GUIDO: Hybrid Verification by Distance-Guided Simulation

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 is still a major challenge when using this technique. 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 cost function derived from the circuit structure, and 2) it has a novel, fine-grained 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 major components of a picoJava microprocessor, and reach a verification goal in many 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 of this preponderant resource demand lie in the growing complexity of digital integrated systems, paired with shrinking design cycle times. Available verification technologies are unable to tackle the complexity of current designs either in terms of confidence of correctness, commonly expressed as coverage, or in terms of sheer design size. While the predominant verification strategy in the industry still remains centered on simulation-based approaches because of 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 complements barebone logic-simulation: these range from testbench design languages [3, 9], to coverage evaluation tools [4, 12, 2] and to constrained random stimulus generators [21, 15]. Constraint-based random simulation, in particular, has the prominent advantage of relieving the engineer from the development of specific testbenches, and relying simply on the specification of a set of interface constraints (called the "design environment") to produce a voluminous amount of valid input vectors in a short time span. However, the challenge remains of producing input stimuli that are not only valid, but can also produce high coverage simulation traces.

The semi-formal verification landscape is populated by tools that draw techniques from both the formal verification and the simulation-based domains and use them in a collaborative manner to achieve good coverage quality on complex designs. Within this space, one group of solutions entails using a wide range of verification tools against the verification task at hand in a time-interleaved fashion (simulation may be one of them or not) [11, 10, 17]. Another family of solutions attempts to apply one specific formal verification technique to the problem and provides smart fall-back mechanisms to overcome the limitations of that technique [20, 19]. 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 it deploys small-scale formal verification techniques to guide the simulation.

1.1 Contribution

Guido is a new hybrid verification solution that enhances the coverage density of random simulation by guiding the simulation towards a specific verification goal. The workhorse of state exploration in Guido is provided by random simulation. However, in contrast with plain random simulation, the simulator's search is directed by a "trace sequence controller" that forces the simulator to get incrementally closer to the verification goal. To this end, the trace sequence controller relies on a hill-climbing algorithm and a cost function associated to each state of the design. The controller complements its baseline hill-climbing technique with mechanisms that exploit the random nature of the simulation when beneficial, and bypass it by inserting single-step deterministic improvements, when the search is not fruitful. A verification goal in Guido is specified by monitoring the value of a design signal; this signal can encode a property or checker to be falsified, a coverage goal, or a state/set of states that the designer wants to target. 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 circuit structure and, in particular, 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 a novel way of balancing random steps with deterministic improvement. This technique leads Guido to exploit random simulation alone to reach a verification goal, when this goal is easily achievable. When the goal requires stepping the simulation through a narrow passage (in terms of state transitions), the trace sequence controller deploys deterministic formal techniques to accomplish the single narrow step towards the goal.

1.2 An overview of Guido

From a dynamic simulation standpoint, Guido can be viewed as a technique to tunnel the exploration trace of random simulation in a narrow path leading to the verification goal. Figure 1 shows schematically this dynamic exploration, where the areas at different 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 brings us closer to the goal is selected. Because of the coarse granularity of the cost function, it is possible that none of the potential next states is closer to the goal than the present state. In these situations, the trace sequence controller uses additional mechanisms based on random selection, backtracking, or deterministic search, to select the next state. Note that, as suggested by the diagram, once the simulation trace has climbed up to a given cost layer, Guido does not allow the simulator to perform a transition back to a layer of lower cost.

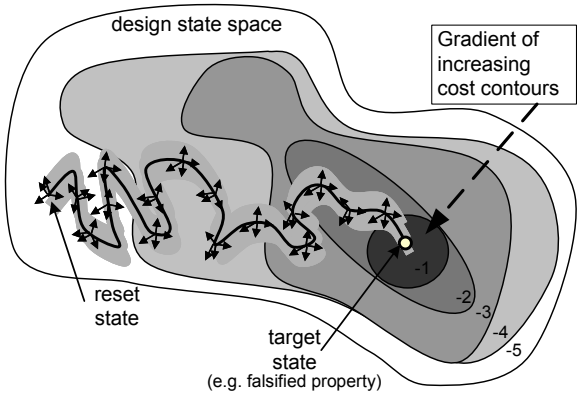

Figure 1: Guido's trace sequence controller guides the random simulation towards the verification target by classifying the design's state space into equidistant layers on which the simulator hill-climbs.

