Using Counter Example Guided Abstraction Refinement to Find Complex Bugs

Per Bjesse, James Kukula
Synopsys Inc.
{bjesse,kukula}@synopsys.com

ABSTRACT

In this paper, we present a method for finding failure traces for safety properties that are out of reach for traditional approaches to counter example generation. We do this by guiding Bounded Model Checking (BMC) with information gathered from counter example guided abstraction refinement. Unlike previously described approaches based on reconstructing abstract counter examples on the concrete machines, we do not limit ourselves to search for failures of the same length as the current abstract counterexample. We also describe a combination of previously known methods for choosing registers to include in the abstraction that we have found works very well together with our technique for finding failures. Our experimental results show that the resulting method can find counter examples that are out of range for both standard BMC and two previously published approaches to abstraction-guided BMC.

1. INTRODUCTION

This paper presents a method for detecting very hard to find safety property failures on large systems. Our aim is to leverage information generated during counter example guided abstraction refinement [10, 4] in Bounded Model Checking (BMC) [1]. Our key idea is to generate *stepping stones* from the abstract system—sets of states that we can guarantee any counterexample to a property will need to traverse in a particular order. When the capacity limits of our search engine makes it too hard to find a path directly from initial states to the failing states, we use the stepping stones to divide the search into a number of shorter searches. Unlike previous approaches, we can use a given abstraction to find counterexamples that are many times longer than the current failure length on the abstraction. We get this feature at the price of not being able to guarantee minimal failures.

The core idea for finding counterexamples that we present can be used together with any of the abstraction refinement schemes that have been presented in the literature. However, we have found that the particular abstractions that are used to guide the search for counterexamples are very important, and can mean the difference between whether a given failure can be found or not. We therefore also present a blend of different techniques for choosing how to extend the current abstraction that we have found works quite well with our method.

Our focus in this paper is on finding bugs that are out of range of any of the battery of techniques that are currently at our disposal. However, as a side effect of the work we present, we have arrived at a system for counterexample guided abstraction refinement that is complete in the sense that it can outperform exact approaches to property verification both for proving systems correct, and for finding complex bugs.

We demonstrate the bug finding capabilities of our system by presenting experimental results from using our tool to find very complex failures of real properties on five industrial circuits. We compare our approach to pure bounded model checking, and to two previous approaches that use abstraction data to guide the failure search [12, 8]. Our results show that our tool is significantly more robust than the other methods, and that it can be used to find failures that previously has been out of range for any methods known to us.

2. RELATED WORK

The core search technology that we are using builds on the work on bounded model checking [1]. BMC can be a very efficient method for generating counterexamples, but its capacity is limited. As a consequence, a number of attempts to guide BMC using abstractions of the system under analysis have recently appeared in the literature.

In [2], Cabodi and coworkers present an idea for guiding BMC using approximate BDD-frontiers. Here, the system under analysis is the concrete system—the approximation comes from simplifying BDDs during the image and preimage operations. The resulting frontiers are then used as constraints for the BMC engine during the search. As a result of the constrained search space, up to one order of magnitude speedups are demonstrated on four ISCAS circuits with automatically generated properties. A significant difference between our and Cabodi and coworkers's method is that our abstraction comes from cutting out a larger and larger part of the design from its environment. In contrast, Cabodi and coworkers dynamically approximate BDDs. We (as do the authors of [12]) believe that structural abstraction provides a more predictable method for controlling the tradeoff between size and expressivity of abstractions.

The DiVer framework introduced in [8] guides BMC using a similar approach to ours. As opposed to the work of Cabodi and coworkers, the abstraction used is structural pruning on the concrete system—images and preimages are done exactly. Each BMC check searches for counterexamples of the same length as the current abstract counterexample, and restrict the search to only traverse states in the abstract forward and backward shells.

