An Effective Guidance Strategy for Abstraction-Guided Simulation *

Flavio M. De Paula Alan J. Hu
Department of Computer Science, University of British Columbia, {depaulfm, ajh}@cs.ubc.ca

ABSTRACT

Despite major advances in formal verification, simulation continues to be the dominant workhorse for functional verification. Abstraction-guided simulation has long been a promising framework for leveraging the power of formal techniques to help simulation reach difficult target states (assertion violations or coverage targets): model checking a smaller, abstracted version of the design avoids complexity blow-up, yet computes approximate distances from any state of the actual design to the target; these approximate distances are used during random simulation to guide the simulator. Unfortunately, the performance of previous work has been unreliable — sometimes great, sometimes poor.

The problem is the guidance strategy. Because the abstract distances are approximate, a greedy strategy will get stuck in local optima. Previous works expanded the search horizon to try to avoid dead-ends. We explore such heuristics and find that they tend to perform poorly, adding too much search overhead for limited ability to escape dead-ends. Based on these experiments, we propose a new guidance strategy, which pursues a more global search and is better able to avoid getting stuck. Experiments show that our new guidance strategy is highly effective in most cases that are hard for random simulation and beyond the capacity of formal verification.

Categories and Subject Descriptors: B.6.3 [Logic Design]: Design Aids – Verification

General Terms: Verification, Algorithms

Keywords: simulation, RTL, model checking, abstraction

1. INTRODUCTION AND BACKGROUND

Formal verification continues to progress, through advances such as model checking [5, 17], symbolic model checking [4], bounded model checking [1, 2], and counterexample-guided abstraction refinement [13], which have greatly expanded the capacity of formal verification tools. Conventional simulation, however, remains the dominant workhorse for industrial hardware validation. Simulation provides unparalleled capacity for handling design size and complexity, but (or because) it performs no analysis of the state space of the design. Abstraction and model checking, on the other hand, derive considerable information exploring the entire state space, but (therefore) suffer from capacity limitations.

Researchers have long tried to combine the completeness of formal techniques with the speed, capacity, and ease-of-use of simulation. The earliest ideas involved augmenting simulation by small amounts of bounded formal (exhaustive) search on the concrete design itself. For example, we could compute a few pre-images of the target states, in order to create a larger set of target states, which might be easier to hit via random simulation [20, 22, 21]. Dually, we could exhaustively explore a small neighborhood around heuristically promising states encountered during simulation, an approach that has demonstrated usefulness in practice (e.g., "Ketchum" [11], the SCH engine in SixthSense [15], or methodologically [9]). Because these methods perform formal analysis on the full design under verification, the extent of formal analysis must be limited, to prevent complexity blow-up.

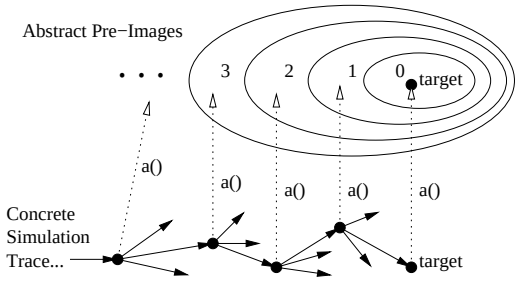

Figure 1: Abstraction-Guided Simulation. The actual (concrete) design is simplified into an abstract design. An abstraction function $a()$ maps concrete states into abstract states. Formal verification of the abstract design partitions the abstract states into "onion rings" — all states in the ith onion ring can reach the abstract target in i cycles. During simulation, the testbench looks for inputs that move the (concrete) simulation state to one that maps into the next closer onion ring. Because the abstraction is conservative, all concrete paths to the target have corresponding abstract paths, but not vice-versa.

A complementary approach, which has also been researched extensively over the past decade, is to formally analyze a simplified abstract version of the design. The abstract design can be simplified enough to be amenable to full formal verification, and the analysis gives a "big picture" global view of the structure of the state space, which can direct the simulator in promising directions. For example, the earliest works along these lines [12, 22] abstracted away all datapath, and then directed the simulator to make (concrete) state changes to cover all (abstract) control state transitions. Subsequent work tried to cover more general abstractions [8]. Most of the research on abstraction-guided simulation, however, has used abstract pre-images from abstract target states as an approximate distance metric, to help the simulator "home-in" on concrete target states (e.g., [21, 14, 10, 7, 19, 16, 6, 18]). The target states could be an assertion violation, or a difficult-to-reach coverage target. Fig. 1 sketches this approach.

