A Simplifier for Propositional Formulas with Many Binary Clauses

Ronen I. Brafman

***Abstract*— Deciding whether a propositional formula in conjunctive normal form is satisfiable (SAT) is an NP-complete problem. The problem becomes linear when the formula contains binary clauses only. Interestingly, the reduction to SAT of a number of well-known and important problems – such as classical AI planning and automatic test pattern generation for circuits – yields formulas containing many binary clauses. In this paper we introduce and experiment with 2-SIMPLIFY, a formula simplifier targeted at such problems. 2-SIMPLIFY constructs the transitive closure of the implication graph corresponding to the binary clauses in the formula and uses this graph to deduce new unit literals. The deduced literals are used to simplify the formula and update the graph, and so on, until stabilization. Finally, we use the graph to construct an equivalent, simpler set of binary clauses. Experimental evaluation of this simplifier on a number of bench-mark formulas produced by encoding AI planning problems prove 2-SIMPLIFY to be a useful tool in many circumstances.**

I. INTRODUCTION

PROPOSITIONAL satisfiability (SAT) is the problem of deciding whether a propositional formula in conjunctive normal form (CNF) is satisfiable. SAT was the first problem shown to be NP-complete [6] and has important practical applications. In the last decade we have witnessed great progress in SAT solution methods, first with the introduction of efficient stochastic local search algorithms [21], [22], and more recently with a number of efficient systematic solvers, such as REL-SAT [4] and SATZ [16] and their randomized versions [10], and more recently, CHAFF[18].

Among AI researchers, interest in SAT solution algorithms has increased since Kautz and Selman showed that some classical planning problems can be solved more quickly when they are reduced to SAT problems [12]. Kautz and Selman's *planning as satisfiability* approach is based on generic SAT technology, and, aside from the translation process itself, makes no use of properties specific to planning problems. However, SAT-encoded planning problems have an important syntactic property: they contain a large fraction of binary clauses. Interestingly, this property is found in other important domains – automatic test-pattern generation for circuits [15] and bounded model checking [23].

Unlike the general SAT problem, 2-SAT, the problem of deciding whether a propositional formula containing binary clauses *only* has a satisfying assignment is (constructively) solvable in linear time. One would hope that this property would make it easier to solve SAT instances containing a large fraction of binary clauses. In this article we describe the 2-SIMPLIFY preprocessor, a simplifier that is geared to such formulas. Like other simplifiers (e.g., Crawford's COMPACT), this algorithm takes a propositional formula in CNF as input and outputs a new propositional formula. Naturally, for this process to be worthwhile, the new formula should be easier to solve, and the overall time required for simplification and solution of the simplified formula should be less than the time required to solve the original SAT instance. Experiments on a number of bench-mark formulas derived from planning problems show that 2-SIMPLIFY's performance depends on the difficulty of the problem, whether it is satisfiable or not, and the solver used. In many classes of problems, 2-SIMPLIFY leads to a combined simplification and solution time that is lesser than the original solution time.

2-SIMPLIFY efficiently implements and combines well-known 2-SAT techniques, a limited form of hyper-resolution, and novel use of transitive reduction to reduce formula size. The basic idea is as follows: each clause of the form $p \vee q$ is equivalent to two implications: $\neg p \rightarrow q$ and $\neg q \rightarrow p$. We use this property to construct a graph (known as the *implication graph* [1]) from the set of binary clauses. This graph contains a node for each literal in the language and a directed edge from a literal l to another literal l' if the (disjunction equivalent to the) implication $l \rightarrow l'$ appears among the set of binary clauses. After constructing this graph, we compute its transitive closure and check each literal to see whether its negation appears among its descendants. If $\neg l$ is a descendant of l, we can immediately conclude that $\neg l$ is a consequence of the original formula. Once we know that l holds, we can simplify the original formula. The simplified formula may contain new binary clauses, which are immediately added to the graph. 2-SIMPLIFY utilizes these and other ideas to quickly derive unit literals from the original formula and produce a simpler equivalent formula as its output.

In the next section we discuss the background of this work in more detail. In Section 3 we present the simplification algorithm used by 2-SIMPLIFY. In Section 4 we present some experimental results, and we conclude in Section 5.

II. BACKGROUND

We start with some SAT background and then we briefly explain the *planning as satisfiability* approach, which motivated this work.

A. The SAT Problem

The SAT problem is defined as follows: given a propositional formula in conjunctive normal form (CNF) output YES if the

Author's address: Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel (email :brafman@cs.bgu.ac.il)

formula is satisfiable and NO otherwise. In practice, a positive answer is accompanied by some satisfying assignment.

There are two classes of SAT algorithms: stochastic and systematic. Stochastic methods, such as G-SAT[21] and WALK-SAT[22], perform stochastic local search in the space of truth assignment. Often, they can find solutions quickly, but they cannot identify an unsatisfiable instance. Their performance is extremely sensitive to the choice of heuristic and various other parameters. Systematic methods systematically search the space of truth assignments. Thus, they can identify unsatisfiable instances. Modern systematic algorithms are quite fast and stable thanks to improved branch choice heuristics and backtracking techniques. In addition, systematic solvers can be improved by introducing some randomization into their search procedure, e.g., their choice of branch variable. See [10] for more information on this topic.

