An Effective Guidance Strategy for Abstraction-Guided Simulation

by

Flavio M. de Paula

M.Sc. in Electrical Engineering, Universidade Federal de Minas Gerais, 1999

B.Sc. in Electrical Engineering, Universidade Federal de Minas Gerais 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

April 2007

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 promise has yet to be realized, as the performance of previous work has been unreliable — sometimes great, sometimes poor. The problem is the guidance strategy.

In this thesis, we first develop a platform to enable flexible exploration of abstraction-guided simulation — different guidance heuristics and formal tools are easily inserted — while providing the capacity, speed, and Verilog compatibility of a leading industry-standard (logic-simulation) tool, Synopsys VCS. Then, we start by exploring some greedy heuristics and find that they tend to perform poorly, adding too much search overhead for limited ability to escape dead ends (local optima). Based on these experiments, we propose a new guidance strategy, which 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.

Contents

List of Tables

List of Figures

Acknowledgments

I would like to acknowledge Alan J. Hu for his support and guidance. I would also like to acknowledge: my graduate professors, especially, Anne Condon and David Kirkpatrick, for kindly guiding me as I entered into the world of the theory of computation; and, my fellow colleagues in the Integration Systems Laboratory, Xiushan Feng and Domagoj Babic, for the priceless theoretical conversations we had.

I dedicate this thesis to you...

Chapter 1

Introduction

1.1 Motivation and Philosophy

Abstraction-Guided Simulation (AGS) is a verification technique that combines model checking and dynamic verification (simulation). A formal definition of AGS will be given in later sections. For now, consider the task of verifying a design according to its specification. AGS divides this task into two phases: first, a model checking phase, in which AGS checks an abstract version of the design under verification (DUV) and generates a coarse map leading to the specification goal; second, a dynamic verification phase, in which AGS guides the DUV's simulation towards the specification goal by heuristically choosing transitions based on the coarse map.

AGS has drawn a lot of attention from the hardware verification community. AGS is appealing because the model checking phase considers only an abstract version of the DUV, and in doing so, it is not so much affected by the DUV's size. It is no surprise with this appeal that several researchers have proposed their own AGS schemes, (e.g. [19, 34, 14, 33, 22, 17, 13, 30, 26, 29]). However, results are often worse than with other techniques like (directed) pseudo-random simulation. The reasons for this inconsistency are twofold: the abstraction techniques used do not always generate good abstract models (precise maps), and the proposed guidance heuristics perform poorly with these imprecise maps.

The problem is *dead-end states.* A *dead-end state* is a state from which no progress is made towards a specification goal because imprecise abstractions create transitions that do not exist in the concrete model. Dead-end states create challenges to AGS for three reasons: a priori, there is no way to know where dead ends are located in the search space; it is hard to know whether the current simulation state is a dead end or not; and if a dead end is found, it is hard to know whether a dead end will be re-visited or not.

This thesis focuses on finding guidance heuristics that demonstrate consistent and effective results by properly handling dead-end states. To achieve this goal, the research is divided into several stages: first, to develop a research platform capable of supporting industrial designs; second, to investigate and to draw conclusions about common search algorithms; third, to investigate a recently proposed advanced search algorithm [30]; and finally, to propose my own guidance heuristic and to demonstrate its effectiveness.

1.2 Background and Related Work

This section presents background on hardware verification techniques. While presenting these techniques in the order in which they were developed, I will discuss their advantages, disadvantages, and related work.

1.2.1 Dynamic Verification

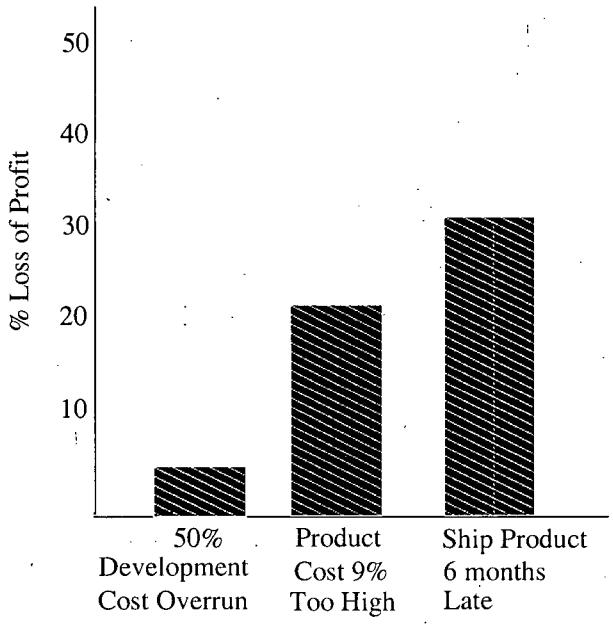
Dynamic verification[1] (DV) is a technique that uses logic simulation to check the behaviors of a design-under-verification (DUV). The role of a logic simulator is to simulate input stimuli to the DUV and to compute the DUV's next state and outputs. Therefore, dynamic verification involves coordinating the input stimuli generation and checking the DUV's response against the DUV's expected behavior.

[1] Also known as "simulation" in the integrated circuit industry.

DV has been one of the most used verification techniques for hardware designs. Several reasons contribute to this success. I classify these reasons into two groups: direct and indirect. The direct reasons are related specifically to the dynamic verification technique, whereas in the indirect group, we have social and historical reasons. The direct reasons for DV's success are its ease-of-use, its speed, and its scalability. In dynamic verification, each simulation run has only one execution trace, which is a sequence of input stimuli, the DUV's state transitions, and the output changes. DV is easy to use because we need to consider only an individual trace at a time while debugging the DUV. Also, each simulation run requires the tracking of only a small number of states (compared to the entire state space), making simulation fast and more independent of the DUV's size.

The indirect reasons for the success of dynamic verification are its mature technology and strong tool support. Dynamic verification is as old as integrated circuits (ICs). Because of this long history, it is common in academic departments of computer science and engineering to have DV courses for undergraduate students. At the same time, electronic design automation (EDA) companies like Synopsys Inc., Cadence Design Systems Inc., and Mentor Graphics Corporation have been at the forefront of dynamic verification with their tools [31, 6, 24], and EDA companies have already established a solid relationship with IC companies. Therefore, the large support coming from academia coupled with those commercial tools have immensely contributed to DV's success.

Today's design complexity, however, is pushing dynamic verification to its limit. The problem is that the number of possible behaviors of the DUV grow exponentially with the DUV's size. Therefore, this growth requires an ever-increasing number of simulation cycles to examine the long and non-trivial DUV behaviors. Also, because dynamic verification analyzes only a single simulation trace each run, a great number of simulation runs may be required to provide assurance that the DUV is free of bugs. Even then, several bugs may slip into the silicon. Once a bug



Source: McKinsey and Co.[11]

Figure 1.1: Time-to-Market Cost

is found in the silicon, it usually means the IC needs to be fabricated again (re-spin). This results in two extra costs: the re-spin itself and the cost of the increased time-to-market. Figure 1.1 shows the profit losses due to delayed production [11]. Therefore, we need a more thorough and conclusive verification technique to find bugs before they slip into the silicon. Model checking is one example of such a verification technique and will be described in the next section.

1.2.2 Model Checking

Model checking [7, 28] is a formal[2] verification technique. In contrast with dynamic verification, model checking exhaustively searches the state space. Given sufficient resources (memory and cpu time), model checking will always conclude whether the system satisfies its specification or not.

[2]In the sense of mathematical formalism, rather than just applying test stimuli.