The most closely related method to ours is the counterexample generation used in RFN [12]. Just as in our case, and in the case of DiVer, this approach is based on abstraction refinement using structural pruning. Given an abstract failure, ATPG-based BMC is used to attempt to transfer the trace to a concrete failure by searching for a concrete state

1530-1591/04 $20.00 (c) 2004 IEEE

sequence of the same length that visits states corresponding to the abstract states in order.

A key difference between the approaches used in the DIVER and RFN frameworks and our proposed method is that the previous methods only look for counterexamples that have the same length as the current counterexample length on the abstract system. In contrast, we can generate failures that are many times longer using the same abstractions. We will show that this makes our approach very competitive on industrial circuits.

There are other techniques that attempts to find long failures for realistic systems. Many of these techniques are based on augmenting simulation with formal searches, as done in the KETCHUM and SIVA [9, 5] systems. As we will discuss in Section 4, simulation plays a part in our approach, but it is minor and could be omitted. In a spectrum from completely formal to classically semi-formal our method and these approaches thus represent points at different ends of the spectrum. However, it is interesting to note that our analysis can be cast as a particular automatic way of generating *lighthouses* in the sense of [5], which we attempt to link using formal searches rather than augmented simulation.

3. ABSTRACTION REFINEMENT

We will now introduce the traditional framework for abstraction refinement based on counter example guidance. We follow the presentation in [12].

Let us define a *structural abstraction* A of a given circuit C as a circuit that can be derived from C by (1) removing a number of registers, (2) substituting free inputs for the removed register outputs, and (3) removing A's unconnected logic. The observation that structural abstraction refinement makes use of is that if we can prove a safety property for A, then it necessarily holds for C. The reason for this is that since we have proved the safety property for A with free inputs substituted for severed connections, the property holds in any environment of A—including A's environment in C.

Structural counter example guided abstraction refinement proceeds as follows:

1. Start with an structural abstraction A containing a small number of registers.

2. Attempt to prove the safety property for A. If we succeed, then the concrete system is correct, and we are done.

3. If we fail, check whether the current counterexample on the abstraction can be transferred to a counter example on the real system. If we can transfer the counter example to the concrete system, then we are done.

4. Otherwise we analyze the failure behavior of the structural abstraction, and attempt to guess good registers to add to the next abstraction. Goto 2.

In every iteration in the abstraction refinement algorithm, a proof of the property is attempted on the current abstraction. Note that in this step of the process, we are free to use any techniques that we want to prove the system correct. In this paper, we will be using BDD-based fixpoint calculations.

If the attempted proof fails, then we check whether it is possible to use the abstraction information to generate a failure on the concrete system. We present the new technique that we use to construct the concrete counterexamples in Section 4. Unlike previous approaches to generating failures in abstraction refinement, this technique does not attempt to transfer an abstract counterexample to the concrete machine. Instead it analyses the state space of the abstract machine, and uses it to generate a concrete trace directly.

Once we know that the current abstraction is insufficient both for proving the system correct and as an aid for generating a real counterexample, we compute an augmentation of the model that we believe will be a good refinement. In counterexample guided abstraction refinement, the idea is to try use the information in the failure on the abstract system to guide which registers in the design are added next. The strength of the resulting method will largely depend on the robustness of this heuristic. We present our choice for an analysis in Section 5.

4. COUNTEREXAMPLE GENERATION

We will now describe our approach to abstraction guided counterexample generation by (1) introducing the notion of backshell lighthouses, by (2) presenting the repeat extender algorithm that attempt to navigate from a given state towards states that come closer to the goal, and by (3) presenting our main algorithm for using a given abstraction to find a concrete counterexample.

4.1 Backshell Lighthouses

Given a current abstraction that is insufficient to prove the property at hand correct, we compute a number of *backshell lighthouses* as follows.

We first compute the set of reachable states on the abstract model, and then generate sufficiently many reverse images (backshells) that every state that can reach a goal state on the abstract system is in some backshell. We designate backshell number zero to be the set of goal states and backshell number N to be the most distant reverse image. Next, we intersect each backshell with the reachable states of the abstraction to form backshell lighthouses $B_0 \dots B_N$. The concrete states in backshell lighthouse B_{i+1} are thus potentially reachable states that can reach one or more states in backshell lighthouse i in one abstract transition. These shells will be our stepping stones.