Often, a formula simplifier is applied before the SAT solver. Simplifiers use specialized, efficient deductive methods to reduce the original formula into a simpler formula which is typically easier to solve. The best known simplification method is unit propagation. When one of the clauses in the formula contains a single literal, it must be assigned the value *true* in any satisfying assignment. For example, if a formula contains the clause $\{\neg p\}$ then $\neg p$ must be *true* in any satisfying assignment, i.e., p must be *false*. Once we deduce this fact, we can use it to simplify other clauses: clauses that contain $\neg p$ can be removed since their satisfaction is guaranteed when p is *false*, and the literal p can be removed from any clause containing it (e.g., $\{p, \neg s\}$ will be transformed into $\{\neg s\}$) because it is equivalent to *false*. As we just saw, the simplification process can yield additional unit clauses, which are used to produce additional simplifications. If during the simplification process an empty clause is discovered (e.g., if we assigned s the value *true* and there is a unit clause $\{\neg s\}$) we can conclude that the formula is unsatisfiable.

There are a number of additional simplification methods, such as failed unit literal and failed binary literal, where one or two unit clauses are added to the current formula and we attempt to show (e.g., using unit propagation) that the resulting formula is inconsistent. In that case, the negation of the added clause is implied by the original formula, and we update the truth assignment accordingly. For example, if our original formula becomes inconsistent once we add the clauses $\{p\}, \{q\}$, we know that either p or q must be assigned *false*, i.e., that $\{\neg p, \neg q\}$ is implied by the formula.

B. 2-SAT

2-SAT is a subclass of SAT in which clauses contain no more than two literals. While SAT is NP-complete [6], 2-SAT can be solved in linear time. The key step in solving 2-SAT problems is the construction of the *implication graph* [1]). The nodes of the implication graph correspond to the literals in the formula. The graph contains an edge between the literal l and the literal l' if the clause $\{\neg l, l'\}$ appears in the formula. That is, edges in the graph correspond to implications (since $\{\neg l, l'\}$ is equivalent to $l \rightarrow l'$). Since implication is transitive, we have that $l_1 \rightarrow l_2$ is implied by the formula whenever there is a path in the graph between node l_1 and node l_2. In particular, if we have a path between l_1 and $\neg l_1$, we know that l_1 cannot hold. Therefore, $\neg l_1$ is implied by the formula. If, in addition, we have a path from $\neg l_1$ to l_1, then neither l_1 nor $\neg l_1$ can hold, and so the formula is unsatisfiable. Finally, we know that in every satisfying truth assignment, if l is assigned *true* then any literal implied by l, i.e., any descendant of l in the graph, must be assigned *true* as well.

Instance	% Binary Clauses
log-dir.a	49%
log-dir.b	55%
log-dir.c	55%
log.d	80%
log-gp.a	98%
log-gp.b	98%
log-gp.c	98%
log-un.a	98%
log-un.b	98%
log-un.c	99%
bw-dir.a	70%
bw-dir.b	71%
bw-dir.c	74%
bw-dir.d	78%

TABLE I
PERCENTAGE OF BINARY CLAUSES IN SAT-ENCODED PLANNING PROBLEMS

C. Planning As Satisfiability

The planning problem is defined as follows: given a description of an initial state, a goal state, and a set of operators (=Actions) for changing the state of the world, find a sequence of operators that, when applied in the initial state, yield the goal state. An important development in planning algorithms was Kautz and Selman's *planning as satisfiability* approach [12]. Kautz and Selman showed that by reducing planning problems to satisfiability problems, we can often solve them more quickly than by using standard planning algorithms. Planning problems can be encoded as satisfiability problems in a number of ways (e.g., see [8] for a description and analysis of some of these methods).

As [5] points outs, encoded planning problems contain a large number of binary clauses. In Table 1 we show this for a number of instances of SAT-encoded planning problems. This is no accidental phenomenon. Close inspection of the types of constraints expressed within encoded planning problems makes it apparent that many classes of these constraints generate binary formulas. For example, the constraint that if an action is executed at some time point then all its preconditions must hold prior, produces binary clauses. Similarly, the constraint asserting that if an action is executed at some point then all its effects must hold after the execution yields binary clauses as well. In the encoding used by the BLACKBOX planner [13] mutual exclusion constraints (on actions and on state variables) play a prominent role. These constraints are expressed using binary clauses as well.

Interestingly, it turns out that the SAT encodings of other important problems exhibit the same large percentage of binary

