Guiding Simulation with Increasingly Refined Abstract Traces*

Kuntal Nanshi, Fabio Somenzi
University of Colorado at Boulder

Abstract

We combine abstraction refinement and simulation to provide a more efficient approach to checking invariant properties whose only counterexamples are very long traces. We allow each transition of an abstract error trace to map to multiple transitions of the concrete error trace and simulate pseudorandom vectors to build segments of the concrete trace. This approach addresses the capacity limitation of the formal verification engine as well as the short-sightedness of the simulator, thus providing a more effective technique for deep, subtle bugs.

Categories and Subject Descriptors
B.6.3 [**Logic design**]: Design aids—*Verification*
General Terms: Verification, Algorithms
Keywords: model checking, abstraction refinement, simulation

1. Introduction

Formal methods are increasingly applied in the verification of complex digital systems for their ability to detect subtle bugs. Compared to simulation, techniques like model checking [9] are much more thorough, but also much more restricted in capacity. There is therefore a clear incentive to combine the two approaches (dubbed by some *static* and *dynamic*) in an attempt to mitigate the limitations of either.

An integration of simulation with model checking is particularly desirable for very deep bugs—bugs that require perhaps many thousands of clock cycles to be excited. While model checkers, even SAT-based bounded model checkers, have trouble with the large analyses required by the long counterexamples, the probability of pseudorandom vectors to reach error states often vanishes rapidly with the depth of the search.

In this paper we present an approach that leverages the strengths of static and dynamic verification by combining them in an abstraction refinement scheme. In abstraction refinement [15], an *abstract* model is increasingly refined until it has enough detail to either prove the property of interest true, or produce a counterexample that can be reproduced in the original, *concrete* model. Counterexamples to the property in the abstract model that cannot be *concretized* are spurious and lead to refinement of the abstraction.

A counterexample to a property is a sequence of states and inputs such that each successive input causes a transition from one state in the sequence to the next. In this paper we concentrate on invariants: The property to be verified is a predicate that should hold in all reachable states; a counterexample is an execution of the model from an initial state to a state where the invariant is violated.

An abstract counterexample is in terms of abstract states and inputs. The common approach to concretization requires that a trace be found in the concrete model that is in one-to-one correspondence with the abstract one. In other words, every transition of the concrete conterexample must match one transition of the abstract counterexample. Even approaches that consider multiple (all) abstract counterexamples of a certain length insist on this corespondence, which allows them to set up the concretization check as a simple satisfiability check. The model is unrolled as many times as there are transitions in the abstract counterexamples, and each concrete state is constrained to match the corresponding abstract state (or abstract states in case of multiple abstract counterexamples).

While the approach we have just outlined has been undeniably successful in extending the reach of model checking, it breaks down when very long error traces are the only counterexamples to a given property, for two reasons: First, the resulting concretization checks may be too much, even for today's sophisticated SAT solvers; second, the analysis of the concrete model becomes problematic, because of the many refinement steps that bring the abstract model dangerously close to the concrete one and make it too complex for the model checker.

Even with a long error trace, it is often the case that a suitable abstraction may provide a *summary* of that trace in the form of an abstract counterexample such that each of its states corresponds to a sequence of states of its concrete counterpart. Our approach is to extend the concretization algorithm so that multiple concrete transitions may map to one abstract transition. The number of concrete transitions required to match an abstract transition is not known in advance; it is not even known whether the target state of the abstract transition corresponds to any reachable concrete state. Therefore, we impose heuristic resource bounds on the attempt to connect two abstract states with a sequence of concrete transitions. Specifically, we simulate pseudorandom vectors on the concrete design.

Resource bounds, then, in practice, are bounds on the number of vectors and vector sequences that are simulated during an attempt. If the bounds are exceeded, we refine the abstract model. This may improve the chances of concretization in two ways: By lengthening the abstract counterexample, and consequently reducing the distance between the preimages of the abstract states in the concrete model; and by removing truly spurious abstract counterexamples—counterexamples that go through abstract states that are the images of unreachable states only.

