Activity-based Refinement for Abstraction-guided Simulation

Debapriya Chatterjee, Valeria Bertacco
Department of Computer Science and Engineering
University of Michigan
{dchatt, valeria}@umich.edu

ABSTRACT

Semi-formal verification tools are gaining popularity because of their ability to balance the performance of logic simulators with the goal-focused capabilities of formal verification. Within this domain, abstraction-based simulation is a technique that has been proposed in several research works and has also emerged in a few commercial solutions. Abstraction-based simulation performs reachability analysis on a design abstraction to gather approximate information on the distance of each design state from a goal state, and then uses this information in a guided search by the logic simulator. Unfortunately, so far, the quality of the abstraction has been the weakest link in this semi-formal solution, because of its impact in enabling a simulator to reach a verification goal.

This paper presents a novel solution for abstraction refinement that operates in an abstraction-based simulation framework. Our solution collects switching activity information during simulation and determines how to modify and improve an abstraction based on analysis of this information. By using refinement, the original abstraction crafted by the tool is no longer a critical aspect of the semi-formal search. Instead, initially the abstraction may be weak, improving over time to enable the simulator to reach the goal state.

1. INTRODUCTION

Verification is one of the most demanding and time consuming tasks in commercial digital design processes. Moreover, current trends of increasing design complexity and shrinking times-to-market further exacerbate the situation, with the result that it is becoming commonplace to exclude some components or features in a system under development because of the scarcity of verification resources. It is estimated that in the near future verification will become a "show-stopping barrier to further progress in the semiconductor industry"[13]. Two main families of techniques are used to perform verification: simulation-based and formal methods. Simulation-based verification is most widespread in the industry, because of its linear scalability with design complexity and ease of use. However, this methodology requires much human effort in generating meaningful direct and random tests, and produces very low coverage, because only those execution scenarios that can be reproduced in a test can actually be validated [2, 20, 26]. On the other end of the spectrum, formal techniques are capable of guaranteeing that a design satisfies (or not) specific functional properties by means of mathematical derivations, without requiring to recreate all the relevant execution scenarios. The downside of these formal techniques is that they have very limited scalability, due to the computational complexity of the algorithms involved, and thus are impractical for any industrial-size design [14, 8]. Because of the promising high-quality coverage that formal techniques can provide, much effort has been dedicated in the past few decades to boosting their scalability, through more scalable underlying solver engines, better methodologies and abstraction techniques [12, 7], which allow the formal tool to operate on a small (and manageable) abstract version of the design under verification, and then reconnect the outcomes of the analysis to the original complex design. One important benefit of abstraction techniques in formal verification is that, if a property holds in the abstract design, then it also holds in the original design. However, if the property fails in the abstract design, it may or may not hold in the original one: hence, the state-of-the-art in formal verification is much stronger in proving valid properties than in disproving invalid ones.

Another promising direction explored in the past few years is that of semi-formal verification solutions: techniques that integrate simulation with formal approaches in an attempt to boost both the scalability of the latter and the coverage of the former [12, 10, 15, 21]. Semi-formal solutions employ a range of mechanisms to integrate the two families of approaches: from time-sharing, to controlled symbolic simplification (down to the Boolean values of logic simulation), to guided search, where simulation is used for coarse global search and formal techniques for fine local analysis, or vice versa. Simulation-heavy guided-search techniques have been often more successful because they are more scalable and can be applied to complex designs, and in some cases have become commercial electronic design automation tools. Their main weakness remains their limited ability to disprove false properties and provide the user with a bug trace (as mentioned earlier, purely formal techniques are often already capable of proving valid properties).

In this paper, we propose a novel solution to address this weakness of guided-search semi-formal tools. Our solution improves the quality of abstract design models used in guiding the simulation search, so to improve the capability of this family of semi-formal methods in disproving false properties. We developed an abstraction-refinement solution, inspired by the corresponding techniques common in purely formal methods. However, note that our refinement mechanism cannot leverage the formal models used in traditional refinement methods, since these are not available in this context. In contrast, we must leverage logic simulation activity to refine the guiding abstract model, that is, to determine which components should be retained in the abstraction, and which should be replaced. Our experimental results indicate that activity-based abstraction refinement is effective in boosting the convergence of semi-formal search solutions by enabling them to reach a given "goal" state (a state disproving a property) in many fewer simulation cycles than pure abstraction-guided hybrid solutions. More importantly, we observe that, in many cases, traditional hybrid solutions often fail to reach the goal, particularly for properties that involve hard-to-reach states, while our technique is capable of refining the abstraction even for these challenging situations leading the simulator to the goal.

