Distance-Guided Hybrid Verification with GUIDO

Smitha Shyam
smithash@umich.edu

Valeria Bertacco
valeria@umich.edu

Advanced Computer Architecture Lab
University of Michigan, Ann Arbor, MI 48109

ABSTRACT

Constrained random simulation is a widespread technique used to perform functional verification on complex digital designs, because it can generate simulation vectors at a very high rate. However, the generation of high-coverage tests remains a major challenge even in light of this high performance. In this paper we present Guido, a hybrid verification software that uses formal verification techniques to guide the simulation towards a verification goal. Guido is novel in that 1) it guides the simulation by means of a distance function derived from the circuit structure, and 2) it has a trace sequence controller that monitors and controls the direction of the simulation by striking a balance between random chance and controlled hill-climbing. We present experimental results indicating that Guido can tackle complex designs, including a picoJava microprocessor, and reach a verification goal in far fewer simulation cycles than random simulation.

1. INTRODUCTION

Functional verification has become the most critical development factor for digital designs in terms of cost and time resources. The reasons for this preponderant resource demand lie in the growing complexity of digital integrated systems, paired with the shrinking of design cycle times. Available verification technologies are unable to tackle the complexity of current designs, neither in terms of coverage, nor in terms of sheer design size. While the predominant verification strategy in industry still remains centered on simulation-based approaches due to their linear scalability with design size, there has been a rising trend in recent years towards the complementary deployment of semi-formal verification techniques, which promise to provide high-coverage results at an acceptable performance cost. To support the verification engineer, the design automation industry provides a rainbow palette of tools and technologies that complement barebone logic-simulation. These tools and technologies range from testbench design languages [3, 10], to coverage evaluation tools [4, 13, 2] and constrained random stimulus generators [21, 16] to semi-formal verification tools [12, 11]. In contrast with these solutions, the objective of this paper is to propose a novel hybrid verification approach that relies heavily on the positive aspects of scalability and fast performance of random simulation while deploying small-scale formal verification techniques to guide the simulation.

1.1 An overview of Guido

Guido is a new hybrid verification solution that enhances coverage density of random simulation by guiding the simulation towards a specific verification goal. The simulator's search is directed by a "trace sequence controller" that relies on the distance function associated with each state of the design and forces the simulator to move incrementally closer to the verification goal.

Guido can be used to reach a coverage target expressed by a set of states in the design, or to disprove a general safety property, or to target a verification "checker" in the context of random simulation methodology. The random simulation-centered solution proposed by Guido differentiates itself from previous solutions in two main aspects:

- The cost function used in Guido provides a high-quality evaluation of the distance from the goal since it is derived from the portion of the design that most closely affects the verification goal.
- The trace sequence controller is based on a hill-climbing approach that employs an innovative way of balancing random steps with deterministic improvement. This technique leads Guido to exploit random simulation alone when the verification goal is easily achievable; When the goal requires stepping the simulation through a narrow passage (in terms of state transitions), the trace sequence controller resorts to a shallow SAT-based search to accomplish this transition.

From a dynamic simulation standpoint, Guido can be viewed as a technique to tunnel the exploration trace of random simulation that leads to the verification goal. This dynamic exploration is shown schematically in Figure 1, where the areas of varying grey intensity represent the partitions of the search space in layers of increasing costs, based on the evaluation of the cost function. The trace sequence controller guides the simulator through a random walk, where at each step of simulation, a range of potential next states are considered and the one that will bring us closer to the goal is then selected. Because of the coarse granularity of the cost function, it is possible that none of the potential next states are closer to the goal than the present state. In these situations, the trace sequence controller uses additional mechanisms based on random selection, backtracking, or a deterministic search, to select the next state. One by-product of the transition of current verification practices towards a methodology that makes use of semi-formal techniques, is the increased use of formal properties to describe the correct behavior of the design, usually derived from design specifications. Additionally, random simulation requires designers to embed "checkers" of some sort into the design, which are then used to detect additional bugs. Guido can target both checkers and properties to flush out many of the bugs present in the design. We envision Guido to be complementary to formal verification software. In fact, it can be deployed in the first stage of property verification to expose

