Conflict-Guided Simplification for SAT

Michael L. Case[1,2], Sanjit A. Seshia[1],
Alan Mishchenko[1], and Robert K. Brayton[1]

[1] University of California, Berkeley
[2] IBM Systems and Technology Group, Austin, TX

Abstract. Boolean satisfiability (SAT) solvers are the computational engines used in a variety of applications, including verification and synthesis. The NP-completeness of SAT implies that solvers often run out of space and time resources. In this paper, we present a method called conflict guided simplification (CGS), which uses conflict clauses, generated during a limited solution attempt, to formulate a set of simpler problems to be solved. This set is not equivalent to the original problem, but their proofs (or counterexamples) can be used as a guide to a proof of the original. This method represents a modification to any SAT solver, which can solve "easy" problems with no overhead, but for difficult problems, CGS kicks in at a user specified time limit, yielding impressive speedups on "hard" instances. We believe this is the first method that uses partial results generated during a proof attempt on an *unsat* problem to simplify it and shorten its proof.

We demonstrate this approach by applying it to Bounded Model Checking (BMC) as a source of hard SAT problems. IBM's industrial BMC tool was modified to utilize CGS. We experimented by testing the maximum depth that BMC could be done within a time limit of 15 minutes (with CGS enabled to kick in at 5 seconds or 1000 backtracks) on a set of hard model checking benchmarks. The new method on average was able to probe 10.71 times deeper than the unmodified BMC. Thus, hundreds of time steps on some designs can 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 exhaust time and space resources, resulting in an inconclusive answer after using significant computational effort. For verification, a timeout can lead to a loss of valuable engineering time and effort possibly delaying the release of a product.

Several methodologies have been proposed to scale up verification, including abstraction, compositional reasoning, and symmetry reduction (see, e.g., [4]). Automatic abstraction-refinement has been arguably one of the most effective techniques, especially for 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 anticipate when a SAT problem might timeout, and to switch to a simplification mode. At any point in the solution attempt, a solver has a set of *conflict clauses* learned as it attempted to construct a proof of unsatisfiability. Using this partial proof, we construct a set of simpler problems which have been made simple enough be solved more easily. Their solutions are "lifted" from the simplified problems to the originals.

In the case that the original problem was already simple enough, it is solved directly with no overhead. Thus this technique speeds up the solvers performance on the difficult problems without hurting it 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 satisfiable, the satisfying assignment must be lifted to one for the original problem and checked. We believe this technique is the first that uses partial information from a solver to simplify an *unsat* problem.

To ground this work and as a source of hard SAT problems, an application in hardware verification was 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 CGS embedded in a standard SAT solver can substantially improve the scalability of BMC. This approach was implemented in the IBM internal verification tool *SixthSense*, modifying the basic BMC implementation to utilize CGS. We show that under a time limit of 15 minutes, the enhanced BMC is able to unroll the transition relation on average ten times further than the BMC without CGS. Thus it enables hundreds of time steps of some designs to be checked rather than tens.

2 Background

Most SAT solvers use Conjunctive Normal Form (CNF), a conjunction of clauses, which are disjunctions of a number of Boolean literals (either a variable or its complement). It will 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. A satisfying assignment is returned.

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

Timeout means that the solver exhausted its computational resources without any conclusion.

While the concepts presented in this paper are general, our implementation was done using an And-Inverter Graph (AIG) representation of a logic network from which a CNF is extracted. Problem simplification is done by injecting constants directly into the AIG. This leads to two simplifications, standard in any AIG package, which can dramatically reduce the size of the logic network and hence its CNF:

constant propagation allows injected constants to simplify all downstream logic, and *structural hashing* discovers and simplifies equivalent gates through a quick structural-based method.

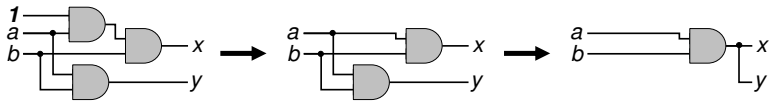

Fig. 1. Constant propagation and structural hashing in a Boolean circuit.