Given a state and a set of backshell lighthouses, we define the state *depth* to be the index of the lowest numbered backshell it is in. If a state is in no backshell, we assign it depth ∞. For the remainder of this presentation we will assume that our system has a single initial state. We define the *start depth* of a system with respect to an abstraction as the depth of the initial state. Note that the start depth will be finite regardless of whether the goal is reachable on the concrete system or not since N is larger than the current abstract failure length. Also note that all goal states by construction have depth 0.

It is easy to see that the depth of a given state is an indicator of its minimum distance to the goal states on the concrete machine: If a state has depth k then the shortest trace (if one exists) on the concrete machine that can reach the goal has length lower bounded by k. However, it is important to realize that due to the over-approximative nature of the backshells, the depth of a state may be an inaccurate

measure for how far it is from the goal states in reality—a given state with depth d may be a *deadend* state in the sense that (1) there may not be any sequences at all that lead to states with lower depth, or (2) the necessary sequence length may exceed our search capacity on the concrete machine. A heuristic for searching for counterexamples guided by backshells must therefore take steps to not get locked into states from which no progression can be made.

We designate states that get assigned finite depth to be states with *measurable* depth. A nonmeasurable state is thus a state that provably can not reach the goal states on the concrete system.

The state depths in a given trace segment on the concrete system can only vary in a very simple way: If state n has measurable depth $k+1$, then if state $n+1$ is measurable, its depth is lower bounded by k. In other words, a legal transition in the concrete system that starts in a measurable state may either correspond to a one shell progression towards the goal, or it may correspond to one or more steps backwards. However, the transition can not correspond to a jump from one shell to a shell closer to the goal states if they are not in succession. This gives rise to the following theorem:

THEOREM 1. *Assume that state s has measurable depth $d > 0$. If there exists a trace from s that can reach a failure state, then this trace will have to visit a state at depth $d-1$ before reaching the failure state. In fact, it will have to go through a state at depth $d-1$ before reaching* any *state of depth $d-2$ or lower.*

PROOF. Follows directly from the definition of measurable depth. □

The lighthouses can thus be seen as stepping stones that we will need to visit on the path to the goal in a particular order if we can reach the goal at all. This insight into the depths of successors of measurable states suggests a simple heuristic for using the backshells to generate counterexample traces: Assuming that we are currently in a state at depth i, we know that any sequence reaching the goal state, must eventually transition through shell $i-1$, $i-2$, ... before reaching the final goal at depth 0. It thus makes sense to search for continuations of the trace that end up in these shells in sequence. Note however, that the state progression forwards towards lower numbered shells may be interrupted at any time by a jump backwards by one or more steps.

4.2 Repeat Extender Algorithm

We will now put our insight into the order stepping stones have to be visited to use. Assume that we have generated a trace to a state `CurrState` with measurable depth `CurrDepth` > 0. The following is a simple approach to generating an extension of this path that may come closer to the goal in the sense that it has a lower depth:

1. `Len` $= 1$, `GoalDepth` $=$ `CurrDepth` $- 1$, `CurrExt` $= [\,]$
2. Given a resource bound for the BMC engine, try to find a length `Len` trace from `CurrState` to some state at depth `GoalDepth`.
3. If such a trace exists, then this is the new `CurrExt`. If `GoalDepth` $= 0$ we are done, otherwise increment `Len` and decrement `GoalDepth`. Goto 2.
4. If no such trace exists, increment `Len` and goto 2.
5. If we ran out of resources, give up.

When this *trace extension algorithm* exits with a nonempty trace, we have either hit the final goal shell, or we have a found a path into an intermediate shell with a lower depth.