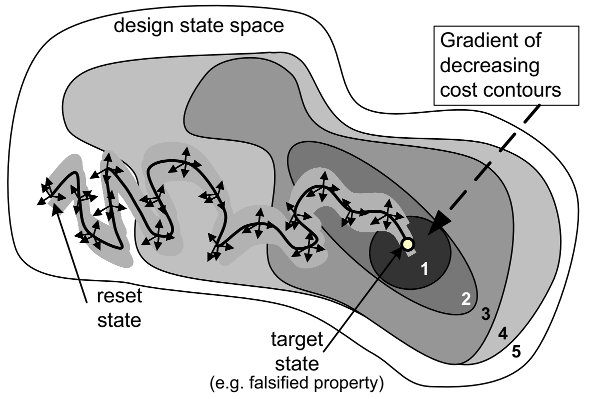
design state space
Gradient of decreasing cost contours
reset state
target state
(e.g. falsified property)
1
2
3
4
5

Figure 1: Guido's trace sequence controller guides the random simulation towards the verification target by classifying the design's state space into equidistant layers on which the simulator hill climbs.

bugs, providing scalability and performance comparable to a logic simulator. Once Guido cannot find any more bugs, heavier formal verification tools are brought in to flush out the remaining issues. When random simulation is the technique of choice, Guido can be viewed as a smart random simulator that boosts the bug-finding rate by keeping the simulator focused on the goal of invalidating checkers.

The remainder of the paper is organized as follows. The next section discusses related work. Section 3 introduces the Guido architecture, its components and the verification flow. Section 4 discusses advanced heuristics to overcome the limitations of the abstract model. Experimental results and conclusions are given in Section 5 and 6.

2. RELATED WORK

Traditional formal verification techniques provide the highest confidence in the correctness of a design by simply proving or disproving specific properties associated with its functionality. When formal verification finds that a property is not valid, it can automatically produce a *bug trace*, that is, a compact test vector that will pinpoint the problem [7, 14]. Because of the exponential nature of these techniques, pure formal verification can be applied only to small designs, with sizes up to a few hundred latches, or to properties affecting only a very small portion of a complex design.

To cope with the aforementioned limitation, within the past few years we have witnessed the emergence of a range of hybrid verification approaches. In this domain, a solution which has also become a commercial product is presented in [12], where the authors time-interleave a random simulation with a symbolic simulation to prove properties that are hidden deeply in the sequential behaviour of complex block. Additionally, a parallel reachability analysis is also ran on an abstract model of the design with the objective of ruling out unreachable configurations and, thus, pruning the space to be explored. Another example which has also been used in an industrial context, is by Aagaard *et al.*[1], where theorem-proving techniques are used to coordinate multiple model checking runs. An approach of a guided counter-example generation based on abstraction was proposed in [6]. Similar to Guido, the authors use the results of the analysis on an abstract model to guide the search. However in that solution the actual search is performed by an ATPG/BMC engine, whereas in Guido the bulk of the work is done by a logic simulator and, consequently, is much more scalable. Furthermore the abstraction strategy of Guido is modular compared to the heuristic register addition technique used in this work.

In the specific domain of target-driven logic simulation, one of the first efforts is by Yang *et al.* [20]. This work proposes to direct a random simulator to hit a goal by enlarging a verification target through backward traversal, so that it is sufficient for the simulator to hit any of the states in the enlarged target. However, the pre-image computation required in this algorithm cannot usually go past 4 or 5 time steps. Consequently it is often difficult for the simulator to reach any of the states in the enlarged target. In [15] a probabilistic guiding algorithm is presented, which assigns values to design states based on their estimated probability of leading to the target state. As values are assigned by approximate analysis, there is no apparent mechanism to escape from dead-end situations. An approach that attempts to reach a target by exploring a range of potential next states in a simulation environment was suggested by [9]. Their solution is a cost function based on the hamming distance between the current configuration reached by logic simulation and the target state. At each step of simulation a set of alternative next states is considered and the one leading to the minimum hamming distance state is chosen. The advantage is that the computation of the hamming distance can be performed very efficiently at the time of simulation. The downside of this approach, however, is that this measure is usually not a good indication of the distance to the target state and could mislead the simulator, as it is possible that two adjacent states in a state transition graph of a sequential system have very high hamming distances. Subsequent work in this direction [19] adds the use of automatically-generated "lighthouses", intermediate goals to direct the simulator toward a goal deep in the design.

