

# Conflict-Guided Simplification for SAT

Michael L. Case<sup>1,2</sup>, Sanjit A. Seshia<sup>1</sup>,  
Alan Mishchenko<sup>1</sup>, and Robert K. Brayton<sup>1</sup>

<sup>1</sup> University of California, Berkeley

<sup>2</sup> IBM Systems and Technology Group, Austin, TX

**Abstract.** Boolean satisfiability (SAT) solvers are the computational engines for a variety of applications, including those in verification and synthesis. The NP-completeness of SAT implies that solvers often run out of time and space resources in practice, resulting in an inconclusive answer even after using up significant computational resources. In this paper, we present a method to automatically generate a simplification of the SAT problem in the event of a timeout, guided by conflict clauses generated before the timeout. The solver then operates on the simplified problem and the results are lifted back to the original problem domain. Easy problems solvable without a given timeout are solved directly with no overhead, and difficult problems that timeout are decomposed a simpler problem and series of lemma proofs, yielding impressive speedups on these hard instances. The technique differs from traditional abstraction approaches in two key ways. First, the simplification is not guaranteed to be either a sound or a complete abstraction, so a SAT or UNSAT result on the simpler problem must be lifted to the original problem; if this is not possible, the simplification is refined. Second, our method is able to simplify the problem using only the limited information available from the solver after timeout. We believe this is the first method that is able to simplify based on partial results available after a timeout.

We demonstrate our approach by applying it to Bounded Model Checking (BMC). Modifying IBM’s industrial BMC tool to utilize Conflict-Guided Simplification, we show that under a timeout of 15 minutes the enhanced BMC package is able to unroll the transition relation 10.71 times further than the previous BMC implementation. Our approach enables hundreds of time steps of some designs to be checked rather than tens, an order of magnitude improvement.

## 1 Introduction

Boolean satisfiability (SAT) solvers have become essential tools in a wide variety of applications, including verification, synthesis, static and dynamic program analysis, and planning. They are the computational engines for verification systems including model checkers and solvers for satisfiability modulo theories (SMT). However, the NP-completeness of SAT implies that solvers often run out of time and space resources in practice, resulting in an inconclusive answer, even after using significant computational resources. We will refer to such a situation as a *timeout*. For verification, a timeout corresponds not only to wasted computational resources, but also loss of valuable engineer time and effort before a product can be released.

Several methodologies have been proposed to scale up verification, including abstraction, compositional reasoning, and symmetry reduction (see, e.g., [4]). Automatic abstraction-refinement has arguably been one of the most effective techniques, especially with regard to scaling up SAT-based verification. However, all abstraction-refinement approaches require the computational engine (SAT solver) to report a precise binary answer: either *satisfiable* or *unsatisfiable*. In the case where neither answer is available due to a timeout, the abstraction-refinement loop is stuck.

We propose a method to simplify the SAT problem after a solver timeout. Upon timeout, the solver has neither a proof of unsatisfiability nor a satisfying assignment, but it does have a set of *conflict clauses* it learned as it attempted to construct a proof of unsatisfiability. Using this partial proof we build a simpler problem to feed to the solver in the next iteration. This problem is made simple enough to avoid a second timeout, and the solver results are lifted from the simplified problem to the original.

In the case that the original problem was simple enough to be decidable before the timeout expires, it is solved directly with no overhead. Therefore this technique speeds up the solving of difficult problem instances without hurting the performance on easy instances.

This *conflict-guided simplification* (CGS) is similar to abstraction, but the simpler problem is not guaranteed to be either an under-approximation or an over-approximation of the original problem. A proof of unsatisfiability for the simpler problem does not imply that the original problem is unsatisfiable, so we present a method to lift the proof to the original problem. Likewise, if the simpler problem is reported to be satisfiable, the satisfying assignment must be lifted to one for the original problem and checked. We believe this technique is the first that can simplify a problem instance based solely on the partial information from a solver timeout.

To ground this work, an application in hardware verification is explored. Bounded model checking (BMC) [6, 3] is a technique to show that a safety property holds for a bounded number of time steps in a hardware design. A time-unrolled model of the design is constructed and passed to a SAT solver. Because of the complexity of satisfiability solving, BMC typically is limited in the number of time steps that can be checked. In this paper we demonstrate that our simplification technique is effective in substantially improving the scalability of BMC on industrial designs. We have implemented our approach in the IBM internal verification tool *SixthSense*, modifying the basic BMC implementation to utilize Conflict-Guided Simplification. We show that under a timeout of 15 minutes the enhanced BMC package is able to unroll the transition relation 10.71 times further, on average, than the previous BMC implementation. Our approach thus enables hundreds of time steps of some designs to be checked rather than tens, an order of magnitude improvement.

## 2 Background

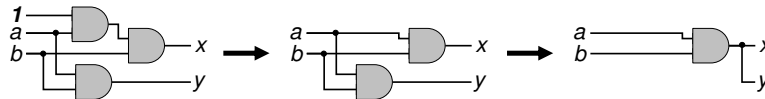
Most SAT solvers represent their problem in Conjunctive Normal Form (CNF). A clause is a disjunction of a number of Boolean literals (either a variable or its complement). A conjunction of clauses forms the CNF problem that the solver operates on. If the solver's computational resources are bounded, it will produce one of three possible outputs:

**Satisfiable** (SAT) means that an assignment to the Boolean variables has been found such that all of the clauses are satisfied. The satisfying assignment is returned.