Because of this refinement step, the algorithm is complete, when resources are unbounded, because in the worst case, it will model check the concrete system to decide whether a property passes. In practice, the advantage of our approach, which is illustrated by the results of Section 6, stems primarily from its ability to complete the analysis (e.g., find a counterexample) using a much more abstract

*This work was supported in part by SRC contract 2004-TJ-920.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

model than other abstraction refinement techniques. It also reduces the risk that concretization check will fail for lack of memory or time, because it avoids a one-shot check of a large model.

2. Related Work

The notion of *guideposts* was introduced in [25]. Abstraction refinement and concretization has been discussed in [15, 7, 8, 5, 16, 23, 11, 4, 22]. Considerable progress has been made in refinement and concretization techniques. In these methods, however, concretization still produces a concrete trace that is in one-to-one correspondence with the abstract one. Another concretization method introduced in [2] does not insist on one-to-one correspondence, however its effectiveness is dampened by (1) capacity limits of a satisfiability (SAT) solver and (2) its difficulty in dealing with a concrete counterexample that corresponds to multiple loops over a sequence of abstract states. These limitations trigger refinements of the abstract model, bringing it closer to the concrete model. While [21, 17] do not use concretization of abstract traces, they also suffer when counterexamples are long.

An altenative to SAT based concretization is to use pseudorandom vector based simulation techniques, a hybrid approach for verification. Early attempts at hybrid verification [10, 1] aimed at improving coverage of a model, by generating psuedorandom test vectors from the information provided by the abstract model for the circuit. [13, 24], discussed the use of an abstract model, along with a combination of BDD-based, SAT-based and pseudorandom simulation based techniques, to prove that certain states are not reachable in the concrete model. These methods are not complete, i.e., they do not guarantee verification or falsification of a property.

In [12] the authors proposed a hybrid verification scheme, where they used the outcome of symbolic simulation on a complete model to complete the initialization sequence of the abstract model. The property is then checked over the abstract model. This method is also not complete as it cannot falsify the property. Another hybrid verification approach was proposed in [19], where state transitions in the complete model during simulation are mapped into abstract transitions—the abstraction map is provided by the user. A model checker then checks the property over the abstract transitions. The generated abstract model is an under-approximation, hence this method can only falsify properties.

Recently [18] proposed guided simulation to concretize abstract paths. They computed a cost function—shortest distance of each state to the goal—for each state in the abstract model. The simulation path is then guided by selecting a state which is closest to the goal based on the cost function. Their approach is different from our approach, in many ways (1) it does not refine the abstract model and can therefore only falsify a property (2) it does not take into account that the abstract and concrete paths may not have a one-to-one correspondence, and hence chooses a larger abstract model.

3. A Motivating Example

Controller logic designs are often built around finite state machines. These machine are typically small, but some of its states may be waiting states, where events—for instance counters filling up—must take place before a transition to the next state is enabled. These designs tend to be very deep and complex, making reachability analysis a hard task.

Consider one such example: B12 from the ITC'99 benchmark circuits [14]. It consists of a state machine with state variable *gamma* that can take 26 different values and whose transitions are controlled by counters and events stored in other latches in the design. A failing invariant property, *nloss* $\rightarrow$ *nl[3:0]=15*, has a long counterexample, because the path involves multiple nested loops over the state machine states. (The shortest counterexample is known to include more than 31898 states.) The counterexample must go through a state satisfying *gamma = W0*. Inspection of the design shows that the counterexample path loops over a nested sequence of states. One such sequence is {*G2, G3, . . ., G12, G2*}.

Model checking an abstract model of B12 comprising *gamma* and the variables mentioned in the property produces a short counterexample (14 states) in about one second. However, this counterexample cannot be directly applied to the original model because such a short error trace does not exist there. Since traditional approaches to abstraction refinement require a one-to-one correspondance between the states of the abstract trace and those of the concrete trace, they are unable to deduce from the abstract counterexample the failure of the property; hence, they refine the abstract model repeatedly until analysis becomes infeasible.

In contrast, the approach we propose uses the values that *gamma* takes in the abstract trace as *guide points*. The complete B12 circuit is simulated with a sequence of pseudorandom vectors. The guide points constrain the path a simulation trace can take, by restricting the sequence of values of *gamma* to those of the guide points. If the sequence of vectors is long enough (at least 35,000 vectors in this case), the path to *gamma = W0* is eventually found.