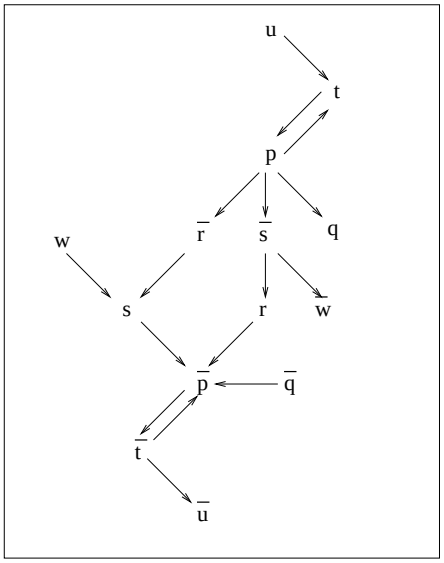

Fig. 1. The Implication Graph

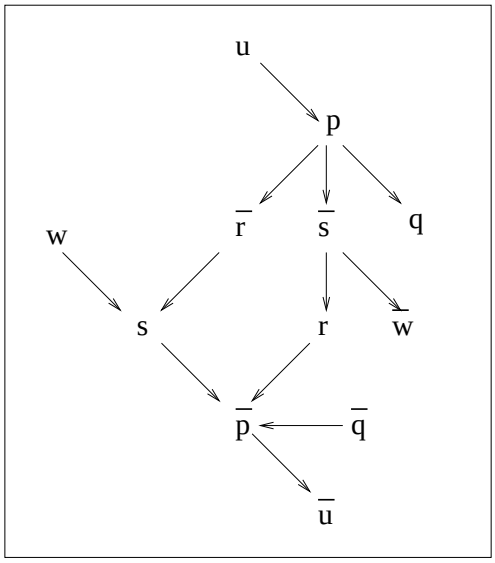

Fig. 2. Removing Strongly Connected Components

clauses. These include test-pattern generation for circuits [15] and bounded model checking [23].

III. The 2-simplify PreProcessor

We now explain the algorithm implemented by the 2-SIMPLIFY preprocessor using the following formula:

$$\{\neg p, q\}, \{\neg p, \neg r\}, \{r, s\}, \{\neg w, s\}, \{\neg p, t\}, \{\neg t, p\}, \{\neg u, t\}$$

$$\{\neg p, \neg s\}\{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\}$$

$$\{s, v, \neg m\}$$

(1) *Construct Implication Graph.* A graph containing all literals in the language is constructed with directed edges from l to l' if $\{\neg l, l'\}$ is a binary clause. Figure 1 shows the implication graph for the formula above. Note that in the figures, we use $\bar{p}$ to denote $\neg p$.

(2) *Collapse Strongly Connected Components.* A subgraph in which there is a path between every pair of nodes is called a strongly connected component (SCC). When a path from node l to node l' exists, we know that $l \rightarrow l'$ is a consequence of our formula. Therefore, all nodes within an SCC imply each other, and they must all be assigned the same value.

Once we discover an SCC we replace it by a single node. The children of this node are the children of the nodes in the SCC, and the parents of this node are the parents of the nodes in this SCC. In addition, all literals in the SCC must be replaced by the literal corresponding to this new node within all non-binary clauses.

Because of the symmetric nature of the implication graph, for every SCC we discover, another SCC containing the negation of the literals of this SCC exists. Thus, whenever we replace an SCC by a new node labelled by the literal l, we replace the symmetric SCC by a new node that is labelled by $\neg l$.

In our example, the nodes t and p form a strongly connected component, and so do their negations, $\neg t$ and $\neg p$. We choose p to represent the first SCC and we choose $\neg p$ to represent the second SCC. The reduced graph is shown in Figure 2.

(3) *Generate Transitive Closure.* Now, we generate the transitive closure of the graph. This can be done with one traversal of the (now acyclic) graph in reverse topological order (i.e., by adding to the adjacency list of each node the children of its children). We know that if l' is a child of l then $l \rightarrow l'$ is implied by the original formula. We can deduce l if either:

1) for some proposition p, both p and $\neg p$ are children of $\neg l$.
2) l is a child of $\neg l$.

Once we deduce l, we can perform unit propagation: all children of l are assigned the value *true*, and we can remove all occurrences of $\neg l$ from within the clauses of our formula. If the reduced formula contains new binary clauses, we add the appropriate edges to the graph and update the transitive closure.

In Figure 3 we can see the effect of this step. First, we compute the transitive closure of the current graph, shown in Figure 3A. In this graph, we see that u has $\neg u$ as a descendant and that p has $\neg p$ as a descendant. Therefore, we conclude that p and u must be assigned the value *false*. We can remove nodes that correspond to assigned propositions (i.e., $p, u, \neg p, \neg u$ in our case). The resulting graph is shown in Figure 3B. Next, we perform unit propagation, and our initial ternary clauses: $\{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\}$ are reduced to $\{\neg q, \neg s\}, \{\neg w, q\}, \{\neg q, \neg r, s\}$. The first clause was removed because it is satisfied, and a (false) literal was removed from the next two clauses. Since we have new binary clauses, we can update the graph, as shown in Figure 3C, making sure it is transitively closed. In the resulting graph, $\neg w$ is a child of w, and we can deduce that $w =$ *false*. The reduced graph is shown in Figure 3D.

(4) *Derive Shared Implications.* Let $\{l_1, \ldots, l_k\}$ be some non-binary clause in the formula. Let L_i be the set of literals implied by l_i for $i = 1, \ldots, k$. Let $L = L_1 \cap \cdots \cap L_k$. All literals in L are consequences of our formula, and we can use