Although abstraction-guided simulation is intuitively appealing, it has yet to deliver on its promise. Results have been inconsistent — sometimes it works amazingly well, but often it doesn't. The core problem is dead-end states, as shown in Fig. 2. Worse, the effect of dead-end states propagates through the abstract pre-images, because a shorter, but false, path through a dead-end state

*This work was supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

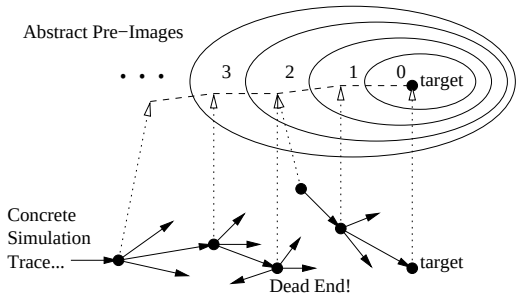

Figure 2: Dead-End States. Because two different concrete states may map to the same abstract state (e.g., in onion ring 2, above), an abstract trace might not correspond to any concrete trace. If so, the abstraction will lead the simulator to a dead-end. Because of the information loss in the abstraction, the simulator doesn't know whether it is headed for a dead-end and should backtrack, or whether it must search harder to make forward progress.

will pull the simulator away from longer, real paths. Unfortunately, the simulator doesn't know whether it is headed for a dead-end, or whether it must search harder to make progress. Some researchers resort to full-formal techniques (e.g., explicit model checking [7], SAT [19], or abstraction refinement [16]) as a back-up tactic to ensure the simulation makes progress. Nevertheless, the fundamental research issue is good guidance strategies for the simulator, in the presence of possibly erroneous distance information from the abstract pre-images.

This paper directly addresses the problem of good guidance strategies. Because the abstract distances are approximate, a greedy strategy will get stuck in local optima. Previous works expanded the search horizon to try to avoid dead-ends. In this paper, we first explore such heuristics and find that they tend to perform poorly, adding too much search overhead for limited ability to escape dead-ends. Based on these experiments, we propose a new guidance strategy, that pursues a more global search and is better able to avoid getting stuck. Experimental results show that our new guidance strategy is highly effective in most cases that are hard for random simulation and beyond the capacity of formal verification.

2. RESEARCH METHODOLOGY

Because this research is an exploration of heuristics, good research methodology is paramount to avoid misleading results.

We make the following assumptions about the verification flow: (1) The target states are specified logically, as would be the case for an assertion violation or an unreached coverage target. (2) Random simulation is used to hit the easy targets quickly. (3) Formal verification is applied to any target that isn't hit via random simulation, as formal is the only way to prove that a target is *not* reachable. (4) Accordingly, abstraction-guided simulation is relevant only when simulation fails to reach the target, and formal verification fails to verify unreachability or generate a concrete trace to the target.

We conducted our research using the EverLost platform [6]. We used VCEGAR [13] version 0.9 and VIS [3] version 2.1 as our formal engines; these are the only free formal tools we are aware of that can handle substantial Verilog designs. We used Synopsys VCS version 7.2 as our simulator. EverLost automatically generates a testbench that controls VCS via DirectC.

We use real, publicly available benchmarks for all of our experiments. In particular, our experiments were conducted on design units from the USB 2.0 Function Core, the USB 1.1 PHY, and the Ethernet MAC 10/100 Mbps designs from www.opencores.org. VCEGAR and VIS were unable to handle the original Verilog of these test cases, so we modified them by hand, then verified equivalence to the original using Synopsys Formality version V-2004.06-SP1. All data, modified Verilog models, and the EverLost platform are available at http://www.cs.ubc.ca/∼depaulfm/EverLost.

Runtime results for random simulation have enormous variance, so statistical analysis is needed to draw valid conclusions. Resource limitations prevented running all experiments with the same number of trials, so we report the number of trials for each experiment. (Indeed, we could not even complete all of our experiments on the same speed processors, but the processor for each benchmark is reported, and we always compare a single benchmark across different heuristics on the same speed processor.) We report the sample mean runtime for each experiment, as well as a 95% confidence interval for the true mean, based on Student's t-distribution. We also report minimum and maximum data points for each experiment.

With tunable heuristics, there is always the danger of over-tuning to a specific benchmark, akin to over-fitting to data in statistics. We prevent this problem using standard experimental design: for our proposed new guidance strategy, we tune using one design and set of properties (the training set), then evaluate using a different version of the design and different properties (the test set), with no changes whatsoever to the heuristic. As a further test, we apply the identical heuristic to a completely different design, again with no further tuning. These results are reported in Section 5.