1.1 Contributions

This work proposes a novel abstraction-refinement technique for simulation-centered semi-formal verification solutions. We considered the family of semi-formal solutions where random simulation operates in symbiosis with a formal search: the simulator traverses

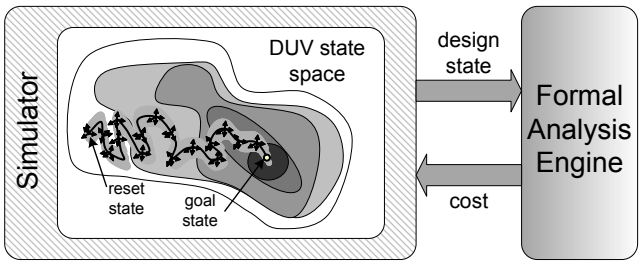

Figure 1: High level overview of an abstraction-guided simulation solution where a simulator communicates with a formal analysis engine providing data on the design's current state and receiving feedback on its progress towards a goal state in the form of a cost value, or approximate distance from the goal.

the design's state space with random, legal input stimuli. The objective of the simulation is to reach a "goal state" (possibly one in a pool of goal states). A goal state can be specified in many ways; typically, it is a state, or states, that falsify a property under verification. It is also possible that it represents states to be reached to improve verification coverage. In order to guide the simulation towards the goal state, at each simulation step, the state of the design is transferred to a formal analysis engine. This engine, in turns, computes a cost function based on the state received and communicates it back to the simulator. Based on the cost, the simulator may choose to transition to the new state, if it is an improvement towards the goal state, or backtrack, if the new cost indicates that the search is moving away from it. A high-level schematic of this approach is showed in Figure 1.

Several solutions have been proposed in this area, using different cost functions – such as hamming distance, abstract reachability analysis, *etc.* – and different algorithms to guide the "walk" of the simulation towards the goal – ranging from hill climbing, simulated annealing, and many others [9, 21, 10]. In addition, in some cases, the formal engine does not compute a new cost at each simulation step, particularly if the computation is time consuming, instead it is called less frequently. For this work, we assume a formal engine based on abstract reachability analysis, that is, the engine computes the cost function based on the distance between the present state and the goal state. Distances are computed as minimum vertex distance on an abstract design model, represented as a finite state machine (FSM). Due to the abstraction process, this FSM may contain spurious transitions, which may lead to myopic cost values returned to the simulator. Consequently, the simulator may hill climb into an apparent local minimum, from which it may not be able to reach the goal state.

Our technique provides the ability to refine the abstract model by collecting and analyzing simulation activity data on selected design signals. An abstraction may be refined several times during a simulation, until a proper and sufficient set of components are selected for it, so that the cost function becomes sufficiently accurate to lead the simulator to the goal state. Note that the accuracy of the cost function is defined with respect to the goal state under study.

We evaluated our solution by using a semi-formal verification framework with a logic simulator and a reachability analysis engine operating on an abstract FSM of the design, and we compared the quality of the results of the system with and without the addition of our refinement solution. We found that signal activity is a good indicator of what portions of the design must be modeled more accurately in the abstraction, and we show that our refinement technique leads to verification runs requiring up to 25 times fewer simulation cycles to converge to the goal state. For a number of testbenches, a goal state could only be reached when our proposed refinement scheme was activated, while the fixed abstraction solution failed to converge within few hours.

2. BACKGROUND AND RELATED WORK

Formal hardware verification solutions have been the focus of much research since the inception of digital design. In a classic model checking framework [6, 5, 8, 3], for instance, a user would specify a formal property to be validated, and the tool then checks that, for any state in which the design may operate, the property holds. In practice, formal algorithms usually operate by checking the dual of the property, that is, that there is no design state for which the property does not hold. If such a state exists, then the property is deemed false and a sequence of input patterns, called a bug trace, that leads to the falsifying state is provided to the user. By studying the bug trace, the user should be able to identify the source of the problem and correct the design. While successful in many aspects, often these techniques are limited in the complexity of the design that they can tackle.

To address the scalability limitation of classic formal verification solutions, several hybrid techniques have been investigated in recent years, both integrating a diverse pool of formal verification techniques, as well as proposing the cooperation between formal methods and a logic simulator. A common technique in these hybrid solutions involves constructing an abstraction of the original circuit and applying formal analysis only to the abstract model, thus circumventing the complexity wall. However, the abstraction process does generate spurious transitions in the abstract model; as a result, often bug traces assembled in the abstraction might not be transferable to the original circuit. To determine if an abstraction bug trace, also called abstract counter-example (ACE), can be transferred to the original design, the ACE is simulated in the original system and the final state reached should falsify the property under consideration. Often, however, this transfer fails to work, and a different abstraction must be created, in the hope to obtain either a proof of validity, or a transferable bug trace.