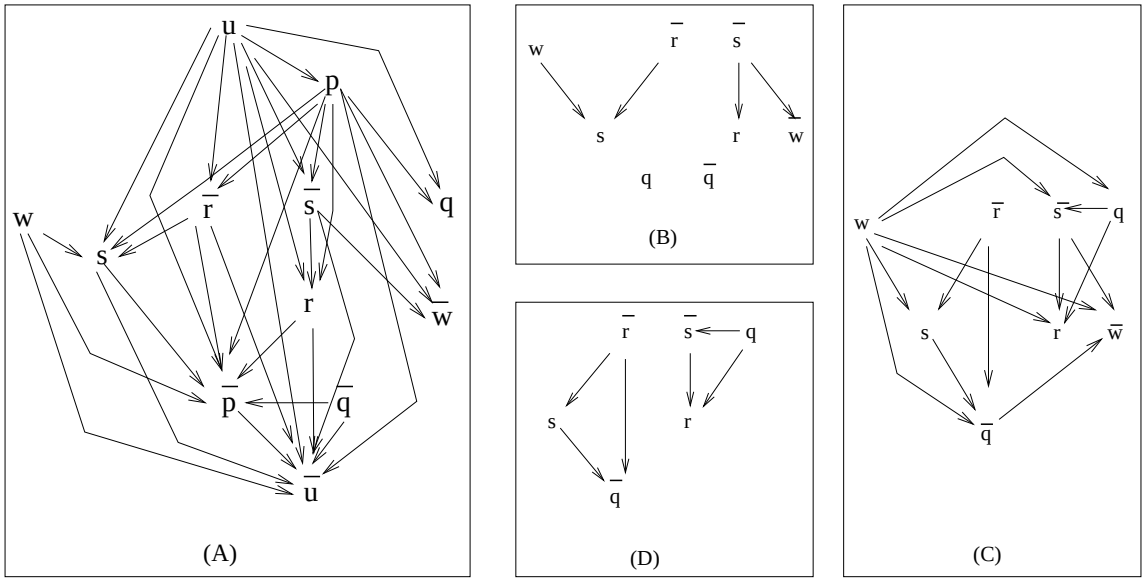

Fig. 3. (A) Initial Transitive Closure (B) Removal of Assigned Nodes (C) Update with New Binary Clauses (D) Removal of Assigned Nodes

them to perform unit propagation.

Consider the clause $\{\neg q, \neg r, s\}$, the sets of literals implied by each of the literals in this clause are $(\neg q), (\neg r, s, \neg q), (s, \neg q)$, respectively. Their intersection contains $\neg q$. Hence, we can deduce that q is *false*.

(5) *Subsumption Elimination.* A clause $C = \{l_1, \ldots, l_k\}$ is *subsumed* by $C' = \{l'_1, \ldots, l'_m\}$ if $m < k$ and $l'_j \in \{l_1, \ldots, l_k\}$ for every $1 \leq j \leq m$. If C' subsumes C, then the constraint C' is stronger than C. Thus, C is redundant.

We use the following method to quickly detect whether one non-binary clause is subsumed by some binary clause implied by the implication graph. Given a clause $C = \{l_1, \ldots, l_k\}$, we generate a set S containing all the children of the negations of the literals in C, i.e., the children of $\neg l_1, \ldots, \neg l_k$. If any of the literals in C appears in S, we know that C is subsumed.

(6) *Pure-literal Removal.* A pure literal is a literal whose negation does not occur in any of the clauses in the formula. In that case, we can assign that literal the value *true* without affecting the satisfiability of the formula.

(7) *Compute Transitive Reduction.* The transitive reduction of a graph G is a graph G' with the same nodes as G but with a minimal set of edges such that a path between l and l' exists in G iff a path between l and l' exists in G'. Thus, G' is a minimal sub-graph of G that maintains node connectivity.

We compute the transitive reduction of the current graph in order to reduce the size of the formula. Our implementation of this step relies on the fact that we start with a transitively closed graph. Thus, if we remove from the list of children of a node all of its grandchildren, starting at the root nodes and progressing in topological order, we obtain a reduced graph.

(8) *Output Simplified Formula.* We output a formula whose clauses consist of the non-binary clauses remaining at this stage, and all binary clauses corresponding to edges in the transitive reduction of the graph. Given the assignments deduced so far and the mappings between elements of strongly connected components, the simplified formula is equivalent to the original formula. For example, the output for our original formula will be: $\{r, s\}, \{s, v, \neg m\}$ together with the partial assignment $u = \mathit{false}, p = \mathit{false}, w = \mathit{false}, q = \mathit{false}$.

Step (4) is a novel implementation of an old technique (hyper-resolution [20]) and step (7) is new. Both have important impact on 2-SIMPLIFY's performance. The *Derive Shared Implications* step enhances the ability of 2-SIMPLIFY to derive unit literals. In some cases, it can derive hundreds of new unit literals quickly. In fact, 2-SIMPLIFY uses a more sophisticated version of this procedure: if no shared unit literals exist, we attempt to derive new binary clauses by intersecting the implications of all literals but one. These binary clauses are then added to the implication graph. The *Compute Transitive Reduction* step leads to a minimal sufficient set of binary clauses, leading to smaller and simpler formulas. We have found this reduction to have an important positive influence on systematic solvers.

IV. EXPERIMENTAL EVALUATION

We run extensive tests that examine different aspects of 2-SIMPLIFY on a set of bench-mark instances of encoded planning problems which were used to test the REL-SAT solver. In addition, to see the usefulness of 2-SIMPLIFY on other problems we checked its performance on a host of verification bench-mark problems.