3. LOCAL SEARCH EXPERIMENTS

As mentioned above, current abstraction-guided simulation heuristics typically search the local neighborhood of a concrete state, trying to find a successor that maps to the next closer onion ring. For example, the original EverLost heuristic was, from a given concrete simulation state, to simulate b different random traces, each d cycles long, and then move to the "best" state on those traces, according to the abstract onion rings. We explore this heuristic space, first varying the breadth b, and then the depth d.

For these experiments, we used as our design under validation (DUV) two design units from the USB Function Core and USB PHY designs. Because we needed a large number of experiments, we focused on two small units from these designs, but as often arises in practice, we examined the integration of two separate designs. In particular, the DUV is the USB Packet Disassembly Unit (usbf_pd) from the USB Function Core integrated with the USB Receive Unit (rx_phy) from the USB PHY. The DUV contained 121 latches, 4 inputs and 56 outputs. We manually abstracted the DUV using structural abstraction: the abstract design was the usbf_pd unit alone, which had 74 latches, 11 inputs, and 42 outputs.

We selected 4 properties to try on the DUV, relating to receiving tokens and/or data with proper acknowledgment:

p1 Can usbf_pd receive a token?

p2 Does usbf_pd acknowledge receiving data?

p3 Can usbf_pd receive a valid token or pid acknowledgment?

p4 Does usbf_pd acknowledge receiving a valid token?

We used VIS to model check the abstract design, generating 5 abstract onion rings for p1–p3, and 6 for p4.

Keep in mind that guided simulation imposes a substantial performance penalty over conventional simulation. Any guidance mechanism needs to know the design state, so the guided simulator must make additional function calls and memory accesses on each simulation cycle. What's worse is that making the simulation state visible at each cycle can disable some compiler optimizations, imposing a substantial slowdown.[1] Therefore, abstraction-guided

[1]Thanks to Valeria Bertacco for explaining this source of overhead.

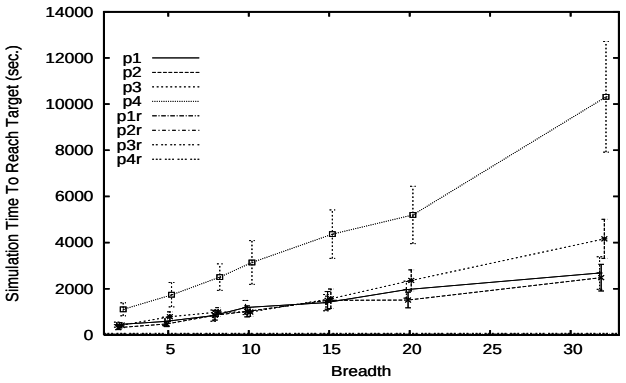

Figure 3: Mean Simulation Time for Varying Search Breadth. The overhead swamps the benefit of guidance and grows with breadth. Pure random simulation times for each property (denoted p1r, p2r, p3r, and p4r) average 29.1, 19.7, 27.1 and 67.9 seconds, respectively. The error bars show 95% confidence intervals for the true mean.

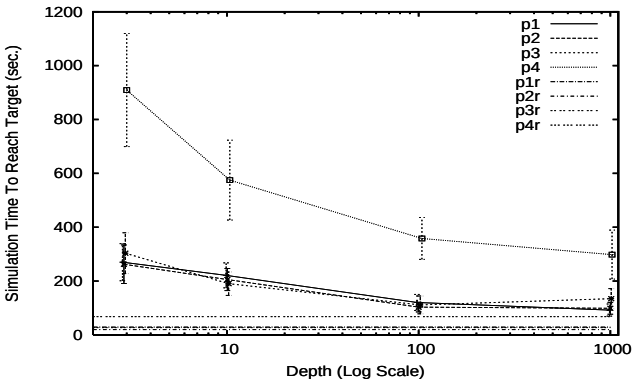

Figure 4: Mean Simulation Time for Varying Search Depth. As search depth increases, guided search becomes pure random simulation (whose results are as in Fig. 3), but with a constant factor overhead .