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 of the simplified problem.

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 are CNF based. Translation from one representation to the other can be done efficiently [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 becomes an over-approximation of the design, so if the simpler problem is unsatisfiable, the unsatisfiability of the original problem is guaranteed. Otherwise the satisfying assignment produced must be validated in the original problem. If it cannot be validated, the abstraction must be refined.

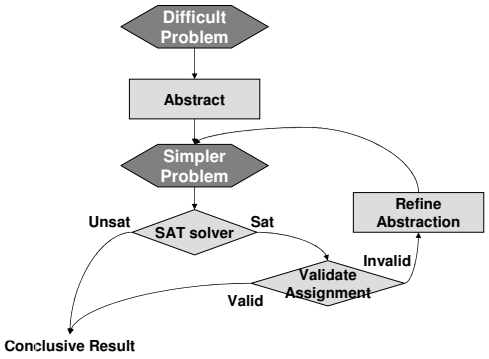

Fig. 2. Typical abstraction-refinement.

Our work does not generate over-approximations and so is not an abstraction technique, but the method is inspired by the approach depicted in Figure 2.

This paper uses BMC in a case study as a source of difficult SAT problems. Others have attempted to apply abstraction techniques to BMC, notably [11] [12]. Gupta et. al. [11] use a partial assignment 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 BMC using domain-specific knowledge,

and preserve equivalence between the original and simplified BMC problems. Our method does not require domain-specific knowledge and does not preserve equivalence when simplifying. This additional freedom to approximate is part of why it works well in practice.

4 Conflict-Guided Simplification

The CGS framework is depicted in the flowchart of Figure 3. Initially the solver works on the unmodified problem Ψ_O. A user specified time limit controls when the solver should switch to simplification. If SAT or UNSAT is determined before then, the algorithm terminates. Otherwise the solver kicks in to CGS mode.

At the point where CGS is invoked, the solver has learned some information about the problem in the form of a set of conflict clauses which are used to form a simplified problem, as discussed in Section 4.1. The solver then recurs on the simplified problem Ψ_S.

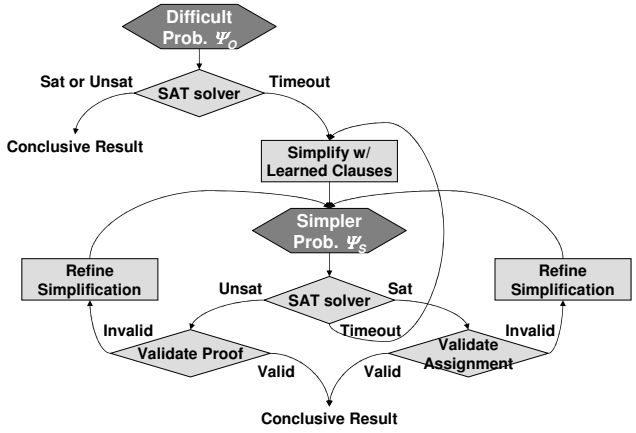

Fig. 3. Our Conflict-Guided Simplification algorithm.

If UNSAT is returned on the simplified problem, a proof of unsatisfiability is returned also. This proof is checked if it applies to the original problem in a step called *proof lifting*, described in Section 4.2. If the proof lifts successfully then UNSAT is returned, but if the lifting fails the simplification must be refined.

If the solver finds a satisfying assignment for the simplified problem then, as in abstraction-refinement, it is checked if it satisfies the original problem. This is called *assignment lifting* and is discussed in Section 4.3. If lifting is succesful, SAT is returned, but if it fails then the simplification must be refined. In any recursive call, the solver may again kick into CGS mode, which causes more simplification.