**Unsatisfiable** (UNSAT) means no satisfying assignment exists. Techniques exist to obtain a proof as evidence of the unsatisfiability of the input CNF problem.

**Timeout** means that the solver exhausted its computational resources without any conclusion. The appearance of a “timeout” result in place of a “satisfiable” or “unsatisfiable” depends on how the resources were bounded.

While the concepts presented in this paper are general, the implementation for this paper was done using a Boolean circuit representation called an And-Inverter Graph (AIG), a logic network consisting only of And-gates and inverters. Problem simplifications in this paper are done by injecting constants, and we inject these constants directly into the AIG representation. This leads to two simplifications which are standard standard in any AIG package that can dramatically reduce the size of the logic network: *constant propagation* allows injected constants to simplify all downstream logic, and *structural hashing* discovers and simplifies equivalent gates through a quick structural-based method.



**Fig. 1.** Constant propagation and structural hashing in a Boolean circuit.

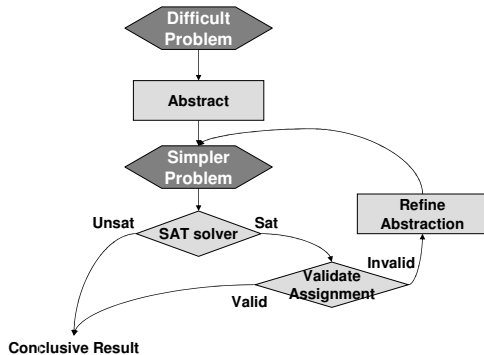
Constant propagation and structural hashing work together as illustrated in Figure 1. They are effective in dramatically decreasing the size of the AIG and corresponding CNF problem after the the problem has been simplified.

AIGs and CNFs work together to solve a circuit based problem. All simplification is done on the AIG, and all SAT solving and resultant unsatisfiable proofs involve a CNF representation. Translation from one representation to the other is an intermediate step in our representation [9].

### 3 Related Work

The closest related work is that on automatic abstraction-refinement for finite-state model checking. The early work in this area introduced *localization reduction* [10] and counterexample-guided abstraction-refinement [5]. Since then, there have been significant extensions to the approach, including *proof-based abstraction* [7, 2], hybrid approaches [1], and applications to bounded model checking [11, 12]. Baumgartner [8] describes the application of these techniques in an industrial setting.

Figure 2 outlines the basic abstraction-refinement approach. First, an abstraction of the design is formed, say using the *localization* technique [10] where parts of the design are replaced with fresh circuit inputs. This abstraction is an over-approximation of the design, so if the simpler problem is unsatisfiable, the unsatisfiability of the original problem is guaranteed. If however the abstraction is satisfiable, the satisfying assignment must be validated in the original problem. If it cannot be validated, the abstraction must be refined.



**Fig. 2.** Typical abstraction-refinement.

None of the prior work addresses SAT solver timeouts. In practice, all calls to a SAT solver are resource bounded, and the lack of a way to handle the timeout is a major obstacle to the adoption of a standard abstraction-refinement algorithm in some industrial settings. We believe our work is the first to directly address the timeouts by constructing a simplified problem based on the partial information obtained from the SAT solver after timeout. Our work does not generate over-approximations and so cannot strictly be called an abstraction technique, but the method presented in this work is inspired by the approach depicted in Figure 2.

This work uses BMC as a case study in building a simplified problem upon a solver timeout. There have been researchers who have attempted to apply abstraction techniques to BMC, notably [11] [12]. Gupta et. al. [11] use a partial assignment taken from a solver after timeout on an abstract model, simulate it on the concrete model, and use the results to prune the search on the abstract model. In contrast, our work constructs simplified problems from the solver’s conflict clauses, and both SAT and UNSAT results on the simplified problem are lifted to the original problem domain. Armoni et. al. [12] simplify the BMC problem using domain-specific knowledge, and they preserve equivalence between the original and simplified BMC problems. In contrast, our method is more general in that it does not require domain-specific knowledge. It does not preserve equivalence when simplifying a BMC instance, and this freedom to approximate is part of why it works well in practice.

## 4 Conflict-Guided Simplification

Our Conflict-Guided Simplification framework is depicted in the flowchart of Figure 3. Initially we attempt to solve the unmodified problem  $\Psi_O$  using a SAT solver. If the solver returns either SAT or UNSAT then the algorithm can terminate, but often the solver will exceed its computational resource bounds; we will refer to this latter case as TIMEOUT.

In the case of TIMEOUT, the solver has learned some information about the problem before it timed out. This information is recorded in a set of conflict clauses which can be used to simplify the problem, as discussed in Section 4.1. Then the simplified problem  $\Psi_S$  is given to a SAT solver.

If the solver returns UNSAT on the simplified problem, it also returns a proof to demonstrate the unsatisfiability. We check that this proof applies to the original problem in a step called *proof lifting*, described in Section 4.2. If the proof lifts successfully then we can terminate with UNSAT, but if the lifting fails then the simplification must be refined.

