Abstraction Refinement for Bounded Model Checking

Anubhav Gupta[1] and Ofer Strichman[2]

[1] School of Computer Science, Carnegie Mellon University, USA
anubhav@cs.cmu.edu
[2] Information Systems Engineering, Technion, Israel
ofers@ie.technion.ac.il

Abstract. Counterexample-Guided Abstraction Refinement (CEGAR) techniques have been very successful in model checking large systems. While most previous work has focused on model checking, this paper presents a Counterexample-Guided abstraction refinement technique for Bounded Model Checking (BMC). Our technique makes BMC much faster, as indicated by our experiments. BMC is also used for generating refinements in the Proof-Based Refinement (PBR) framework. We show that our technique unifies PBR and CEGAR into an abstraction-refinement framework that can balance the model checking and refinement efforts.

1 Introduction

One of the most successful techniques for combating the infamous state-explosion problem in model checking is abstraction combined with automatic refinement. Starting from Kurshan's *localization reduction* [Kur94] in the 80's and later on a long list of symbolic techniques based at least partially on BDDs [CGKS02] [BGA02] [CCK+02] [GKMH+03], this framework proves to be highly efficient in solving model checking problems, many of which simply cannot be solved without it. The reason abstraction works, is that sometimes the checked property can be proved or refuted with only partial information about the model. Algorithms following the abstraction-refinement framework try to identify small subsets of the original model that on the one hand contain enough information to get the correct answer, and on the other hand are small enough to be handled by a model checker.

Abstraction techniques are mostly *conservative*: they preserve all the behaviors of the original model, but may introduce additional behaviors in the abstract model. This means that the model checker can produce *spurious counterexamples*, i.e. traces that are only possible in the abstract model, and not in the original model. In such cases we need to *refine* the abstraction in order to remove the spurious behavior. If the refinement is done based on an analysis of the spurious counterexample, which is the typical case, it is known as *Counterexample-Guided Abstraction Refinement* (CEGAR).

In this paper, we present a *Counterexample-Guided* abstraction-refinement technique for *Bounded Model Checking*, called CG-BMC. SAT-based Bounded

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 112–124, 2005.

Model Checking (BMC) [BCCZ99] has gained wide acceptance in industry in the last few years, as a technique for refuting properties with shallow counterexamples, if they exist. Given a model M, a property φ and a positive integer k representing the depth of the search, a Bounded Model Checker generates a propositional formula that is satisfiable if and only if there is a counterexample of length k or less to φ, in M. BMC iteratively deepens the search for counterexamples until either a bug is found or the problem becomes too hard to solve in a given time limit. The motivation for making this technique more powerful, i.e. to enable it to go deeper in a given time limit, is clear.

The guiding principle behind CG-BMC is similar to that of any good CEGAR technique: attempt to eliminate spurious behavior in few iterations, while keeping the abstract model small enough to be solved easily. CG-BMC therefore focuses on eliminating those spurious transitions that may lead to an error state.

Our abstraction makes parts of the model non-deterministic as in [Kur94], i.e. the defining logic of some variables (those that are known in literature as *invisible variables*) is replaced with a nondeterministic value. This abstraction works well with a SAT-solver since it corresponds to choosing a set of 'important' clauses as the abstract model. Our initial abstraction is simply the empty set of clauses, and in each refinement step we add clauses from the original, concrete model.

We start the CG-BMC loop with search depth $k = 1$. In each iteration of the loop, we first try to find a counterexample of length k in the current abstract model, with a standard BMC formulation and a SAT-solver. If the abstract model has no counterexamples of length k, the property holds at depth k and we move to depth $k+1$. Otherwise, if a counterexample is detected in the abstract model, we check the validity of the counterexample on the original model. More specifically, we formulate a standard BMC instance with length k, and restrict the values of the visible variables (those that participate in the abstract model) to their values in the counterexample. If the counterexample is real, we report a bug and exit. If the counterexample is spurious, the abstract model is refined by adding to it gates that participated in the proof of unsatisfiability. This refinement strategy has been previously used by Chauhan et al.[CGKS02] in the context of model checking. We chose this technique because for our purposes, it provides a good balance between the effort of computing the refinement and the quality of the refinement, i.e. how fast it leads to convergence and how hard it makes the abstract model to solve.

