule* to each variable [3] [7]. Common assignments to variables are identified, by detecting and merging equivalent branches [8]. Observe that variable probing is covered by reasoning conditions established with Theorem 2.1.

6 JQuest2 entered in the second stage of the industrial category in the SAT'2003 Competition (see http://www.satlive.org/SATCompetition/2003/comp03report/).

7 The branch-merge rule is the inference rule used in the *Stålmark's Method* [12].

8 In addition, variable probing is often used as part of look-ahead branching heuristics in SAT solvers [9].

Clause Probing is similar to variable probing, even though variable probing is based on variables and clause probing is based on clauses. Clause probing consists of evaluating clause satisfiability requirements for identifying common assignments to variables. Common assignments are deemed necessary for a clause to become satisfied and consequently for the formula to be satisfied. These techniques have been applied to SAT in [10] and more recently in [4]. In our framework, clause probing is captured by Theorem 2.2. To the best of our knowledge, no other work proposes the joint utilization of variable and clause probing.

The notion of *literal dropping*, that considers applying sets of simultaneous assignments for inferring clauses that subsume existing clauses, is described for example in [6]. As mentioned earlier, some of the clause inference conditions proposed by Theorem 2.8 can be related with previous techniques for literal dropping, proposing more general conditions for inferring clauses, but based on less powerful unit propagation.

4.2. Other Manipulation Techniques

Two-variable equivalence is a well-known formula manipulation technique that has been integrated in ProbIt (see Section 2.4). Additional two-variable equivalence conditions can be established, namely by the identification of *strongly connected components* [1]. It is interesting to observe that the existing strongly connected components in a CNF formula are captured from the construction of the assignment table and the application of Theorem 2.4. Furthermore, sophisticated techniques have been developed to detect chains of biconditionals [8, 11].

The *2-closure* of a 2CNF sub-formula [7] allows to infer additional binary clauses. The identification of the transitive closure of the implication graph is obtained from the construction of the assignment table and the application of Theorem 2.4: if $\langle y, 1\rangle \in \mathrm{BCP}(\langle x, 1\rangle)$ then create clause $(\neg x \vee y)$.

More recently, a competitive SAT solver incorporating hyper-resolution with binary clauses has been proposed [2]. Given the set of clauses $(\neg l_1 \vee x) \wedge (\neg l_2 \vee x) \wedge ... \wedge (\neg l_k \vee x) \wedge (l_1 \vee l_2 \vee ... \vee l_k \vee y)$, hyper-resolution allows inferring $(x \vee y)$. Once again, observe that this technique is covered by the construction of the assignment table and the application of Theorem 2.4: if $\langle x, 0\rangle \in \mathrm{BCP}(\langle y, 1\rangle)$ then create clause $(x \vee y)$.

Compared with existing work, probing-based preprocessing techniques not only naturally capture *all* the above mentioned formula manipulation techniques, but also further allow the development of new techniques. In addition, the proposed unified framework also allows relating and comparing different formula manipulation techniques.

5. Conclusions

This paper introduces ProbIt, a new probing-based formula manipulation SAT preprocessor. ProbIt has been implemented as a unified formula manipulation framework, based on probing assignments, that captures a significant number of formula manipulation techniques. Moreover, this new approach integrates for the first time most formula manipulation techniques and allows developing new techniques. In addition, the obtained experimental results clearly indicate that ProbIt is effective in increasing the robustness of state-of-the-art SAT solvers.

References

[1] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979.

[2] F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *AAAI'02*, August 2002.

[3] D. L. Berre. Exploiting the real power of unit propagation lookahead. In *LICS Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[4] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In *IJCAI'01*, August 2001.

[5] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI'93*, pages 22–28, 1993.

[6] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *IJCAI*, August 2001.

[7] A. V. Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.

[8] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI'00*, pages 291–296, July 2000.

[9] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *CP'97*, pages 341–355, October 1997.

[10] J. P. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proc. of the ACM/IEEE Design, Automation and Test in Europe Conference*, pages 145–149, March 1999.

[11] R. Ostrowski, Éric Grégoire, B. Mazuren, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *CP'02*, pages 185–199, September 2002.

[12] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, 1989. Swedish Patent 467 076 (Approved 1992), US Patent 5 276 897 (approved 1994), European Patent 0 403 454 (approved 1995).