Searching for Counter-Examples Adaptively

Sharon Keidar¹ and Yoav Rodeh²

¹ IBM Haifa Research Laboratory
sharon@il.ibm.com

² IBM Haifa Research Laboratory, and Weizmann Institute of Science, Rehovot, Israel
yrodeh@wisdom.weizmann.ac.il

Abstract

We describe a framework for finding counter-examples in a transition system. This framework constructs an on-the-fly heuristic for choosing states that are good candidates for exploration, and refines this heuristic when it realizes it made mistakes. The refinements are always relative to the mistakes made, and therefore the amount of overhead needed for constructing the heuristic is minimized. This heuristic is used for partial exploration of the state space, directed toward reaching a user specified set of states. Empirical results show this method to be a strong competitor to the standard BFS based methods.

Keywords: model checking, directed search, over approximation, under approximation

1. INTRODUCTION

Interest in formal verification has been increasing through the past years because of the promise it holds, especially for the hardware community. While formal verification set its foot in the industry, use of it is limited due to the fact that current methods cannot deal with full designs, but only with small blocks. This is because the size of the state space to be checked is typically exponential in the size of the design description.

The classical, most practical problem in formal verification is given a set of initial states, rules for the allowed transitions between states, and a set of bad states, decide whether there is a path between an initial state and a bad state that does not violate any of the transition rules. We call this path a *counter-example*. For example, given a hardware design in VHDL, the user may define the set of bad states as the set of states where two processes are in the critical section together. Then, if such a state is reachable, the verification tool should give a path (execution sequence) from an initial state of the design to a state that is in this bad set. Otherwise it should tell the user no such state is reachable.

Lately [2, 7], there has been an understanding that perhaps a more important question than the question of the existence of a path, is the question of finding such a path assuming it exists. This is because counter-examples are more useful than simple yes answers. This diverts the attention from trying to prove the validity of formulas into finding counter-examples.

The most common solution is to do a Breadth First Search (BFS) type search of all states reachable from the initial states and see if this set intersects the set of bad states. The problem with BFS is that a great deal of effort will be put into exploring states that can by simple methods be ruled out as candidates for states that can reach the set of bad states. An even greater waste can be seen: perhaps there is a very simple counter-example that can be found easily, while the BFS search goes over all possibilities.

Some work has been done with *directed search*, where instead of doing blind BFS search, one chooses the states to explore using a heuristic that should direct the search to the set of bad states faster. The main idea is to try to give scores to states: the lower the score, the better chance the state has for reaching the set of bad states fast. The choice of heuristic for scoring states is clearly critical in this method since a bad heuristic may actually prolong the run by misdirecting the search engine. In [1] the score of a state is its hamming distance from the set of bad states. In [7] the score of a state is its approximate distance from the set of bad states, which is calculated before hand using over-approximate

backwards steps from the set of bad states. The basic approach of this paper is similar, except we refine the heuristic when we see fit.

Two papers [5, 8] suggest automatic methods for refining abstractions using false counter-examples that occur in the abstracted system. They find a small set of variables such that their addition to the abstracted system will prevent the recurrence of the same counter-example in the new system. They then proceed and basically start over all calculations with the new abstraction. Although our implemented method for refining the heuristic of scoring states has some similarity with this method, it differs very significantly in that we use previous results much more efficiently, and in that our refinement is done in a very local way, and not to the whole system.

In this paper we propose an automatic trial and error process for building a heuristic for scoring states on-the-fly, and refining it when we have evidence that it has misled us. We perform this refinement using this evidence, and therefore not only do we improve the heuristic only as much as needed, but also only where we know it has misled us before. We do not spend excess energy on refining it in places that we may not even reach. Our experimental results show that this method can work very well in combination with the BFS method, since there are many instances where their efficiency differs by much — some where the new adaptive method works much faster, and some where the BFS does.

2. THE PROBLEM

We are given a transition system, specified by a set of initial states and a relation that specifies the allowed transitions between consecutive states. We are also given a set of bad states. The problem is: assuming there are paths between the set of initial states and the set of bad states using only allowed transitions, find one of them (preferably a short one).