Recently, automatic techniques to refine abstract formal models have been proposed. In [7], the authors present a solution to analyze a non-transferable ACE to determine how to expand the abstraction (by including additional storage elements) so that to improve the quality of the bug trace obtained from it. Note, however, how this basic approach may ultimately grow the abstraction until the formal search is no longer computationally feasible, without converging on a determination for the property. Several works have strengthened and expanded on this idea, through improved analysis of the ACE, often employing other formal tools [4, 11, 24, 17, 27].

As an alternative to purely formal verification solutions, semi-formal, or hybrid, techniques have been investigated in the past decade, also with the purpose to boost scalability, while still benefiting from the high quality of results achievable by formal tools. The common trait of these solutions is the use of a logic simulator to support the search in the original digital system, and working in cooperation with one or more formal engines either operating on an abstract model or on a small portion of the system. Examples in this area include [30], where goal states are "enlarged" by backward traversal in the hope that they become easier targets for simulation; [10] strives to simplify the formal analysis by relying on hamming distances as a metric to determine the simulator's distance to the goal state. This research direction later attempted to strengthen the distance metric by complementing the search with automatically-generated "lighthouses", intermediate goals to guide the simulator towards the final goal state [29]. In [16], probabilities are assigned to design states, indicating if they are part of paths reaching the goal states; however this solution suffers from abstrac-

tion approximation in assigning probabilities to individual states, and may consequently lead the simulator to dead-end regions.

Within the family of semi-formal techniques, one group, which we call *abstraction-guided simulation*, uses a logic simulator to explore the design, while interacting with a distinct formal engine to gather an approximate measure of the search progress. For instance, [10] uses hamming distance computation as its low-cost formal engine; the distance metric in [21, 9] is derived as the vertex distance in an abstract FSM of the design under verification: this metric is much more accurate, but also more expensive, since distance computations must leverage a reachability analysis algorithm. The Everlost platform used in [9] attempts to improve the simulation guidance strategy, however it does not address the problem of deriving a useful abstraction automatically.

Both the latter two solutions complement the approximate distance metric with clever heuristics in the simulation walk through the design state space, including variants of hill climbing, backtracking, simulated annealing, *etc.* The selection of the components to be used in the abstraction is also varied, from individual register selections in [7, 4], to design modules in [21], to the use of data mining and cultural algorithms in [28]. Dynamically switching abstractions to provide simulation guidance for hard to reach states has also been considered in [19].

The common goal of all the solutions discussed is to boost the complexity of designs that can be tackled by formal verification. However, on one hand, abstraction-refinement in the context of formal tools provides only partial improvements. On the other hand, abstraction-guided simulation can only use statically generated abstractions, without refinement, which are often too inaccurate to reach the goal state. In this work, we propose a novel abstraction-refinement techniques for abstraction-guided simulation, which analyzes simulation data to craft an improved abstraction. In addition, our solution allows to both expand and reduce the original abstraction, thus it does not suffer from the growing complexity of traditional abstraction-refinement solutions.

Other solutions complement the simulation-based effort with additional formal engines, for instance to prune the search space, by proving that a set of states cannot be reached in the design [12, 18], or to coordinate the search at a high level [1]. The goal is to further boost the size of designs that can be tackled, and these solutions have reached a sufficient scalability level to become commercial tools and/or the mainstream solution within a large company. Our solution is complementary to them, and can be deployed within these multi-engine tools to further boost coverage and scalability.

Finally, properties in the frameworks discussed can be specified in several forms: using a formal property specification language, or a hardware description language. Typically the property would be translated into a small circuit sharing signals with the design under verification. One or more states of this circuit would correspond to the property being false. Thus, the design and the property could be treated uniformly by the formal analysis solution, and the goal of the tool would be to attempt to reach one of the falsifying states in the "property circuit". These states are also called "goal states". A very similar framework allows the user to directly specify the goal states of interest, having the formal tool simply target the generation of traces leading to them. This framework is valuable in scenarios where a verification team wants to boost the design coverage and requires a formal solution to target hard-to-reach states that plain logic simulation has failed to cover.