3. GUIDO ARCHITECTURE

Guido consists of three main components: an abstraction engine, a distance function generator and a trace sequence controller. The trace controller makes decisions based on the analysis of the other two components and thereby controls the state exploration path of a random simulator. The distance function in Guido is based on an exact reachability analysis performed on a design's abstraction. The abstraction is generated by considering the verification goal at hand, as well as some of the design components that most closely affect it. During random simulation a set of candidate next states are explored and the one with the best potential of reaching the target is chosen based on the computed distance function. Thus, the distance associated with the abstract states guides the simulation.

From an architectural standpoint, Guido operates as a module interacting with a logic simulator and a random stimulus generator guiding the logic simulator towards a property goal. If Guido finds an input assignment that violates the property, then a bug trace is produced by simply logging the simulator input sequence. Figure 2 shows how Guido interfaces with the simulator and the random stimulus generator by selecting the direction of the next sim-

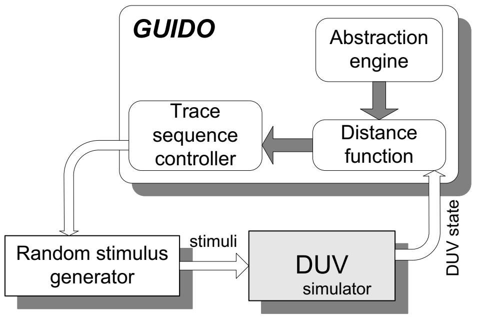

Figure 2: Guido uses a distance function to control the random stimulus generator and direct the simulator towards the goal.

ulation step among a range of possible steps suggested by the random generator. Guido also samples the current state from the simulator to evaluate the progress towards the verification goal. The three main components of Guido and their interaction are also represented in Figure 2. A detailed analysis of each of these components is presented below.

3.1 Abstraction Engine

Guido uses an abstract model of the design to compute a cost function, which is then used to guide the random simulator. The abstraction engine selects a small number of critical design modules, together with the property description, to generate a product finite state machine(FSM).

A digital design is commonly described by a hierarchical structure of modules (simpler design components) interconnected together. If we represent each module as a single FSM, it is easy to see how any subset of the design's modules can be represented by a product machine obtained by composing the FSMs of the component modules. At the limit, the complete design can be represented by an FSM obtained by computing the product of all instantiated modules. This machine represents the full design behavior, while all the intermediate products correspond to design abstractions. From a practical standpoint, the computation of the product machine is intractable for all but those abstractions involving just a few components. To overcome the computational complexity of performing a full state traversal, we select only a few "critical" design modules for our abstraction. The selection process is automated. It always includes the checker module, that is, the module describing the verification goal. The additional modules to be included into the abstraction are selected based on the following criteria:
1. Proximity to the checker module: This is based on the observation that closely interacting components are more prone to removing spurious behavior from the abstract machine, compared to the product of two non-interacting modules. Interacting modules are modules instantiated at the same level that communicate directly through I/O signals, or modules which are instantiated hierarchically from within each other. For instance, with reference to Figure 3, the modules controller and cliA are directly interacting with the checker, and, thus, they belong to our first layer of consideration for inclusion in the product machine.
2. Complexity of each module: We maintain an estimation of how complex the resulting product machine will be based on the number of memory elements that each module contains. If the inclusion of the closest layer of modules does not generate a product FSM that is deemed "too complex", based on our estimation, then we consider components at the next layer, that is modules interacting directly with the ones already included. If the inclusion of a module leads to a machine that is estimated to be too complex, we slice through it or skip it and consider another component in the same layer.