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 in the design, which are used to detect if a bug is found. Guido can target both checkers and properties to flush out many of the bugs present in the design by invalidating them. We envision Guido as complementary to formal verification software. In fact, it can be deployed in the first stage of property verification and expose bugs with 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 the verification methodology is based on random simulation, Guido can be seen as a smart random simulator that boosts the efficacy of this methodology by keeping the simulation on track towards the goal of invalidating checkers.

The remainder of the paper is organized as follows. The next section discusses related work and reviews background concepts. Section 3 introduces the Guido architecture and its verification flow. The abstraction, guiding cost function and trace sequence controller are also presented in this section. Section 4 discusses heuristics and techniques to avoid dead-ends during the simulation. Experimental results and conclusions are given in Section 5 and 6.

2. RELATED WORK AND BACKGROUND

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 a formal verification technology 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 [6, 13]. Because of the exponential complexity nature of these techniques, pure formal verification can be applied only to small designs, with size up to a few hundreds latches, or to properties affecting only a very small portion of a complex design.

To cope with this limitation, the past few years have witnessed the emergence of a range of hybrid verification approaches, both in the academic environment and in the design-automation industry. The common thread among these efforts is the attempt to deploy a diverse set of formal verification and simulation-based techniques to boost the size of the designs that can be formally verified, and to obtain at least some formal verification results, that is, property proofs, at a low additional performance cost. In this domain, an example which has also become a commercial product is [11], where the authors time-interleave random simulation with symbolic simulation to gain the ability to prove properties that are hidden in a complex design block far away from the initial state configuration. Additionally, reachability analysis is run in parallel on an abstract design model with the objective of ruling out unreachable configurations and thus prune the design 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. Other solutions that have combined a range of formal verification techniques are also in [10, 17].

In the specific domain of target-driven logic simulation, one of the first efforts is by Yang *et al.* [20]. This work proposes to help a random simulator 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,

with the consequence that it is often difficult for the simulator to reach any of the states in the enlarged target. In [14] 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 the dead-end states. An approach that attempts to reach a target by exploring a range of potential next states in a simulation environment was suggested by Ganai *et al.* in [8]. Their solution 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 that leads to the minimum hamming distance state is chosen. The advantage is that the computation of the hamming distance can be performed very efficiently at simulation time. 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, since it is possible and common that two adjacent states in a state transition graph of a sequential system have actually very high hamming distances. Subsequent work in this direction [19] adds the use of automatically-generated "lighthouses", intermediate goals that serve the purpose of directing the simulator toward a goal deep in the design.

2.1 Composing design modules

The abstraction engine of Guido performs two main tasks: it selects the components of the design that will be used for the abstract finite state machine and it generates the product machine. The selection algorithm is discussed in Section 3.1, here we discuss how the composition of the selected components is performed. We model each design module, or component, as a finite state machine, formally defined below:

DEFINITION 1. *A finite state machine (FSM) $\mathcal{M}$ is a 6-tuple $\mathcal{M}(I, O, S, \delta, \lambda, S_0)$ where I is the n length input vector, O is the k length output vector and S is the m length vector that represents the set of states of the machine $\mathcal{M}$. $S_0 \in S$ is the set of initial states and $\delta : S \times I \rightarrow S$ is the next state transition function. Finally, $\lambda : S \rightarrow O$ is the output function.*

In the following we refer to finite state machines and design modules interchangeably.

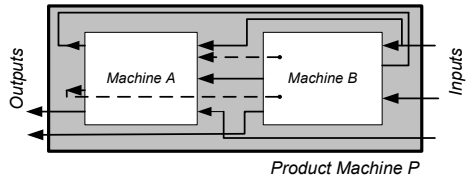

Figure 2: Composing two design modules in the abstract machine: the matching input/outputs are connected inside the product machine P, while the other I/Os become I/Os of P. The dashed lines consider the case where one module is instantiated inside the other.

The composition of multiple components is performed by generating iteratively the product of pairs of FSMs. The product machine is reported schematically in Figure 2 where we indicate all the possible ways in which inputs/outputs from one machine may be connected to inputs/outputs of the other. The dashed lines consider the case where one machine is instantiated inside the other. In composing two FSM $\mathcal{A}$ and $\mathcal{B}$, we first identify all the signals that are inputs and outputs of the resulting product machine $\mathcal{P}$, by analyzing the module descriptions and instantiations. We then derive the set of state elements of $\mathcal{P}$, by simply composing the set of state elements of the two source FSM. Finally, we compute all the components of the transition function $\delta_{\mathcal{P}}$ of $\mathcal{P}$ by composing the elements of $\delta_{\mathcal{A}}$ and $\delta_{\mathcal{B}}$ that are connected and existentially quantifying the $I_{\mathcal{A}}$ and $I_{\mathcal{B}}$ variables that are consequently eliminated.