An alternative to our two-stage heuristic, corresponding to checking the abstract and concrete models, is to try to emulate this process within a SAT-solver by controlling the decision heuristic (focusing first on the parts of the model corresponding to the abstract model). Wang et al. [WJHS04] went in this direction: they use the unsatisfiable cores from previous cycles to guide the search of the SAT-solver when searching for a bug in the current cycle. Guidance is done by changing the variable selection heuristic to first decide on the variables that participated in the previous unsatisfiable cores. Furthermore, McMillan observed in [McM03] that modern SAT-solvers internally behave like abstraction-refinement

engines by themselves: their variable selection heuristics move variables involved in recent conflicts up in the decision order. However, a major drawback of this approach is that while operating on a BMC instance, they propagate values to variables that are not part of the abstract model that we wish to concentrate on, and this can lead to long phases in which the SAT-solver attempts to solve *local conflicts* that are irrelevant to proving the property. In other words, this approach allows irrelevant clauses to be pulled into the proof through propagation and cause conflicts. This is also the drawback of [WJHS04], as we will prove by experiments. Our approach solves this problem by forcing the SAT-solver to first find a complete abstract trace before attempting to refute it. We achieve this by isolating the important clauses from the rest of the formula in a separate SAT instance.

Another relevant work is by Gupta et al. [GGYA03], that presents a top-down abstraction framework where proof analysis is applied iteratively to generate successively smaller abstract models. Their work is related because they also suggest using the abstract models to perform deeper searches with BMC. However, their overall approach is very different from ours. They focus on the top-down iterative abstraction, and refinement is used only when they cannot go deeper with BMC.

We take the CG-BMC approach one step further in Section 5.2, by considering a new abstraction-refinement framework, in which CG-BMC unifies CEGAR and Proof-Based Refinement (PBR) [MA03, GGYA03]. PBR eliminates all counterexamples of a given length in a single refinement step. The PBR refinement uses the unsatisfiable core of the (unrestricted) BMC instance to generate a refinement. Amla et al. showed in [AM04] that CEGAR and PBR are two extreme approaches: CEGAR burdens the model checker by increasing the number of refinement iterations while PBR burdens the refinement step because the BMC unfolding without the counterexample constraints is harder to refute. They also present a hybrid approach that tries to balance between the two, which results in a more robust overall behavior. We show that by replacing BMC with a more efficient CG-BMC as the refinement engine inside PBR, we also get a hybrid abstraction-refinement framework that can balance the model checking and refinement efforts.

In the next section we briefly describe the relevant issues in BMC, unsatisfiable cores produced by SAT-solvers, and the CEGAR loop. In Section 3 we describe our CG-BMC algorithm, which applies CEGAR to BMC. We also describe a variant of this algorithm, called CG-BMC-T, which uses timeouts in one stage of the algorithm to avoid cases in which solving the abstract model becomes too hard. In Section 4 we describe our experiments with these two techniques and compare them to both standard BMC and [WJHS04]. Our implementation of CG-BMC on top of zChaff [MMZ$^+$01] achieved significant speed-ups comparing to the other two techniques. In Section 5, we describe how this approach can very naturally be integrated in a hybrid approach, which benefits from the advantages of both CEGAR and PBR. We conclude in Section 6 by giving some directions for future work.

2 Preliminaries

2.1 Bounded Model Checking

SAT-based Bounded Model Checking [BCCZ99] is a rather powerful technique for refuting properties. Given a model M, a property φ and a positive integer k representing the depth of the search, a Bounded Model Checker generates a propositional formula that is satisfiable if and only if there is a counterexample of length k or less to φ, in M. In this case we write $M \not\models_k \varphi$. The idea is to iteratively deepen the search for counterexamples until either a bug is found or the problem becomes too hard to solve in a given time limit. The extreme efficiency of modern SAT-solvers make it possible to check properties typically up to a depth of a few hundred cycles. More importantly, there is a very weak correlation, if any, between what is hard for standard BDD-based model checking, and BMC. It is many times possible to refute properties with the latter that cannot be handled at all by the former. It should be clear, then, that every attempt to make this technique work faster, and hence enable to check larger circuits and in deeper cycles, is worth while.