3. OVERVIEW

The abstraction-refinement solution that we developed in this work operates on a framework of the type outlined in Figure 1:

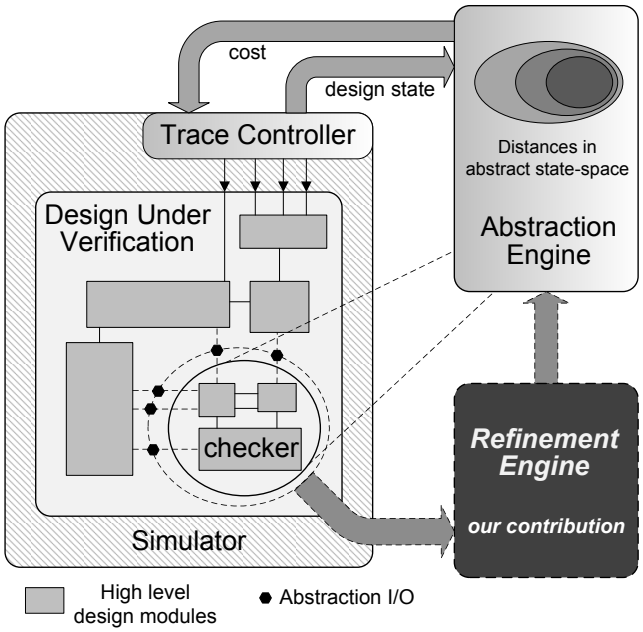

Figure 2: An abstraction-guided simulation framework augmented with a refinement engine. The system comprises an abstraction engine, computing an abstraction on a design subset and providing cost information to a trace controller. The latter resides within the simulator and it is responsible for guiding its progress towards the goal state. Our novel contribution, the refinement engine, collects simulation data at the periphery of the abstraction and communicates to the abstraction engine which components to add/drop to refine the abstraction.

a simulation engine navigates through the design's state space and interacts at regular intervals with a formal analysis engine to receive updates on its progress towards the goal state. Since FSM-based abstractions have shown more promising results than others, we assume that the formal engine computes a distance metric from the goal state by performing reachability analysis on an abstract FSM of the design. Figure 2 shows a more detailed schematic with the addition of our contribution highlighted. The *abstraction engine* operates on a small set of components (modules or sequential logic portions) – the abstraction – of the design under verification, which is undergoing constrained-random simulation. This engine computes a cost function by applying reachability analysis on the abstraction and can be queried dynamically during simulation to report on the cost of the current design's state. The logic simulator includes a *trace controller*, responsible for guiding the simulation towards the goal state, based on the cost information received by the abstraction engine. The trace controller may employ a range of robust search algorithms, such as hill climbing, backtrack, simulated annealing, *etc.*, to compensate for the approximate distance measures provided by the abstraction engine.

Our contribution, highlighted in the picture, is the *refinement engine*, which monitors dynamically the activity at the periphery of the abstraction components and, after internal analysis, provides feedback to the abstraction engine on how to refine the selection (that is, which additional components to include, and which to exclude). The refinement engine collects data continuously, however, the refinement is triggered at a coarser granularity, for instance after the cost has not improved for a certain number of simulation cycles, or if the goal state cannot be reached within a certain time. Below, we discuss the structure of each of the components in this system, while Section 4 is devoted to the refinement engine architecture.

3.1 Abstraction Engine

The abstraction engine is responsible for computing the cost function used to communicate to the simulator the current distance to the goal state. Many metrics have been proposed for this module, however, abstractions based on approximate reachability analysis tend to provide more accurate results. In our framework, the abstraction engine is responsible for selecting a portion of the design, either critical design modules, or a set of critical latches with some or all of their fanin cone of logic, and create an internal representation of the corresponding FSM (typically through Binary Decision Diagrams). Then, this engine performs backward reachability analysis [8] on this FSM, thus classifying each state in the abstraction based on their distance to the goal state. Note that it is advisable to include the logic corresponding to the checker or property in the abstraction, in order to achieve a viable abstraction. Moreover, note also that the abstraction process, that is, the selection of just a few design components instead of the whole design, affects the accuracy of the distances computed so that design states may appear to be closer to the goal than they really are. Unfortunately reachability analysis is computationally demanding, thus this engine is forced to only rely on a very small design abstraction (50 to 100 storage elements) to compute the cost function.

Abstraction engines for abstraction-based simulation solutions select the components to be included by performing a static analysis of the design. For instance, a set of storage elements that are closely affecting the checker logic, or the set of design modules that directly interact with the checker logic, *etc.*. Since the selection is based on a static analysis, there is a high risk that the abstraction is not the best suited for the goal of the verification task at hand.