4. Preliminaries

We are concerned with the verification of digital systems that are modeled as finite transition structures. Without loss of generality, we assume that a model has a finite set of binary state variables. Each valuation of those state variables is a state.

Let $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$ be sets of Boolean variables. We designate by V' the set $\{v'_1, \ldots, v'_n\}$ consisting of the primed version of the elements of V, and by V^i the set $\{v^i_1, \ldots, v^i_n\}$. Likewise, $W^i = \{w^i_1, \ldots, w^i_m\}$. An *open system* is a 4-tuple $\langle V, W, I, T\rangle$, where V is the set of (current) state variables, W is the set of combinational variables, $I(V)$ is the initial state predicate, and $T(V, W, V')$ is the transition relation. The variables in V' are the next state variables. We assume that the transition relation is given by

$$T(V, W, V') = \bigwedge_{1 \le i \le n} v'_i \leftrightarrow \delta_i(V, W) \ , \qquad (1)$$

that is, the next value of each state variable is a function of the current state and the primary inputs. While this assumption amounts to determinism, it is not restrictive, because nondeterministic relations can be determinized by adding auxiliary inputs [6].

We consider the form of abstraction known as *localization reduction*, which is obtained by dropping from the transition relation (1) some of the conjuncts, thus turning the corresponding state variables into primary inputs. Localization reduction relies on the observation that many properties of interest are *local*, in the sense that they can be verified by considering only the variables mentioned in the property and those in their vicinity. In *abstraction refinement*, one starts with a very coarse abstraction—typically including in the transition relation only the variables mentioned in the property to be proved—and then adds more variables as required to eliminate the spurious behaviors that impede the proof of the property or the discovery of a true counterexample. The variables that are retained in the abstract model are called *visible*; the others are *invisible*.

The process of checking counterexamples to the property in the abstract model is known as *concretization*. A popular approach translates the concretization problem into propositional satisfiability. The full transition relation is unrolled as many times as there are transitions in the counterexample to be checked, and the SAT

solver is asked to decide whether there is a path in the resulting circuit that is consistent with the values of the visible variables in the abstract counterexample.

Localization reduction is popular, but it is not the only approach to abstraction in practical use. Predicate abstraction, for instance, is applied especially in software verification. Both approaches can be described in terms of an *abstraction function* α that maps concrete states to abstract states. For localization reduction, a concrete state is mapped to the abstract state that agrees with it on the visible variables; in other words, state abstraction in localization reduction amounts to projection of the state on the visible variables. We have implemented our technique on top of an abstraction refinement scheme that uses localization reduction. The principles, however, apply to other abstraction schemes as well.

5. Algorithm Overview

The overall algorithm for abstraction refinement is presented in Fig. 1. It accepts, as inputs, the concrete model M and the property to be verified Φ. In this paper, Φ is an invariant, that is, a propositional formula p over the state variables.

The algorithm starts by computing an initial abstraction M_a of the concrete model, with only those state variables that are mentioned in the property and their next state logic. The property is checked in the abstract model with a model checker. A BDD-based forward reachability analysis is used to find a path from some initial state to a state satisfying $\neg p$. Because of the nature of the abstraction, if a state satisfying $\neg p$ cannot be reached in the abstract model, it cannot be reached in the concrete model either. (Since M_a simulates M, all universal properties, including invariants, are preserved in going from M_a to M.) Property Φ, therefore holds in the concrete model and the algorithm returns TRUE.

If a state satisfying $\neg p$ is reached in the abstract model, the model checker returns a set of Abstract Counter Examples (ACEs) of length L. The abstract counterexamples are of minimum length for the given abstract model and are captured in a data structure called Synchronous Onion Rings (SORs). The SORs are the pairwise intersection of forward and backward reachable onion rings. The sets of new states encountered in the breadth-first search during forward reachability analysis form the forward reachable onion rings. Similarly, the sets of new states encountered during backward reachability analysis starting from the states labeled $\neg p$, i.e., the reached "bad states" form the backward reachable onion rings.