If the solver finds a satisfying assignment for the simplified problem then, just as in abstraction-refinement, we must check that the assignment satisfies the original

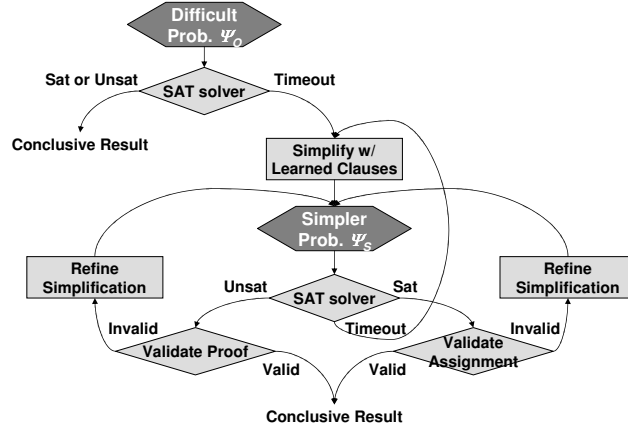


Fig. 3. Our Conflict-Guided Simplification algorithm.

problem. This process is called *assignment lifting* and is discussed in Section 4.3. If an assignment is lifted successfully the algorithm can terminate with SAT, but if the lifting fails then the simplification must be refined.

It is possible that the solver hits the resource limit while solving the simplified problem as well. In this case, we simplify the problem still more, ensuring that it gets easier to solve.

The process of refinement involves reverting some of the simplifications that were made to the problem. By analyzing the cause of the failed lifting attempt (either proof or assignment lifting), a small number of simplifications can be reverted to create a new problem that is simpler than the original yet not too-simple to produce spurious proofs or assignments. This process is discussed in Section 4.4.

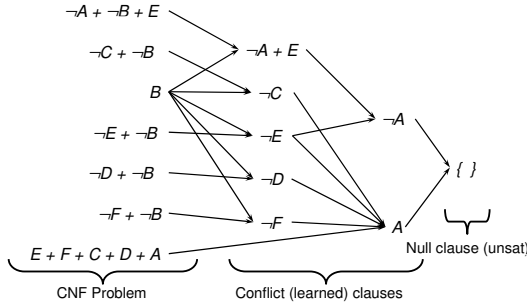
Our approach shown in Figure 3 is similar to the sample abstraction refinement flow shown in Figure 2. One important difference is that this proposed flow intelligently handles solver timeouts, and it also utilizes a proof-lifting method to compensate for the simplification not being a strict over-approximation of the original problem. Because both proofs and satisfying assignments are lifted, our overall framework is both sound and complete, given infinite computational resources. While modern DPLL-style SAT algorithms are also sound and complete, we demonstrate in Section 6 that, in practice, our approach can scale much better.

#### 4.1 Problem Simplification

If given adequate resources, the solver will run until either a satisfying assignment or proof of unsatisfiability is obtained. The solver can be thought of as performing a series of *resolution* steps, combinations of existing clauses to form stronger clauses, to prune its search space. The proof of unsatisfiability is given as a series of resolutions that derive the null clause from the initial set of CNF clauses. However, after a timeout only some of these resolution steps are available, and the solver has only a partial resolution proof.

Figure 4 shows a sample resolution proof. A CNF problem is input to the solver, and the solver derives a number of learned clauses as resolutions of the original clauses. These learned clauses are usually derived as a result of a conflicting partial

assignments and are called *conflict clauses*. If the null clause is resolved then there are no legal assignments, and the original problem is unsatisfiable.



**Fig. 4.** A sample resolution proof.

marked. Each corresponds to a signal (variable) in the original SAT problem, and a subset of these signals are replaced with constants 0 or 1, consistent with the conflict clauses from the solver.

As an example, consider the resolution proof shown in Figure 4. Suppose the solver has derived by resolution only the conflict clauses  $\neg A + E$ ,  $\neg C$ , and  $\neg E$ . The literals  $\neg A$  and  $\neg C$  only appear in one polarity. In the original problem  $A$  and  $C$  can both be replaced with a constant 0, and the resultant problem is consistent with the three conflict clauses.

---

**Algorithm 1** Simplifying with a set of conflict clauses.

---

```

1: function simplify(problem, conflict_clauses)
2:   literals := all literals from all conflict_clauses
3:   monotones := literals from literals that only appear in 1 polarity
4:   compute rank(l) for each literal l ∈ monotones // described in text below
5:   target literals := N literals with smallest rank
6:   for all literals lit ∈ target literals do
7:     var := variable referenced in lit
8:     if lit is complemented then
9:       Replace var with 0 in problem
10:    else
11:      Replace var with 1 in problem
12:    end if
13:  end for
14: end function

```

---

In Algorithm 1, selecting which monotone literals to simplify and how many such literals to simplify is important to the overall success of Conflict-Guided Simplification. This selection is performed based on a rank function *rank*. In our experiments with simplifications on logic circuits, we found two factors to be particularly important in selecting a literal to simplify:

1. The literal must correspond to a wire in the circuit close to the circuit inputs, since this helps to maximize the amount of downstream logic that can be simplified. For a literal  $l$ , let  $\delta(l)$  denote the shortest distance of  $l$  from a circuit input, where distance is measured in number of gates.

After a timeout, a number of conflict clauses have been generated, and this information can be leveraged to simplify the problem. The simplification should be done so that conflict clauses are still valid, thus preserving any future proof that depends on these clauses.