The model checking verification flow consists of three major steps: modeling, specification, and verification. The modeling step is a translation of the system into a formal model, capturing the properties of the design. In model checking, a Kripke structure is a typical formal model used to describe the system. A Kripke structure is a nondeterministic finite state machine given by the 4-tuple: $M = \{S, S_0, R, L\}$, where

- S is the finite set of states,
- S_0 the set of initial states,
- $R \subseteq S \times S$ is a total transition relation,
- $\mathcal{L} : S \mapsto 2^{AP}$, is a function that labels each state with a set of atomic propositions, AP, that are true in that state.

The specification step uses some logical formalism (e.g. CTL or LTL [8, 27]) to capture the system's properties. In this thesis, we are interested only in safety properties. Intuitively, a safety property states that something bad never happens in the system (e.g. deadlock, division by zero). In CTL, safety properties are usually encoded as AG$\neg p$ — for all paths and in every state in those paths, p is false — where p is a propositional formula, denoting the bad event. Although focusing only on safety properties seems restrictive, we are still able to capture many important design properties.

Finally, the verification step consists of a methodical search of the state space for a state violating the specification. Recall that we are using a Kripke structure to model our design. Therefore, we need to traverse the Kripke structure to verify whether it is a model of the CTL formula AG$\neg p$. Backward reachability is a standard mechanism used in model checking to verify such formulas. We compute backward reachability by successive pre-image (defined below) operations. For example, consider a set P_0, which satisfies p, i.e., the set of bad states. The pre-image of the state P_0 is the set of states that can reach P_0 in one step. We then compute

the pre-image of this new set of states and iterate. Eventually, this computation reaches a fixed-point, having computed all the states that can reach P_0. Formally, the pre-image operator is given by

$$PreImg(S) = \{s \in S | \exists s' \cdot s' \in S \wedge R(s, s')\} \tag{1.1}$$

and, the backward reachability computation is given by

$$\mathcal{F}_0 = P_0$$

$$\mathcal{F}_1 = \mathcal{F}_0 \cup PreImg(\mathcal{F}_0)$$

$$\mathcal{F}_2 = \mathcal{F}_1 \cup PreImg(\mathcal{F}_1)$$

$$\vdots$$

In practice, for efficiency, model checking tools implement two termination conditions while checking safety properties: first, the computation reaches a fixed-point that does not include any initial state, i.e., the model checking tool shows that the set of bad states are unreachable from the set of initial states; second, the successive pre-images include an initial state, at which point, the model checking tool asserts that it is possible to reach the set of bad states and generates a counterexample. We illustrate these two scenarios in Fig. 1.2.

Counterexample generation is one of the important features of model checking. A counterexample is an error trace demonstrating how the model can reach a bad state from an initial state. Consider the model-checking tool found that an error trace exists; that the model-checking tool computed n backward reachability iterations, and that the result of each $PreImg$ call, $P_1, P_2, \ldots, P_n$, is available. The model-checking tool will generate the counterexample by:

1. choosing an initial state from $P_n \cap S_0$;

2. computing its image (dual of the pre-image operator);

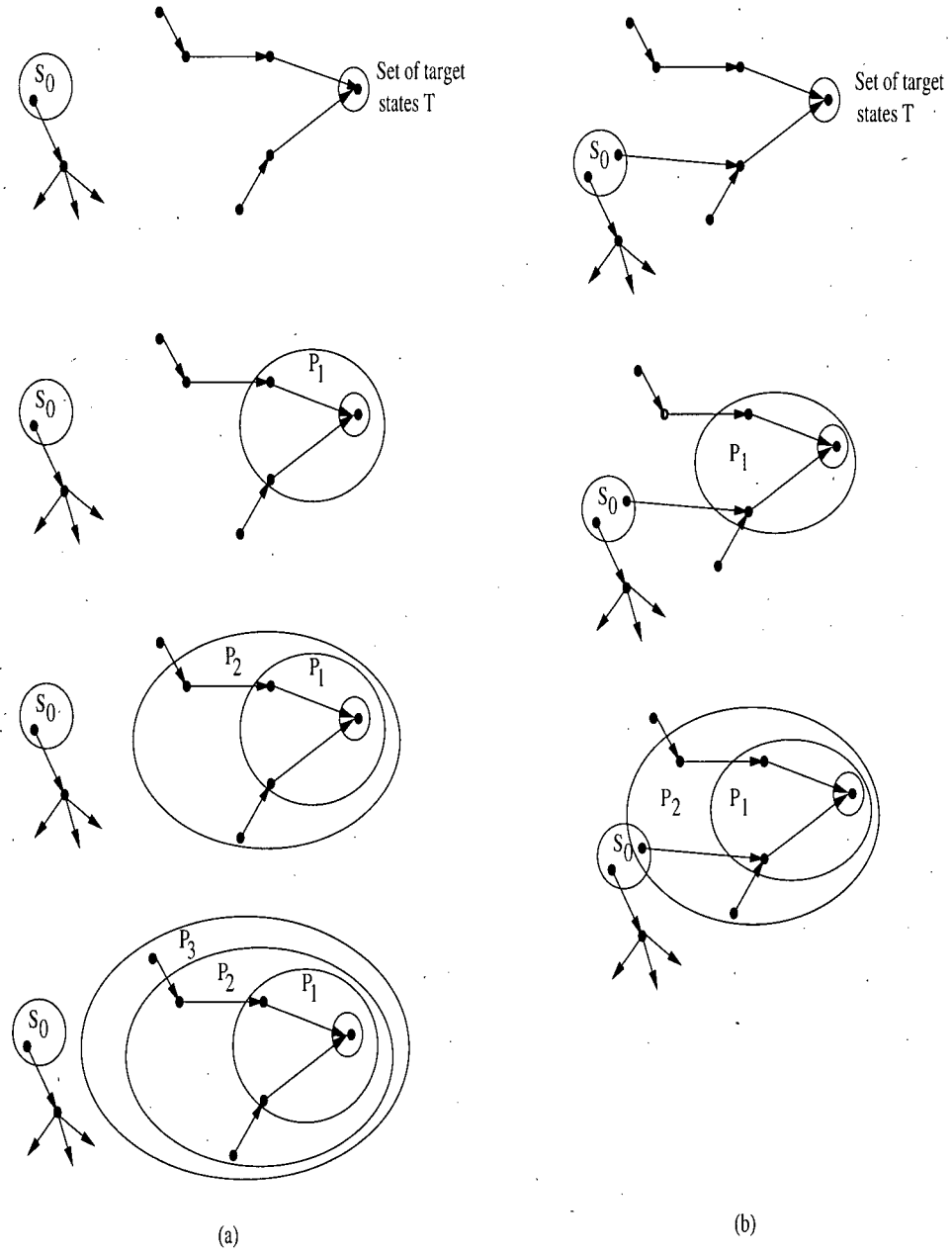


Figure 1.2: Model Checking AG$\neg p$ via backward reachability computation. S_0 is the set of initial state; T is the set of target states in which p is true. Termination conditions: (a) Successive pre-images reach a fixed-point without including any initial state; (b) Successive pre-images reach an initial state.

3. intersecting the resulting set from step (2) with the set P_{n-1};

4. choosing a state from this intersection set and repeating steps (2)-(4) until $n = 0$.

The main disadvantage of model checking is its complexity. The state space of meaningful systems is enormous and computers have only a fraction of the memory space needed for model checking such systems. This problem, called the state explosion problem, has been an active area of research. The most prominent techniques used to lessen the state explosion problem have relied on saving memory by using different representations [5], bounding the search [2, 3], and creating simpler models via abstraction [21]. I will discuss abstraction techniques in the next section, since the model checking of abstract models is an essential part of this thesis.

1.2.3 Abstraction

Abstraction reduces complex systems to simpler models while still preserving important behaviors of the original system. Then model checking tools can use this simpler model (abstract model) to verify properties of the original design.

In this thesis, we focus on conservative abstraction. A conservative abstraction is such that if a property holds (model checking proved the property to be true) on the abstract model, the property also holds on the original design. However, if a property fails in the abstract model, we cannot assert, without further inspection, that the property also fails on the original design.