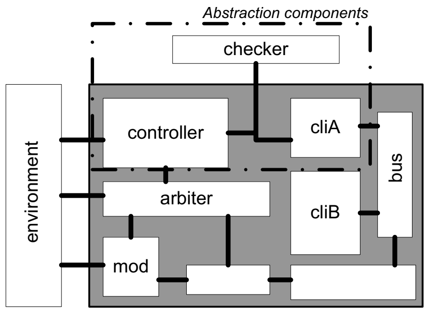

Figure 3: Guido computes the abstraction using the checker module and the components that most closely interact with it. In the example above, controller and cliA are included in the abstract FSM.

3.2 Distance Function

The distance function assigns a value to all reachable abstract states. This value measures the shortest distance of each state to the goal. The cost is stored as a set of characteristic representations of all states that have a specific distance from the goal. During simulation, each visited design configuration is mapped to one of the distance values by sampling the values of the latches that belong to the abstract FSM.

Since the characteristic equidistant functions are represented by BDDs, we strive to maintain a set of BDDs of minimal size. For this objective, we store for each distance k a BDD of minimal size in the interval:

$$[Pre(R_{k-1})/R_{k-1}, Pre(R_{k-1}) \cup R_{k-1}] \qquad (1)$$

where R_{k-1} represents the set of states at distance $k-1$ from the goal and $Pre()$ is the pre-image function. The accuracy of the cost function depends on the refinement quality of the abstraction. In general, because the cost function is computed on an abstract representation, the abstract FSM will include state transitions that do not exist in the real design. The implication is that it is possible for the simulator to reach a state at cost C from which there is no transition to a state at cost $C-1$, even if in the abstract machine such a transition was present, as indicated by the derived cost function. Section 4 discusses how to handle these dead-end situations.

3.3 Trace sequence controller

The trace sequence controller in Guido uses the cost associated with each state in the abstract machine to guide the random simulator towards the goal. At each simulation step, the trace sequence controller tries different sets of random input vectors and then selects, among all the possible "next states" obtained, the one with the shortest abstract distance. The best search is an informed search algorithm; it uses a heuristic to rank the potential "next states" based on their estimated cost [8].

During the simulation, we maintain a queue $\mathcal{Q}$ of states that we have already visited and that are good candidates as starting points for the "next state" transition. At each step of the search, we first consider the "current state" CS, that is, the state from which we are going to perform the upcoming transition. A "current state" under consideration is obtained by removing it from the queue. If its cost is 0, then we stop, having reached the goal. Otherwise we remove it from the queue $\mathcal{Q}$ and the random simulator starts generating a pre-determined number of successor states from CS. Each of these successors is evaluated by the cost function and added to the queue $\mathcal{Q}$. At this point, the best of the candidates in the queue is selected as the new "current state" and the process is repeated until a goal is found.

When a search plateaus at a certain cost, there is a non-zero probability of selecting from the queue, a candidate from the past simulation steps, which is equi-distant,but belongs to a search path that was previously abandoned. Retrieving search directions that had not been explored is the first mechanism that allows Guido to move away from dead-end simulation paths.

4. ADVANCED HEURISTICS

The best search algorithm described in Section 3.3 is often not sufficient by itself to guide the simulator to a target. Because our cost function is computed on an abstract machine, situations may arise that force the random simulation to a corner from which there is no transition to a lower-cost configuration. In some cases, the only path in the real design to a lower-cost configuration is through a higher cost configuration. With the support of an example we illustrate some of the situations that may arise.

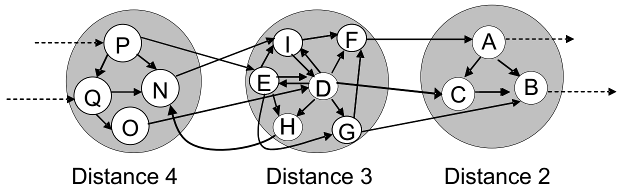

Figure 4: An example clustering of real design states based on a Guido cost function.