Guided simulation is used to concretize the current ACEs. The concrete model is simulated with pseudorandom vectors, where the resulting trace is prevented by the SORs from taking a path not leading to "bad states". Hence the notion of *guided* simulation. The SORs also serve as "milestones" along the simulation trace. The use of SORs instead of a single counterexample increase the chance of success of the (rather expensive) concretization check. Procedure simulAbs, if successful, returns a Concrete Counter Example (CCE). It is explained in detail in Section 5.1.

If simulAbs does not return a concrete counterexample, all ACEs are declared spurious and the current abstract model M_a is refined by adding *invisible* state variables to the abstract model. The selection of these variables can be guided by the analysis of the spurious counterexamples. We adopted the refinement algorithm from GRAB of [23, 22]. It is briefly explained in section 5.3. The process repeats until a concrete counterexample is found or the property is proved true.

In the event, the abstract counterexamples are not concretized, the refined abstract model eventually reaches the concrete model. In such a case, if the property fails, the SORs need not be concretized. Our algorithm returns FALSE, and a path extracted from the SORs as the counterexample. A path with one state selected from each SOR is a concrete counterexample for this model. The algorithm is thus complete, i.e., it does return a pass/fail for every property. Soundness follows from the fact that M_a retains all possible behaviors of M over the visible variables, and from failure being declared only if a concrete counterexample to Φ is found.

```
1   abstractionRefinement(M, Φ) {
2       M_a = getInitialAbstraction(M, Φ)
3       while(TRUE) {
4           SORs = modelCheckAbstactModel(M_a, Φ)
5           if(SORs empty) return TRUE
6           if (M_a = M)
7               return (FALSE, extractOnePath(SORs))
8           CCE = simulAbs(M, SORs)
9           if(CCE not empty) return (FALSE, CCE)
10          M_a = refineAbstraction(M, M_a, SORs) } }
```

Figure 1: Overall algorithm.

5.1 Simulation Based Concretization

We attempt to concretize ACEs by simulating the concrete model with guided pseudorandom vectors. The pseudo-code is given in Fig. 2. The procedure takes as inputs the concrete model M and a set of abstract counterexamples in the form of SORs.

Let $\{R^0, R^1, \ldots, R^L\}$ be the length-L synchronous onion rings, where R^0 is a set of abstract initial states and R^L is a set of abstract states labeled $\neg p$. With $a_i \in R^i$, let $a_0, a_1, \ldots, a_L$ represent an abstract counter example, such that for $0 \leq i < L$, there is an abstract transition between a_i and a_{i+1}. We designate the concrete state to which the input stimulus is applied as *current state* and the resulting state as the *next state*. A *Counter Example Segment* (CES) is a sequence of states on a path from state s_i such that $\alpha(s_i) \in R^i$ to state s_{i+1} such that $\alpha(s_{i+1}) \in R^{i+1}$. A concrete counterexample is composed of L such CESs.

First, we select an initial state s_0 in the concrete model M such that $\alpha(s_0) \in R^0$. We then get a set of *num* pseudorandom vectors. Each vector supplies stimulus to all primary inputs of the concrete model. The concrete model is then sequentially simulated with the set of vectors, starting from the initial state.

After every simulation cycle, the resulting *next state* is checked to see if the states labeled $\neg p$ can still be reached from it. A *care set* is a set of states in the abstract model that lie on the paths from the initial states to those labeled $\neg p$. If *next state* is in the care set, it is set as the *current state* for the next simulation cycle. Otherwise, the result is discarded, and the *current state* is retained for the next simulation cycle.

When a CES is found, it is tentatively added to the CCE. The last state in the CES is chosen as the starting point for the next CES search. We term these starting points *milestones*. We get a new set of vectors to continue the search for the complete counter example. A CCE is found when the simulation procedure reaches s_L such that $\alpha(s_L) \in R^L$. In this case the CESs found so far are concatenated to form a concrete counterexample.

Guided simulation of pseudorandom vectors does not guarantee that the simulation trace progresses towards the target states. In event the simulation procedure does not find a CES with the given set of *num* vectors, backtracking is triggered. Backtracking, involves returning to a previous *milestone* state before starting a search for a new CES. The CES that starts from the selected *milestone* and all subsequent CESs are discarded. Only CESs representing a path from initial state to the *milestone* state are kept. A search for new CES begins with a fresh set of vectors.