2.2 Images and pre-images

The cost function in Guido relies on a backward reachability analysis over an abstraction of the design. This section reviews the basic terminology that we use in referring to this computation. The exploration of the set of states that can be reached from an initial state in a FSM is done through the *image* operation. The image operation computes the set of states that can be reached in one simulation step from a given starting set. It's inverse is called a *pre-image* and it computes the set of states that are one step away from a target state set. Let X be a subset of the state set S. The image and pre-image of X are formally defined as follows:

DEFINITION 2. *Given a subset X of S for a FSM $\mathcal{M}$, the image of X is defined as $Img(X, \mathcal{M}) = \{y | \exists x, i \in X, I, y = \delta(x, i)\}$.*

DEFINITION 3. *Given a subset X of S for a FSM $\mathcal{M}$, the pre-image of X is defined as $Pre(X, \mathcal{M}) = \{y | \exists x, i \in X, I, x = \delta(y, i)\}$.*

2.3 Backward traversal

In order to rank the states of the abstract machine based on the distance from the goal state, Guido performs a backward FSM traversal. The general algorithm for backward traversal is given in Figure 3. From a target set of states that describe the verification goal, the previous states are iteratively enumerated until a fix point is obtained. With reference to the figure, R_i represents the set of states that can be reached in up to i steps of traversal, while `previous_states` represent the set of states reached in the current traversal step, often called frontier set in the literature.

```
1  BackwardTraversal(){
2   R_i = goal_state;
3    do {
4        R_i=R_{i+1};
5        previous_states = Pre(δ,R_i);
6        R_{i+1} = previous_states ⋃ R_i;
7    } while(R_i ≠ R_{i+1})
8  }
```

Figure 3: Backward traversal algorithm.

All the functions and operations involved in the backward traversal algorithm are usually performed using BDD representations. Because of the BDD limitations in representing very complex functions, backward traversal computations

are typically limited in application to small FSM. The complexity limitation applies to all the functions involved: from the reached sets, to the computation of the pre-image and transition functions. In Guido, we apply a design-based abstraction to avoid reaching this complexity limit by selecting only a few critical modules of the design.

3. GUIDO ARCHITECTURE

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

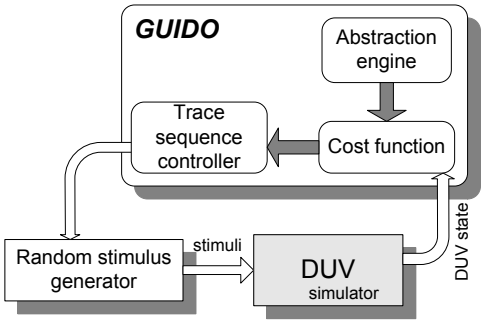

Figure 4: Guido uses a cost function to control the random stimulus generator and direct the simulator towards the goal.

Guido can be used to verify 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 a random simulation methodology.

From an architectural standpoint, Guido operates as a module interacting with a logic simulator and random stimulus generator and guides 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 4 shows how Guido interfaces to the simulator and random stimulus generator by selecting the direction of the next simulation 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 is also represented in the figure. A detailed analysis of each of these components is presented in the following sections.

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 and generates a product finite state machine. The cost function module performs a backward traversal on this abstract machine and computes a cost value for all reachable states. This value measures, for each abstract state, its shortest distance to the goal. This is equivalent to the distance to any of the states falsifying the property. In particular, all the states that falsify the property have cost 0, the states at distance one from the goal have cost -1, and so on. The abstract machine used in the cost function generation is created by considering the product of the finite state machine representing the property to be falsified and other "critical" design modules, also modeled as FSM.

A digital design is commonly described by a hierarchial 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 FSM of each instantiated module. This last machine represents the full design behavior, while all the other intermediate products correspond to design abstractions. From a practical standpoint, the computation of the product machine is intractable for all but the smaller designs, or an abstraction involving 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 includes always the checker module, that is, the module describing the verification goal. The additional modules included are selected based on how they interact with this checker module. We expand this set of modules based on the estimated complexity of the product machine, each module is included based on how close is its interaction with the target module. Practically, the selection is based on two criteria: 1) close interaction with the checker module, and 2) complexity of each module considered as a candidate for the abstraction machine.