2.2 Unsatisfiable Cores Generated by SAT-Solvers

While a satisfying assignment is a checkable proof that a given propositional formula is satisfiable, until recently SAT-solvers produced no equivalent evidence when the formula is unsatisfiable. The notion of generating resolution proofs from a SAT-solver was introduced in [MA03]. From this resolution proof, one may also extract the *unsatisfiable core*, which is the set of clauses from the original CNF formula that participate in the proof. Topologically, these are the roots of the resolution graph. The importance of the unsatisfiable core is that it represents a subset, hopefully a small one, of the original set of clauses that is unsatisfiable by itself. This information can be valuable in an abstraction-refinement process as well as in other techniques, because it can point to the reasons for unsatisfiability. In the case of abstraction-refinement, it can guide the refinement process, since it points to the reasons for why a given spurious counterexample cannot be satisfied together with the concrete model.

2.3 Counterexample-Guided Abstraction-Refinement

Given a model M and an ACTL property φ, the abstraction-refinement framework encapsulates various automatic algorithms for finding an abstract model $\hat{M}$ with the following two properties:

- $\hat{M}$ over-approximates M, and therefore $\hat{M} \models \varphi \rightarrow M \models \varphi$;
- $\hat{M}$ is smaller than M, so checking whether $\hat{M} \models \varphi$ can be done more efficiently than checking the original model M.

This framework is an important tool for tackling the state-explosion problem in model checking. Algorithm 1 describes a particular implementation of the Counterexample-Guided Abstraction-Refinement (CEGAR) loop. We denote by

$BMC(M, \varphi, k)$ the process of generating the length k BMC unfolding for model M and solving it, according to the standard BMC framework as explained in Section 2.1. The loop simulates the counterexample on the concrete model using a SAT-solver (line 4), and uses the unsatisfiable core produced by the SAT-solver to refine the abstract model (lines 5,6). This refinement strategy was proposed by Chauhan et al.[CGKS02].

Algorithm 1 Counterexample-Guided Abstraction-Refinement

CEGAR (M, φ)
1: $\hat{M} = \{\}$;
2: **if** $MC(\hat{M}, \varphi) = TRUE$ **then return** 'TRUE';
3: **else** let C be the length k counterexample produced by the model checker;
4: **if** $BMC(M, k, \varphi) \wedge C = SAT$ **then return** 'bug found in cycle k';
5: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
6: $\hat{M} = \hat{M} \cup U$;
7: **goto** line 2;

3 Abstraction-Refinement for Bounded Model Checking

The underlying principles behind the CEGAR framework are the following:

- The information that was used to eliminate previous counterexamples, which is captured by the abstract model, is relevant for proving the property.
- If the abstract model does not prove the property, then the counterexamples in the abstract model can guide the search for a refinement.

We apply these principles to guide the SAT-solver, thereby making BMC faster.

3.1 The CG-BMC Algorithm

The pseudo-code of our Counterexample-Guided Bounded Model Checking algorithm is shown in Algorithm 2. We start with an empty initial abstraction and an initial search depth $k = 1$. In each iteration of the CG-BMC loop, we first try to find a counterexample in the abstract model (line 3). If there is no counterexample in the abstract model, the property holds at cycle k and the abstract model now contains the gates in the unsatisfiable core generated by the SAT-solver (lines 4,5). Otherwise, if a counterexample is found, we simulate the counterexample on the concrete model (line 8). If the counterexample can be concretized, we report a real bug. If the counterexample is spurious, the abstract model is refined by adding the gates in the unsatisfiable core (line 10). Like standard BMC, CG-BMC either finds an error or continues until it becomes too complex to solve within a given time limit.

3.2 Inside a SAT-Solver

Most modern SAT-solvers are based on the DPLL search procedure [DP60]. The search for a satisfying assignment in the DPLL framework is organized as a binary search tree in which at each level a *decision* is made on the variable to split

Algorithm 2 Counterexample-Guided Bounded Model Checking

CG-BMC (M, φ)
1: $k = 0$; $\hat{M} = \{\}$;
2: $k = k + 1$;
3: **if** $BMC(\hat{M}, k, \varphi) = UNSAT$ **then**
4: Let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
5: $\hat{M} = U$;
6: **goto** line 2;
7: **else** let C be the satisfying assignment produced by the SAT-solver;
8: **if** $BMC(M, k, \varphi) \wedge C = SAT$ **then return** 'bug found in cycle k';
9: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
10: $\hat{M} = \hat{M} \cup U$;
11: **goto** line 3;

on, and the first branch to be explored (each of the two branches corresponds to a different Boolean assignment to the chosen variable). After each decision, Boolean Constrain Propagation (BCP) is invoked, a process that finds the implications of the last decision by iteratively applying the *unit-clause rule* (the unit-clause rule simply says that if in an l-length clause $l-1$ literals are unsatisfied, then the last literal must be satisfied in order to satisfy the formula). Most of the computation time inside a SAT-solver is spent on BCP. If BCP leads to a conflict (an empty clause), the SAT-solver backtracks and changes some previous decision.