Example 1. Consider the diagram in Figure 4. The states labeled by a capital letter represent all the real states of a DUV. The large circles group these states in equivalence classes of equal cost based on the cost function and its related abstract FSM. The following are possible scenarios:
1. that the present simulation state is D. Among the six "next states" from D, five are other states at the same cost and only one is a state of lower cost. Moreover, all the other states at cost 3 are tightly interconnected, with only a few transitions toward lower cost states. Statistically, there is a low probability that the random simulator will generate a transition to state C. If this does not happen the simulation will continue iterating among states of cost 3 and never progress toward the goal.
2. Due to the abstraction, it is possible that configurations in the real design that are at greater distance from the goal are incorrectly assigned a low cost by the cost function. This can happen because the abstract machine may have extra transitions that are not available in the real design. With reference to Figure 4, if for instance, state D is effectively at cost 3 from the goal, then state E must be at cost 4. However, the cost function clamps them together because of the additional behavior seen by the abstract machine.
3. The set of modules in the abstract machine are not selected carefully, meaning, the abstract machine is composed by two disconnected components, then all the states in one of the components will have to have the highest cost (since the goal is not reachable from there). In the real machine that will be inaccurate. The additional modules of the real machine bring the disconnected component to a finite distance from the goal. This problem can be avoided by carefully selecting the modules for the abstraction.

The situations described in the example suggest that special techniques need to be deployed to provide sufficient liklihood of progress. This is necessary to provide a good probability of forward progress even in complex cases, when plain hill-climbing approaches are not sufficient. We are proposing two techniques that help to steer the simulation away from a plateau region. These techniques, called SimSearch and SimSAT, are presented below.

4.1 SimSearch

Guido uses a modified version of the best search algorithm described in Section 3.3, called SimSearch. The pseudo-code of SimSearch is given in Figure 5. When the cost of a candidate "next state" is the same as the "current state" CS, the candidate state is not added to the queue $\mathcal{Q}$. Simulation is continued along that "next state" for NUM_FWD steps in the hope of discovering a state with higher cost. If such a state exists within NUM_FWD steps, then it is a good candidate and it is added to the queue. The number NUM_FWD of forward steps is parameterized to allow experimenting with different trade-offs.

```
SimSearch(){
CS = initial_state
while(CS!=goal_state)
  loop Num_Successors
    curr_sample = sample_next_state(CS)
    loop NUM_FWD
     if Cost(curr_sample)≠Cost(curr_state)
             AND Is_not_in_queue(curr_sample)
         add_priority_queue(curr_sample)
         break
     else
     curr_sample=sample_next_state(curr_sample)
    end;end
  best_next_state = priority_queue.head
  CS = best_next_state
end
```

Figure 5: Pseudocode of the SimSearch algorithm.

With reference to Example 1, if the present state is E, and the randomly generated vectors lead to states D, H and I, SimSearch can discern between H, a state whose only outgoing transition is to a higher cost state, and D and I, which are also at the same cost level as all the others, but have outgoing edges leading to lower cost states. Among these alternatives, H will be discarded in favor of D or I.

4.2 SimSAT

When the simulator reaches a state with a very tight filter to a lower cost configuration, random generation is not sufficient to progress in the simulation, thereby making all our previous techniques fall short. A situation, such as this, was described in the first case of Example 1,where only one of the possible outgoing transitions from state D led to a lower cost configuration. In this type of situation we use SimSAT, a SAT checking procedure that simply checks if there is a one-step transition from the present state to a lower cost state in the real design. If the problem is satisfiable, we use the answer to perform the transition, otherwise we must backtrack to a previous state. The inputs to the SAT solver are the present state of the real design (which has a cost C), the combinational portion of the design and all the states at cost $C-1$ in the abstract model, as shown in Figure 6. To this end, the values of all the present state registers are gathered on the fly and translated into the CNF_{PS} formula. The combinational logic is also converted to CNF_{CKT}. Finally the minimal BDD stored for the cost level $C-1$ is translated and written as CNF_{C-1}.