Consider two models $M = \{S, S_0, R\}$ and $M^A = \{S^A, S_0^A, R^A\}$, where S and S^A are the sets of states; S_0 and S_0^A, the sets of initial states; and $R \subseteq S \times S$ and $R^A \subseteq S^A \times S^A$, the transition relations. Also, consider $\alpha : S \rightarrow S^A$, the abstraction function that maps states from the model M to states in the model M^A. We say that model M^A is a conservative abstraction of model M if $S_0^A = \{a_0 | \exists s_0 \in S_0 \; . \; a_0 = \alpha(s_0)\}$; if every transition $s \rightarrow s' \in R$ can be simulated

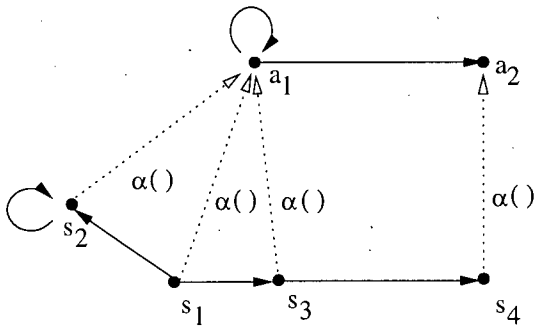


Figure 1.3: Conservative Abstraction. Concrete model M, its abstract model A, and respective transitions. The function $\alpha(\cdot)$ is the abstraction function.

by a transition $a \rightarrow a' \in R^A$, such that $\alpha(s) = a$ and $\alpha(s') = a'$. Formally, the abstract transition relation R^A is given by

$$R^A = \{(a, a') \mid \exists s, s' \in S \;.\; R(s, s') \wedge \alpha(s) = a \wedge \alpha(s') = a'\} \tag{1.2}$$

In Fig.1.3, we illustrate the effects of conservative abstraction. Observe that, in the concrete model, M, we have transitions $s_1 \rightarrow s_2$, $s_1 \rightarrow s_3$, and $s_3 \rightarrow s_4$. The transition $a_1 \rightarrow a_2$ exists in the abstract model A because $s_3 \rightarrow s_4$ exists and $\alpha(s_3) = a_1$ and $\alpha(s_4) = a_2$. Notice that transition $a_1 \rightarrow a_1$ exists in the abstract model A because, for example, $s_1 \rightarrow s_2$ exists and $\alpha(s_1) = a_1$ and $\alpha(s_2) = a_1$. This shows that conservative abstraction can overapproximate the behaviors of the concrete system. For example, the abstract model in Fig.1.3 gives an impression that s_4 is reachable from all states in the concrete model, which is clearly not true.

In Chapter 3 and 4, we will use two instances of conservative abstraction: localization reduction and predicate abstraction. Localization reduction [23] is an abstraction technique that simplifies a design by removing some of its components. More specifically, in hardware designs, latches and logic elements that are not directly mentioned in the properties being verified are good candidates for being removed (abstracted away) from the design. The hope is that the fundamental be-

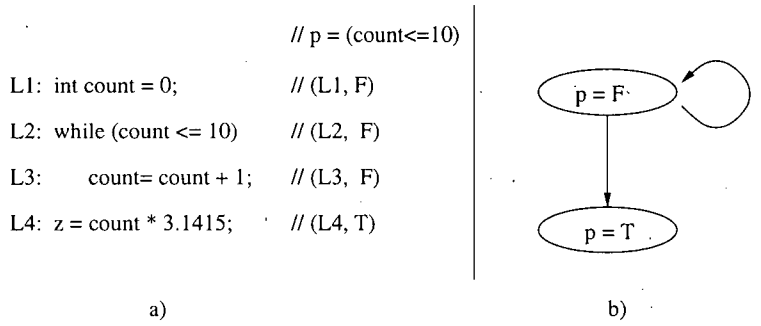


Figure 1.4: Predicate-Abstraction Example. a) Small Code Excerpt and Predicate Valuations per Line; b) Resulting Abstract Model.

havior of the design does not change in the absence of these removed components.

Predicate abstraction focuses on data abstraction. For example, consider the program in Fig. 1.4 *a)*. The type of the variable *count* is *int*. Instead of reasoning about all possible values for *count* (i.e., all 2^{32} possible values, if the size of *int* is 32 bits), a smaller model could be created based on the truth values of the predicate $count \leq 10$. Fig. 1.4 *b)* gives the abstract model for this program.

More precisely, predicate abstraction [16] generates an abstract model whose state variables are boolean variables. The truth values of these boolean variables represent valuations of data predicates in the concrete model. While localization reduction implicitly constructs the abstract transition system (the current and next-state logic of the components in the abstract model), predicate abstraction needs explicitly to compute — e.g. using theorem provers [16] or SAT solvers (e.g. [10]) — the abstract transition system over the boolean variables of the abstract model.

The advantage of abstraction is that it allows for model checking to verify larger systems using simpler, smaller, abstract models. However, the main disadvantage of abstraction is that the abstract model is often too coarse. A coarse model is prone to false negatives — the property fails in the abstract model but not in the concrete model — due to overapproximation. To eliminate false negatives, we need to refine the abstract model until the property is either proved or disproved.

This iterative refinement technique is called Counterexample Guided Abstraction Refinement (CEGAR) (e.g. [9, 1]).

CEGAR has being very successful in extending the capacity of model checking tools (e.g. [21]). However, with ever-increasing design complexity, even abstract model checking often exhausts the memory available. While we are limited in what we can achieve running abstract model checking on such very large designs, we can still collect important information like abstract pre-images. The abstract pre-images of a design contain approximate distances information of the search space and can be used to guide a simulator during dynamic verification. This technique is called abstraction-guided simulation, and we describe it in the next section.

1.2.4 Abstraction Guided Simulation

Abstraction-guided simulation is a general framework for automatically harnessing, during simulation, information obtained by model checking an abstraction of the design. The abstract design can be simplified enough to be amenable to model checking, and the analysis gives a "big picture" global view of the structure of the state space, which can direct the simulator in promising directions.

The earliest work in this area [19, 34] 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 [14]. 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., [33, 22, 17, 13, 30, 26, 12, 29]). Fig. 1.5 sketches this approach.

More specifically, abstraction-guided simulation consists of the following:

- The goal of verification is to find an execution sequence that reaches a specified set of target states, e.g., error states or a hard-to-reach coverage target.
- Any conservative abstraction technique is used to create a model small enough

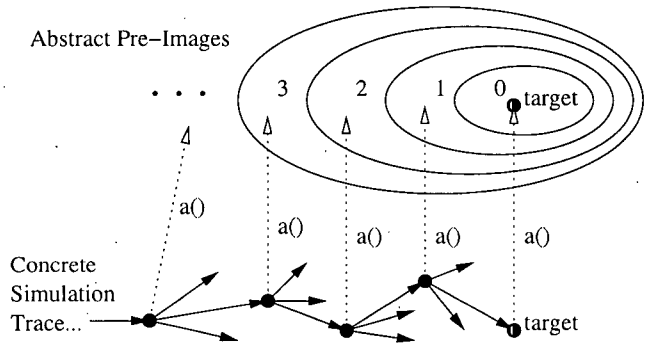


Figure 1.5: Abstraction-Guided Simulation.

for model checking. Recall that the abstract model preserves existence of any paths to the error (target) states, but may introduce paths that do not correspond to any concrete path.

- If formal verification succeeds (either finding no abstract error paths, or deriving a concrete path to the target states from an abstract error path), the verification is done. The interesting case for simulation is when formal verification fails (and attempts at abstraction refinement fail to create a tractable model), as can occur typically with large hardware designs.