The experiments in the first section below, dealing with encoded planning problem were carried out on a DELL Latitude CPi notebook with a Pentium II-400 processor with 64MB RAM running Linux. These problems were obtained from tt ftp://ftp.research.att.com/dist/ai/logistics/tar.Z and `satplan.data.tar.Z`. The experiments in the second section, dealing with different encoded verification problems

Instance	Time	Assigned+Mapped	Clause Number Ratio
bw-dir.a	0.16	198 out of 459	1944/4675
bw-dir.b	0.57	391 out of 1087	6814/13772
bw-dir.c	2.63	890 out of 3016	29770/50457
bw-dir.d	9.28	1760 out of 6325	82453/131973
log-dir.a	0.07	238 out of 828	5019/6718
log-dir.b	0.08	269 out of 843	5325/7301
log-dir.c	0.13	302 out of 1141	8309/10719
log.d	4.05	967 out of 4713	16602/21991
log-gp.a	0.31	281 out of 1782	5511/20895
log-gp.b	0.49	323 out of 2069	6725/29508
log-gp.c	0.85	374 out of 2809	9915/48920
log-un.a	0.21	416 out of 1415	3725/14346
log-un.b	0.34	307 out of 1729	5355/21943
log-un.c	0.58	347 out of 2353	7975/37121

TABLE II
RUNNING TIME AND DEDUCTION POWER OF 2-SIMPLIFY

Instance	SATZ	2-SIMPLIFY	SATZ on 2-SIMPLIFY	Total
log-dir.a	**60.15**	0.07	5.44	**5.51**
log-dir.b	**0.32**	0.08	0.3	**0.38**
log-dir.c	**1.99**	0.13	2.64	**2.77**
log.d	**896.3**	4.05	1.88	**5.93**
log-gp.a	**0.7**	0.31	0.17	**0.48**
log-gp.b	**1.9**	0.49	0.58	**1.07**
log-gp.c	**2.91**	0.85	0.34	**1.19**
log-un.a	**0.4**	0.21	0.09	**0.30**
log-un.b	—	0.34	1630.94	**16311.28**
log-un.c	—	0.58	158.07	**158.65**
bw-dir.a	**0.16**	0.16	0.06	**0.22**
bw-dir.b	**0.56**	0.57	0.25	**0.82**
bw-dir.c	**2.53**	2.63	1.39	**4.02**
bw-dir.d	**675.66**	9.28	59.7	**68.98**

TABLE III
SOLUTION TIMES FOR SATZ AND 2-SIMPLIFY+SATZ.

(which were much more difficult) were performed on a PC running Linux with a Pentium 4, 180Ghz processor and a 256KB cache. These problems were taken from the benchmark of Miroslav Velev at `www.ece.cmu.edu/~mvelev` (the fvp and 2dlx instances), Ofer Shtrichman (ibm instances) obtainable from satlib (`www.satlib.org`), and the BMC generated instances of Biere, Cimatti, Clark, and Zhu, `www-2.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html`. 2-SIMPLIFY is written in C++ and all time measurements refer to CPU time.

A. Encoded Planning Problems

First, we examined 2-SIMPLIFY's ability to deduce unit literals. In Table 2 we show 2-SIMPLIFY's running time on each of the instances, the number of variables it was able to assign or map, and the ratio between the number of clauses in the simplified formula and the original formula.

To assess the utility of 2-SIMPLIFY we generated simplified formulas for each of the instances and compared the solution time of the original formulas with the combined simplification and solution times for the simplified formulas. We performed this comparison using two systematic solvers: SATZ and REL-SAT. The results for SATZ are shown in Table 3. In 5 out of the 14 problems, the use of 2-SIMPLIFY leads to degraded performance. This occurs on the relatively smaller problems, and typically, with a small overhead that stems from simplification costs. On 9 out of the 14 problems, 2-SIMPLIFY leads to improved performance. Of particular importance is the fact that 2-SIMPLIFY performs better on the problems that harder for SATZ and on those that SATZ cannot solve without simplification.

The results for REL-SAT are shown in Table 4. The simplified formulas were solved using REL-SAT, but with its preprocessor disabled. In many cases, this leads to improved performance, a natural consequence of the fact that 2-SIMPLIFY provides stronger simplification capabilities. For example, in the largest instance, bw-dir.d, REL-SAT without the preprocessor took 571 seconds, instead of 847. However, there are cases in which the REL-SAT preprocessor leads to better running times on the simplified formulas. For instance, in the log-gp and log-un instances, it was always better. Overall, we see that

Instance	REL-SAT	2-SIMPLIFY	2-SIMPLIFY+REL-SAT
log-dir.a	**0.43**	0.07	**0.27**
log-dir.b	**0.48**	0.08	**0.21**
log-dir.c	**1.16**	0.13	**0.66**
log.d	**9.41**	4.05	**10.28**
log-gp.a	**1.51**	0.31	**1.43**
log-gp.b	**2.22**	0.49	**2.4**
log-gp.c	**4.43**	0.85	**5.00**
log-un.a	**0.58**	0.21	**0.52**
log-un.b	**11.92**	0.34	**8.09**
log-un.c	**22.96**	0.58	**17.3**
bw-dir.a	**0.25**	0.16	**0.21**
bw-dir.b	**1.65**	0.57	**0.98**
bw-dir.c	**26.9**	2.63	**14.07**
bw-dir.d	**>1000**	9.28	**560.26**

TABLE IV
SOLUTION TIMES FOR REL-SAT, 2-SIMPLIFY, AND 2-SIMPLIFY+REL-SAT.