The question is now how to continue the search if the trace extension algorithm finished with `CurrExt` containing a path to a shell of depth > 0. In practice this will be he vast majority of cases, compared to how often we find a path directly to the goal, as we will need to work around a limited search capacity for realistic examples. Our solution is to take the first transition in the `CurrExt` sequence, delete the first state in `CurrExt`, decrement `Len`, and update `CurrState` and `CurrDepth`. We then call the trace extender again, which hopefully now will be able to go deeper as we have taken a transition that have moved us closer to the lowest numbered shell we have encountered. We iterate this operation until we either (1) hit the goal states, or (2) the trace extension procedure returns an empty trace. We will refer to this as the *repeat extender* algorithm.

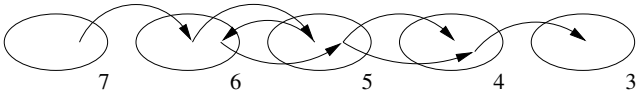

Figure 1: Repeat extension example.

As an example of how the repeat extension algorithm works, consider Figure 1, where we are attempting to find a path forwards from the initial state in shell 7. The first invocation of the trace extension algorithm runs out of steam after having found a length 5 extension that leads to shell 4. The repeat extender then moves to the state in shell 6, and attempts to reach shell 3 again using a length 5 extension. This results in the discovery of a new extension that visits a different state in shell 4 before arriving in shell 3.

4.3 Counterexample Generation Algorithm

We now have the building blocks necessary for a counterexample generation algorithm.

The simplest possible way to make use of the repeat extender algorithm is to generate the stepping stone lighthouses corresponding to our current abstraction, and then call the repeat extender with an empty trace starting from the initial state at the starting depth. If we can not reach the goal, then we give up and attempt to refine our abstraction.

However, as we will be mixing proof attempts on the abstract machine with searches for counterexamples, we want to balance the level of effort we put into each. When the size of the abstraction grows larger, we will be spending significant time on proof attempts. It would therefore be beneficial to be able to extend the search using some notion of restarting when the repeat extender has advanced when it terminates, but can not make further progress.

One possibility for a restart heuristic would be to return to the initial state and force the search engine to return a different first trace extension path than the one we explored the first time. We choose the simpler solution of taking a number of random steps from the current state in order to move in an arbitrary direction and then restart the trace extension process from this new state. If the random steps

lands us in a state that does not have measurable depth, then we abort this iteration's counterexample search altogether. We repeat this process until we reach a resource threshold or abort. Our current heuristic for the number of random steps is to take as many steps as the length of the last `CurrExt` that was consumed by the repeat extender.

Note that as all our trace construction is done on the concrete system, every failure trace that our algorithm generates to the goal states will be a true counterexample.

5. ROBUST REFINEMENT GENERATION

There is a plethora of methods for analyzing counterexamples from the abstract system and extracting candidates for registers to add in the next refinement iteration. We will now present a blend of three different techniques that has been instrumental in arriving at our current counterexample search capacity.

We first present the three individual techniques that we are making use of, and after this we present our combination method. The three individual methods have all appeared previously in the literature. Our combination method is novel.

5.1 Multiple Counterexample Analysis

Assume the shortest counterexample on a given abstract model has length l, and that we are using BDD fixpoint computations to find it. Let S_i for $0 \leq i < l$ denote the set of states that can occur at time i in some length l abstract counterexample. It is observed in [6] that it is straight forward to modify the standard algorithms for generating counterexamples to return BDDs characterizing $S_0 \ldots S_{l-1}$. Given these BDDs, the idea in [6] is to analyze each S_i to figure out whether an excluded register input is

- *Essential*, in the sense that the input has the same value in every state in S_i.
- *Irrelevant*, in the sense that if there exists a state in S_i where the input has value v, then S_i also contains the state that differs only in that the input has the opposite value.
- *Regular*, in the sense that the input is not irrelevant, but it is not essential.

The multiple counterexample (MCE) analysis thus studies the impact of individual registers not only in a particular counterexample of length l, but in *every* such counterexample.