- The model checker has computed a series of pre-images from the error states in the abstract model as in Section 1.2.2. These pre-images represent sets of abstract states whose shortest (abstract) path to an error state is i abstract states long. For example, as in Fig. 1.5, visualize these sets as concentric ("onion rings") around the error states. A concrete state that abstracts to an abstract state in ring i is at least i clock cycles away from an error state.

- During dynamic verification, the simulator can consult the abstraction information for guidance by periodically computing the abstraction of the current simulation state using the abstraction function, illustrated in Fig. 1.5 as $a()$, and querying which ring it is in. Thus, the simulator can benefit from considerable information computed by model checking the abstract model.

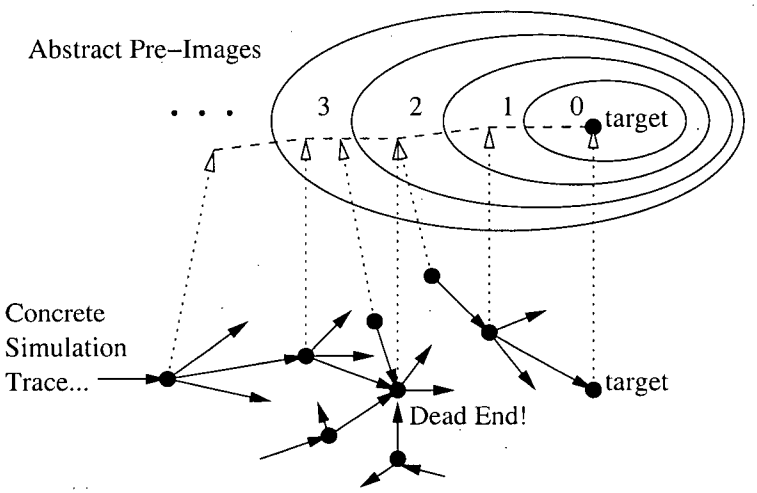


Figure 1.6: Dead-End States.

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 does not. The core problem is dead-end states. For example, in Fig.1.3 in page 9, analyzing the abstract model, one could assume that any time a concrete state is abstracted to a_1 would mean that the simulation is making progress (since a_2 is reachable from a_1); however, s_2 abstracts to a_1 but notice that, from s_2, there is no path to any state that abstracts to a_2, i.e., the simulation is stuck. Because two different concrete states may map to the same abstract state (e.g., in onion ring 2, in Fig. 1.6), an abstract trace might not correspond to any concrete trace. If so, the abstraction will lead the simulator to a dead end. Unfortunately, the simulator has no way of knowing 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 [13], SAT [30], or abstraction refinement [26]) 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.

Chapter 2

Building a Research Platform

Three goals motivated building a research platform. First, because this thesis researches guidance heuristics, I needed to have a platform in which different heuristics could be easily deployed. Second, because abstraction-guided simulation involves model checking and abstraction — each having a variety of techniques to choose from such as, for example, symbolic or bounded model checking, and predicate abstraction or structural abstraction — it was important to have a flexible platform such that different formal engines could be used. Finally, because of the need of validating my experiments using real, industrial hardware designs as benchmarks, a logical choice was to leverage the speed, capacity, ease-of-use and Verilog[1] [20] compatibility of commercial logic simulators. All these considerations helped us decide which third-party tools to use, and what tools to develop for our research platform.

Our research platform consists of three major components: the logic simulator, the abstraction/model-checking engine, and a tool, EverLost[2], we developed to implement abstraction-guided simulation. For tight integration and highest performance, we had to target a specific logic simulator, although the tool could be retargeted easily. We chose Synopsys VCS [31] because it is one of the most widely-

[1]Hardware designs in industry are typically described using the hardware description language Verilog.

[2]The name "EverLost" is a play on a pioneering, widely-deployed in-car GPS navigation system.

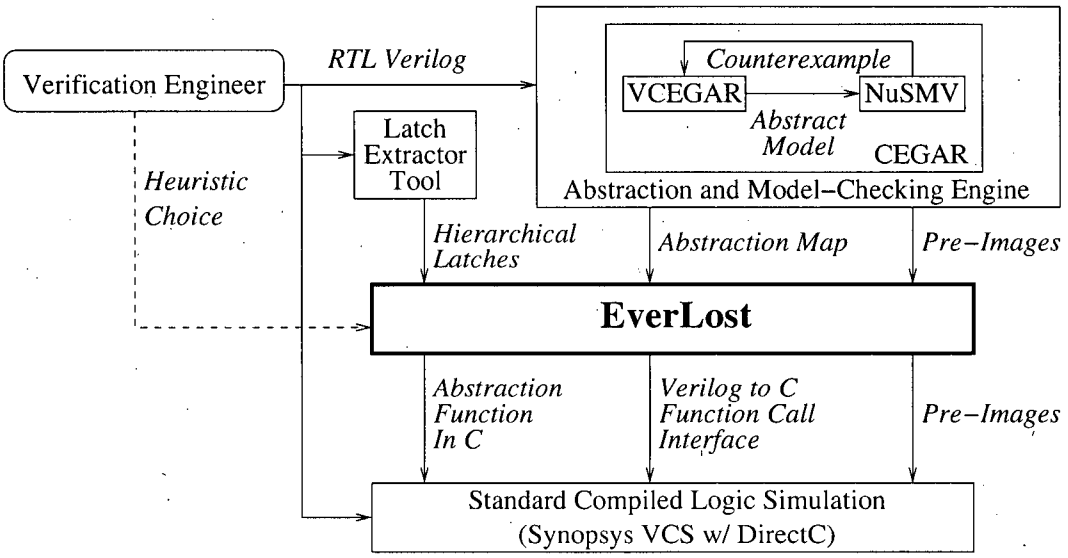


Figure 2.1: Everlost Architecture.

used logic simulators in industry. We used Synopsys VCS version 7.2.

For the interface with the abstraction/model-checking engine, we designed for maximum flexibility: all we require are a list of the design's latches, the abstraction function, and the BDD pre-images that are a by-product of symbolic model checking tools. In Fig. 2.1, the darker box illustrates the fact that we can use different abstraction/model-checking techniques, e.g., from CEGAR (counterexample-guided abstraction refinement) to structural-abstraction based model checking. In this thesis, we either used VCEGAR [21] version 0.9 or VIS [4] version 2.1 as our formal engines; these are the only free formal tools we are aware of that can handle substantial Verilog designs. Chapter 3 and 4 describes in detail when each abstraction/model-checking technique and tool is used.

Given the needed inputs, EverLost generates a simulation guidance driver in C, the abstraction function in C, and a C interface in Verilog, which are passed to VCS along with the Verilog files and the BDD pre-images. The user can specify different simulation guidance heuristics via EverLost options.