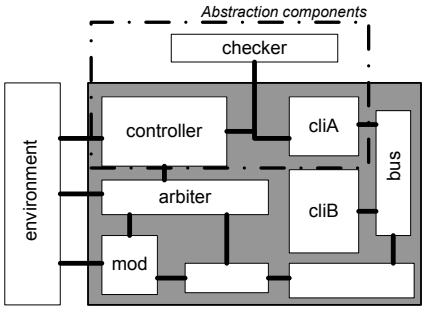

Figure 5: Guido computes the abstraction using the checker module and the components that most closely interact with it.

The selection of which modules to include in the generation of the abstract machine is based on the observation that closely interacting components are more prone to remove spurious behavior from the abstract machine compared to the product of two non-interacting modules. Inter-

acting modules are modules instantiated at the same level that communicate directly through I/O signals, and modules which are instantiated hierarchically within each other. For instance, with reference to Figure 5, 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. We maintain an estimation of how complex the resulting product machine will be based on the number of memory element that it includes. If the inclusion of the closest layer of modules does not generate a product machine that is deemed "too complex", based on our estimation, then we consider components at the next layer, that is, components that interact directly with the ones already included. If the inclusion of a module leads to a product machine size that is estimated to make the computation too complex, we skip it and consider an other component in the same layer.

3.2 Cost Function

The cost function assigns a cost 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 design configuration is mapped to one of the distance values by abstracting the real state to the abstract state in the cost function and then finding the distance set to which the abstract state belongs.

Since the characteristic equi-distant functions are represented by BDDs, we strive for maintaining a set of BDDs of minimal size. For this objective, we store for each distance k a BDD with minimal size in the interval:

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

where R_{k-1} represents the set of states at distance $k-1$ from the goal. 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 are not existing in the real design. The implication is that it is possible that the simulator reaches 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 highest cost, that is, the one closest to the goal. The best search is an informed search algorithm; it uses a heuristic to rank the potential next states based on their estimated cost [7].

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 since we have 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.

Notice that by using this process, when the search plateaus at a certain cost, there is a non-zero probability to select from the queue a candidate from the past simulation steps that has equal cost but belongs to a search path that was previously abandoned. This mechanism of retrieving search directions that had not been previously explored is the first mechanism that allows Guido to move away from a dead-end simulation path.

4. ESCAPING FROM DEAD-END STATES

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

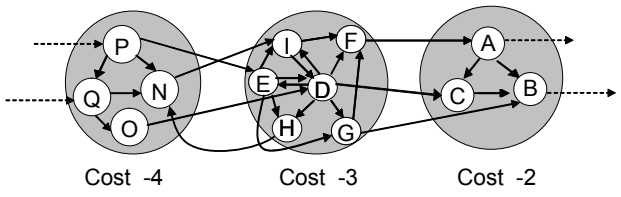

Figure 6: An example clustering of real design states based on a Guido cost function.

Example 1. Consider the diagram of Figure 6. 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 followings are possible scenarios:

1. Assume that the present simulation state is D. Among the six next states from D, five are to other states at the same cost and only one is to a state of higher cost. Moreover, all the other states at cost -3 are tightly interconnected, with only few transition towards lower cost states. Statistically , there is a low probability that the random simulator will generate a transition to the state C at a higher cost. If this does not happen the simulation will just keep iterating among states of cost -3 and never progress toward the goal.

2. Due to the abstraction used in computing the cost function, it is possible that configurations that in the real design are at greater distance from the goal, are assigned a high cost by the cost function. This can happen because the abstract machine used to generate the cost function may have extra state transitions

that are not available in the real design. With reference to the Figure, if, for instance state D is effectively at distance 3 from the goal, then state E must be at distance 4. However, the cost function clamped them both at the same distance because of the additional behavior seen by the abstract machine.

3. Finally, if the set of modules in the abstract machine is not selected carefully, that is, the abstract machine is composed by two disconnected components, the result is that all the states in one of the components are going to have the lowest cost (since the goal is not reachable from there). However, in the real machine that would be inaccurate since the additional modules of the real machine bring the disconnected component at a finite distance from the goal. This problem can be avoided just by carefully selecting the modules in the abstract machine.