2-SIMPLIFY+REL-SAT is almost always faster than REL-SAT alone, with three exceptions that stem from relatively long simplification times. The improvement is especially significant in the hardest instances. We note that the REL-SAT figures represent average running times (because REL-SAT has a stochastic element) and that in the case of bw-dir.d, REL-SAT timed out on the original problem in some of the iterations and the result provided is a lower bound on its true average running time.

We run another sequence of experiments to compare 2-SIMPLIFY with Crawford's COMPACT simplifier. COMPACT provides various simplification options. The basic COMPACT simplifier performs unit resolution, removes satisfied clauses, and renames variables to be contiguous. In addition, there are a number of optional flags. With the p flag, COMPACT performs pure-literal elimination, with s it resolves away literals that occur only once, with l it performs the unit-failed test – that is, it checks for each literal whether adding this literal to the formula leads to an inconsistency (using unit propagation). If so, it assigns that literal the value *false*. Finally, the b option adds the binary-failed test. In this case, pairs of literals are added each time, and unit resolution is performed. If an inconsistency is detected, the negation of the conjunction of this pair of literals

Instance	COMPACT+SATZ	2-SIMPLIFY+ SATZ	psl + SATZ
bw-dir.a	0.31	0.22	0.18
bw-dir.b	0.95	0.82	1.06
bw-dir.c	3.67	4.02	6.76
bw-dir.d	688.21	68.98	125.2
log-dir.a	1.22	5.51	2.33
log-dir.b	0.56	0.38	0.66
log-dir.c	1.33	2.77	404.96
log.d	454.57	5.93	4.12
log-gp.a	1.12	0.48	5.95
log-gp.b	1.75	1.07	0.58
log-gp.c	4.95	1.19	23.41
log-un.a	0.7	0.3	3.35
log-un.b	-	1631.28	13.18
log-un.c	-	158.65	107.22

TABLE V
SIMPLIFICATION WITH COMPACT AND 2-SIMPLIFY, SOLUTION WITH SATZ.

Instance	COMPACT+REL-SAT	2-SIM.+REL-SAT	psl + REL-SAT
bw-dir.a	0.24	0.21	0.18
bw-dir.b	0.89	0.98	1.03
bw-dir.c	17.42	14.07	15.42
bw-dir.d	579.51	580.26	283.41
log-dir.a	11.15	0.27	6.48
log-dir.b	0.33	0.21	0.62
log-dir.c	2.59	1.66	0.88
log.d	7.2	10.28	2.41
log-gp.a	2.18	1.43	6.04
log-gp.b	3.58	2.4	10.37
log-gp.c	7.26	5.00	25.07
log-un.a	1.11	0.52	3.08
log-un.b	10.08	8.09	11.49
log-un.c	23.1	17.3	34.47

TABLE VI
SIMPLIFICATION WITH COMPACT AND 2-SIMPLIFY, SOLUTION WITH REL-SAT.

is added to the formula. This latter test is quite powerful, but it is almost always too costly to be worthwhile. After testing various combinations of options on the above instances, we found that the best performance is obtained almost always using either no flags or using the *psl* flags, and this is what we show here.

In Tables 5 and 6 below, we compare the simplification + running times of SATZ and REL-SAT on COMPACT simplified formulas and on the 2-SIMPLIFY simplified formulas. In Table 5, we see the results for SATZ. The columns correspond to the combined running time of SATZ and the simplification algorithms on each of the instances. There are three cases in which COMPACT with no options leads to better performance, but overall, 2-SIMPLIFY leads to much better results, especially on the more difficult instances. The results when the *psl* option is used are more varied. 2-SIMPLIFY still does better in more cases, but on the last two instances, COMPACT with *psl* does much better. This may indicate that an enhanced version of 2-SIMPLIFY with *psl*-like capabilities could perform better than either simplifiers. However, as we shall see below, the relative performance is greatly affected by the solver used.

In Table 6 we show the corresponding results for REL-SAT (without its preprocessor). Again, we see that 2-SIMPLIFY is better than COMPACT with no options, and that 2-SIMPLIFY and *psl* succeed on different instances. However, notice that *psl* does noticeably better only on 3 instances. Moreover, notice the large change in performance on the two problems that were most difficult for SATZ – log-un.b/c. Again, it would be excellent if we could get the best of both worlds.

Finally, we examined 2-SIMPLIFY's influence on the performance of WALKSAT, a stochastic solver. As noted, stochastic solvers require tuning, and we tried to find the best parameters in each case. As Table 7 shows, the results are, again, quite positive. Although there are three instances in which 2-SIMPLIFY leads to reduced performance, on most instances it leads to improved performance. Moreover, on all the harder instances, 2-SIMPLIFY leads to noticeable improvements.

Instance	WALKSAT	2-SIMPLIFY+WALKSAT
bw-dir.a	0.04	0.19
bw-dir.b	0.83	1.49
bw-dir.c	84.18	38.26
bw-dir.d	321.90	186.72
log-dir.a	0.29	0.14
log-dir.b	0.39	0.21
log-dir.c	0.78	0.30
log.d	0.7	4.26
log-gp.a	3.93	1.06
log-gp.b	5.92	1.55
log-gp.c	35.19	3.45

TABLE VII
SOLUTION TIMES USING WALKSAT.