The code generated by EverLost is compiled with VCS into a single exe-

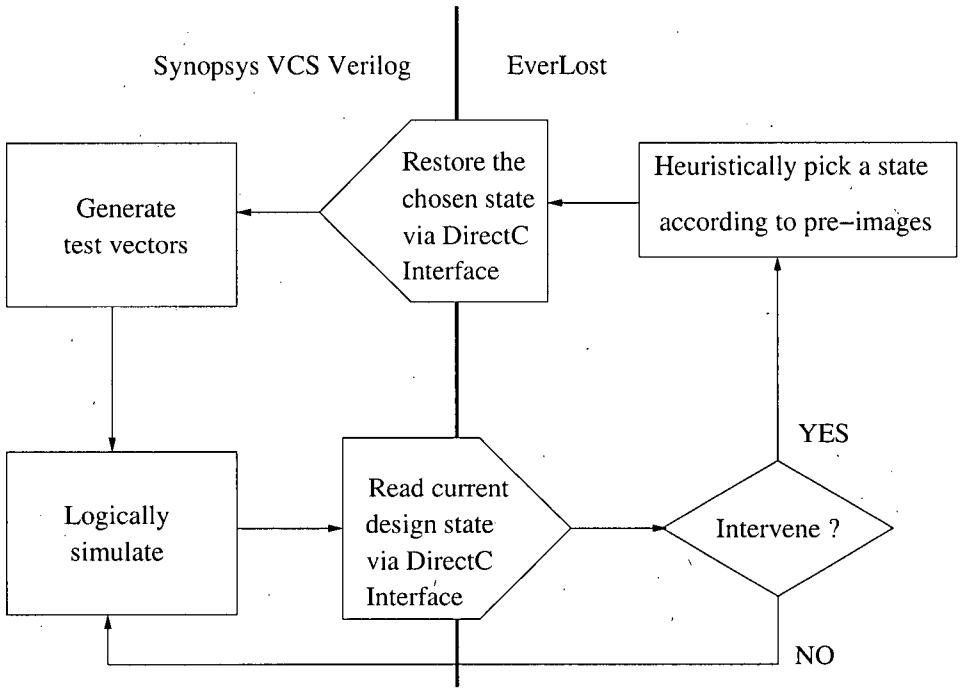


Figure 2.2: Everlost Plaftorm: Abstraction-Guided Simulation Flow.

cutable. Internally, the simulator calls the EverLost driver every clock cycle. The EverLost code can read the current simulation state, possibly save it, and possibly evaluate it using the abstraction information. The EverLost code can then allow the simulation to continue, or it can force the simulator to jump to a particular state. Fig 2.2 sketches this flow.

This platform is available at http://www.cs.ubc.ca/~depaulfm/EverLost.

Chapter 3

Guided-Search Strategies

We start experimenting with some simple, greedy, local-neighborhood search heuristics. Next, we implement a version of a proposed advanced guidance strategy [30]. The conclusions we draw from these experiments illustrate the problems most of the current abstraction-guided simulation techniques face. We then propose our own guidance strategy to address this problem.

3.1 Bounded 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. As illustrated in Fig. 3.1 for example, consider a heuristic that, from a given concrete simulation state, simulates b different random traces, each d cycles long, and then moves 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 verification (DUV) two design units from the USB 2.0 Function Core and the USB 1.1 PHY designs from OpenCores.org [1]. Because we needed a large number of experiments, we focused

[1] http://OpenCores.org is an open source repository for hardware designs.

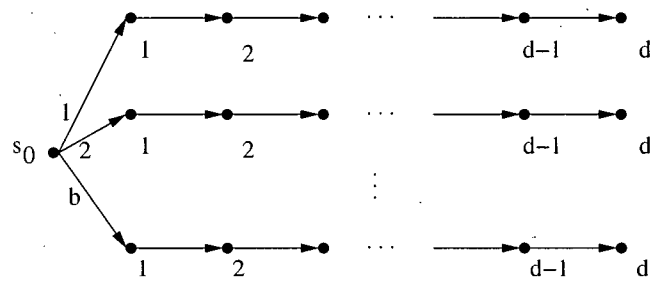


Figure 3.1: Simple Local-Neighborhood Heuristic. Exploring b different random traces, each d cycles long.

on two small units from these designs, but as often arises in practice, we examined the integration of two units, each one coming from a separate design. 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, as shown in Fig. 3.2. 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

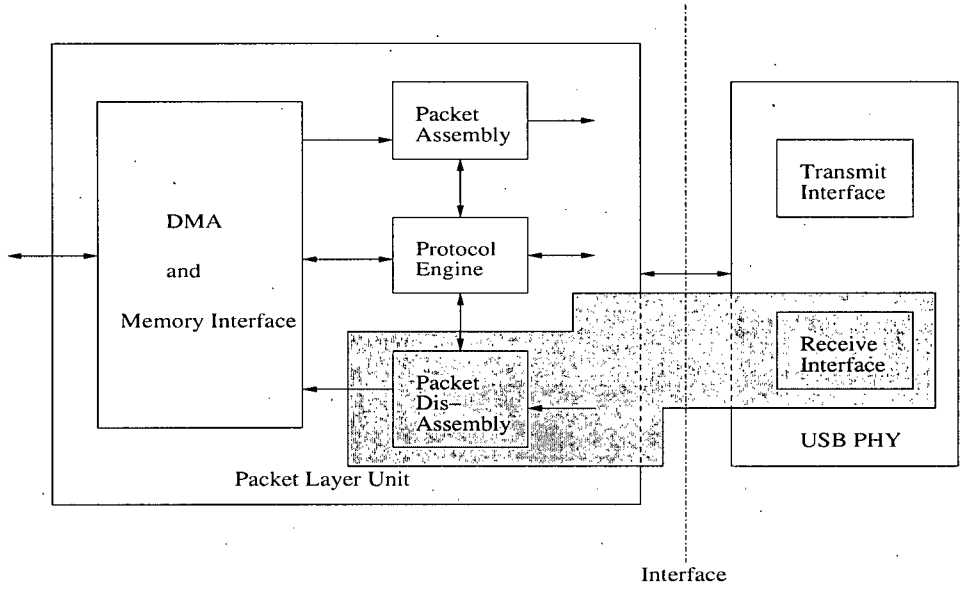


Figure 3.2: Design Under Verification. Shaded area represent the integration of two units from separate designs: the Packet Disassembly from the USB Function Core Packet Layer Unit and the Receive Interface unit from the USB PHY. The DUV is the integration of the Packet Disassembly and the Receive Interface units.

state visible at each cycle can disable some compiler optimizations, imposing a substantial slowdown.[2] Therefore, abstraction-guided simulation is useful only if the guidance is good enough to overcome the large overhead.

3.1.1 Varying Search Breadth

The most straightforward search strategy is greedy hill-climbing. From a simulation state s, we generate b possible next states 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 pre-images were perfectly accurate, then a

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

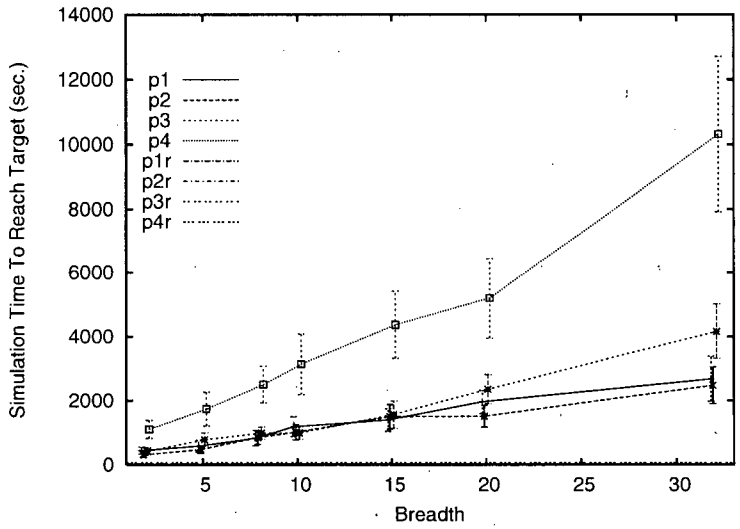


Figure 3.3: Mean Simulation Time for Varying Search Breadth. The overhead swamps the benefit of guidance and grows with breadth. Pure random simulation times for each property (denoted p1r, p2r, p3r, and p4r) average 29.1, 19.7, 27.1 and 67.9 seconds, respectively. The error bars show 95% confidence intervals for the true mean, based on Student's t-distribution.

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 computed the estimated mean, the estimated standard deviation and the 95% confidence interval for the true mean based on Student's t-distribution. We also ran conventional random simulation. We report the sample mean runtime for each experiment and the confidence interval for the true mean in Fig. 3.3. 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.