3. HIGH LEVEL DESCRIPTION OF THE SOLUTION

We construct a sequence of sets of states: $D_0, \dots, D_n$, where D_0 is the set of bad states, and each D_i is a superset of the set of states that can reach D_{i-1} in one step. The last set, D_n , is the first set in the sequence that intersects the set of initial states. In Section 4.2 we show that n is no more than the length of the shortest path between the set of initial states and bad states.

This sequence represents our heuristic for giving scores to states. The *score* of a state is the smallest i s.t. the state is in D_i . This makes sense, since the closer i is to zero, the closer the state should be to the set of bad states in terms of steps needed to reach it.

We choose from the set of states we wish to explore (initially, this is the set of initial states), some that have the lowest score (best candidates), and calculate their set of successors using the transition relation. If the score of the original states was i , and our heuristic is accurate enough, some of their successors should be in D_{i-1} . If so, we continue in exactly the same fashion. If nothing goes wrong, we will always decrease the score, and eventually reach D_0 , which is the set of bad states, meaning we found a path.

Since our D_i 's are not exact, we may come to a point where our chosen states were in D_i but none of their successors are in D_{i-1} . So our set D_i is not accurate enough, since it should represent the set of predecessors of D_{i-1} . This is where the refinement mechanism comes in. We refine D_i by over-approximating¹ the set of predecessors of D_{i-1} , so the result does not contain the set of states we made the mistake on — those we explored and found that none of their successors are in D_{i-1} .

After the refinement, we return to choosing the lowest score states by our new refined heuristic and proceed to explore them.

¹An *over-approximation* of a set of states is a superset of the original set of states, that has a shorter description. By a short description we mean that the amount of memory needed to describe the set is small. For example, if the set is represented by a Binary Decision Diagram (BDD, see [3] for details on this data structure), the number of nodes in the BDD is the length of the description.

4. DETAILS OF THE SOLUTION

We will now proceed to describe the inner workings of the algorithm discussed in the last section. We keep a set of the states we reached so far: *Reached*, that is incremented throughout the algorithm. It is initialized to be the set of initial states, and each time we calculate the set of successors to a set of states, we add the result to *Reached*.

We also keep the list $D_0, \dots, D_n$. At first we only have D_0 , and we calculate using any over-approximation we wish a superset of D_0 's predecessors. Denote this set by D_1 . We continue in the same fashion: calculating a superset of D_1 's predecessors to get D_2 , and so on, until D_i contains a state from *Reached*.

Now, we take some of the highest ranking states (smallest i), by choosing them from the intersection of *Reached* with the last D_i . Denote this set by *Candidates*. We proceed to calculate *Candidates*'s set of successors: *Successors*. If we moved up in our ranking — the result has an intersection with D_{i-1} , we pick states from there and proceed.

Otherwise, we refine D_i using D_{i-1} and the last mistake we made (we made a mistake — we picked *Candidates* that were supposed to have successors in D_{i-1} but they don't). This refinement process is the core of our framework and is described in Section 4.1. Now we see if after the refinement there are still any of the reached states in D_i . If so we explore them. Otherwise, we do the same as we did when we started out: do any sort of over-approximation to calculate a superset of the set of predecessors of the new D_i , and go on doing over-approximated backward steps until we find a reached state.

1. $Reached \leftarrow$ set of initial states.
2. $D_0 \leftarrow$ set of bad states.
3. $i \leftarrow 0$ (i is the smallest s.t. there are states from *Reached* in D_i).
4. While $Reached \cap D_0 = \emptyset$ do
 - (a) If $Reached \cap D_i \neq \emptyset$ then
 - i. Pick a set of states from $Reached \cap D_i$. Set *Candidates* to be this set.
 - ii. Calculate the set *Successors* which is *Candidate*'s set of successors.
 - iii. Add *Successors* to *Reached*.
 - iv. If $Successors \cap D_{i-1} = \emptyset$ then refine D_i according to *Candidates*.
 - v. Otherwise, set $i \leftarrow i - 1$.
 - (b) Else
 - i. Calculate an over-approximation of the set of predecessors of D_i . Denote the resulting set A .
 - ii. If we have a previous D_{i+1} , set $D_{i+1} \leftarrow D_{i+1} \cap A$.
 - iii. Otherwise, set $D_{i+1} \leftarrow A$.
 - iv. Set $i \leftarrow i + 1$.