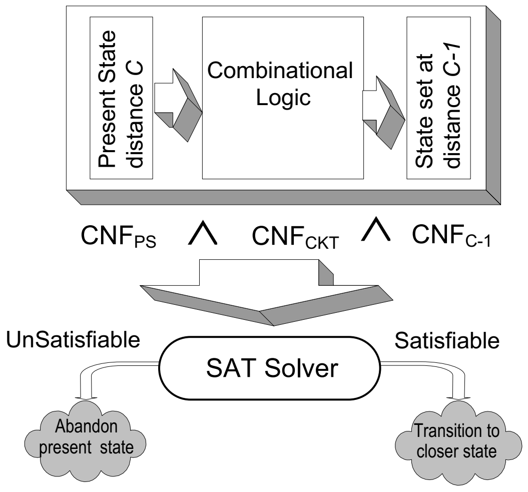

Figure 6: The SimSAT flow.

SimSAT's objective is to find if a transition to a lower cost state exists. If a solution is found, then the SAT solver returns a valid input assignment for the transition. The inputs are fed into the present state and a transition to the next state with a lower cost is made. If no solution exists, and the transition found in the abstract model was due to over-approximation, we can infer that the present state is not at cost C from the goal in the real design, and, therefore, abandon that state. Note that our SimSAT procedure uses a SAT instance that includes only one copy of the circuit's combinational logic, in contrast with SAT/BMC verification techniques [5], which require unrolling the circuit many times. The relative compactness of our SAT instances contributes to control the overall complexity of SimSAT.

A crucial aspect of incorporating SAT techniques into Guido is the decision of when to deploy SimSAT. It is a tempting option to use SimSAT to move from state to state raising the cost level at each step all the way to the goal. However, due to the complexity of SAT solvers, this solution would become prohibitively expensive after just a few simulation cycles. Hence, only when the trade-off is advantageous and the random simulation-based algorithms are unable to take the simulation towards better cost states in the given time window, is it wise to recur to the deterministic SimSAT. SimSAT not only validates the abstraction model, but also accelerates the random simulator towards the goal with small computational overhead.

5. EXPERIMENTAL RESULTS

We tested Guido on a number of publicly available testbenches, namely, a MSI cache-coherence protocol, a PCI bus from the VIS benchmark suite [18], and a picoJava processor from SUN [17]. A few relevant properties were targeted for each of these testbenches. In evaluating the quality of Guido, we compared it with a baseline constraint-based random simulation and a commercially available semi-formal verification tool.

5.1 Designs and properties for the experiments

The MSI design is a cache-coherence protocol used in a multiprocessor environment with shared memory. In this testbench, individual processors monitor the cache bus and respond accordingly to the activities of the other processors. We checked two known-false properties that were available with the benchmark suite. The second testbench is a PCI bus model interconnecting peripheral components with a core processor or memory. We changed this design to be synthesizable from the benchmark suite.

We also tested properties on the entire picoJava design processor and a few of its individual modules, specifically the ICU (Instruction Cache Unit) and a home-crafted testbench obtained by combining ICU with the Stack Management Unit (SMU) and the Bus Interface Unit(BIU). We refer to this testbench as "BSI" in the table. We verified a property on the validity of the buffer control signal for both of these testbenches. For BSI we also checked some additional properties related to the SMU unit. For all our experiments, a rough design environment was created to generate valid stimuli during simulation.

Testbench	In/Out	Latches	Logic Gates
MSI	14/15	43	1674
ICU	28/80	64	1797
BSI	84/62	108	4778
PCI	20/4	275	10078
PJava	44/76	3646	189895

Table 1: Design size and complexity.

5.2 Results

Table 2 shows the quality of the Guido exploration in terms of simulation cycles executed before reaching the verification goal. We ran the experiments on a Linux machine running at 1Ghz and equipped with 1GB of memory. The results show the comparison between the trace lengths generated by Guido, a plain constrained random simulator and an industrial semi-formal tool. Each row of the table corresponds to one design-property pair of those described in the previous section. The first column reports the simulation length of the random simulator. The second and third columns report the performance of Guido, the leftmost being the final trace length to the bug that Guido finds (column "Guido trace"), and the rightmost (column "Guido total") the total number of simulation steps executed by Guido. As described in Section 3.3, Guido explores different search directions at each step and then selects the best option. In column "Guido total" we report the total number of simulation steps which includes "exploration" steps. The fourth column reports the trace lengths found by a commercial semi-formal verification tool. All simulations were run with a wide range of random seeds in an attempt to gain a sense of the quality of these results independent of the random factor. We found that results were fairly consistent and we reported the best results for both the random simulator and Guido.