B. *Verification Problems*

A natural question is whether the performance of 2-SIMPLIFY carries over to other SAT-encoded problems. In particular, those originating from the verification community. To check this, we tested 2-SIMPLIFY on a host of model-checking and bounded model-checking problems. These problems are much larger than the planning bench-marks, and SATZ and REL-SAT have a very difficult time solving them. Luckily, a recent solver, CHAFF, is able to solve these problems rather easily. Thus, we compared the running times of CHAFF with and without 2-SIMPLIFY on these problems.

In Table 8 we see the performance of 2-SIMPLIFY on satisfiable instances. CHAFF solves these problems without difficulty, usually in less than a second. Although CHAFF works faster on the simplified problems, the simplification time is roughly 40 seconds, making the use 2-SIMPLIFY inappropriate.

In Table 9, we see the results on a set of unsatisfiable problems. Here, we see precisely the opposite picture. Except for two cases, on all instances that require more than 10 seconds, 2-SIMPLIFY leads to considerable reduction in combined running time. The hardest problem in this class, fvp-7pipe, was not solvable by CHAFF within over 1000 minutes, whereas its simplified version was solved within less than an hour.

Instance	CHAFF	CHAFF on simplified	2-SIMPLIFY	2-SIMPLIFY+ CHAFF
2dlx...f2-bug01	0.31	1.14	41.01	42.15
2dlx...f2-bug02	0.25	0.74	42.06	42.8
2dlx...f2-bug03	0.44	1.08	41.66	42.74
2dlx...f2-bug04	0.15	0.84	40.79	41.63
2dlx...f2-bug05	2.22	0.23	42.02	42.25
2dlx...f2-bug06	0.25	0.75	41.93	42.68
2dlx...f2-bug07	0.16	0.59	41.29	41.88
2dlx...f2-bug08	2.62	0.52	41.36	41.88
2dlx...f2-bug09	2.61	0.52	42.77	43.29
2dlx...f2-bug10	0.05	0.06	72.16	72.22
ibm-1	2.01	OOM	–	–
ibm-2	0.01	0.23	0.22	0.23
ibm-5	0.17	0.04	6.05	6.09
ibm-7	0.07	0.02	1.46	1.48

TABLE VIII

SOLUTION TIMES FOR CHAFF ON ORIGINAL AND SIMPLIFIED SATISFIABLE PROBLEMS(OOM STANDS FOR OUT OF MEMORY).

The explanation for the performance differences of 2-SIMPLIFY with respect to the class of satisfiable and unsatisfiable instances most likely lies in the different nature of the search required. CHAFF seems to be able to generate a single solution, when one exists, quickly. This makes the advantages of the reduction relatively small – even if we save during solution time, the difference is too small to compensate for the simplification cost. On the other hand, to prove that a problem is unsatisfiable, an exhaustive search of the space of assignments is required. Now, the smaller search space of the simplified formula pays off.

V. CONCLUSION AND RELATED WORK

SAT instances with many binary clauses arise naturally in a number of important applications. The abundance of binary clauses in such problems can be exploited using 2-SAT solution methods and other specialized inference algorithms. Here, we presented 2-SIMPLIFY, a principled and efficient simplification algorithm that uses the transitive closure of the implication graph together with a novel implementation of hyper-resolution (i.e, the *derive shared implications* step) and *transitive reduction* to obtain a smaller equivalent formula. This leads to an approach that is faster, more powerful, and more efficient than the ad-hoc resolution of binary clauses used in [5].

Our experiments show that the performance of 2-SIMPLIFY depends crucially on the solver used, the satisfiability of the problem, and its difficulty for the solver. Such irregular behavior of simplifiers was noticed by [17]. However, it seems safe to say that on problems that are difficult for a solver, 2-SIMPLIFY is quite useful. Indeed, we saw that on the encoded planning problems 2-SIMPLIFY was beneficial in conjunction with both SATZ and REL-SAT in the bulk of cases. On the larger satisfiable verification problems we saw that 2-SIMPLIFY's simplification time was much larger than the solution times. However, on the larger unsatisfiable verification times, we saw that 2-SIMPLIFY leads to two to three-fold improvement, and even more. Another interesting observation is the complementary effect of COMPACT's psl options and 2-SIMPLIFY, raising the natural question of whether the capabilities of these two simplifiers can be combined effectively. Finally, we note that the current implementation of 2-SIMPLIFY leaves much for improvement, and we believe that a more careful design can lead to much improved simplification times. This is of particular importance when one recalls that in almost all cases the simplified formulas require less time to solve. Thus, reduced performance is often due to the simplification time overhead.

We are not the first to utilize binary resolution in this area. Larrabee used the implication graph to devise a SAT algorithm in the context of test-pattern generation [15]. Larrabee systematically generates satisfying assignments consistent with the implication graph. Any assignment that satisfies the non-binary clauses is a satisfying assignment for the whole formula. This method exploits the binary portion of the formula, but it does not utilize the power of contemporary variable ordering and search techniques.

2CL [9] is a solver based on the Davis-Putnam-Logemann-Loveland algorithm [7]. At each branch point, 2CL constructs the transitive-closure of the current implication graph and uses it to choose the next branching variable. Thus, 2CL is a dynamic extension of a key aspect of 2-SIMPLIFY. It does not incorporate our *derive shared implications* step. We did not experiment with 2CL but currently, it is not considered a competitive solver.