Given the resulting classifications of the excluded registers at time i, we assign higher scores to essential variables and lower score for irrelevant variables. The overall MCE score for an excluded register is the sum of time instance scores. The intuition is that if a variable is essential one or more times, then it is likely that the addition of this register to the abstraction will remove many counterexamples. Conversely, if it almost always is irrelevant then it is unlikely that the addition of this register will affect the current set of counterexamples.

5.2 Conflict Analysis

Conflict analysis [12] was one of the first described approaches for using a counterexample to rank register candidates for possible abstraction inclusion.

Given a counterexample on the abstract machine that is *spurious* in the sense that it is not transferable to a concrete counterexample, the conflict analysis separates out the stimuli on real inputs in the concrete system from the stimuli on inputs corresponding to excluded registers. As the counterexample is spurious, it is likely that the value assignments appearing on the one or more inputs corresponding to excluded registers are inconsistent with what the registers would contain in the same situation. Excluded registers with a lot of conflicts are thus good candidates for inclusion in the abstraction as they probably will remove the counterexample at hand.

The conflict analysis in [12] calculates conflicts using three valued simulation. The concrete machine is first initialized with the initial state. The concrete machine is then repeatedly driven one step with an input vector that is all X except for whatever values were extracted for the real inputs at this time instance. The resulting state is compared with what is specified by the counterexample input values corresponding to excluded registers. If the values differ, then this is regarded as a conflict for that register at the current time instance, and the state contents is updated with the value from the counterexample. This process is repeated until the whole counterexample has been processed. The overall score for a variable is equal to the number of conflicts the variable had during the whole simulation run.

5.3 SAT-based Register Scoring

When a standard DPLL-based SAT-solver detects that a formula is unsatisfiable, it is possible to reconstruct an explicit proof of this fact if we have taken care to save some data during the execution [13, 7]. The idea in SAT-based register scoring [3] is to analyze appropriate proof objects and use this data to guide the refinement process. In the original presentation of SAT-based register scoring, the following idea is put forth for finding a upper bound on the subset of excluded registers that can remove a spurious abstract counterexample.

Let us define two traces from different machines as *agreeing*, if they have the same length, and the state values for all registers that are present in both machines always have the same value. Now, as our current abstract counterexample is spurious, there exists a shortest prefix of this trace for which there exists no agreeing concrete trace. If we construct a formula characterizing all concrete traces agreeing with the shortest prefix, we have thus constructed a formula without models. If the last time instance of a register does not appear in the proof of unsatisfiability for the characterizing formula, then the addition of this register will not affect the fact that the same prefix will be nontransferrable. The set of excluded registers whose last time instance appear in the proof are thus an upper approximation of the set of registers that we need to consider to add to the abstract model in order to remove the spurious counterexample.

5.4 Our Refinement Strategy

Our refinement strategy is a combination of the methods in Sections 5.1, 5.2, and 5.3. In the case of SAT-based scoring, we are applying our own variation of the original idea.

Our refinement strategy first compute the conflict score and MCE scores for all registers. Essential, regular, and irrelevant registers are assigned score 100, 1, and 0 respectively. We then form an overall register ranking by a lexi-

cographical ordering that uses a higher priority for the conflict score. Following [12], we next greedily attempt to find a minimal subset of registers that removes all counterexamples of the current length. We do this by adding registers one by one starting with the highest ranked register, until all counterexamples of this length disappear on the abstract machine. We then attempt to remove the added registers in reverse order, leaving registers in if counterexamples of the current length reappears. We only consider registers for adding that either have a nonzero conflict score or a nonzero MCE score. If we end up with a subset of registers that removes all counterexamples of the current length, then we are done.

If we can not find a subset of registers that removes all counterexamples, then we use a version of the SAT scoring technique in Section 5.3. However, rather than analyzing why a particular prefix of a counterexample can not be transferred, we study why we can not progress from the lowest numbered shell we have visited.