We illustrate our backtracking scheme by an example. Suppose the simulation trace reaches a milestone state s_4, which is mapped to abstract state $a_4 \in R^4$. Starting from s_4 we begin a search for a CES. If the simulation vectors are exhausted, we need to backtrack. At first, only the current trace—the one that started from s_4—is discarded. With a new set of vectors, another attempt is made at finding a CES from s_4. If this attempt also fails, the backtrack procedure moves back one *milestone* to s_3. The current CES from s_3 to s_4 is discarded, and a new seach begins from s_3. At every successive failed attempt, we back-track by one CES. In case the backtrack procedure reaches the initial state s_0, a new initial state is chosen as a starting point and the simulation begins again.

When the number of attempts exceeds a set number, our algorithm declares the SORs spurious and requests a refinement of the abstraction. The number of vectors and number of attempts are an implementation issue and are discussed in Section 6.

5.2 Care Set States

The *care set* is a set of abstract states that lie on a potential counter example path from initial state to states labeled $\neg p$. The *care set* is used to prevent the simulation trace to deviate from a potential counterexample path. This set should consist of states (1), that are in the SORs and (2), that did not get included in the SORs because of abstraction. The example in Fig. 3 describes the motivation for the construction of the care set. It represents a state transition diagram of an abstract machine. The ACEs are represented by the synchronous onion rings R^0, R^1, R^2, R^3. Notice that abstract states E and G are not included in any SORs. Also notice that there exists a path from abstract state E back to state $C \in R^2$. However, no such path leads back from state G.

Assume that the abstract counter examples $\{A, B, C, F\}$ and $\{A, B, D, F\}$ are spurious, i.e., there is no one-to-one correspondence with any path in the concrete model. Also assume that transition $C \rightarrow F$ in the concrete model requires a *visit* to state E, which, because of abstraction, is not reflected in the SORs. In an abstract model, the invisible variables are considered as free inputs, thus creating transitions that would not exist otherwise. For algorithms based on checking a one-to-one correspondence between ACE and CCE, these types of transitions pose a difficult challenge. In such cases the abstract model has to be refined—most likely, multiple times—to achieve the correspondence.

The simulation trace can reach states in the concrete model that are mapped to E and G. It can also be seen that if the simulation trace is allowed a transition to state G, the trace will remain there and the counterexample would never be found. On the other hand the simulation trace reaching E can still reach F, the "bad state."

The care set solves the problem, by allowing the simulation to *visit* states like E, even though they are not in the SORs, while keeping the simualtion trace away from states like G. The care set is a set of states that can be reached by a series of pre-image computations starting from $\neg p$. It is computed as a fixpoint backward reachability analysis on the abstract model starting from $\neg p$ states.

Since backward reachability is performed on the abstract model, the overhead for the algorithm is limited. In our implementation, we limit the size of BDDs during backward reachability by confining our search to only reachable states in the abstract model. The reachable states were computed when performing forward reachability analysis for checking the property in the abstract model.

5.3 Refinement of the Abstraction

The refinement process selects a small set of *invisible* variables to be added to the abstract model M_a along with their associated transition relations. We use the algorithm defined in [23] for picking a good set of refinement variables and refinement minimization. The central idea in selecting refinement variables is explained in Fig. 4. Let f and g be the invisible variables that are considered for refinement. Before refinement every abstract state in S^i has a successor in S^{i+1}. If f is added to the abstract model, two out of four new states in S^i have a successor in S^{i+1} while, $(1, f)$ and $(2, \neg f)$ have no successors and are considered dead-end states. On the other hand, if g is added to the abstract model, three out of four new states in S^i have a successor in S^{i+1}. We can therefore conclude that variable f is a better candidate for refinement. Even if both f and g are added to the abstract model, the spurious ACEs may still exist: A larger set, consisting of more invisible states may be needed for refinement. The procedure selects from the invisible states those with a higher chance of removing all spurious ACEs.