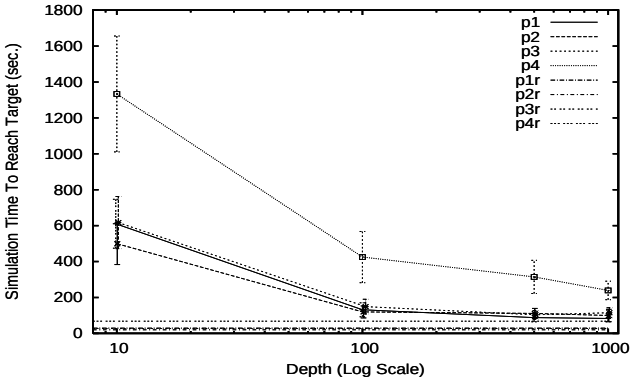

Figure 5: Mean Simulation Time Varying Search Depth with Breadth of 3. Combining breadth and depth doesn't help.

simulation is useful only if the guidance is good enough to overcome the large overhead.

3.1 Varying Search Breadth

The most straightforward search strategy is greedy hill-climbing. From a simulation state *s*, we generate *b* successors and evaluate all of them. If any successor is better (maps to a closer onion ring) than *s*, we pick the best one. Otherwise, we pick a successor randomly. The simulation then proceeds from the chosen successor.

The obvious first experiment is to vary the search breadth *b*: how many next states do we try when looking for a state that maps to a better onion ring? If the distances computed from the abstract preimages were perfectly accurate, then a greedy search with enough breadth is guaranteed to find an optimum trace to the target, so one might assume that greater search breadth will yield better results.

We simulated 60 runs for each property, with varying breadth. We also ran conventional random simulation. Fig. 3 shows the results. Despite the large error bars, two things are clear: the guided simulation is much slower than conventional simulation, and the slowdown gets *worse* with greater breadth. The overhead of running *b* simulation cycles for every cycle of progress dominates the results; guidance is ineffective, and the guided simulator is apparently getting stuck in dead-ends and then wandering randomly.

3.2 Varying Search Depth

Another common heuristic is to allow the simulator to randomly search deeper: from a simulation state *s*, run random simulation for *d* cycles, and evaluate all states on that trace. If any successor is better than *s*, pick the best one. Otherwise, pick a random state on the trace. Continue the simulation from the chosen state.

As before, we simulated 60 runs for each property, varying *d*. Fig. 4 presents the results. Exploring depth does much better than breadth, but still much worse than random. As *d* increases, the performance improves. The explanation is that as $d \to \infty$, the depth heuristic becomes pure random simulation. Indeed, the results appear to be asymptotically approaching the constant factor slowdown of guided simulation. In other words, guidance isn't working.

We can try combining breadth and depth, to get a larger sample of the local neighborhood of the simulation state. Fig. 5 shows that the results are similar: breadth (which would help if the distance metric were perfect) imposes an $O(b)$ slowdown (vs. Fig. 4), and depth approaches a slowed-down version of random simulation as $d \to \infty$. The standard heuristics do not work.

3.3 GUIDO's SimSearch

To evaluate a sophisticated, state-of-the-art guidance heuristic, we tried out the search heuristic proposed in GUIDO [19]. The GUIDO verification tool contains two search modes: an abstraction-guided simulation mode *SimSearch* that fits the framework of this paper, backed up by an exhaustive, formal, SAT-based procedure *SimSAT* for when SimSearch gets stuck.

Since the focus of this paper is guidance heuristics, we implemented and evaluated SimSearch. SimSearch explores a bounded breadth *b* and depth *d* from a given state, similarly to the previous heuristics, but stores all states that reach a different onion ring into a priority queue. The simulation then proceeds from the best state in the priority queue.[2] In [19], specific values for neither *b* nor *d* are given. We ran 60 simulations for each property, trying out $d = 5$, 10, 50, and 100. These simulations found the target only when $d = 100$. Next, we tried several values for *b*, simulating 60 runs for each property, keeping $d = 100$. The results, in Fig. 6, show that increasing breadth has limited impact on simulation time, particularly compared with the random simulation results. SimSearch alone is not an effective guidance strategy, necessitating the more expensive SimSAT mechanism in GUIDO.

3.4 Hard Gains, Easy Losses

The intuition behind abstraction-guided simulation is that the

[2] If the priority queue is empty, the description in [19] of SimSearch is undefined. Our implementation continues from the current state.