If this procedure terminates, then we have a state that is in *Reached* and in D_0 ; i.e., there is a path within *Reached* from an initial state to a bad state.

The procedure itself does not generate a path, but it builds a subset of the reachable state space, *Reached*, that contains such a path. The simplest way of finding it, would be to do BFS search inside this set, until we reach the set of bad states. This is in fact rather efficient, since *Reached* should be significantly smaller in size than the actual reachable state space, and so the search is a great deal faster.

4.1. The Refinement Procedure

One important detail left unexplained, is how we do the refinement. The idea is to find the least expensive refinement (in terms of description size) such that it removes *Candidates* from D_i , because we know they are not in the set of predecessors of D_{i-1} — after all, we saw that none of their successors are in D_{i-1} . The simplest method would be to remove *Candidates* from D_i , and this would be the new D_i . But this way we've learned nothing extra.

We proceed to calculate an over-approximation to the set of predecessors of D_{i-1} , such that the over-approximation does not intersect *Candidates*.

One preferable way of doing this, is by first finding a short description of a superset of D_{i-1} that does not contain *Successors* — denote this set by D'_{i-1} . We know that not only the set of predecessors of D'_{i-1} is a superset of the set of predecessors of D_{i-1} , it also does not intersect *Candidates* (because *Successors* are all the successors of *Candidates*, and *Successors* does not intersect D'_{i-1}). We now find A' , an over-approximation of the set of predecessors of D'_{i-1} that does not contain *Candidates*. We explain how this is implemented in our system in Section 5.4.

A' is a superset of the set of predecessors of D_{i-1} that does not contain *Candidates*. The natural next step would be to get the new refined D_i by intersecting the old one with A' . However, we first find A , a short description superset of A' that does not contain *Candidates*. Now we intersect the old D_i with this A to get the new D_i . This may seem counterintuitive: we already got a good result, why should we make it less exact? The reason is we will want to use this result in the future, calculate its set of predecessors for example. So we want it to have a short description so it will be easy to work with later.

Figure 1 describes the refinement process.

FIGURE 1: The refinement procedure: The approximation creates larger sets that are simpler to describe. The direction of the arrows is the direction of the successor operation.

4.2. Comments

Notice that in our list of approximations $D_0, \dots, D_n$, n is always smaller than the shortest path between the set of initial states and the set of bad states. This is because in the set of reached states we will always have a state that is at most that distance from the set of bad states, and therefore it will appear in the corresponding D_i or before, since D_i is a superset of the set of states that are at distance i from the set of bad states. Therefore, the algorithm will not need to expand the list beyond this length.

One drawback of this framework, is that if there is no path between the initial states and the set of bad states, the algorithm will run forever. This is something we are willing to accept, since we are more interested in finding counter-examples than proving validity of formulas.

5. IMPLEMENTATION DETAILS

We have implemented this framework as a part of RuleBase [9], a formal verification tool developed in IBM's Haifa Research Laboratory. We proceed to describe the important details of this implementation.

5.1. The Problem

First, we describe the specific problem:

1. **The input:** The set of states S is the set of assignments to n boolean variables $x_1, \dots, x_n$. The sets of initial states S_0 and bad states B are given as a single BDD on the n variables. For each one of the n variables, x_i , we have its next state relation r_i , which is an $n + 1$ -ary relation. r_i is also given in BDD format.
The allowed transitions are defined as follows: $(\alpha'_1, \dots, \alpha'_n)$ is a successor of $(\alpha_1, \dots, \alpha_n)$ iff for every i , $r_i(\alpha_1, \dots, \alpha_n, \alpha'_i)$ is true. In other words, the i -th next state relation specifies what values of x_i (the $n + 1$ operand) are allowed as a function of the current state (specified in the first n operands).
2. **The output:** Find a sequence of states $(s_0, \dots, s_d)$, where each s_i is an assignment to $x_1, \dots, x_n$. We require that s_0 is in S_0 , s_d is in B , and for every $i > 0$, s_i is a successor of s_{i-1} .