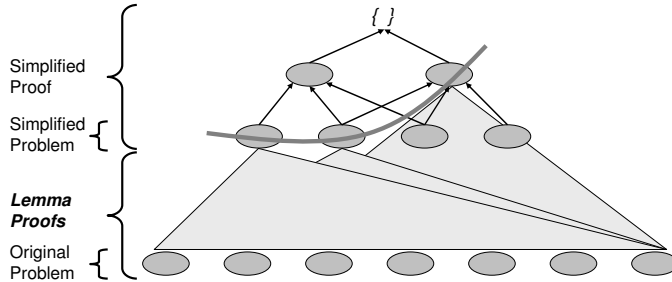
The conflict-clause-preserving simplifications are simple, as illustrated in Algorithm 1. The *monotone* literals that only appear in one polarity across all conflict clauses are

2. Simplifying with monotones from short clauses is thus less likely to result in spurious counterexamples and proofs, because a short conflict clause corresponds to a stronger lemma that constrains the search space of the SAT solver. Therefore, we measure the length of the shortest clause in which a monotone literal  $l$  appears, denoted by  $\lambda(l)$ .

To obtain a ranking function  $rank$ , we combine the above quantities linearly, so that  $rank(l) = a_1\delta(l) + a_2\lambda(l)$ , where  $a_1$  and  $a_2$  are chosen heuristically. A parameterizable number  $N$  of the highest scoring monotone literals are then selected for simplification. In our implementation we found that  $a_1 = 2$ ,  $a_2 = 1$ , and  $N = 10$  worked well.

## 4.2 Proof Lifting

If the SAT solver returns UNSAT on the simplified problem then it has produced a proof of the unsatisfiability of the simpler problem but not necessarily of the original. In this section we discuss how to *lift* this proof to the original problem.



**Fig. 5.** Lifting a simplified proof to the original problem.

Figure 5 illustrates the lifting of a resolution proof. Any cut across the simpler resolution proof gives a set of clauses termed *lemmas*. If each of these can be shown to hold in the original problem, then a composition of the lemma proofs and the simplified problem proof is sufficient to demonstrate the unsatisfiability of the original problem. In this way, the lemmas provide a way to decompose a complicated proof into a series of easier proofs, a decomposition that can give significant speedups in practice.

Algorithm 2 is used to lift the resolution of the simplified problem to the original. It uses a depth-first traversal over the simplified proof and attempts to lift the lowest-possible clauses that appear in that proof. Upon failure, it will try to lift a clause that appears higher in the resolution proof. In this way, the cut across the simplified proof that gives the set of lemmas is dynamic. It starts as a cut across the lowest levels of the resolution proof and rises only as the proofs of lemmas fail.<sup>3</sup>

At the heart of Algorithm 2 is an attempt to verify whether a clause  $C$  holds in the original problem  $\Psi_O$ . This can be implemented by checking whether  $\Psi_O \wedge \neg C$

<sup>3</sup> Our optimized implementation executes Algorithm 2 twice. The first execution limits the child CGS engine to 50 backtracks with no simplification. In this case, a timeout when attempting to lift a child clause triggers an attempt to lift a parent clause. This helps to find an “easy” set of lemmas, if such a set exists. The second execution of Algorithm 2 uses a backtrack limit of 500 with CGS simplifications enabled.

---

**Algorithm 2** Lifting a resolution proof.

---

```
1: function liftProof(proof)
2:   return justifyClause(proof.nullClause)
3: end function
4:
5: function justifyClause(C)
6:   for all child clauses c of C do
7:     justifyClause(c) // recurse to the leaves of the proof DAG for C
8:   end for
9:   if all children clauses lifted then
10:    return “lifted” // children lifted  $\Rightarrow$  parent lifted
11:   else if C = proof.nullClause &&  $\exists$  unlifted child clause then
12:    return “not lifted” // null clause’s children must be lifted
13:   else
14:     // recursive call to CGS,  $\Psi_O$  is the original SAT problem
15:     result := CGS_satSolve( $\Psi_O \wedge \neg C$ )
16:     return (result = SAT) ? “falsified” : “lifted”
17:   end if
18: end function
```

---

is unsatisfiable. As this satisfiability check can also be hard, instead of invoking a regular SAT solver, we invoke another instance of a SAT solver based on CGS. Note that the input to each such invocation of CGS is a simpler problem than  $\Psi_O$ , as  $\neg C$  is a partial assignment to the variables in  $\Psi_O$ , so the second invocation of CGS operates on a simpler problem than the first invocation. Therefore, the chain of recursive CGS invocations is of finite length.

It is possible for the iterated lifting to fail. In this case, the cut will rise to the top of the resolution proof and there exists a path from the null clause to the bottom of the resolution proof for the simplified problem such that each lemma along the path failed to lift. The next step is refinement where some of the simplifications that were done to the original problem are reverted such that the lemmas that failed to lift will be eliminated.

### 4.3 Assignment Lifting

If SAT solver finds a satisfying assignment to the simplified problem  $\Psi_S$ , it does not necessarily satisfy the original SAT instance  $\Psi_O$ . Therefore the assignment must be *lifted*.