Indeed, extending 2-SIMPLIFY to a full solver would seem to be a natural next step. This solver will be based on the DPLL algorithm. Every time an assignment is made, the implication graph will be used to detect all of its immediate implications. New binary clauses resulting from the reduction of ternary clauses will be added to the implication graph, and the *derive shared implications* step will be executed. Moreover, the implication graph may be able to provide us with valuable information for branch selection.

Unfortunately, our initial efforts in this direction were not successful. There appear to be two reasons for this. First, in our implementation, maintaining the graph transitively closed after each assignment (which typically results in a number of new binary clauses) appeared to be a serious bottleneck. Second, we were not able to come up with a quick, yet powerful branch selection heuristics that is competitive with that used by current solvers. For instance, it appears that SATZ is able to gain

Instance	CHAFF	CHAFF on simplified	2-SIMPLIFY	2-SIMPLIFY+ CHAFF
barrel8	60.27	33.02	0.12	33.14
barrel9	361.9	159.6	0.23	159.83
queueinvar16	0.28	0.35	0.71	1.06
queueinvar18	1.71	1.28	12.12	13.4
queueinvar20	2.07	1.85	20.48	22.33
fvp-3pipe	4.33	3.06	1.75	4.81
fvp-3pipe-1	4.01	4.38	0.73	5.11
fvp-3pipe-2	5.87	5.12	1.16	6.28
fvp-3pipe-3	6.81	5.64	1.73	7.37
fvp-4pipe	149.59	64.47	14.37	78.84
fvp-4pipe-1	59.19	32.3	5.34	37.64
fvp-4pipe-2	141.27	55.13	7.41	62.54
fvp-4pipe-3	153.93	65.72	9.78	75.5
fvp-4pipe-4	63.64	165.58	13.6	178.18
fvp-5pipe	383.42	68.84	79.07	145.65
fvp-5pipe-ooo	716.15	308.26	84.22	392.48
fvp-6pipe	4798.88	667.88	382.04	1049.92
fvp-6pipe-ooo	2650.02	2482.34	465.03	2947.37
fvp-7pipe	>60,000.0	2647.79	1513.25	4161.04

TABLE IX

SOLUTION TIMES FOR CHAFF ON ORIGINAL AND SIMPLIFIED UNSATISFIABILITY PROBLEMS.

much information quickly using its unit propagation step, while we have not been able to emulate that using the information in our implication graph. Very recently, Bacchus reported on his effort to extend our approach to a full solver [2]. The resulting solver is competitive with CHAFF on some SAT instances. In particular, Bacchus suggests that repeated application of our *derive shared implications* step can be very useful.

Our use of graph-based techniques, motivated by well-known 2-SAT technique is somewhat reminiscent of graph-based simplification techniques used in work on automated theorem proving such as Kowalski's Clausal Graphs [14] which formed the basis of the Markgraf Karl Refutation Procedure [19] (the implication graph can be viewed as a special case of this graph), and some rewrite-based simplification methods [11], [3]. However, the methods used there are much more sophisticated and focus on aspects of first-order theories that do not come up in the propositional setting in which we work.

Acknowledgments: I am grateful to Yefim Dinitz and Avraham Melkman for their help and advice on graph algorithms and for important comments on previous versions of this paper, and to the anonymous reviewers for their useful suggestions and comments. This work was supported in part by the Paul Ivanier Center for Robotics and Production Management.

REFERENCES

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

[2] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Proc. AAAI'02*, 2002.

[3] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[4] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI-97*, pages 203–208, 1997.

[5] R. I. Brafman. Reachability, relevance, resolution, and the planning as satisfiability approach. In *IJCAI'99*, pages 976–981, 1999.

[6] S. A. Cook. The complexity of theorem proving procedures. In *Proc. of the 3rd ACM Symposium on Theory of Computing*. ACM, 1971.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5(7):394–397, July 1962.

[8] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.

[9] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. American Mathematical Society, 1996.

[10] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of 15th Nat. Conf. AI*, pages 431–437, 1998.

[11] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (rrl). In *Proc. of RTA'89, Lecture Notes in Computer Science 355*, pages 559–563. Springer-Verlag, 1989.

[12] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th National Conference on AI (AAAI'96)*, pages 1194–1201, 1996.

[13] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proc. 16th Intl. Joint Conf. on AI (IJCAI'99)*, pages 318–325, 1999.

[14] R. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22:572–595, 1975.

[15] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, pages 4–15, January 1992.

[16] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. IJCAI-97*, 1997.

[17] I. Lynce and J. P. Marques-Silva. The interaction between simplification and search in propositional satisfiability. In *Proc. of CP'01 Workshop on Modeling and Problem Formulation*, 2001.

[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of 39th Design Automation Conference*, 2001.

[19] Hans Jürgen Ohlbach and Jörg H. Siekmann. The Markgraf Karl refutation procedure. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 41–112. MIT Press, 1991.

[20] J. A. Robinson. Automatic deduction with hyper-resolution. *Int. J. of Com. Math*, 1:227–234, 1965.

[21] B. Selman, H. J. Levesque, and D. Mitchell. Gsat: A new method for solving hard satisfiability problems. In *Proc. of the 10th National Conf. on AI (AAAI '92)*, pages 440–446, 1992.

[22] Bart Selman, Henry A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. Nat. Conf. on AI*, pages 337–343, 1994.

[23] O. Shtrichman. Tuning sat checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification 2000*, 2000.