The performance of a SAT-solver is determined by the choice of the decision variables. Typically SAT-solvers compute a score function for each undecided variable that prioritizes the decision options. Many branching heuristics have been proposed in the literature: see, for example, [Sil99]. The basic idea behind many of these heuristics is to increase the score of variables that are involved in conflicts, thereby moving them up in the decision order. This can be viewed as a form of refinement [McM03].

An obvious question that comes to mind is why do we need an abstraction-refinement framework for BMC when a SAT-solver internally behaves like an abstraction-refinement engine. A major drawback of the branching heuristics in a SAT-solver is that they have no global perspective of the structure of the problem. While operating on a BMC instance, they tend to get 'distracted' by local conflicts that are not relevant to the property at hand. CG-BMC avoids this problem by forcing the SAT-solver to find a satisfying assignment to the abstract model, which contains only the relevant part of the concrete model. It involves the other variables and gates only if it is not able to prove unsatisfiability with the current abstract model.

The method suggested by Wang et al. [WJHS04] that we mentioned in the introduction, tries to achieve a similar effect by modifying the branching heuristics. They perform BMC on the concrete model, while changing the score function of the SAT-solver so it gives higher priority to the variables in the abstract model (they do not explicitly refer to an abstract model, rather to the unsatisfiable core of the previous iteration, which is what we refer to as the abstract model). Since

their SAT-solver operates on a much larger concrete model, it spends a lot more time doing BCP. Moreover, many of the variables in the abstract model are also present in clauses that are not part of the abstract model and their method often encounters conflicts on these clauses. CG-BMC, on the other hand, *isolates* the abstract model and solves it separately, in order to avoid this problem.

3.3 The CG-BMC-T Algorithm

The following is an implicit assumption in the CG-BMC algorithm: Given two unsatisfiable sets of clauses C_1 and C_2 such that $C_1 \subset C_2$, solving C_1 is faster than solving C_2. While this is a reasonable assumption and mostly holds in practice, it is not always true. It is possible that the set of clauses C_2 is *over-constrained*, so that the SAT-solver can prove its unsatisfiability with a small search tree. Removing clauses from C_2, on the other hand, could produce a set of clauses C_1 that is *critically-constrained* and proving the unsatisfiability of C_1 could take much more time [CA93].

We observed this phenomenon in some of our benchmarks. As an example, consider circuit *PJ05* in Table 1 (see Section 4). The CG-BMC algorithm takes much longer than BMC to prove the property on *PJ05*. This is not because of the overhead of the refinement iterations: the abstract model has enough clauses to prove the property after an unfolding length of 9. The reason for this is that BMC on the small abstract model takes more time than BMC on the original model.

In order to deal with such situations, we propose a modified CG-BMC algorithm, called CG-BMC-T (T stands for Timeout). The intuition behind this algorithm is the following: if the SAT-solver is taking a long time on the abstract model, it is possible that it is stuck in a critically-constrained search region, and therefore we check if it can quickly prune away this search region by adding some constraints from the concrete model. The algorithm is described in Algorithm 3. In each iteration of the CG-BMC-T loop, we set a timeout T for the SAT-solver (line 4) and try to find a counterexample in the abstract model (line 5). If the SAT-solver completes, the loop proceeds like CG-BMC. However, if the SAT-solver times-out, we simulate the partial assignment on the concrete model with a smaller timeout ($T \times \beta,\ \beta < 1$) (lines 14,16). If the concrete model is able to concretize the partial assignment, we report a bug (line 17). If the concrete model refutes the partial assignment, we add the unsatisfiable core generated by the SAT-solver to the abstract model, thereby eliminating the partial assignment (line 21). If the concrete solver also times-out, we go back to the abstract solver. However, for this next iteration, we increase the timeout with a factor of α (line 14). The CG-BMC-T algorithm is more robust, as indicated by our experiments.