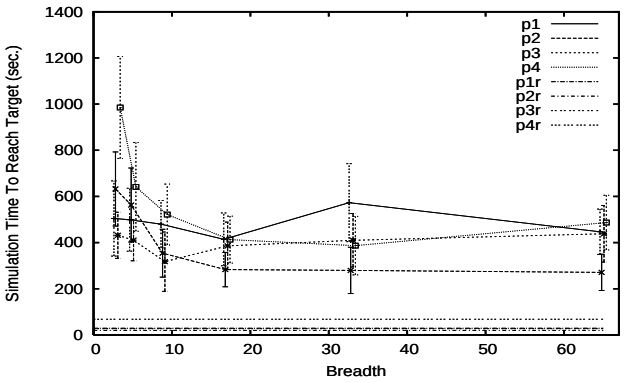

Figure 6: Mean Simulation Time for SimSearch. Even a sophisticated, recent heuristic loses to random simulation. We vary search breadth, with search depth fixed at 100. Error bars and random simulation times are as in Figs. 3–5.

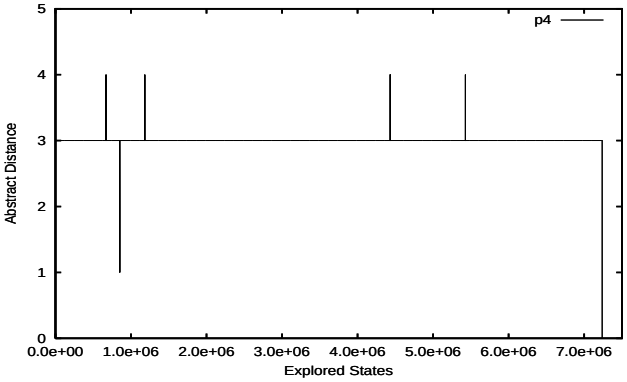

Figure 7: Simulation Trace using Depth of 100 and Breadth of 1.

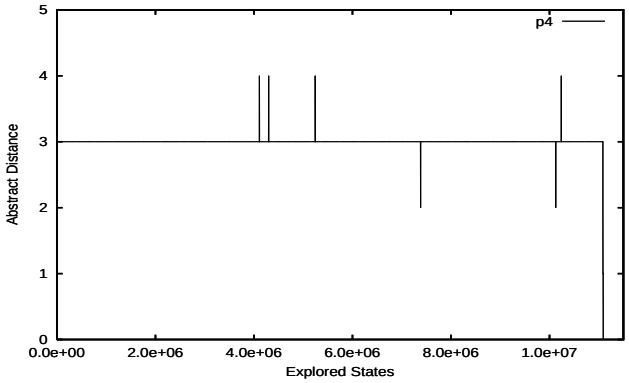

Figure 8: Simulation Trace using SimSearch with Depth 100 and Breadth 16. We see the same pattern of hard gains, easy losses.

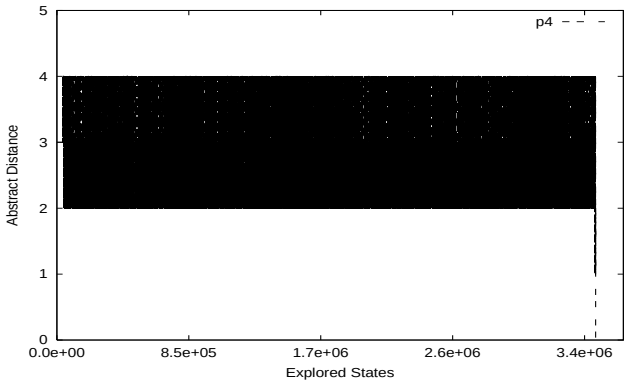

Figure 9: Simulation Trace using Algorithm 1 with Depth 100 and Breadth 1. The behavior is radically different.

simulation trace will gradually work its way into closer onion rings, perhaps with some delays or detours due to dead-end states. However, an informative picture of the progress of a search strategy emerges by plotting the onion ring number of the simulation state over time. Although each trace is unique, Fig. 7 is a typical trace. What is striking is how hard it is to make progress, but how easy to lose it. In this trace, the heuristic spends almost all of its time stuck at onion ring 3, almost never breaking through. It quickly reached onion ring 1 a bit before 10,000 cycles, which may or may not have been a dead end, but then immediately gave up this progress for more than 60,000 cycles before finally succeeding. All of the traces we have plotted for previous heuristics are qualitatively similar. Even SimSearch produces a similar graph (Fig. 8). The challenge is to develop a heuristic that doesn't get stuck near dead-ends, yet aggressively pursues promising states.

4. A NEW GUIDANCE STRATEGY

Two key ideas underlie our new guidance strategy: remembering multiple states from which to search, and balancing between greed and relaxation.