The situations described in the example suggests that special techniques needs to be deployed to provide sufficient chance of progress. This is necessary to provide a good probability of forward progress even in these complex cases, when plain hill-climbing approaches are not sufficient. We propose here two techniques that help in steering 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 pseudocode of SimSearch is given in Figure 7. 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 T steps in the hope to discover a state with higher cost. If such a state exists within T steps, then it is a good candidate and it is added to the queue. The number T of forward steps is parameterized to allow experimenting with different trade-offs.

```
1  SimSearch(){
2  CS = initial_state
3  repeat
4    loop Num_Successors
5      curr_sample = sample_next_state(CS)
6      loop T steps
7        if Cost(curr_sample)≠Cost(curr_state)
                AND Is_not_in_queue(curr_sample)
8         add_priority_queue(curr_sample)
9         break
10       else
11       curr_sample=sample_next_state(curr_sample)
12     end
13   end
14   best_next_state = priority_queue.head
15   CS = best_next_state
16 end
```

Figure 7: Pseudocode of the SimSearch algorithm. From each current state, Num_Successors forward paths are searched for a maximum of T steps or until a higher cost state is found.

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 lower cost state, and D and I, which are also states at the same cost level as all the others involved, but at least have outgoing edges leading to the possibility of transitioning to higher cost states. Among these alternatives, H will be discarded in favor of D or I.

4.2 SimSAT

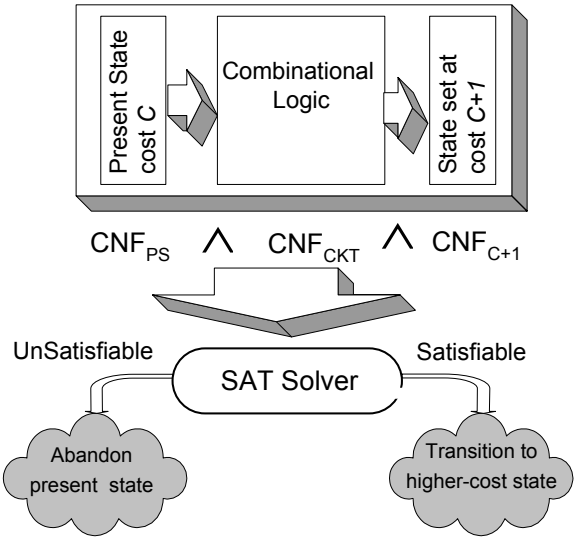

Figure 8: The SimSAT flow. The CNFs of the present state and the combinational logic of the real design and the abstract better cost states is fed to a SAT solver to ascertain if a transition to a higher cost state from the present state exists.

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, and usually all our previous techniques fail. 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 was leading to a lower cost configuration. In this type of situations we use SimSAT, a SAT checking procedure that simply checks if there is a 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 transition relation of the design and all the states at cost $C+1$ in the abstract model, as shown in Figure 8. To this end, the values of all the present state registers are gathered on the fly and translated into the CNF_{PS} formula. The transition relation 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}.

SimSAT's objective is to find if a transition to a higher 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 higher cost is made. If no solution exists

and the transition found in the abstract model was due to the over-approximation of the design behavior, we can infer that the present state is not at cost C from the goal in the real design, and 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 just after 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 lower cost states in the given time window, it is wise to recur to the deterministic SimSAT. SimSAT not only validates the abstraction model, but also accelerates the random simulator towards the goal with minimal computational overhead.

Referring to case 1 in Example 1, if the random simulator is held at cost -3 after a predetermined number of simulation steps, then SimSAT would intervene and guide the simulator to state C at cost -2. On the other hand, for case 2, when the present state is E, a call to SimSAT would establish that no higher cost transition is feasible and the current state E would be abandoned.

5. EXPERIMENTAL RESULTS

We tested Guido on a number of publicly available testbenches, namely, a cache coherence protocol and a PCI bus from the VIS benchmark suite [18], and vaious modules from the picoJava processor from SUN [16]. 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 with a commercially available semi-formal verification software.

The next sections provide relevant information on our tests setup and discuss the results of the experiments.

5.1 Designs and properties for the experiments

All the properties that we considered for the Guido evaluation were known to be false. 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. The protocol under verification has 43 latches and 1674 combinational gates. We checked two known-false properties that were available with the benchmark suite. The second testbench is a PCI bus model used as interconnect between peripheral components and a core processor or memory. The PCI module has 275 latches. We checked three properties for this design, also derived from the available properties in benchmark suite.