5.2. Implicitly Conjoined BDD Lists

In [6], Hu describes a new data structure based on BDDs. Instead of representing a set using one BDD f , it is represented using a list of BDDs $f_1, \dots, f_k$, where $f = \bigwedge_{i=1}^k f_i$. This new data-structure loses the uniqueness of representation that BDDs have, yet gains in flexibility. Intersecting two BDD lists is easy: simply join them to be one longer list. In [6], a heuristic algorithm for minimizing BDD lists is described, which is used whenever the BDD list is changed in some significant way.

We use this data structure to describe each D_i — the set that represents our approximation to states that are in distance i from the set of bad states.

5.3. Large vs. Small Steps

To make our refinement process as fast as possible, we would like to take the set *Candidates* to contain a small number of states. This way the refinement is easy, but we lose the power of symbolic image calculation, and our set *Reached* will be very small. To remedy this we combine the two approaches by sometimes taking *Candidates* to be a large set (using the heuristic of [10] to limit the BDD size), and in this case do not try to refine our set D_i even if *Successors* does not intersect D_{i-1} .

5.4. The Refinement Procedure

The refinement procedure is the core of our framework, and if it is not implemented with care, the overhead involved when using it may be unacceptable. The most crucial point is that the resulting BDD from the procedure has as little variables as possible. There are a number of reasons for this:

1. When calculating the set of predecessors of a set A , the number of variables in the support of A is the best practical estimate for how difficult the calculation will be.
2. The number of variables is also a good estimate for the size of the BDD.
3. Maybe the most important reason is that many times most of the information needed is encoded in a small number of the variables, and all we have to do is find out which ones.

Therefore, most of our approximations are based on keeping a small number of the variables, so the approximations are on one hand effective — reduce the memory and time consumption, and on the other hand preserve most of the information we need.

5.4.1. Differ:

We use the function $Differ(A, B)$ in our implementation. *Differ* gets as input two non-intersecting sets A and B (represented as BDDs), and returns a small set of variables W such that they are “enough” to distinguish between A and B . More formally, if $Differ(A, B) = W$ then $\Pi_W(A) \cap \Pi_W(B) = \emptyset$ where Π_W of a set is its projection on variables W :

$$\Pi_W(A) = \{\alpha \mid \exists s \in A \text{ such that } s \text{ and } \alpha \text{ agree on all the variables in } W\}$$

Differ is implemented in the following fashion:

1. Choose some arbitrary order on the variables : $x_1, x_2, \dots, x_n$.
2. $Result \leftarrow \emptyset$
3. For $i = 0$ to n do if $(\exists_{x_i} A) \cap B \neq \emptyset$
 - (a) Then $Result \leftarrow Result \cup \{x_i\}$
 - (b) Else $A \leftarrow \exists_{x_i} A$
4. Return $Result$

Recall that we have two lists of BDDs, one describing D_i and one describing D_{i-1} . We also have a set *Candidates* (a single BDD) which is contained in D_i , and a set *Successors* (a single BDD) that does not intersect D_{i-1} .

We wish to find a BDD representing a set A that we will append to the list describing D_i . This set A has to satisfy the following properties:

1. A is a superset of the set of predecessors of D_{i-1} .
2. A does not intersect the set *Candidates*.

We first construct a single BDD which is a superset of D_{i-1} and does not intersect *Successors*. Since D_{i-1} does not intersect *Successors*, we find (using a greedy approach) a sublist of the BDD list D_{i-1} such that its intersection with *Successors* is empty. We intersect these BDDs to get a BDD F , and so F is a superset of D_{i-1} that does not intersect *Successors*. Since *Candidates* is a small set, *Successors* is usually a small set, and therefore only a small number of BDDs from D_{i-1} is needed.

The next step is to further over-approximate F , so that it will contain a small number of variables. We set W to be the result of $Differ(F, Successors)$, and set D'_{i-1} to be $\Pi_W(F)$. The definition of *Differ* guarantees that D'_{i-1} will not intersect *Successors*. Also, D'_{i-1} is obviously a superset of D_{i-1} .