To remember multiple states from which to continue the search, we keep "buckets" of previously visited states at each onion ring distance. The buckets for the closest onion rings track the best states encountered during the simulation, overcoming the problem of easily giving up hard-earned progress. Equally important, having buckets for all distances allows flexibly backing up different distances to avoid dead-ends. Recall that a dead-end is caused by an abstract transition with no corresponding concrete transition, so one dead-end will affect many nearby states. The multiple states in each bucket provide a much more global concept of breadth, spreading the breadth across the history of the simulation, rather than the local neighborhood of one state. We implement the buckets as bounded FIFOs, guaranteeing no blow-up in space. Furthermore, using a bounded bucket for each onion ring means that states at distances that are hard to reach will persist, whereas states at onion rings where we are stuck will be quickly replaced.

The other challenge is to determine when to push forward from the current state, when to return to previously visited promising states, and when to back up to outer onion rings to escape the influence of a dead-end. The right balance will be different for different designs and different properties, and even for different parts of the search space of one simulation: in a region of the search space where the distance metric is wrong, leading to a dead-end, a guidance heuristic should abandon the current state; in a region where the distance metric is right, the guidance heuristic should press ahead. We use randomization to solve this problem. In particular, we start from the closest (lowest numbered) onion ring with a non-empty bucket and flip a (fair) coin. Heads means we continue simulation from a random state in that bucket. Tails means we go on to the next non-empty bucket. If we reach the outermost onion ring without choosing a bucket, we repeat this process. This process gives an exponential decrease of the probability of choosing each non-empty bucket, from the closest to the farthest. This prob-

Algorithm 1 New Abstraction Guided Simulation Algorithm

```
 1: procedure AGS()
 2: CS = initial_state
 3: while (CS!=goal_state) do
 4:    loop BREADTH
 5:     curr_sample = sample_next_state(CS)
 6:     loop DEPTH
 7:      distance = abstract_and_evaluate(curr_sample)
 8:      save_in_bucket(distance, curr_sample)
 9:      curr_sample = sample_next_state(curr_sample)
10:     end loop
11:   end loop
12:   bkt_index = 1; restore_bkt_index = 0
13:   while TRUE do
14:     if (flip_coin AND bucket_is_not_empty[bkt_index]) then
15:       restore_bkt_index = bkt_index
16:       break
17:     end if
18:     bkt_index++
19:     if (bkt_index >= onion_rings) then
20:       bkt_index = 1
21:     end if
22:   end while
23:   CS = bucket.random_pick(restore_bkt_index)
24: end while
```

ability distribution is important because it favors persisting with promising states (hard gains) while keeping a more global search (avoiding dead ends). The algorithm is presented in Algorithm 1.

Fig. 9 shows a typical simulation trace with our new heuristic. This is for the same design and property as in Figs. 7 and 8, but note that the guided simulation reaches the target 2-3x faster. Qualitatively, the difference is striking: once the simulation reaches a closer onion ring, it persists at that distance, but it's also flexible enough to back out to outer onion rings.

5. EXPERIMENTAL EVALUATION

5.1 Tuning the Heuristic on the Training Set

Our new heuristic presented in Section 4 has only two parameters: *depth*, and *breadth*. From the experiments in Section 3, we selected *depth* and *breadth* to be 100 and 1. As noted in Section 2, we use these parameters for all evaluations of our heuristic, with no further tuning.

5.2 A Clean Test Set

The task now is to evaluate the heuristic in a different design. In this section, we report results for the USB Function Core Packet Layer Unit (usbf_pl). Although, this design shares one unit with the DUV of the training set, namely the usbf_pd, none of the properties verified in this section relates to the training set. Furthermore, the interconnects we are interested in do not share any signals with the ones in the DUV of the training set.

We looked into four usbf_pl properties:

usb_p0 After receiving a transfer command request from the host processor, does the usbf_pl time out if the host does not follow the request with a packet?

usb_p1 Has a packet been received and is it ready to be DMAed to Memory?

usb_p2 After sending data to the host in response to a host command, does the usbf_pl time out if no acknowledgment is properly signaled by the host?

usb_p3 Upon receiving data, is the data PID in sequence?

Property	Concrete Model		VIS on Abstract Model	
	VCEGAR	VIS	CPU Time	onion rings
usb_p0	2128.8s	MemOut	66.8s	26
usb_p1	42809.2s	MemOut	32277.7s	12
usb_p2	MemOut	MemOut	71.0s	28
usb_p3	MemOut	MemOut	72.5s	5

Table 1: Formal Verification Trials. VCEGAR runs were on Intel P4@3.2GHz; VIS, on Sparcv9@900MHz. MemOut is 800MB.