Refinement involves undoing some of the simplifications (constants injected) that were made. By analyzing the cause of the failed lifting attempt (either proof or assignment lifting), a small number of simplifications can be undone 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. An important difference is that our method utilizes proof-lifting to compensate for a 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 scales much better.

4.1 Problem Simplification

A SAT solver can be thought of as performing a series of *resolution* steps, combinations of existing clauses to form stronger clauses, to prune the search space. The proof of unsatisfiability is given as a series of resolutions that derives the null clause from the initial set of CNF clauses. However, at any point in time, only some of these resolution steps are available and thus only a partial resolution proof.

Figure 4 shows a sample resolution proof. A CNF is input to the solver, and the solver derives a number of learned clauses as resolutions of the original clauses. Each learned clause is usually derived as a result of a conflicting partial assignment and is called a *conflict clause*. If the null clause is resolved, then there are no legal assignments, and the original problem is unsatisfiable.

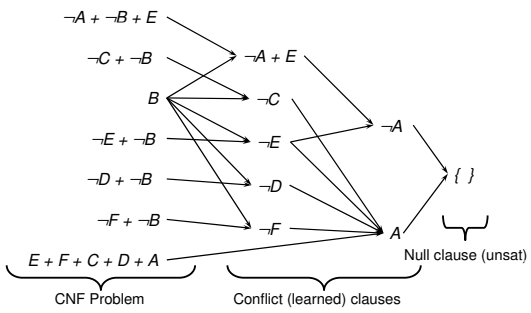

Fig. 4. A sample resolution proof.

Att any time, a number of conflict clauses has been generated, which can be leveraged to simplify the problem. The simplification should be done so that all conflict clauses are still valid, thus preserving any future proof that depends on these.

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

As an example, consider the resolution proof shown in Figure 4. Suppose the solver has derived by resolution 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.

In Algorithm 1, selecting which and how many monotone literals to simplify is important to the overall success of CGS. This selection is based on a rank function *rank*. In our experiments, we found two factors to be particularly important in selecting a literal to simplify:

1. The literal should 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.
2. Simplifying with monotone literals from short clauses is less likely to result in spurious counterexamples and proofs, because a short conflict clause corresponds

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
```

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 rank 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 the proof of unsatisfiability is not necessarily valid for the original problem. In this section we discuss how to *lift* this proof to the original problem.

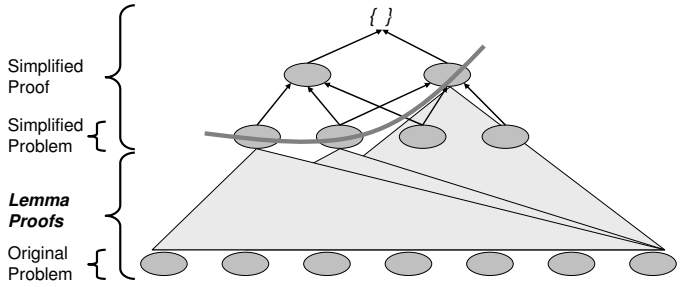

Fig. 5. Lifting a simplified proof to the original problem.

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 lemmas 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. This decomposition can give significant speedups in practice. **Doesn't this mean that if we put all the proofs together into a**

single monolithic resolution proof (like what Satrajit did for CEC) then we would have a shorter proof?

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. On 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. [3]

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 ⇒ parent lifted
11:   else if C = proof.nullClause && ∃ unlifted child clause then
12:     return "not lifted" // null clause's children must be lifted
13:   else
14:     // recursive call to CGS, Ψ_O is the original SAT problem
15:     result := CGS_satSolve(Ψ_O ∧ ¬C)
16:     return (result = SAT) ? "falsified" : "lifted"
17:   end if
18: end function
```

At the heart of Algorithm 2 is an attempt to verify if a clause C holds in the original problem Ψ_O. This can be implemented by checking if $\Psi_O \wedge \neg C$ is unsatisfiable. As this satisfiability check can also be hard, we recur by calling the CGS-based solver again. Note that the input to each such invocation is a simpler problem than Ψ_O, as $\neg C$ is a partial assignment to the variables in Ψ_O, so a subsequent invocation operates on a simpler problem than the preceding one. Thus, the chain of recursive CGS calls 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 a path from the null clause to the bottom of the resolution proof will exist such that each lemma along the path failed to lift. This will cause a refinement step to be done where some of the simplifications of the original problem are undone in such a way that the lemmas that failed to lift will be eliminated.