Denote by $Pred(D'_{i-1})$ the set of predecessors to the set D'_{i-1} . Recall our original set $T = Reached \cap D_i$ from which we picked *Candidates*. Any state which is in T and not in $Pred(D'_{i-1})$ should not belong to D_i (note that *Candidates* are such states). In Section 5.5 we show how to calculate an over-approximation A' to the set $Pred(D'_{i-1})$ which will agree with $Pred(D'_{i-1})$ on the set T .

Once we've got A' , our first option is to add it to the list describing D_i . This is problematic because A' usually has a relatively large BDD with a large number of variables. We therefore over-approximate A' :

1. $W \leftarrow Differ(A', T \cap \neg A')$
2. $A \leftarrow \Pi_W(A')$

We get that A is an over-approximation of A' and therefore is an over-approximation of the set of predecessors of D_{i-1} . Also, $A \cap Candidates = \emptyset$ as we required. If A 's BDD is too large, we can replace Step 1 by $W \leftarrow Differ(A', Candidates)$. Then A would generally be smaller, but we do not eliminate some states from *Reached* we could have.

5.5. Calculating the Pre-image

Given a set G (which is D'_{i-1} in our case) and a set T we wish to calculate an over-approximation A' to the set of predecessors of G such that A' agrees with $Pred(G)$ on T : $A' \cap T = Pred(G) \cap T$.

Recall that r_i is the next state relation of variable x_i . It is an $(n+1)$ -ary relation where $r_i(\alpha_1, \dots, \alpha_n, \alpha'_i)$ if and only if α'_i is a legal value of x_i in a successor of the state $(\alpha_1, \dots, \alpha_n)$. Instead of r_i we take $\hat{r}_i$ that satisfies:

1. For every $(\alpha_1, \dots, \alpha_n)$ we have $r_i(\alpha_1, \dots, \alpha_n, \alpha'_i) \implies \hat{r}_i(\alpha_1, \dots, \alpha_n, \alpha'_i)$
2. For every $(\alpha_1, \dots, \alpha_n) \in T$ we have $r_i(\alpha_1, \dots, \alpha_n, \alpha'_i) = \hat{r}_i(\alpha_1, \dots, \alpha_n, \alpha'_i)$

If we use these $\hat{r}_i$ in the pre-image calculation, we will indeed get a set B that satisfies:

1. $Pred(G) \subseteq B$
2. $Pred(G) \cap T = B \cap T$

The point of all this is that we can find such $\hat{r}_i$ with BDD size that is smaller than that of r_i (using a variation of the simplify operation in BDDs), making the pre-image calculation much faster.

In our case the set T is a subset of the reached states so far, so it is relatively small, and therefore the resulting $\hat{r}_i$ is usually significantly smaller in size than r_i . The only problem is, the result B is a lot larger than the set $Pred(G)$, and we lose too much in exactness.

The way we constructed B , we know that $P = T \cap Pred(G)^c = T \cap B^c$. So we can calculate P . We then set $W = Differ(P, B)$, and now calculate $A' = \Pi_W(Pred(G))$. A' is obviously a superset $Pred(G)$, and also does not intersect P — meaning it agrees with $Pred(G)$ on T .

The naive way of calculating A' would be to first calculate $Pred(G)$ and then perform the existential quantification. However, this way we earn nothing since we calculated the exact $Pred(G)$, which is something we wish to avoid. So, we combine the quantification with the pre-image computation. This is done by a trivial generalization of the method proposed in [4] for image and pre-image calculation. Note that all stages before the last were only used to find W .

6. EXPERIMENTAL RESULTS

We implemented the framework suggested in this paper as an alternate execution mode of RuleBase. RuleBase is a state of the art BFS based model checker developed by the formal verification group of IBM Haifa Research Laboratory. Perhaps the best evidence for this paper's contribution is that it is many times used for practical formal verification by RuleBase users. This is because on a good fraction of real life examples we outperform RuleBase by an order of magnitude (or more) in running time. Of course, there are many examples where the opposite is true, and therefore RuleBase users are advised to try both methods, classical and new, to achieve the best results.