We chose these properties to meet three criteria: first, they are real properties, describing interesting behavior of the design; second, the properties are non-trivial for simulation; and third, they are challenging to the formal tools as well.

Recall that we use VCEGAR and VIS as our formal tools. VCEGAR automatically abstracts the design, whereas for VIS, we manually created a structural abstraction by removing design units not directly mentioned in the properties being verified. The usbf_pl comprises 4 units: Packet Assembly, Packet Disassembly, DMA and Memory Interface, and Protocol Engine. Altogether, it has 536 latches, 157 inputs, and 143 outputs. The abstract model included only the Protocol Engine and the DMA and Memory Interface units, and had 397 latches, 170 inputs, and 159 outputs.

Table 1 presents the formal verification results. Both tools had trouble with the concrete design, but VIS was able to model check the structural abstraction for all four properties. Because the structural abstraction also generated more onion rings, we used those results for the guided simulation runs.

Table 2 compares guided simulation using the new heuristic to random simulation. In three of the four cases, the guided simulation performed better than both random simulation and formal verification. More specifically, for the property usb_p0, VIS blows up when model checking the concrete design, and guided simulation is two orders of magnitude faster than VCEGAR or conventional simulation. For usb_p1, VIS again blows up, but the other methods succeed. Random simulation is more than 10x faster than VCEGAR or guided simulation (including the abstract model checking time). On the harder properties, usb_p2 and usb_p3, both formal tools ran out of memory, and the random simulations timed out on every trial, despite running for several days for each trial. Guided simulation took only hours, and never timed out.

5.3 Case Study on a Separate Design

As an additional test of the robustness of our guided-search strategy, we selected a completely different design and followed the verification flow methodology assumptions made in Section 2. The design is the Ethernet MAC 10/100 Mbps from www.opencores.org. The verification focused on the core functions of the design comprising four units: MAC Control, Transmit, Receive, and Status units. We tried to hit 14 properties in all. We started with random simulation and quickly reached 12 of these. The remaining two properties seemed reasonably difficult for simulation, so we tried to formally verify them.

After some hand modifications (verified with an equivalence checker) to accommodate the Verilog limitations of VCEGAR and VIS, we attempted to formally verify the remaining two properties. Both tools exhausted the memory available (memory limit was 800Mbytes). For VIS, we manually abstracted the design, selecting the Receive unit to be the abstract model, since all the properties were related to this unit. During the abstraction process, we realized a problem with the model: it had multiple-clocks, and neither formal tool supports this feature. We updated all four

Property (Run)	#of Trials	Avg (s)	(95% Conf. Interval)	(Min; Max) (s)
usb_p0 (Random)	30	1011.3	(656.8; 1365.8)	(27.5; 3999.3)
usb_p0 (Guided)	30	1.4	(1.25; 1.72)	(0.4; 2.9)
usb_p1 (Random)	30	3510.1	(2224.2; 4795.9)	(106.8; 10885.5)
usb_p1 (Guided)	30	6681.6	(4015.6; 9347.7)	(150.8; 28865)
usb_p2 (Random)	22	TimeOut0		NA
usb_p2 (Guided)	30	10585.6	(6109.7; 15061.4)	(481; 51444.4)
usb_p3 (Random)	16	TimeOut1		NA
usb_p3 (Guided)	30	71687.4	(53804.9; 89570)	(4424.3; 224962.7)

Table 2: Random vs. Guided Simulation Time. The time to reach the target is measured in seconds. Simulation times for usb_p1 were on a Sparcv9 1.3GHz; others, on a Sparcv9 900MHz. TimeOut0>100 hours. TimeOut1>150 hours.

Property	VIS abstract model (s)	Avg (s) (95% Conf. Interval)	(Min; Max) (s)
eth_p0			
Random	NA	19 out of 30 TimeOut0	NA
Guided	1777	20.9 (13.9; 27.9)	(1.7; 92)
eth_p1			
Random	NA	TimeOut1	NA
Guided	11373	16.1 (12.9; 19.3)	(3.7; 38.8)

Table 3: Random vs. Guided Simulation Time. Times were on a Sparcv9 900MHz. TimeOut0 $>$ 3 hours. TimeOut1 $>$ 6 hours

units (to maintain synchrony with the simulations) by hand (again, equivalence checked) and tried again. VCEGAR was still unable to handle both properties due to either failing to find new predicates or exhausting the memory available. VIS, however, was able to verify the abstract model, so we used the VIS results to guide the simulation. We ran 30 simulations comparing random and guided simulation on these two properties. The results in Table 3 show that on a completely different design, guided simulation helps find the targets, whereas random simulation is usually timing out.