[3] 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 failure to lift a child clause within this limit, 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.

4.3 Assignment Lifting

Any satisfying assignment to the simplified problem Ψ_S does not necessarily satisfy the original SAT instance Ψ_O and must be *lifted*. The assignment has values for all of the variables of the simpler problem, but this is only a subset of the inputs for the original problem. This partial assignment must be extended to a complete assignment which demonstrates the satisfiability of the original problem. This can be done with the following call to a SAT solver:

`satSolve`(*free inputs*, *assigned inputs*, *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 a valid satisfying assignment of original problem. If UNSAT is returned, then this partial assignment is spurious. We then refine the simplified problem by undoing 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, 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 failed to lift: The simplified problem Ψ_S was satisfiable under a given input assignment α, but the original problem Ψ_O is not satisfiable under α. In other words, $\Psi_S(\alpha) = 1$ but $\Psi_O(\alpha) = 0$.

A proof failed to lift: The simplified problem Ψ_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 α be an assignment for which $\Psi_O \wedge \neg C$ evaluates to 1.

In both of the above cases, the injected constants have caused the simplified and original problems to differ on some input assignment. Both of the above situations can be formulated as instances F and F' of Boolean formulas on the same input variables that evaluate differently on assignment α. 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 Ψ_S from Ψ_O. We wish to discover a subset S_{good} of S such that if S_{good} were used in place of S, F and F' would agree on α. Formally, we will say that S_{good} is a *safe simplification set* for F on α.

Algorithm 3 illustrates our procedure to identify the set S_{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

Algorithm 3 Refinement to correct spurious behaviour.

```
1: function refine(simplifying_consts, assign, F)
2:   return refine_rec(∅, 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' = simplify(F, good_consts ∪ unknown_consts)
9:
10:  if F(α) ≠ F'(α) then
11:    if (|unknown_consts| = 1) then
12:      return good_consts // All of unknown_consts are bad.
13:    else
14:      (lhs, rhs) = bisect(unknown_consts) // Bisect the set of constants
15:      lhs' =  refine_rec(good_consts, lhs, assign)
16:      return refine_rec(lhs',          rhs, assign)
17:    end if
18:  else
19:    return good_consts ∪ unknown_consts // All simplifying constants are safe.
20:  end if
21: end function
```

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 Ψ_O.

5 Application to Bounded Model Checking

The previous sections developed an efficient method to decide the satisfiability of complex problems by invoking simplification after a limited amount of computation has been done. To ground these concepts, we explored the use of CGS on hardware model checking.

An RTL specification of a hardware design is equivalent to a network of gates and state-holding elements that together embody a finite state machine. We focus on the verification of safety properties using Bounded Model Checking (BMC), which checks 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 with 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 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.

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

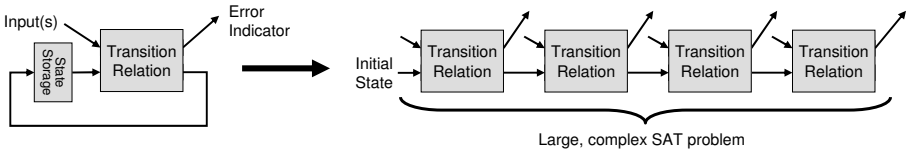

Fig. 6. Bounded model checking.