We give some example results in Table 1. In this table, the prefix R denotes RuleBase's BFS results, and A denotes our adaptive method's. Also, *time* is the running time in seconds, *nodes* is the maximal number of BDD nodes allocated throughout the run, *space* is the size of the state space explored until a counter-example was found, and *ce* is the length of the counter-example generated. The last column # is the number of refinement steps made in the adaptive method.

TABLE 1: Adaptive vs. RuleBase

Example	R-time	A-time	R-nodes	A-nodes	R-space	A-space	R-ce	A-ce	#
biu-N2	288.2	79	188017	94284	$5.2 \cdot 10^9$	$1.0 \cdot 10^8$	32	32	42
Irit-6	338.8	82.4	347604	124109	$5.7 \cdot 10^{14}$	$1.7 \cdot 10^9$	37	37	48
lhs-3-P	123.7	223.1	132809	134379	$1.0 \cdot 10^{11}$	$1.2 \cdot 10^9$	25	25	284
lhs-3-R	516	21.9	329270	80809	$1.0 \cdot 10^{11}$	$5.7 \cdot 10^5$	32	32	0
Ofer-Iris	2078.5	140	968205	129857	$5.2 \cdot 10^9$	$1.0 \cdot 10^8$	32	32	116
sbcntl	1083.8	273	530412	223380	$1.6 \cdot 10^{16}$	$1.1 \cdot 10^{12}$	32	33	126
Galileo1	10522.2	29	2854986	30481	$2.3 \cdot 10^{32}$	$4.2 \cdot 10^6$	36	36	38

7. CONCLUSIONS AND FURTHER RESEARCH

In our implementation we have added on top of the basic framework some heuristics that try to prevent possible bottlenecks. For example, when executing the refinement process, if the BDD sizes increase beyond some threshold, we abort the procedure. This type of greedy heuristics is possible since we always work with approximations: the sets D_i are over-approximations, and the set *Reached* is an under-approximation. Therefore, we have much leverage when using them that does not exist in classical model checking.

In general, this method is a great deal more flexible than classical model checking, and is therefore well suited for implementation of different heuristics. For example, A good heuristic for picking the set *Candidates* from the set of the best candidates for exploration can affect the running time greatly. There are many other options for enhancement of this framework: the employment of user guided hints, analyzing results of earlier runs to guide the current one, etc.

REFERENCES

- [1] J. Abraham, A. Aziz, J. Shen and J. Yuan, "On Combining Formal and Informal Verification", In *Computer-Aided Verification CAV '97*.
- [2] A. Aziz and M. Ganai, "Efficient Coverage Directed State Space Search", In *International Workshop on Logic Synthesis IWLS '98*.
- [3] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", *ACM Computing Surveys*, vol.24, no.3, Sep. '92.
- [4] Ilan Beer and Daniel Geist, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions", In *Computer-Aided Verification CAV '94*.
- [5] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-Guided Abstraction Refinement", In *Computer-Aided Verification CAV 2000*.
- [6] D. L. Dill and A. J. Hu, "Efficient Verification with BDDs using Implicitly Conjoined Invariants", In *Computer-Aided Verification CAV '93*.
- [7] David L. Dill and C. H. Yang, "Validation with Guided Search of the State Space", In *Design Automation Conference DAC '98*.
- [8] Shankar G. Govindaraju and David L. Dill, "Counterexample-Guided Choice of Projections in Approximate Model Checking", In *International Conference on Computer Aided Design ICCAD 2000*.
- [9] Ilan Beer, Shoham Ben-David, Cindy Eisner, Daniel Geist, L. Gluhovsky, Tamir Heyman, Avner Landver, P. Paanah, Yoav Rodeh, G. Ronin, Yaron Wolfsthal, "RuleBase: Model Checking at IBM", In *Computer-Aided Verification CAV '97*.
- [10] Kavita Ravi and Fabio Somenzi, "High-Density Reachability Analysis", In *International Conference on Computer Aided Design ICCAD '94*.