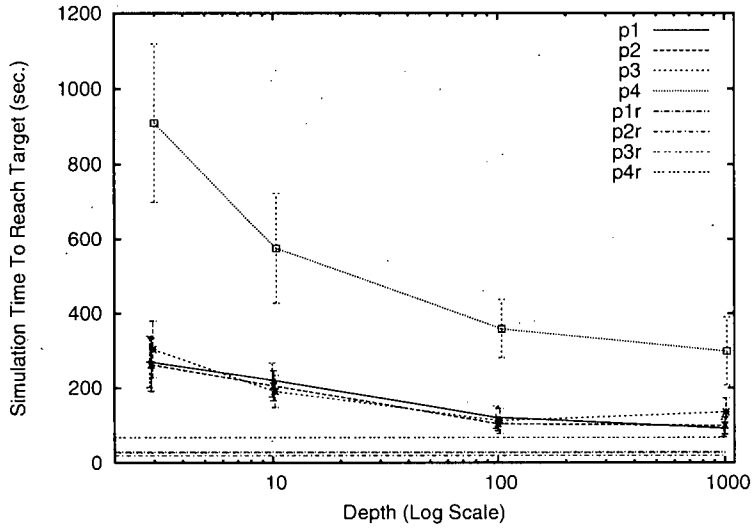


Figure 3.4: Mean Simulation Time for Varying Search Depth. As search depth increases, guided search becomes pure random simulation (whose results are as in Fig. 3.3), but with a constant factor overhead . The error bars show 95% confidence intervals for the true mean, based on Student's t-distribution.

3.1.2 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 (e.g., set $b = 1$ and vary d in Fig. 3.1). 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. 3.4 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 $d \rightarrow \infty$, 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. The results in Fig. 3.3 show that simulation

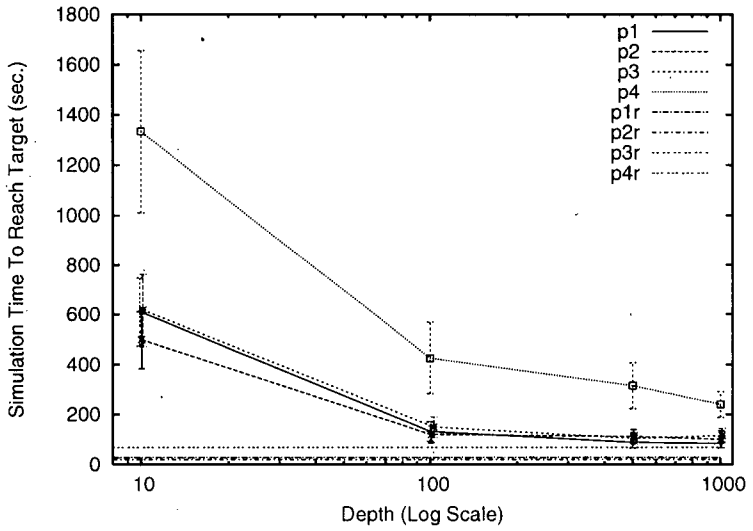


Figure 3.5: Mean Simulation Time Varying Search Depth with Breadth of 3. Combining breadth and depth does not help. The error bars show 95% confidence intervals for the true mean, based on Student's t-distribution.

runtimes are best when $b = 3$. We simulated 60 runs for each property, varying only d. Fig. 3.5 shows that the results are similar: breadth (which would help if the distance metric were perfect) imposes an $O(b)$ slowdown (vs. Fig. 3.4), and depth approaches a slowed-down version of random simulation as $d \to \infty$. The standard heuristics do not work.

3.1.3 GUIDO's SimSearch

To evaluate a sophisticated, state-of-the-art guidance heuristic, we tried out the search heuristic proposed in GUIDO [30]. The GUIDO verification tool contains two search modes: an abstraction-guided simulation mode *SimSearch* that fits the framework of this thesis, backed up by an exhaustive, formal, SAT-based procedure *SimSAT* for when SimSearch gets stuck.

Since the focus of this thesis is guidance heuristics, we implemented and evaluated the SimSearch algorithm, presented in Alg. 1. As in the original implementation, our SimSearch implementation explores a bounded breadth b and depth

Algorithm 1 Adapted GUIDO's SimSearch

```
 1: procedure SimSearch()
 2: CS = initial_state
 3: while (CS!=goal_state) do
 4:   loop b // breadth
 5:     curr_sample = sample_next_state(CS)
 6:     loop d //depth
 7:       if Cost(curr_sample) != Cost(CS)
            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 loop
13:   end loop
14:   if (priority_queue != empty)
15:     CS = priority_queue. head
16: end while
```

d from a given state, similarly to the heuristics in Sec. 3.1.1 and Sec. 3.1.2. However, SimSearch 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. The only difference between the original implementation and ours is that the simulation continues from the current state if the priority queue is empty (lines L14-L15, in Alg. 1). The description in [30] does not define what happens in this case.

In [30], 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. 3.6, 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.

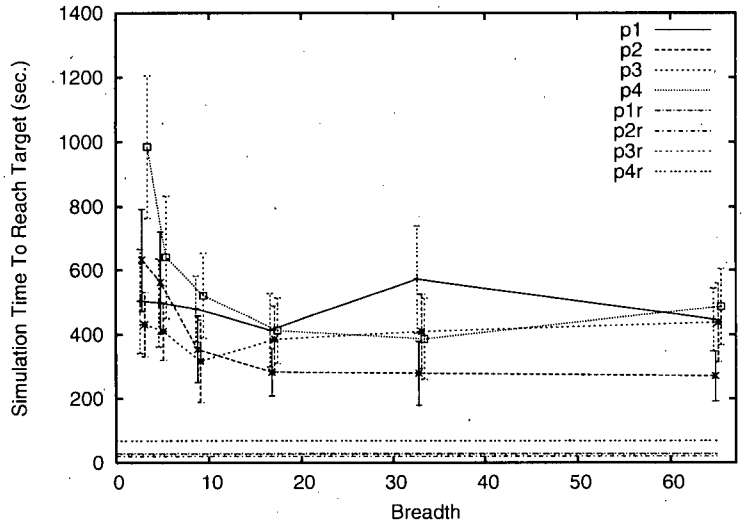


Figure 3.6: Mean Simulation Time for SimSearch. Even a sophisticated, recent heuristic loses to random simulation. We vary search breadth, with search depth fixed at 100. Error bars and random simulation times are as in Figs. 3.3–3.5. The error bars show 95% confidence intervals for the true mean, based on Student's t-distribution.

3.1.4 Hard Gains, Easy Losses

The intuition behind abstraction-guided simulation is that the 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. Recall that we simulated 60 runs for each experiment. I analyzed a sample of 9 runs within the confidence interval: 3 around the bottom, 3 around the estimated mean and 3 around the top. In all of them, the general behavior is very similar, although each trace is unique.

Fig. 3.7 is a typical trace for the experiments with $b = 1$ and $d = 100$. What is striking is how hard it is to make progress, but how easy to lose it. In Fig. 3.7, 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 1,000,000 cycles, which may or

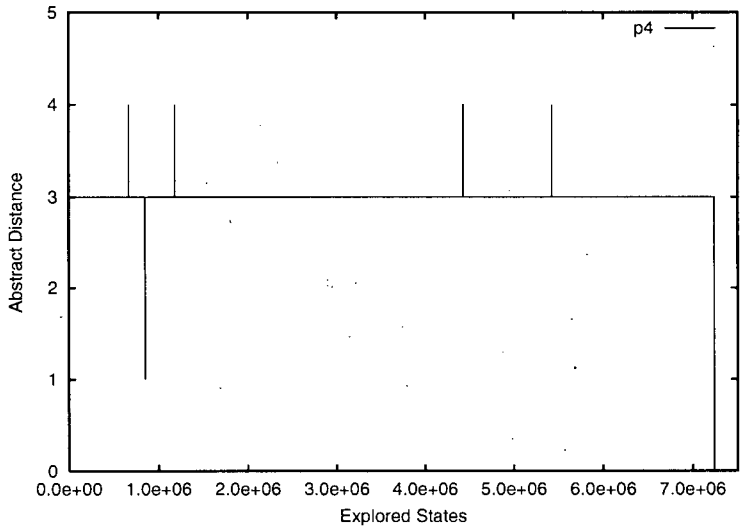


Figure 3.7: Simulation Trace using Depth of 100 and Breadth of 1.

may not have been a dead end, but then immediately gave up this progress for more than 6,000,000 cycles before finally succeeding. All of the traces we have plotted for previous heuristics are qualitatively similar. Even SimSearch produces a similar graph (Fig. 3.8). The challenge is to develop a heuristic that does not get stuck near dead ends, yet aggressively pursues promising states.