BMC grows with the number of time frames. BMC is amenable to the use of a CGS-based solver , and we explore the application CGS 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

CGS was implemented inside the IBM internal verification tool *SixthSense*. An additional engine named CGS was added to *SixthSense*, with 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.[4]

CGS was evaluated on SAT problems encountered during BMC on Intel benchmarks from the Hardware Model Checking Competition, held at CAV 2007 [13]. All experiments were done on a 1.8 GHz Pentium M laptop running Linux. The time limit when CGS invocation happened was set to 5 seconds or 1000 solver backtracks.

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

[4] This modular design allows CGS to replace any SAT solver and hence to be used in any SAT based application.

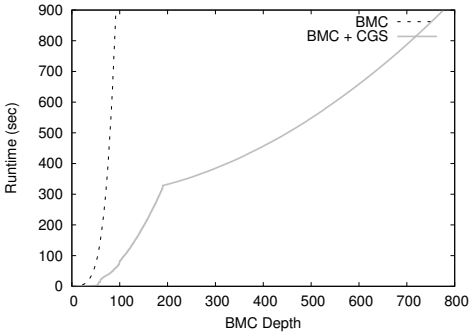

Fig. 7. BMC runtime comparison for intel_026: 3833 ANDs, 349 registers.

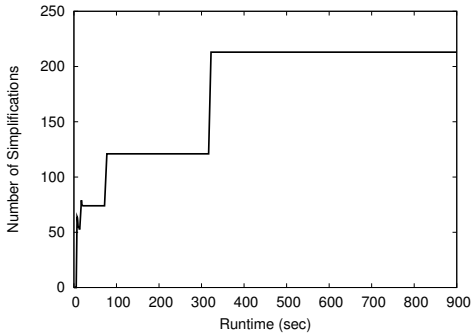

Fig. 8. Number of simplifications used in the "BMC+CGS" run on intel_026.

Table 1 shows the performance of BMC and CGS. BMC was run for 900 seconds on each of these designs. Column "Normal" gives the number of timeframes that BMC could be checked using the existing implementation in *SixthSense*, while Column "CGS" gives the number when CGS was used. On average, CGS enables BMC to proceed 10.71 times deeper into the design.

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 all. Column "Simp. Consts. " gives the average number of simplifying constants that were injected into each problem. The implementation injects 10 additional constants into a design when CGS is invoked, a number that was found to be large enough for the simplified problem to be easily solved yet small enough to prevent an abundance of spurious counterexamples and proofs. 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 "Total iters" 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 average runtime within one CGS. On average, 81% of the time was spent in the SAT solver in 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 to the original problem. This runtime breakdown is sensitive to the time bound on the SAT solver. With a large time limit, CGS is called less often, but the resultant resolution proofs will be larger and more difficult to lift; a shorter timeout allows the lifter runtimes to be manageable.

The time limit before CGS is invoked is a tuning parameter that affects the runtime breakdown but has less impact on the total runtime. Varying this limit 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 on the "intel_026" benchmark in more detail. 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 enhanced with CGS. These two BMC versions have similar performance until CGS begins to kick in, after which BMC 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 problem gets harder and more constants are injected, and the number decreases when spurious behaviour is detected and refinement is called.

Figures 7 - 8 illustrate an important trend that is responsible for the speedups offered by CGS. After some time, the current set of simplifications are able to provide a simplified problem that is easy to solve for subsequent timeframes, and the simplifications lead to a simplified resolution proof that is easy to lift. **This says that the speedup by CGS is mostly due to running a set of SAT problems that are related and taking advantage of this fact, rather than it speeding up individual SAT problems alone. We may have to modify our claims for CGS then.** 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. 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

We presented a method to enhance the solving of difficult SAT problems, creating a CGS-based SAT solver. It works by using conflict clauses, generated during SAT solving up to a given time limit, to simplify a problem. The 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 uses partial information, gathered in an attempt to solve a difficult problem, to simplify the problem.

An application to bounded model checking was explored. A Conflict-Guided Simplification framework was used to assist in the SAT solving of difficult BMC instances. CGS enabled BMC to check an order of magnitude more time frames, on average, in the same amount of runtime.