4 Experiments

We implemented our techniques on top of the SAT-solver zChaff [MMZ+01]. Some modifications were made to zChaff to produce unsatisfiable cores while adding and deleting clauses incrementally. Our experiments were conducted on

Algorithm 3 CG-BMC with Timeouts

```
CG-BMC-T (M, φ)
 1: k = 0; M̂ = {};
 2: k = k + 1;
 3: T = T_init;
 4: Set_Timeout(T);
 5: Res = BMC(M̂, k, φ);
 6: if Res = UNSAT then
 7:     Let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
 8:     M̂ = U;
 9:     goto line 2;
10: else
11:     if Res = SAT then
12:         Let C be the satisfying assignment produced by the SAT-solver;
13:     else
14:         T = T × α; Set_Timeout(T × β);
15:         Let C be the partial assignment produced by the SAT-solver;
16: Res = BMC(M, k, φ) ∧ C;
17: if Res = SAT then return 'bug found in cycle k';
18: else
19:     if Res = UNSAT then
20:         Let U be the set of gates in the unsat core produced by the SAT-solver;
21:         M̂ = M̂ ∪ U;
22: goto line 4;
```

a set of benchmarks that were derived during the formal verification of an open source Sun PicoJava II microprocessor [MA03]. All experiments were performed on a 1.5GHz Dual Athlon machine with 3Gb RAM. We set a timeout of 2 hours and a maximum BMC search depth of 60.

We use the incremental feature of zChaff to optimize the CG-BMC loop as follows. We maintain two incremental SAT-instances: *solver-Abs* contains the BMC unfolding of the abstract model while *solver-Conc* contains the BMC unfolding of the concrete model. The counterexample generated by *solver-Abs* is simulated on *solver-Conc* by adding unit clauses. The unsatisfiable core generated by *solver-Conc* is added to *solver-Abs*. Our algorithm can in principle be implemented inside a SAT-solver although this requires fundamental changes in the way it works, and it is not clear if it will actually perform better or worse. We discuss this option further in Section 6.

Table 1 compares our techniques with standard BMC. For each circuit, we report the depth that was completed by all techniques, the runtime in seconds, and the number of backtracks (for CG-BMC/CG-BMC-T we report the backtracks on both abstract and concrete models). We see a significant overall reduction in runtime. This reduction is due to a decrease in the total number of backtracks, and the fact that most of the backtracks (and BCP) are performed on a much smaller abstract model. We also observe a more robust behavior with CG-BMC-T ($T_{init} = 10\text{s}, \alpha = 1.5, \beta = 0.2$).

Table 2 compares our technique with the approach based on modifying the SAT-solver's branching heuristics, as described in Wang et al. [WJHS04]. We re-

Table 1. Comparison of CG-BMC/CG-BMC-T with standard BMC.

Circuit	Depth	Time(s)			Backtracks				
		BMC	CG-BMC	CG-BMC-T	BMC	CG-BMC		CG-BMC-T	
						Abs	Conc	Abs	Conc
PJ00	35	7020	48	48	139104	378	27	378	27
PJ01	60	273	99	99	169	187	9	187	9
PJ02	39	6817	51	51	79531	807	5	807	5
PJ03	39	6847	51	51	79531	807	5	807	5
PJ04	60	125	99	98	169	184	7	184	7
PJ05	25	751	2812	296	12476	582069	59	64846	445
PJ06	33	2287	2421	364	23110	346150	92	82734	137
PJ07	60	1837	789	449	34064	197843	86	111775	132
PJ08	60	5061	201	201	43564	44468	124	44468	124
PJ09	60	1092	110	110	22858	32453	57	32453	57
PJ10	50	6696	47	46	76153	3285	67	3285	67
PJ11	33	6142	69	70	120158	1484	95	1484	95
PJ12	24	5266	28	28	117420	2029	91	2029	91
PJ13	60	327	103	102	1005	4019	4	4019	4
PJ14	60	5086	295	316	103217	64392	84	62944	93
PJ15	34	6461	117	115	86567	16105	111	16105	111
PJ16	56	4303	172	173	37843	30528	56	30528	56
PJ17	20	7039	815	1153	81326	68728	548	72202	2530
PJ18	43	7197	719	992	170988	102186	1615	126155	1904
PJ19	9	5105	2224	2555	544941	522702	2534	522460	34324

port results for both *static* (Ord-Sta) and *dynamic* (Ord-Dyn) ordering methods. The *static* ordering method gives preference to variables in the abstract model throughout the SAT-solving process. The *dynamic* ordering method switches to the SAT-solver's default heuristic after a threshold number of decisions. Our approach performs better than these methods.