The set of refinement variables thus selected is not minimal. This set is then reduced to include only those variables that are needed [5, 23]. The refinement minimization procedure is greedy, i.e., it tries to reduce the refinement set by removing each variable from the set, one at a time, and check to see if any of the ACEs re-appear. If the ACE does re-appear the variable is important in removing the ACEs and hence is not removed from the refinement set, otherwise the variable can be removed. A satisfiability (SAT) solver is used for the check. For a more detailed explaination on refining the abstraction the reader is referred to the original paper [23].

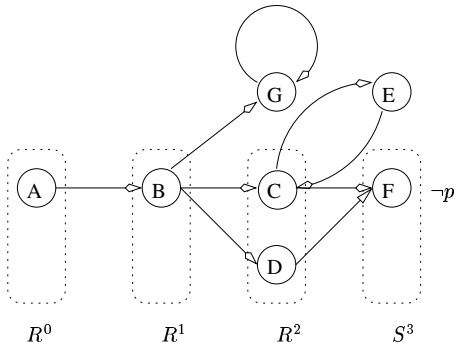
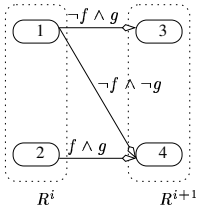
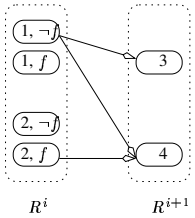


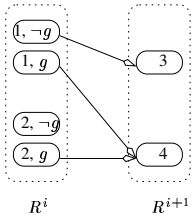
Figure 3: An example for care set.



(a) Segment of spurious abstract SORs



(b) Segment after refining on f



(c) Segment after refining on g

Figure 4: Selecting refinement variables.

```
1   simulAbs(SORs, M) {
2       s_i = getInitialState(M, S^0)
3       while (attempt ≤ Allowed) {
4           vectors = getPseudoRandomVectors(num)
5           CES = simulateWithCheck(vectors, s_i, R^{i+1}, careSet)
6           if (CES is not empty) { // if successful
7               incr i
8               CCE[i] = CES
9               if (i = L) return (TRUE, CCE)
10              s_i = s_{i+1}
11          } else {
12              incr attempt
13              s_i = backtrackSim(CCE, attempt)
14              decri } }
15      return FALSE }
```

```
1   simulateWithCheck(vectors, s, R, careSet) {
2       for i from 0 to N-1 {
3           δ = simulate(vector[i], s)
4           if (δ ∈ R) {
5               add δ to CES
6               return CES
7           } else if (δ ∈ careSet) {
8               add δ to CES
9               s = δ
10          } else discard vector[i] }
11      return FALSE }
```

Figure 2: Concretization algorithm.

6. Experimental Results

We implemented the proposed algorithm in VIS-2.1 [3, 20]. For simulating the concrete design, we modified the simulator in VIS to check the outcome after every vector. The experiments were run on a Linux workstation with a 3.0 GHz Intel Pentium 4 CPU and 2 GB of RAM. The CPU times are in seconds and memory in MB.

Table 1 compares simulAbs against the BDD-based invariant checking algorithm in VIS (CI) and GRAB, [23] also implemented in VIS. The CI algorithm consists of BDD-based forward reachability analysis with early termination. The circuits used for comparision are from both industry and the VIS verification benchmarks [20]. Many of the latter circuits are modified—by adding more processes and/or increasing the length of timeout counters—to make them more complex and lengthen their counterexamples, reflecting the complexity of industrial designs. Each experiment run is limited to 5 hours, with dynamic variable reordering enabled. The number of pseudorandom vectors[1] for each attempt, *num* in Fig. 2, is gradually increased. For our experiments we allow a maximum of four attempts with 500 vectors for the first attempt, and 1000, 10000, and 50000 for the successive attempts.