The assignment has values for all of the inputs in the simpler problem, but this is only a subset of the inputs for the original problem. We must extend this partial assignment to a complete assignment and also verify that the assignment demonstrates the satisfiability of the original problem. This can be done with the following call to a SAT solver:

$$\text{satSolve}(\textit{free inputs}, \textit{assigned inputs}, \textit{original problem})$$

where the *assigned inputs* is the partial assignment from the simpler problem.

If this is satisfiable then the assignment has been extended to the original problem and this is a valid satisfying assignment. If UNSAT is returned, then the assignment from the simplified problem is spurious. We then refine the simplified problem by

reverting some of the simplifications such that this spurious assignment will not appear again.

#### 4.4 Refinement

If either a proof or a satisfying assignment fails to lift from the simplified problem  $\Psi_S$  to the original instance  $\Psi_O$ , the simplified problem must be refined. The simplified problem has been derived from the original by replacing a number of variables by constants (Section 4.1). Due to these injected constants, one of two things has gone wrong:

- An assignment fails to lift:** The simplified problem  $\Psi_S$  was satisfiable under a given input assignment  $\alpha$ , but the original problem  $\Psi_O$  is not satisfiable under  $\alpha$ . In other words,  $\Psi_S(\alpha) = 1$  but  $\Psi_O(\alpha) = 0$ .
- A proof fails to lift:** The simplified problem  $\Psi_S$  was proven unsatisfiable, and clause  $C$  was part of the proof. Thus,  $\Psi_S \wedge \neg C$  is UNSAT. However, we found that  $\Psi_O \wedge \neg C$  is SAT. Let  $\alpha$  be an assignment for which  $\Psi_O \wedge \neg C$  evaluates to 1.

---

**Algorithm 3** Refinement to correct spurious behaviour.

---

```

1: function refine(simplifying_consts, assign, F)
2:   return refine_rec( $\emptyset$ , simplifying_consts, assign, F)
3: end function
4:
5: function refine_rec(good_consts, unknown_consts, assign, F)
6:   // Precondition: good_consts is a safe simplification set for F
7:   // Postcondition: the returned set of constants is a safe simplification set for F
8:    $F' = \text{simplify}(F, \text{good\_consts} \cup \text{unknown\_consts})$ 
9:
10:  if  $F(\alpha) \neq F'(\alpha)$  then
11:    if ( $|\text{unknown\_consts}| = 1$ ) then
12:      return good_consts // All of unknown_consts are bad.
13:    else
14:       $(\text{lhs}, \text{rhs}) = \text{bisect}(\text{unknown\_consts})$  // Bisect the set of constants
15:       $\text{lhs}' = \text{refine\_rec}(\text{good\_consts}, \text{lhs}, \text{assign})$ 
16:      return refine_rec( $\text{lhs}'$ ,  $\text{rhs}$ , assign)
17:    end if
18:  else
19:    return  $\text{good\_consts} \cup \text{unknown\_consts}$  // All simplifying constants are safe.
20:  end if
21: end function

```

---

In both of the above cases, the injected constants have caused the simplified and original problems to differ on some input assignment. We can formulate both of the above situations as instances  $F$  and  $F'$  of Boolean formulas on the same input variables that evaluate differently on assignment  $\alpha$ . In the case of assignment lifting,  $F = \Psi_O$  and  $F' = \Psi_S$ . For proof lifting,  $F = \Psi_O \wedge \neg C$  and  $F' = \Psi_S \wedge \neg C$ .

Let  $S$  be the set of simplifying constants used to obtain  $\Psi_S$  from  $\Psi_O$ . We wish to discover a subset  $S_{\text{good}}$  of  $S$  such that if  $S_{\text{good}}$  was used in place of  $S$ ,  $F$  and  $F'$  would evaluate the same on  $\alpha$ . Formally, we will say that  $S_{\text{good}}$  is a *safe simplification set* for  $F$  on  $\alpha$ .

Algorithm 3 illustrates our procedure to identify the set  $S_{\text{good}}$  of safe constants. The algorithm is passed the current set of simplifying constant assignments *simplifying\_constants*, an assignment *assign*, and a Boolean formula  $F$  such that *simplifying\_constants* is not a safe simplification set for  $F$  on *assign*.

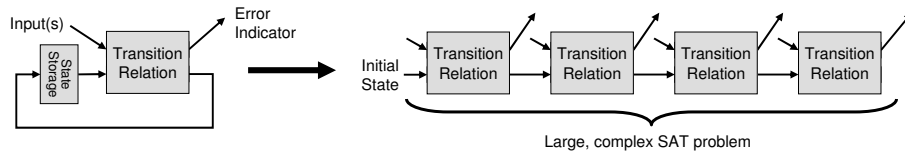
The algorithm works by repeatedly bisecting the set of simplification constants until a single constant simplification is found to be an unsafe simplification for  $F$  on *assign*. At this point, we know that the single constant is responsible for the difference and the constant is discarded. The only simplification constants that survive are those on which  $F$  and the resulting  $F'$  do not differ for assignment *assign*. In fact, because proof or assignment lifting failed, it is guaranteed that the returned set will be a strict subset of *simplifying\_constants*. This in turn guarantees that the refinement procedure makes progress, eliminating at least one simplifying constant in each refinement iteration. In the worst case, all simplification constants are eliminated, leaving us with the original problem  $\Psi_O$ .