3.2 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,

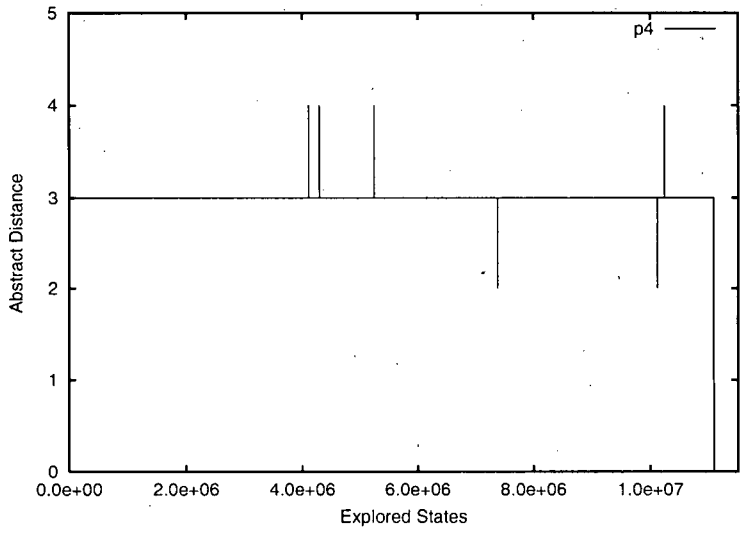


Figure 3.8: Simulation Trace using SimSearch with Depth 100 and Breadth 16. We see the same pattern of hard gains, easy losses.