Our idea for SAT-based scoring builds on the observation that if we look at the set of concrete states with the lowest depth encountered during our search, then these states are miscategorized by the current abstraction in the sense that placing them at this depth is optimistic—even very long sequences can not reach from these states into a shell closer to the goal.

We know that as we can not progress from the encountered states in the lowest numbered visited backshell, a formula expressing that one or more of these states can transition into the next shell will be unsatisfiable. If we use an instrumented SAT-solver to solve this formula, we can thus extract a subset of excluded registers that actually appear in proof of this fact. If we do not add these registers to our abstraction, then the concrete states we could not get out of in the last shell will still be categorized in the same shell. Once we have computed the subset of excluded registers that may affect whether the deadend states will be placed in the lowest numbered visited shell again, we choose the two highest ranked registers according to the lexicographical score.

Note that the first pass of our overall strategy is focused on increasing the length of counterexamples greedily. The reason for this is that we believe that longer counterexamples will imply that the backshells spread out the states more in terms of depth, which will mean that the depth of a state will be a better predictor. When we can not increase the counterexample length, we use the SAT-based scoring to bring registers into the abstraction that will not make the abstract counterexamples longer, but that we think will remove the cause of the failed counterexample construction in the current iteration. In our experience, this later pass is often very important for progressing rapidly towards a failure.

6. EXPERIMENTAL RESULTS

We have implemented the failure generation algorithm in an abstraction refinement framework that we call NCS. We will now study the performance of NCS on real design bugs in five very challenging industrial systems, and compare it to a three alternative approaches to formal property falsification.

In Table 1, we present data on the size of the five falsifiable circuits that we have applied NCS to. The systems range in size from several hundred registers up to close to five thousand registers. The circuits are industrial circuits that have been generated by synthesis from Verilog descriptions, and then augmented with environment constraints. They all are out of range for BDD-based exact model checking. Note that the size of the circuits is a true measure of search complexity as all formal searches are performed on the concrete system. In the cases where we are aware of the minimum lengths of failures, we also present this. For problems three, four and five, no failures have been found previously.

Problem	Registers #	Minimum failure # cycles
1	434	109
2	4895	30
3	839	-
4	4494	-
5	634	-

Table 1: System metrics

In Table 2, we present results on using NCS to find failures of partial correctness statements for the five systems. We compare NCS to (1) using a BMC engine on the whole system directly without any guidance at all, (2) using the intersection of forward and backward shells from the abstract system as constraints to the BMC engine in the manner of the DiVer framework [8], and (3) guiding the BMC engine by the current abstract failure in the manner of RFN [12]. In the table, we present the runtimes for (2) and (3) under the headings BMC Shell and BMC Trace, respectively. The core search engine used by all techniques is an ATPG-based BMC engine developed internally at Synopsys.

Problem	NCS (s)	BMC (s)	BMC Shell (s)	BMC Trace (s)
1	62 763	18 769	>20h	>20h
2	3 918	12 204	>20h	>20h
3	22 190	>20h	>20h	53 501
4	10 299	>20h	>20h	>20h
5	130 886	> 5 days	> 5 days	> 5 days

Table 2: Runtimes for finding failures

Note that there are many ways to implement the proposal in the RFN paper for using counterexamples from the abstract system to guide the search for real counterexamples. We have taken the following approach, which we believe achieves a reasonable balance between speeding up the search, and being too constrictive: Whenever an abstract counterexample has been generated on the current abstraction, we check whether there exists a trace of the same length on the concrete machine where the inputs and real state variables that exists in both models are assigned the values from the abstract counterexample. We do not constrain the value of internal points in the circuit.

As can be seen in Table 2, NCS is the only method that can find the failures on all five systems. NCS outperforms all of the other methods on all examples, with the exception of problem one. The reason that NCS is slower than pure BMC on the first problem is that NCS needs many BMC invocations to construct a single trace. As a result, we may incur a performance penalty compared to a single run of BMC when examples are within capacity limits. However, as

demonstrated by problem two, there are examples within the range of BMC where the decomposition into many simpler searches outperforms a single BMC run.