## 5 Application to Bounded Model Checking

The previous sections have developed an efficient method to decide the satisfiability of complex problems by simplifying after a timeout. To ground these concepts, we explore an application of Conflict-Guided Simplification to hardware model checking.

Consider the RTL specification of a hardware design. The design is a network of gates and state-holding elements that together embody a finite state machine. We focus on the verification of safety properties, specifically Bounded Model Checking (BMC). In BMC we want to check that the safety property is not violated in any of the first  $k$  cycles from the initial state.

BMC is illustrated in Figure 6. A logic design that has internal state can be partitioned into state storage and a state transition relation that computes the next state as a function of the current state and the inputs. Suppose the circuit is *unrolled* by adjoining several copies of the transition relation. The first copy is fed with the design’s initial state, and each subsequent copy is fed with the previous copy’s output state. Each copy of the state-free circuit represents a different time step in the design’s execution, and so the resulting complex model can check the safety property across several time frames.



**Fig. 6.** Bounded model checking.

BMC translates this unrolled circuit to CNF and feeds the problem to a SAT solver. This works well for a small number of time frames, but the complexity of BMC grows with the number of time frames. BMC is amenable to decomposition with Conflict-Guided Simplification, and in the remainder of this paper we explore the application of this framework to BMC.

BMC implementations usually check each frame in a separate SAT call, resulting in many related SAT problems. In this context, it makes sense to preserve problem simplifications between CGS calls, and Figure 3 was modified slightly to immediately

simplify the current problem if the simplifications used in the previous problem are available.

## 6 Experimental Results

Conflict-Guided Simplification was implemented inside the IBM internal verification tool *SixthSense*. An additional engine named CGS was added to *SixthSense*, and this engine has an interface that is identical to the normal SAT solver. This allows it to be used as a drop-in replacement for normal SAT in the industrial BMC flow.<sup>4</sup>

Conflict-Guided Simplification is evaluated on satisfiability problems encountered while doing BMC over the Intel benchmarks from the Hardware Model Checking Competition, held at CAV 2007 [13]. All experiments were run on a 1.8 GHz Pentium M laptop running Linux. The CGS implementation was set to simplify all satisfiability problems that cannot be solved within 5 seconds or 1000 solver backtracks – the solver timeout.

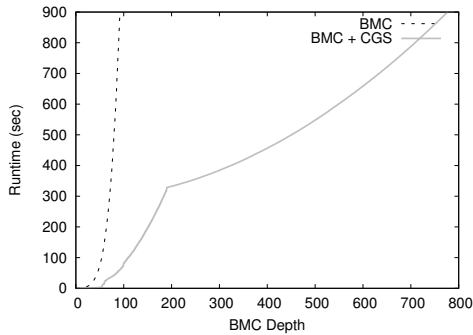
**Table 1.** CGS performance on the CAV '07 Intel benchmarks [13].

Benchmark (Post Synthesis)			BMC Depth (900 sec)		Per-problem Stats.			Normalized Runtime Breakdown		
Name	ANDs	Regs.	Normal	CGS	Simp. Consts	Num. Refines	Total Iters.	Sat Solving	Simp- lifying	Lifting Results
intel_003	683	47	265	830	37.13	0.00	1.00	0.99	0.00	0.00
intel_007	10084	607	24	450	32.02	0.01	1.03	0.88	0.01	0.11
intel_009	60718	2890	20	179	49.00	0.03	1.08	0.73	0.04	0.23
intel_010	5369	367	62	684	151.31	0.03	1.06	0.83	0.01	0.17
intel_011	5312	361	65	774	65.91	0.01	1.01	0.80	0.01	0.19
intel_014	42289	2372	22	180	63.57	0.18	1.37	0.80	0.04	0.17
intel_015	5336	381	63	809	65.36	0.01	1.02	0.79	0.01	0.20
intel_016	18911	1353	42	374	64.90	0.05	1.12	0.78	0.02	0.20
intel_017	4183	401	82	388	150.87	0.10	1.30	0.80	0.02	0.18
intel_018	4152	321	67	901	86.89	0.02	1.02	0.78	0.01	0.21
intel_019	4382	338	67	872	99.64	0.04	1.05	0.79	0.01	0.20
intel_020	3573	233	69	1005	72.71	0.01	1.01	0.81	0.01	0.18
intel_021	3669	244	71	971	87.09	0.02	1.02	0.80	0.01	0.19
intel_022	5318	358	69	783	88.02	0.05	1.05	0.80	0.01	0.19
intel_023	3522	240	75	981	74.85	0.01	1.01	0.80	0.01	0.19
intel_024	3530	239	71	989	95.70	0.02	1.02	0.79	0.01	0.20
intel_025	9047	654	55	566	61.99	0.01	1.04	0.79	0.01	0.19
intel_026	3833	349	92	777	176.89	0.02	1.02	0.77	0.01	0.22
intel_027	55423	2779	19	179	37.41	0.03	1.06	0.70	0.04	0.26
intel_028	76224	3947	16	53	16.09	0.09	1.32	0.90	0.10	0.00
intel_029	5471	389	68	813	77.97	0.03	1.03	0.81	0.01	0.18
Average				10.71x		0.04	1.08	0.81	0.02	0.17

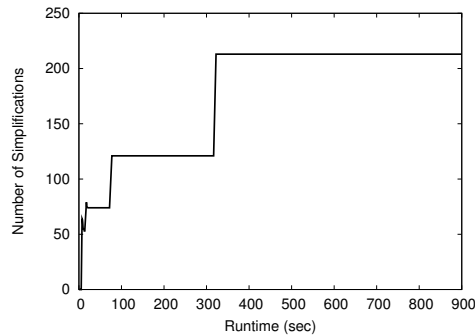
Table 1 shows the performance of BMC and CGS. BMC is run for 900 seconds on each of these designs. The column “Normal” gives the number of timeframes that BMC checked using the existing implementation in *SixthSense*. The column “CGS” gives the number of timeframes that BMC is able to check when its SAT solver is replaced with our CGS framework. On average, CGS enables BMC to proceed 10.71 times deeper into the design before it times out.