e.g., Fig. 1.6 on page 13. 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

Algorithm 2 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
```

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 probability 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 2.

Fig. 3.9 shows a typical simulation trace with our new heuristic. This is for the same design and property as in Figs. 3.7 and 3.8, 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. This behavior can be seen

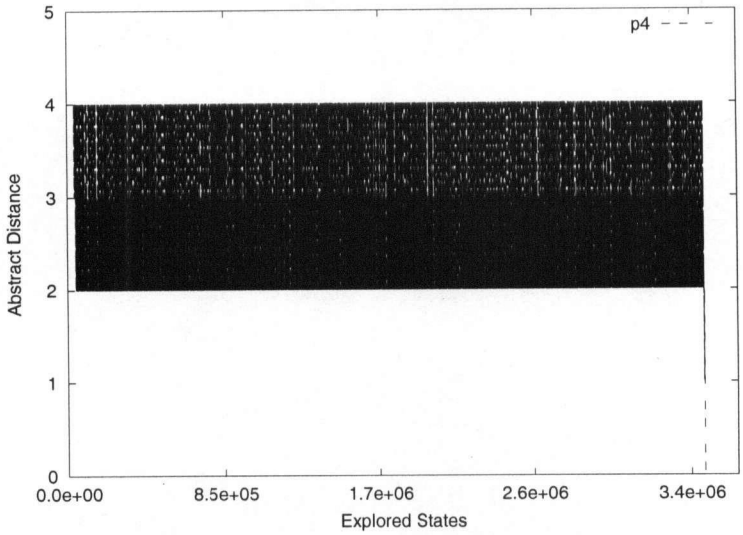


Figure 3.9: Simulation Trace using Algorithm 2 with Depth 100 and Breadth 1. The behavior is radically different.

more clear by zooming in on Fig. 3.9, as in Figs. 3.10 and 3.11.

It seems I have found a good guidance heuristic. I tried greedy heuristics, analyzed their behavior, understood the nature of the problem and devised a new, more effective heuristic. However, at this point, I cannot say I am done because these results are from a small DUV. Moreover, it is not clear if the parameters (i.e. b and d) used here would work well for other, larger DUVs. To get to a conclusive analyses, we needed to devise an experimental evaluation methodology and apply it to larger DUVs. This is what I describe in the next chapters.

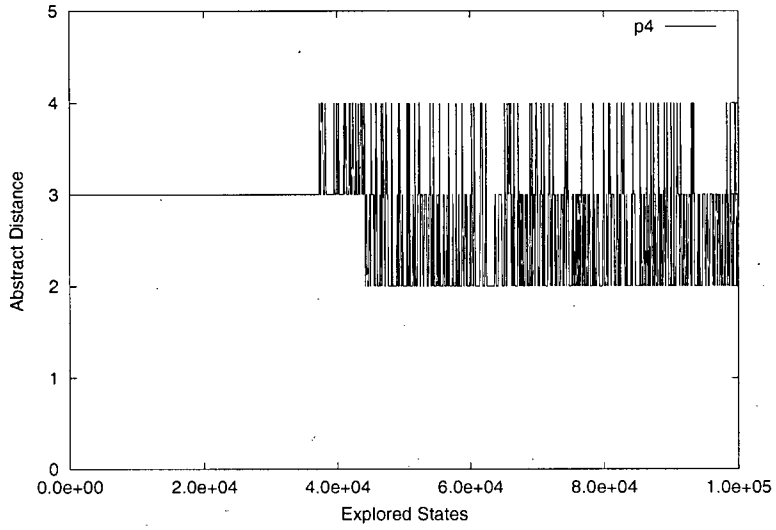


Figure 3.10: Enlargement of Part of (Fig. 3.9). First Thousands of Cycles. The path to a closer ring is through a ring farther away. States at closer rings are tried exponentially harder than states at farther rings.

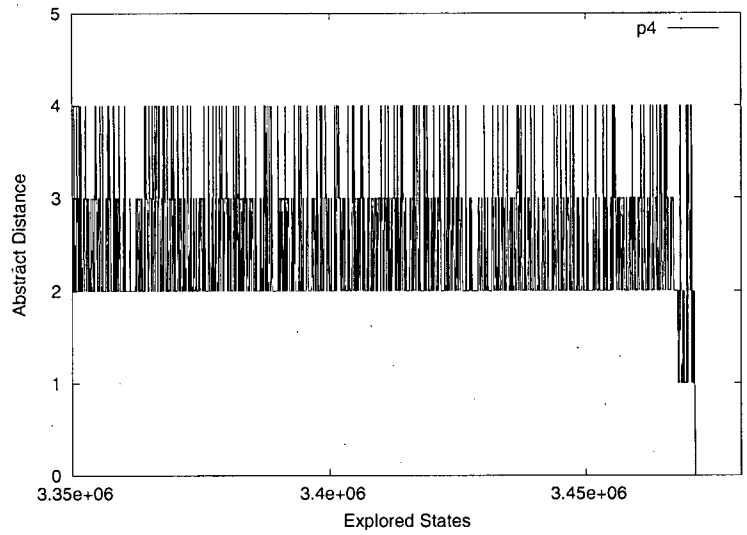


Figure 3.11: Enlargement of Part of (Fig. 3.9). Last Thousands of Cycles.

Chapter 4

Experimental Evaluation

4.1 Analysis 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 is not hit via random simulation, as formal is the only way to prove that a target is *not* reachable;
4. Abstraction-guided simulation is used only when simulation fails to reach the target, and formal verification fails to verify unreachability or generate a concrete trace to the target.

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.

We conducted our research using the EverLost platform described in the Chapter 2. 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 (I will define later which units we are using in this chapter), and the Ethernet MAC 10/100 Mbps designs from 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 and modified Verilog models are available at http://www.cs.ubc.ca/~depaulfm/EverLost.

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 tuned using one design and set of properties, then evaluate using a different version of the design and different properties, with no changes whatsoever to the heuristic. For easier reference, we use "training set" when we refer to the experiments described in Section 3.1, and "test set" when we refer to the experiments described in the next section. As a further test, we apply the identical heuristic to a completely different design, again with no further tuning. These results are reported in following sections.

Our new heuristic presented in Section 3.2 has only two parameters: *depth*, and *breadth*. From the experiments in Section 3.1, we selected *depth* and *breadth* to be 100 and 1.

4.2 The 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 Packet Disassembly unit, shown in Fig. 3.2 in page 19, 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?

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, shown in Fig. 4.1, consists of 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, shown as shaded blocks in Fig. 4.1, included only the Protocol Engine and the DMA and Memory Interface units, and had 397 latches, 170 inputs, and 159 outputs.

Table 4.1 presents the formal verification results. Both tools had trouble with the concrete design, but VIS was able to model check the structural abstraction for

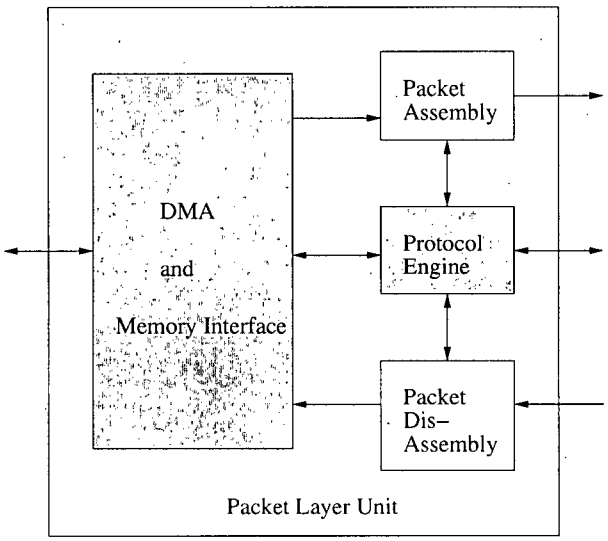


Figure 4.1: Test Set. Design Under Verification: USB Function Core Packet Layer Unit. The shaded blocks represent the abstract model resulting from manual structural abstraction.

all four properties. Because the structural abstraction also generated more onion rings, we used those results for the guided simulation runs.

Table 4.2 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.

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

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 4.2: Test Set: 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.

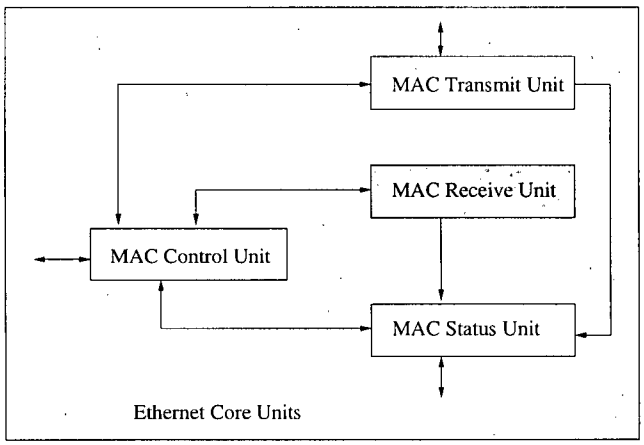


Figure 4.2: Case Study on Ethernet MAC 10/100 Mbps. These core functions represent this case study's DUV. The shaded block represents the abstract model resulting from manual structural abstraction.

4.3 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 4.1. The design is the Ethernet MAC 10/100 Mbps from OpenCores.org. The verification focused on the core functionality of the design, described by the following four units: the MAC Control, the Transmit, the Receive, and the Status units. Fig. 4.2 shows these core functions and their interconnect. 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.

To accommodate the Verilog limitations of VCEGAR and VIS, we modified the implementation by hand and used the equivalence checker tool from Synopsys (as described in Section 4.1) to prove that the modifications did not change the design's behavior. Then, we attempted to formally verify the remaining two properties. Both tools, VCEGAR and VIS, exhausted the memory available (memory limit was 800Mbytes). For VIS, we manually abstracted the design, selecting the

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

Receive unit to be the abstract model, since all the properties were related to this unit. In Fig. 4.2, the shaded block represents the abstract model. During the abstraction process, we realized a problem with the model: it had multiple clocks, i.e., the Transmit and Receive units can be configured at different clock rates. Unfortunately, neither formal tool supports this feature. We updated all four 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 abstract onion rings resulting from VIS to guide the simulation. We ran 30 simulations comparing random and guided simulation on these two properties. We report estimated mean for simulation runtime, the 95% confidence interval using the Student's t-distribution, and minimum an maximum simulation runtimes. The results in Table 4.3 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.

Chapter 5

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.[1] 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.

Another very challenging area for future work is to understand what types of abstractions better suit abstraction-guided simulation. During this thesis, we noticed that predicate abstraction would often generate a map too coarse to provide

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

good guidance. The main challenge is to understand the relationship, if any, between different designs (e.g. DSPs, processors) and different abstractions.

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.

Bibliography

[1] T. Ball, R. Majumdar, T. Millstein and S. K. Rajamani. Automatic predicate abstraction of C programs. *PLDI*, pp. 203–213, ACM Press, 2001.

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

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

[4] 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.

[5] 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.

[6] Cadence Design Systems. Incisive HDL Simulator. http://www.cadence.com/products/functional_ver/incisive_hdl

[7] 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.

[8] E. M. Clarke, E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs*, pp. 52–71, Springer, 1981. LNCS 131.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-Guided Abstraction Refinement. *CAV*, pp. 154–169. Springer, 2000. LNCS 1855.

[10] E. M. Clarke, M. Talupur, H. Veith and D. Wang. SAT Based Predicate Abstraction for Hardware Verification. *SAT*, pp. 78–92, Springer, 2003.

[11] S. Davis. Total Cost of Ownership: Xilinx FPGA's vs. traditional ASIC solutions (white paper). http://www.xilinx.com

[12] 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.

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

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

[15] 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.

[16] S. Graf, H. Sadi. Construction of Abstract State Graphs with PVS. *CAV*, pp.72–83, 1997. LNCS 1254.

[17] A. Gupta, A. E. Casavant, P. Ashar, X. G. S. Liu, A. Mukaiyama, and K. Wakabayashi. Property-specific testbench generation for guided simulation. *ASP-DAC and VLSID*, pp. 524–531. IEEE, 2002.

[18] 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.

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

[20] The Institute of Electrical and Electronics Engineers, Inc. Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Std. 1364-1995.
http://standards.ieee.org/reading/ieee/std_public/description/dasc/1364-1995_desc.html

[21] 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.

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

[23] R. P. Kurshan. Program Verification. *Notices of The American Mathematical Science*, pp. 534–545, Vol. 47, Number 5., 2000.

[24] Mentor Graphics Corporation. Modelsim SE.
http://www.mentor.com/products/fv/digital_verification/modelsim_se

[25] 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.

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

[27] A. Pneuli. The Temporal Logic of Programs. *Symposium on Foundations Of Computer Science*, pp. 46–57., 1997.

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

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

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

[31] Synopsys Inc. Smart RTL Verification. VCS. http://www.synopsys.com/products/simulation/simulation.html

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

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

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