Unfortunately, we later realized that we had not tried VIS on the concrete model (which had blown-up earlier) after fixing the multiple-clock issue. It turns out VIS finds these two targets in less than five minutes. Although our oversight weakens the case study, the results still demonstrate that guided simulation did help find these two hard-to-reach targets much faster than random simulation, on a different design, with no heuristic tuning.

6. CONCLUSION AND FUTURE WORK

Our study of the typical local search heuristics used by most previous works on abstraction-guided simulation shows that they are not effective in avoiding dead-ends. Based on these experiments, we propose a new heuristic that is better able to avoid dead-ends by tracking multiple promising states and backing-off when getting stuck. Experimental results on a variety of designs show excellent results on hard-to-reach targets, with no heuristic tuning.

The direct line of future work is further experimentation to confirm our results and illuminate the way towards better and even more robust guidance strategies. More generally, a challenge for abstraction-guided simulation is how to deal with targets specified via a non-synthesizable software testbench. Handling such targets is necessary to truly and seamlessly bridge formal and simulation.[3] Fortunately, the simulation side needs no modification: anything that can be done in a simulator can be done in a guided simulator. To compute the abstract pre-images, we believe software model checking techniques can apply.

[3]Thanks to Eyal Bin and Gil Shurek for pointing this out.

With better heuristics and broader applicability, abstraction-guided simulation will be a valuable tool in the verification arsenal, filling the gap between formal verification and simulation.

7. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *TACAS*, pp. 193–207. Springer, 1999. LNCS 1579.

[2] V. Boppana, S. P. Rajan, K. Takayama, and M. Fujita. Model checking based on sequential ATPG. *CAV*, pp. 418–430. Springer, 1999. LNCS 1633.

[3] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. *CAV*, pp. 428–432. Springer, 1996. LNCS 1102.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *LICS*, pp. 428–439, 1990.

[5] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Wkshp Logics of Programs*, pp. 52–71, 1981. LNCS 131.

[6] F. M. de Paula and A. J. Hu. EverLost: A flexible platform for industrial-strength abstraction-guided simulation. *CAV*, pp. 282–285. Springer, 2006. LNCS 4144.

[7] S. Edelkamp and A. Lluch-Lafuente. Abstraction in directed model checking. *Wkshp Connecting Planning Theory and Practice*, pp. 7–13, 2004.

[8] M. K. Ganai and A. Aziz. Rarity based guided state space search. *GLSVLSI*, pp. 97–102. ACM, 2001.

[9] S. Gorai, S. Biswas, L. Bhatia, P. Tiwari, and R. S. Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. *DAC*, pp. 731–736. ACM/IEEE, 2006.

[10] A. Gupta, A. E. Casavant, P. Ashar, X. G. S. Liu, A. Mukaiyama, and K. Wakabayashi. Property-specific testbench generation for guided simulation. *ASPDAC and VLSID*, pp. 524–531. IEEE, 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. *ICCAD*, pp. 120–126. IEEE/ACM, 2000.

[12] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. *ISCA*, 1995.

[13] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. *DAC*, pp. 445–450. ACM/IEEE, 2005.

[14] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. *ICCAD*, pp. 574–579. IEEE/ACM, 1999.

[15] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. *FMCAD*, pp. 159–173. Springer, 2004. LNCS 3312.

[16] K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. *DAC*, pp. 737–742. ACM/IEEE, 2006.

[17] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. *Intl Symp Programming*, pp. 337–351. Springer, 1981. LNCS 137.

[18] N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. *FMCAD*, pp. 60–67. IEEE, 2006.

[19] S. Shyam and V. Bertacco. Distance-guided hybrid verification with GUIDO. *DATE*, pp. 1211–1216, 2006.

[20] C. H. Yang and D. L. Dill. SpotLight: Best-first search of FSM state space. *HLDVT*, 1996.

[21] C. H. Yang and D. L. Dill. Validation with guided search of the state space. *DAC*, pp. 599–604. ACM/IEEE, 1998.

[22] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. *CAV*, pp. 376–387. Springer, 1997. LNCS 1254.