The next columns in Table 1 give average statistics per CGS problem solved. Each CGS problem is a check of a single design at a single BMC depth, so there are 13,558 CGS problems in total. The column “Simp. Consts.” gives the number of simplifying constants that were injected into each problem, on average. The implementation

<sup>4</sup> This modular design allows Conflict Guided Simplification to assist many verification algorithms in the future, not just BMC.



**Fig. 7.** BMC runtime comparison for intel\_026: 3833 ANDs, 349 registers.



**Fig. 8.** Number of simplifications used in the “BMC+CGS” run on intel\_026.

injects 10 additional constants into a design in the event of a timeout, a number that was found to be large enough to allow the simplified problem to be easily solved yet small enough to prevent an abundance of spurious counterexamples and proofs. The column “Num. refines” gives the average number of times the set of simplifying constants had to be refined. Because of the small number of constants that were injected, refinement was rarely needed. The last column in this group is “Total iters” that gives the average number of iterations, or attempts to solve a simplified problem. On average, only one attempt was needed in each CGS problem.

The final group of columns in Table 1 give a breakdown of the runtime within the CGS algorithm. On average, 81% of the time was spent in the SAT solver while solving either the original or simplified problem. Simplifying difficult problems was relatively simple, consuming only 2% of the time. 17% of the time was spent trying to lift results from a simplified problem to the original problem. This runtime breakdown is sensitive to the time bound on the SAT solver. With a long timeout, CGS will find less need to simplify difficult problems. However, the resultant resolution proofs will be large and difficult to lift, thus a shorter timeout is necessary for the lifter runtimes to be manageable.

The amount of time spent trying to solve a problem before simplification occurs (the timeout value) is a tuning parameter that affects the runtime breakdown but has less impact on the total runtime. Varying this timeout by  $\pm 20\%$  caused the BMC depth achievable within 900 seconds to decrease between only 2.3 and 2.8%.

Figures 7 - 8 examine the performance of CGS-enhanced BMC in more detail on the “intel\_026” benchmark. Figure 7 shows the runtime as a function of the current BMC step. The “BMC” plot shows the performance of BMC as it is currently implemented in *SixthSense*, and the “BMC+CGS” plot shows the performance after the SAT solver has been replaced with the CGS framework inside of the BMC implementation. These two BMC versions have similar performance until CGS begins to simplify the problem, and after this CGS is significantly faster.

Figure 8 shows the number of simplifying constants injected by CGS as a function of time. The number of simplifications increases whenever the SAT solver times out and more constants are injected, and the simplifications decrease when a spurious behaviour is detected and refinement is called.

Figures 7 - 8 illustrate an important trend that is responsible for the speedups offered by Conflict-Guided Simplification. After some time, the current set of simplifications are able to provide a simplified problem that is easy to solve, and the

simplifications lead to a simplified resolution proof that is easy to lift. This establishes a steady state where no new simplifications are needed and no simplifications are discarded, enabling BMC to proceed rapidly through a large number of time steps. Note that “intel.026” has three steady states. At depths 99 (71 sec) and 189 (316 sec) the simplifications from the previous fixed point no longer adequately simplified the problem, and the solver timed out. This triggered a re-simplification, and CGS settled into a new steady state. This steady state settling means that on average little simplification or refinement is needed, and CGS is able to efficiently partition each difficult problems into a simpler problem plus a set of simple lemma proofs.

Similar plots on some other benchmarks are included in the Appendix.

## 7 Conclusion and Future Work

In this paper we have presented a method to solve difficult SAT problems. It works by using the information from a timeout to simplify the problem. A SAT solver is then applied to the simplified problem and the results are lifted back to the original. To our knowledge, this is the first method that is able to simplify a problem using only the limited information available after a solver timeout.

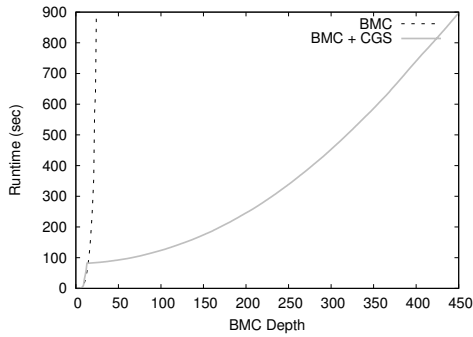
An application to bounded model checking was explored. The Conflict-Guided Simplification framework was used to assist in the SAT solving of difficult BMC instances. Our technique enabled BMC to check a factor of 10.71 more time frames in the same amount of runtime.