In Table 1, the first column refers to the example and the property under study. The second column lists the binary variables in the cone of influence (COI) of the property. The third column lists length of the shortest concrete counter example. For passing properties it lists 'T' followed by the length of the last abstract counterexample reported in GRAB. For failing properties, where both CI and GRAB do not finish, the minimum length of a concrete counterexample is unknown and is not reported. For each of the methods being compared we list **CPU**, the total run time, and **mem**, the total memory allocated to BDDs. In addition, for GRAB and simulAbs we list **regs**, the number of binary state variables in the last abstract model, **iter**, the number of iterations of abstraction refinement performed, and **ratio**, the ratio of variables in abstract model to COI variables. We also list for simulAbs **acex**, the length of abstract counterexample, and **trace** the length of counterexample trace found. Neither of these lengths is applicable to passing properties. When an experiment times out, the values mentioned in brackets are the last values reported by the algorithm.

Our experiments are based on a mixture of passing and failing properties with counterexamples of varying lengths. The runtimes show a significant performance improvement over the pure BDD-based invariant checking algorithm as well as over the GRAB abstraction refinement algorithm when failing properties with long counterexamples are concerned. In fact for several models, neither of the other techniques managed to finish within the allotted five hours, while simulAbs always took less than two hours. As models get more complex, the BDD-based invariant checking algorithm runs out of resources, while as the examples get deeper, GRAB generally does not finish as SAT runs out of steam. For passing properties and properties with short counterexamples, the advantages of our improved concretization approach are felt less, e.g. in circuits D24 p2 and D4 p2 the abstract counterexamples are short and within resource constraints of SAT solvers. Under such circumstances an approach like GRAB is better at proving the counterexamples spurious. It is important to keep in mind, however, that our simulation-based concretization is an initial prototype and in particular simulation speed could be substantially improved. In this light, even for passing properties and short counterexamples, our new technique hold up rather well against the competition.

In theory, unguided pseudorandom simulation could also be used to find target states. With such an approach, given a set of vectors, the concrete model is simulated starting from the initial state. The procedure ends when either a target state is reached or vectors are exhausted. The effectiveness of this approach in finding a target state is clearly based on the vectors generated. With pseudorandom generation, the probability of generating the "correct" set of vectors although non-zero, is still low and is inversely proportional to the complexity of the circuit. For circuits like, b13 p1, Daio p1, ar p1, which are not very complex to begin with, a simulation trace cannot take many divergent paths. Hence, an unguided pseudorandom simulation may be effective in finding a target state. Even in these cases our method is competitive, as our overhead is very small. In most complex circuits a simulation trace ends up taking divergent paths more often. In such a scenario, it is very likely that the unguided pseudorandom simulation will not find a target state even after multiple attempts.

7. Conclusions

We presented a new approach to checking invariant properties whose shortest counterexamples are very long traces. We proposed an efficient guided simulation method, where the simulation trace is constrained to the potential counterexamples with the help of guide points. The method includes backtracking in case the simulation trace gets stuck in a dead-end state. We show with the experimental results that our approach is significantly better at finding long

[1] vectors could be biased by environmental constraints

Table 1: Performance comparision between CI, GRAB and simulAbs.

circuit	COI	cex	CI		GRAB					simulAbs						
	regs	len	CPU	mem	CPU	mem	regs	iter	ratio	CPU	mem	regs	iter	acex	trace	ratio
D1 p1	101	9	17	30	4	22	51	8	0.50	2	23	28	0	1	9	0.28
D1 p2	101	13	3166	326	4	22	54	9	0.53	3	25	28	0	2	100	0.28
D1 p3	101	15	759	270	4	22	54	9	0.53	14	26	29	0	5	640	0.29
D1 p4	101	T(4)	1330	26	3	21	28	0	0.28	3	21	28	0	-	-	0.28
D24 p1	179	9	11055	234	7	29	25	0	0.14	8	35	25	0	9	9	0.14
D24 p2	179	T(10)	>5 h	831	7	28	25	0	0.14	7	28	25	0	-	-	0.14
D24 p4	179	T(9)	>5 h	850	8	29	29	2	0.17	2585	41	30	2	-	-	0.17
rcu1 p1	52	T(72)	38	334	115	571	52	9	1.00	380	1173	52	9	-	-	1.00
rcu1 p2	49	T(2)	57	53	1	22	4	0	0.10	1	22	4	0	-	-	0.10
rcu1 p3	44	48	8	26	39	22	20	7	0.45	126	23	7	3	8	111	0.16
ba1 p1	44	144	67	78	>5 h	348	(40)	(17)	(0.91)	2	67	8	0	10	564	0.18
ba2 p1	120	-	>5 h	1234	>5 h	1500	(120)	(44)	(1.00)	298	97	8	0	10	5126	0.07
b12 p2	110	-	>5 h	153	>5 h	7	(26)	(11)	(0.24)	434	14	10	1	14	32217	0.09
rcu2 p3	136	-	>5 h	427	>5 h	229	(58)	(22)	(0.43)	259	32	4	0	1	34750	0.03
ar p1	111	3081	207	39	>5 h	6	(17)	(10)	(0.15)	18	9	2	0	1	3107	0.02
D4 p2	230	T(24)	423	44	151	72	97	22	0.42	6019	144	97	22	-	-	0.42
Daio p1	27	12795	11	8	>5 h	60	(19)	(4)	(0.70)	25	8	9	0	191	12795	0.33
Daio p2	27	T(192)	2	22	1	21	13	1	0.48	130	24	13	1	-	-	0.48
b13 p1	34	40033	65	44	>5 h	23	(29)	(15)	(0.85)	98	24	2	0	2	40033	0.06