Test	Random	**Guido**		semi-	**Guido Time**	
	simul.	trace	total	formal	Abs	Run
MSI P1	444	47	239	3933	4.2s	8.9s
MSI P2	106	15	85	9	12.1s	18.3s
ICU P1	11917	81	363	2882	30.8s	60.2s
BSI P1	110855	130	650	5168	113.0s	230.0s
BSI P2	61444	33	153	2332	113.2s	179.0s
BSI P3	20	5	21	9	25.2s	30.2s
BSI P4	TO	4	12	10	25.1s	29.1s
PCI P1	TO	245	2386	2000	52.8s	105.0s
PCI P2	TO	10	50	20	28.0s	49.1s
PCI P3	TO	12	56	20	27.5s	45.4s
PJava P1	12364	14	89	9419	64.1s	98.3s
PJava P2	1800K	17	108	250209	64.1s	98.8s

Table 2: Comparison of Guido trace and simulation lengths vs. random simulation and a semi-formal verification software. For each testbench, simulation has been run multiple times with distinct random seeds. In a few cases random simulation timed-out (TO) at the five million cycles cut-off.

We have also reported the time taken by Guido for building the abstraction model and its total execution time. Guido produced shorter counter-example traces with less execution time. When compared to random simulation, run-times were nearly one-half as long for Guido. In some cases, random simulation could not produce a result within 5 million cycles (indicated by TO in the table). Trace lengths for random simulation where orders-of-magnitude longer. Morever, compared to the commercial semi-formal tool, run-times were only slightly faster for Guido but trace lengths were consistently shorter. Clearly, abstraction-guided simulation provides both run-time and trace quality advantages over previous approaches.

6. CONCLUSIONS

In this paper we presented a novel hybrid verification technique which deploys a distance function derived from an abstract model of the design under verification to guide a random simulator towards a verification goal. We discussed various issues that arise due to the difference between the coarseness of the distance function and the real design, and we showed experimental results indicating that the Guido approach is effective for a range of publicly available testbenches. We are exploring alternative mechanisms to select the components of the abstract machine, so that it can be dynamically adaptive to the quality of the random exploration that Guido is undertaking.

7. REFERENCES

[1] M. Aagaard, R. Jones, and C.-J. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proc. DAC*, pages 538–541, June 1998.

[2] S. Asaf, E. Marcus, and A. Ziv. Defining coverage views to improve functional coverage analysis. In *Proc. DAC*, pages 41–44, June 2004.

[3] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd ed., '03.

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

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems, LNCS vol.1579*, 1999.

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

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. DAC*, pages 46–51, June 1990.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[9] M. Ganai, A. Aziz, and A. Kuehlman. Enhancing simulation with bdds and atpg. In *Proc. DAC*, pages 385–390, June 1999.

[10] F. I. Haque, K. A. Khan, and J. Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

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

[12] 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 *Proc. ICCAD*, pages 120–126, Nov. 2000.

[13] R. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In *Proc. ICCAD*, pages 146–151, Nov. 1996.

[14] A. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 677–682, 1997.

[15] A. Kuehlmann, K. McMillan, and R. Brayton. Probabilistic state space search. In *Proc. ICCAD*, pp. 574-580, 1999.

[16] K. Shimizu and D. L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Proc. DAC*, pages 801–806, 2002.

[17] Sun Microsystems. PicoJava technology. *http://www.sun.com/microelectronics/communitysource/picojava.*

[18] Texas 97 benchmark suite. *http://www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97.*

[19] P. Yalagandula, V. Singhal, and A. Aziz. Automatic lighthouse generation for directed state space search. In *Proc. DATE*, pages 237–242, Mar. 2000.

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

[21] J. Yuan, K. Schultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing using bdds in simulation. In *Proc. ICCAD*, pages 584–590, Nov. 1999.