Shell-guided BMC is not able to solve any of the problems. It seems that for the problems at hand, the possible search pruning provided by adding the constraints has a hard time offsetting the cost of increased size of problem specifications. Nevertheless, that we see no positive effects at all are somewhat surprising in the light of the experiences from [2] where pruning based on BDD information provides significant speedups. One possible reason for why we do not see the same effects may be that we are using an ATPG-based search engine that order variables differently than a SAT engine would. It is possible that this gives rise to fewer erroneous decisions that needs to be pruned. The standard SAT variable ordering heuristics known are known to be suboptimal for BMC [11], so one theory may be that BDD-based pruning may be more useful in this context.

Trace-guided BMC can solve problem three, but it substantially slower than NCS. Both engines finds length 1005 traces, but trace-guided BMC needs a larger abstraction (20 registers compared to 15 registers for NCS). Part of the reason for the slowness of trace guided BMC is that the concrete searches fail to reconstruct length 1005 counterexamples several times, and even though each of the failed BMC checks are constricted they are still expensive. It is possible that a better heuristic for how to constrain the trace further would give better results. However, this would come at the price of forcing a closer correspondence between the models so it is not obvious that it would be a good idea.

In Table 3, we present some NCS statistics for the five problems, such as the size of the structural abstraction that is needed to find a counter example, and the length of the failure that is found. The failure that NCS generates for problem one illustrates our tradeoff between capacity and counterexample length—the generated trace is five hundred times longer than the shortest trace. The reason for this is that our current implementation tries hard to use as small abstractions as possible, and that our restarts do not throw away trace prefixes. If we had given up on this trace and built a slightly larger abstraction, we would have generated better lighthouses that would have decreased the failure length to a few hundred cycles.

Problem	NCS size abstraction	NCS cycles #
1	53	50 708
2	28	866
3	15	1 005
4	19	378
5	65	2 726

Table 3: Data for NCS failures

7. CONCLUSIONS

In this paper, we have shown how counterexample guided abstraction refinement can be augmented to be able to search for long counterexamples. Our key idea is to use the BDDs representing the shells generated backwards on the abstract machine as intermediate targets that we traverse forwards and possibly also backwards on the way to the goal.

As our results indicate, the resulting method can extend the range of failures that can be found substantially, and can outperform both stand-alone BMC and related methods for using abstraction information to guide formal searches.

As future work, we are interested in investigating how the methods for improving semi-formal verification and simulation can be used to augment our method further. We are also interested in evaluating to what extent we can generate shorter failure traces by throwing away the current trace and restarting searches from the initial state, rather than just taking a number of random steps when we restart the repeat trace extender. One additional benefit from this is that our current restart facility has the feature that it can not get us out of a terminal strongly connected component that does not contain the goal. However, this has not been much of an issue for us so far, possibly because the searches that veer off into the wrong direction are aborted once our search resource constraint has been exhausted. The correct path may thus be found in later iterations with different abstractions.

8. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC '99*, 1999.

[2] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversals. In *DATE*, 2003.

[3] 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 *FMCAD '02*, 2002.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00*, 2000.

[5] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal. SIVA: a system for coverage-directed state space search. In *Journal of Electronic Testing: Theory and Applications*, February 2001.

[6] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi. Multiple-counterexample guided iterative abstraction refinement. In *TACAS '03*, 2003.

[7] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, 2003.

[8] A. Gupta, C. Wang, M. Ganai, Z. Yang, and P. Ashar. Abstraction and BDDs complement SAT-based BMC in DiVer. In *CAV '03*, 2003.

[9] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD*, 2000.

[10] R. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[11] O. Strichman. Tuning SAT checkers for bounded model-checking. In *CAV '00*, 2000.

[12] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation, and hybrid engines. In *DAC '01*, 2001.

[13] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, 2003.