CGS is a "resource-aware" approach to SAT solving that can enhance any SAT solver to be more efficient on difficult problems. Future work will include exploring the application of CGS-based SAT solving to 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

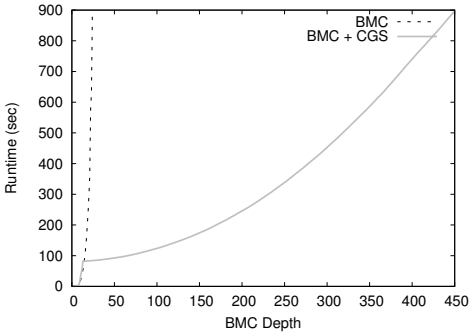

Fig. 9. BMC runtime comparison for intel_007: 10084 ANDs, 607 registers.

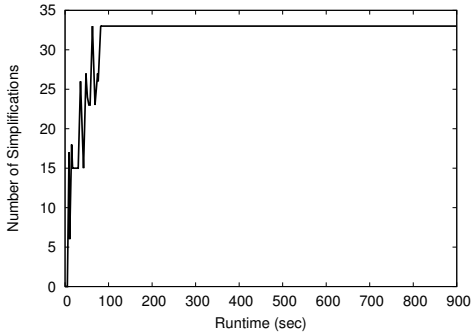

Fig. 10. Number of simplifications used in the "BMC+CGS" run on intel_007.

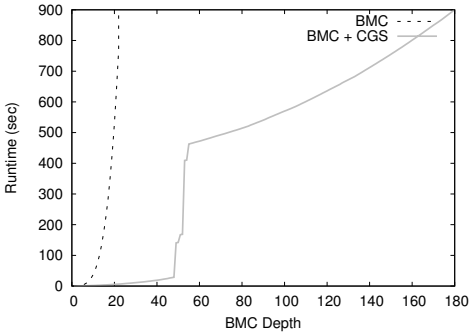

Fig. 11. BMC runtime comparison for intel_014: 42289 ANDs, 2372 registers.

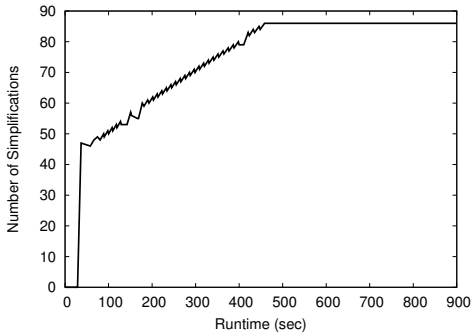

Fig. 12. Number of simplifications used in the "BMC+CGS" run on intel_014.

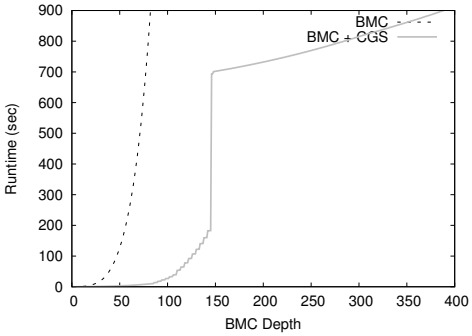

Fig. 13. BMC runtime comparison for intel_017: 4183 ANDs, 401 registers.

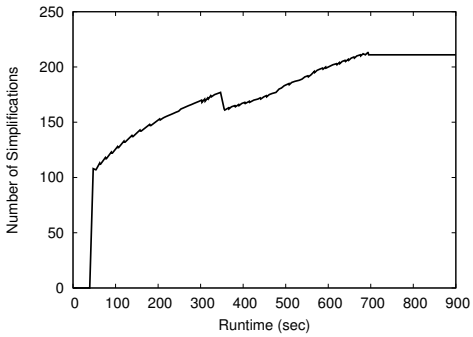

Fig. 14. Number of simplifications used in the "BMC+CGS" run on intel_017.