counterexamples. In the future, we intend to experiment with the use of decision procedures (SAT) in conjunction with simulation. Simulation provides a fast solution to distant justification problems, while SAT may be used for complex and short range justification problems. A combined approach, SAT for short segments, and simulation to fill in long gaps could be more efficient and, along with an approximate *care set*, may lead to shorter concrete counterexamples.

References

[1] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In *ICCAD*, pages 580–583, Nov. 1999.

[2] P. Bjesse and J. Kukula. Using counter example guided abstraction refinement to find complex bugs. In *DATE*, pages 156–161, Feb. 2004.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In *CAV*, pages 428–432, 1996.

[4] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversal. In *DATE*, pages 898–905, Mar. 2003.

[5] P. Chauhan et al. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *FMCAD*, pages 33–51, Nov. 2002.

[6] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comput. Syst. Sci.*, 8:117–141, 1974.

[7] E. Clarke et al. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, Jul. 2000.

[8] E. Clarke et al. SAT based abstraction-refinement using ILP and machine learning. In *CAV*, pages 265–279, Jul. 2002.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[10] D. Geist, M. Farkas, A. Landver, and Y. Lichenstein. Coverage-directed test generation using symbolic techniques. In *FMCAD*, Nov. 1996.

[11] A. Gupta et al. Learning from BDDs in SAT-based bounded model checking. In *DAC*, pages 824–829, Jun. 2003.

[12] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix. A hybrid verification approach: Getting deep into the design. In *DAC*, pages 111–116, Jun. 2002.

[13] P.-H. Ho et al. Smart simulation using collaborative formal and simulation engines. In *ICCAD*, pages 120–126, 2000.

[14] URL: http://www.cad.polito.it/tools/itc99.html.

[15] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[16] B. Li, C. Wang, and F. Somenzi. A satisfiability-based approach to abstraction refinement in model checking. *ENTCS*, 89(4), BMC 2003.

[17] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, Apr. 2003.

[18] S. Shyam and V. Bertacco. Guido: Hybrid verification by distance-guided simulation. IWLS 2005.

[19] S. Tasiran, Y. Yu, and B. Baston. Using a formal specification and a model checker to monitor direct simulation. In *DAC*, pages 356–361, Jun. 2003.

[20] URL: http://vlsi.colorado.edu/∼vis.

[21] C. Wang, R. Bloem, G. D. Hachtel, K. Ravi, and F. Somenzi. Divide and compose: SCC refinement for language emptiness. In CONCUR, pages 456–471, Aug. 2001.

[22] C. Wang, G. D. Hachtel, and F. Somenzi. Fine-grain abstraction and sequential don't cares for large scale model checking. In *ICCAD*, pages 112–118, Oct. 2004.

[23] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In *ICCAD*, pages 408–415, Nov. 2003.

[24] D. Wang et al. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *DAC*, pages 35–40, Jun. 2001.

[25] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, Jun. 1998.