5 A Hybrid Approach to Refinement

5.1 Proof-Based Refinement and a Hybrid Approach

We described the CEGAR loop in Section 2.3. An alternative approach, called Proof-Based Refinement (PBR), was proposed by McMillan et al. [MA03] (and independently by Gupta et al.[GGYA03]). The pseudo-code for this approach is shown in Algorithm 4. In each refinement iteration, PBR performs BMC on the concrete model (line 4) and uses the unsatisfiable core as the abstract model for the next iteration (line 6). As opposed to CEGAR that eliminates one counterexample, PBR eliminates all counterexamples of a given length in a single refinement step.

Amla et. al. [AM04] performed an industrial evaluation of the two approaches and concluded that PBR and CEGAR are extreme approaches. PBR has a more ex-

Table 2. Comparison of CG-BMC/CG-BMC-T with Wang et al. [WJHS04]

Circuit	Depth	Time(s)			
		Ord-Sta	Ord-Dyn	CG-BMC	CG-BMC-T
PJ00	60	961	942	104	104
PJ01	60	729	711	99	99
PJ02	60	694	678	101	101
PJ03	60	693	679	101	100
PJ04	60	656	641	99	98
PJ05	25	219	205	2812	296
PJ06	20	4786	1192	148	154
PJ07	25	4761	124	40	41
PJ08	60	703	712	201	201
PJ09	60	494	483	110	110
PJ10	54	827	6493	54	69
PJ11	60	816	796	135	111
PJ12	60	1229	973	101	101
PJ13	60	673	657	103	102
PJ14	60	3101	2746	295	316
PJ15	60	3488	3456	296	371
PJ16	60	3022	3021	198	199
PJ17	21	3132	6114	1069	1570
PJ18	38	6850	4846	556	721
PJ19	5	5623	176	113	116

Algorithm 4 Proof-Based Refinement

PBR(M, φ)
1: $\hat{M} = \{\}$;
2: **if** $MC(\hat{M}, \varphi) = TRUE$ **then return** 'TRUE';
3: **else** let k be the length of the counterexample produced by the model checker;
4: **if** $BMC(M, k, \varphi) = SAT$ **then return** 'bug found in cycle k';
5: **else** let U be the set of gates in the unsatisfiable core produced by the SAT-solver.
6: $\hat{M} = U$;
7: **goto** line 2;

pensive refinement step than CEGAR, since PBR performs unrestricted BMC while CEGAR restricts BMC to the counterexample produced by the model checker. CEGAR, on the other hand, has a larger number of refinement iterations since it only eliminates one counterexample per refinement iteration, thereby putting more burden on the model checker. To balance the two, they propose a hybrid of the two approaches.

Their hybrid approach also performs BMC on the concrete model after given a counterexample from the abstract model. However, they use the counterexample only to provide the initial decisions to the SAT-solver. They also set a time limit to the SAT-solver. If the SAT-solver completes before the time-out with an Unsat answer, the hybrid approach behaves like PBR. On the other hand if the

SAT-solver times-out, they rerun a BMC instance conjoined with constraints on some of the variables in the counterexample (but not all). From this instance they extract an unsatisfiable core, thereby refuting a much larger space of counterexamples (note that this instance has to be unsatisfiable if enough time was given to the first instance due to the initial decisions). Their experiments show that the hybrid approach is more robust than PBR and CEGAR.

5.2 A Hybrid Approach Based on CG-BMC

Since the CG-BMC algorithm outperforms BMC, it can be used as a replacement for BMC in the refinement step of PBR. For example, in our experiments on *PJ17* (see Section 4), CG-BMC proved the property up to a depth of 29, and model checking could prove the correctness of the property on the generated abstract model. BMC could only finish upto depth of 20, and the resulting abstract model had a spurious counterexample of length 21.

CG-BMC is also a better choice than BMC because it provides an elegant way of balancing the effort between model checking and refinement. Algorithm 5 shows the pseudo code of our HYBRID algorithm, that we obtained after replacing BMC with CG-BMC inside the refinement step of PBR, and adding a choice function (line 2). At each iteration, HYBRID chooses either the model checker (line 3) or a SAT-solver (line 8) to find a counterexample to the current abstract model. The model checker returns 'TRUE' if the property holds for all cycles (line 3). If the SAT-solver returns UNSAT, the property holds at the current depth and the unsatisfiable core is used to refine the current abstraction. If a counterexample is produced by either the model checker or the SAT-solver, it is simulated on the concrete model (line 13). If the counterexample is spurious, the unsatisfiable core generated by the SAT-solver is added to the abstract model (line 15).