The remaining tests are derived from the picoJava design: specifically, 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. All these properties are found to be false, unless they are checked in the context of the complete design. For all our experiments a rough design environment was created to generated valid stimuli during simulation. The same environment was shared by all the systems we tested.

5.2 Results

Table 1 shows the quality of the Guido exploration in terms of simulation cycles executed before reaching the verification goal. We ran the experiments on a Sun Blade 1500 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 is the final trace length to the bug that Guido finds (column "Guido trace") and the rightmost (column "Guido total") is the total number of simulation steps executed by Guido. In fact, as described in Section 1.2, 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 that includes these "exploration" steps. All simulations were run with a wide range of random seeds in an attempt to gain a sense of the quality of these results that is 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. The fourth column reports the trace lengths found by a commercial semi-formal verification tool. Finally, the last column reports the ideal minimum-length trace found by a purely-formal verification software (SAT-BMC in VIS), whenever it was able to complete its execution.

Test		Random simul.	**Guido** trace	**Guido** total	semi-formal	min. leng.
MSI	P1	44	5	37	3933	1
MSI	P2	106	10	57	9	8
ICU	P1	11917	140	953	2882	2
BSI	P1	110855	1532	5216	5168	2
BSI	P2	61444	65	345	2332	3
BSI	P3	20	5	21	9	2
BSI	P4	TO	4	12	10	4
PCI	P1	TO	245	2386	N/A	N/A
PCI	P2	TO	10	50	N/A	N/A
PCI	P3	TO	245	2386	N/A	N/A

Table 1: Comparison of Guido trace length and simulation length vs. random simulation and a semi-formal commercial 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.

Table 2 shows the impact of incorporating the SimSAT procedure into Guido. For example, for the ICU P1 prop-

erty, SimSAT reduces the trace length from 140 to 81 clock cycles. Calls to SimSAT are triggered only when the efforts of the trace sequence controller do not reach the goal within 60 steps. For instance, SimSAT is not used for the MSI testbench, since the random generator guided with by cost function alone can reach the goal in very few steps.

Test		Guido+SimSAT	
		trace	total
ICU	P1	81	363
BSI	P1	130	650
BSI	P2	33	153

Table 2: Impact of SimSAT on trace and simulation length generated by Guido.

We also analyzed Guido in terms of total execution time. We found that, on average and for the cases where the random simulator completed the search, it took about twice as long as Guido to falsify a property. The cases where the random simulator timed out at five million cycles of simulation took one hour to run. Correspondingly, Guido spent approximately 120 seconds to falsify the trace. The commercial semi-formal verification tool is always faster than Guido, averaging at 60 seconds of execution time, for the tests that we could run through it. With reference to this analysis, we note that, even if slower, Guido often finds shorter traces than the commercial tool. We also would like to point out that we compared an industrial quality software development with our preliminary experimental software.

6. CONCLUSIONS

In this paper we presented a novel hybrid verification technique which deploys a cost 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 because of the model difference between the cost function and the real design, and we showed preliminary experimental results indicating that this approach is effective for a range of publicly available testbenches. We are exploring alternative mechanisms to select the components of the abstract machine to generate the cost function, so that it is dynamically adaptive to the quality of the random exploration that Guido is undertaking.

Acknowledgments

The authors would like to thank Karem Sakallah for his valuable comments during our initial efforts in this work.

7. REFERENCES

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

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

[3] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[4] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, 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] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *DAC, Proceedings of Design Automation Conference*, pages 46–51, June 1990.

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

[8] M. Ganai, A. Aziz, and A. Kuehlman. Enhancing simulation with bdds and atpg. In *DAC, Proceedings of Design Automation Conference*, pages 385–390, June 1999.

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

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

[11] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 120–126, Nov. 2000.

[12] R. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 146–151, Nov. 1996.

[13] 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.

[14] A. Kuehlmann, K. McMillan, and R. Brayton. Probabilistic state space search. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 574–580, Nov. 1999.

[15] K. Shimizu and D. L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *DAC, Proceedings of Design Automation Conference*, pages 801–806, 2002.

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

[17] S. Tasiran, Y. Yu, and B. Batson. Using a formal specification and a model checker to monitor and direct simulation. In *DAC, Proceedings of Design Automation Conference*, pages 356–361, June 2003.

[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 *DATE, Design, Automation and Test in Europe Conference*, pages 237–242, Mar. 2000.

[20] C. H. Yang and D. Dill. Validation with guided search of the state space. In *DAC, Proceedings of Design Automation Conference*, 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 *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 584–590, Nov. 1999.