In addition to being effective in improving the capacity of BMC, Conflict-Guided Simplification should be effective in helping to solve any type of difficult SAT problems. For this reason, our technique is really a “resource-aware” approach to SAT solving that was plugged into an industrial BMC implementation. In future work we will explore the application of our CGS SAT solver in other problem domains.

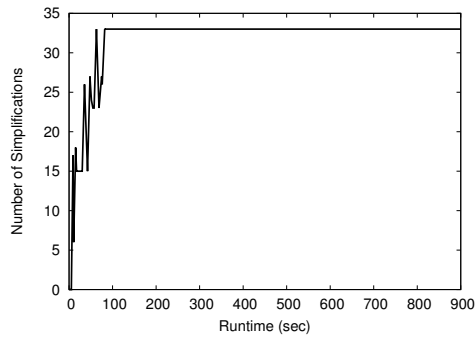
## References

1. N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan, and K.L. McMillan, “An Analysis of SAT-based Model Checking Techniques in an Industrial Environment,” in *CHARME* 2005.
2. N. Amla and K.L. McMillan, “A hybrid of counterexample-based and proof-based abstraction,” in *FMCAD* 2004.
3. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, “Bounded Model Checking,” in *Advances in Computers*, volume 58, Academic Press, 2003.
4. E. Clarke, O. Grumberg, and D. Peled, “Model Checking,” MIT Press, 2000.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, “Counterexample-Guided Abstraction Refinement,” *CAV* 2000, pages 154-169.
6. E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded Model Checking Using Satisfiability Solving,” *Formal Methods in System Design*, volume 19 issue 1, July 2001.
7. A. Gupta, M. Ganai, Z. Yang, and P. Ashar, “Iterative Abstraction using SAT-based BMC with Proof Analysis,” in *ICCAD* 2003.
8. J. Baumgartner, “Integrating FV Into Main-Stream Verification: The IBM Experience,” Tutorial Given at *FMCAD* 2006.
9. M.N. Velev, “Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors,” in *ASPDAC* 2004.
10. R. P. Kurshan, “Computer-Aided-Verification of Coordinating Processes,” Princeton University Press, 1994.
11. A. Gupta and O. Strichman, “Abstraction Refinement for Bounded Model Checking,” in *CAV* 2005.
12. R. Armoni, L. Fix, R. Fraer, T. Heyman, M. Vardi, Y. Vizel, and Y. Zbar, “Deeper Bound in BMC by Combining Constant Propagation and Abstraction,” in *ASPDAC* 2007.
13. Hardware Model Checking Competition 2007 Benchmark Suite, in *CAV* 2007.

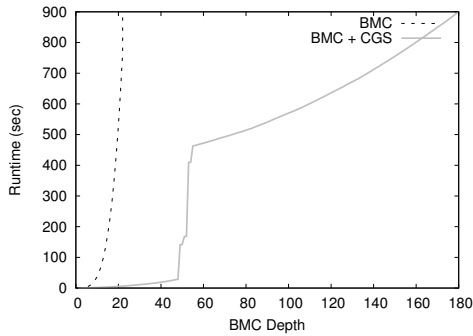
## Appendix: Additional Runtime Plots



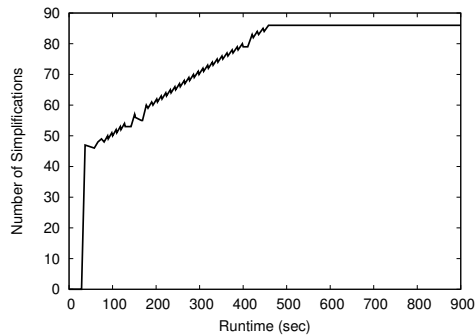
**Fig. 9.** BMC runtime comparison for intel\_007: 10084 ANDs, 607 registers.



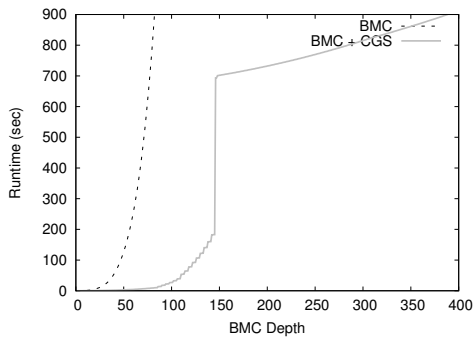
**Fig. 10.** Number of simplifications used in the “BMC+CGS” run on intel\_007.



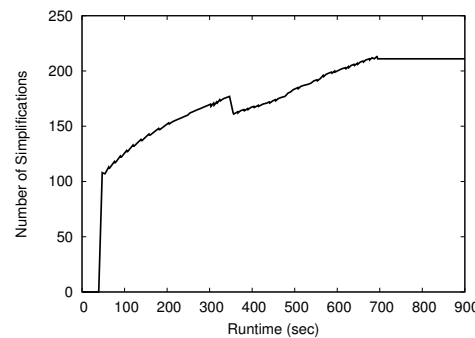
**Fig. 11.** BMC runtime comparison for intel\_014: 42289 ANDs, 2372 registers.



**Fig. 12.** Number of simplifications used in the “BMC+CGS” run on intel\_014.



**Fig. 13.** BMC runtime comparison for intel\_017: 4183 ANDs, 401 registers.



**Fig. 14.** Number of simplifications used in the “BMC+CGS” run on intel\_017.