At a first glance, the HYBRID algorithm looks like a CEGAR loop, with the additional option of using a SAT-solver instead of a model checker for verifying the abstract model. However, the HYBRID algorithm captures both the CEGAR and the PBR approaches. If the choice function always chooses the model checker (line 3), it corresponds to the standard CEGAR algorithm. Now consider a strategy that chooses the model checker every time there is an increase in k at line 10, but chooses the SAT-solver (line 8) in all other cases. This is exactly the PBR loop that uses CG-BMC instead of BMC. Other choice functions correspond to a hybrid approach. There can be many strategies to make this choice, some of which are: 1) Use previous run-time statistics to decide which engine is likely to perform better on the next model, and occasionally switch to give the other engine a chance; 2) Measure the *stability* of the abstract model, i.e., whether in the last few iterations (increases of k) there was a need for refinement. Only if not - send it to a model checker; and, 3) Run the two engines in parallel.

6 Conclusions and Future Work

We presented CG-BMC, a Counterexample-Guided Abstraction Refinement algorithm for BMC. Our approach makes BMC faster, as indicated by our experiments.

Algorithm 5 Hybrid of CEGAR and PBR

```
HYBRID (M, φ)
 1: k = 1; M̂ = {};
 2: goto line 3 OR goto line 8;
 3: if MC(M̂, φ) = TRUE then return 'TRUE';
 4: else
 5:     Let C be the length r counterexample produced by the model checker;
 6:     k = r;
 7:     goto line 13;
 8: if BMC(M̂, k, φ) = UNSAT then
 9:     Let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
10:     M̂ = U; k = k + 1;
11:     goto line 2;
12: else let C be the satisfying assignment produced by the SAT-solver;
13: if BMC(M, k, φ) ∧ C = SAT then return 'bug found in cycle k';
14: else let U be the set of gates in the unsatisfiable core produced by the SAT-solver;
15: M̂ = M̂ ∪ U;
16: goto line 2;
```

There are many directions in which this research can go further. First, the HYBRID algorithm should be evaluated empirically, and appropriate choice functions should be devised. The three options we listed in the previous section are probably still naive. Based on the experiments of Amla et al. reported in [AM04] in hybrid approaches, it seems that this can provide a better balance between the efforts spent in model checking and refinement, and also enjoy the benefit of CG-BMC. Second, it is interesting to check whether implementing CG-BMC inside a SAT-solver can make it work faster. This requires significant changes in various fundamental routines in the SAT-solver. In particular, this requires some mechanism for clustering the clauses inside the SAT-solver into abstraction levels, and postponing BCP on the clauses in the lower levels until all the higher levels are satisfied. Refinement would correspond to moving clauses up (and down) across levels, possibly based on their involvements in conflicts. We are currently working on this implementation. A third direction for future research is to explore the application of CG-BMC to other theories and decision procedures, like bit-vector arithmetic.

Acknowledgements. The first author would like to thank Ken McMillan for useful discussions.

References

[AM04] N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004*, pages 260–274, 2004.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS, 1999.

[BGA02] S. Barner, D. Geist, and A.Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, July 2002.

[CA93] J. M. Crawford and L. D. Anton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27. AAAI Press, 1993.

[CCK+02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS, 2002.

[CGKS02] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, LNCS, pages 265–279. Springer-Verlag, 2002.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[GGYA03] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based bmc with proof analysis. In *ICCAD*, pages 416–423, 2003.

[GKMH+03] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003,*, LNCS, pages 176–191, 2003.

[Kur94] R. Kurshan. *Computer aided verification of coordinating processes.* Princeton University Press, 1994.

[MA03] K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *9th Intl. Conf. on Tools And Algorithms For The Construction And Analysis Of Systems (TACAS'03)*, volume 2619 of *LNCS*, 2003.

[McM03] Ken McMillan. From bounded to unbounded model checking, 2003.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference 2001 (DAC'01)*, 2001.

[Sil99] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

[WJHS04] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *ACM/IEEE 41th Design Automation Conference (DAC'04)*, pages 535–538, 2004.