3.2 Trace Controller

The trace controller is responsible for guiding the logic simulator towards an assigned goal state. It does so by applying a range of informed search algorithms over the design states that it observes in simulation. The trace controller's search relies on the cost information that it receives from the abstraction engine upon communicating to it the present design state. A trace controller would store aside a number of states recently visited with their relative cost and, at each simulation step, either force the design in one of these states, or allow the simulator to apply a random move and visit a new state. A variety of search algorithms may be deployed in the trace controller, usually including some heuristic aspects. In addition, a trace controller must compensate for the approximate cost measure that it receives from the abstraction engine, and incorporate the possibility of backtracking from local minima and/or cycles different from the goal state.

3.3 Refinement Engine

The addition of a refinement engine to this framework is the main contribution of our work. The purpose of this engine is to provide feedback on the quality of the current abstraction and suggestions on how to improve it by adding and/or removing some of its components. One of the main limitations of current abstraction-based simulation tools is in the quality of the abstraction: a poor selection may make it impossible for a trace controller to guide the simulator to the goal. At the same time, because the size of the abstraction is extremely limited, it is challenging to construct one that is both small and accurate, simply based on a static design analysis. Thus, the refinement engine that we propose analyzes simulation data dynamically and can provide feedback to the abstraction engine at regular intervals, indicating which components are most critical for the abstraction selection. The abstraction engine, in turn, can update its selection and recompute the FSM representation and the cost function on the new abstraction. Thus the refinement engine acts as a feedback loop in the system by evaluating the quality of an abstraction and suggesting possible improvements. Note that, after each abstraction update, the cost information contained in the trace controller must be reset, however, the simulation does not need to be re-initialized and may continue from the present state.

The introduction of a refinement engine, not only enables better abstractions, but also allow to attain goal states that may be reached only through a series of different abstractions, operating at different distance ranges from the given goal.

4. ABSTRACTION REFINEMENT

As discussed in the previous section, the task of the abstraction engine is to partition design states into equidistant 'bins', each collecting all the states with the same cost, that is, at the same distance from the goal state (an intuitive sketch of these bins is shown in the left part of Figure 1). The ideal abstraction engine is one that creates very fine granularity partitions, so that a simulator can leverage a high spread among the costs of states it can move into at each step of the search. Note, however, that the ideal abstraction is only required to provide fine granularity partitions for design states in the *surrounding of the simulator search path*. In other words, during a successful simulation, the design transitions from state to state, along a trail starting from an initial state and ending in a goal state. At each step, the system considers several candidate states that can be reached in one step from the present state and then selects the most promising option. If the abstraction engine can provide fine-granularity distance information for the states in the surrounding of the trail that the simulation is tracing, then the trace controller may apply good next-state selections and reach the goal quickly.

The purpose of our proposed refinement engine is to achieve an abstraction as close to the ideal one as possible, throughout the simulation. In our framework we assume that the building blocks of the abstraction are individual design modules. A module is a small portion of the original design, comprising both combinational logic gates and sequential elements. An individual module may be as small as a single gate, and possibly smaller than the permissible size of the entire abstraction, so that there is flexibility in choosing a number of modules to be part of the abstraction. The permissible size of the abstraction depends on the computational capabilities of the reachability engine and on the time that a user accepts spending in computing abstractions vs. simulating the design. To build an abstraction, a number of modules are selected within the design; if the goal state is specified through a checker module, it should always be included in the abstraction, so that the latter always contains at least one state at cost 0 (the goal state). All together these modules should not exceed the permissible size of the abstraction.

The main task of our refinement engine is in observing the signals at the boundary of the abstraction modules during simulation. The insight is that the goal state can always be reached in the abstraction because its inputs are free (that is, they can assume any value during each simulation cycle), while in the original design, the behavior of these signals is constrained by the other modules of the design that are not part of the abstraction. Thus, the refinement engine monitors these signals and tags those that show no or limited switching activity during the abstraction-guided simulation. If those *low activity signals* were to switch more frequently, it would be easier to generate any set of input sequences at the abstraction boundary, including sequences necessary to reach the goal state.

The refinement engine thus proceeds as follows:

1. During simulation collect switching activity counts for each input signal at the boundary of the abstraction logic.

2. When a refinement is deemed necessary (this is discussed in detail in Section 4.2) the refinement engine computes the average switching activity for each design module that i) has outputs connected to the abstraction logic and ii) it is not part of the abstraction logic. The average is computed over the number of signals that are connected to the abstraction.

3. The module with the lowest switching activity is added to the set of modules that are already in the abstraction. At this point the new selection is completed, the abstraction engine may recompute the new distance metric and then simulation regulated by the trace controller may resume.

We speculated that, if a module causing low switching activity were to be added to the abstraction, then the abstract engine would improve the accuracy and granularity of its distance metric for states affected by that module and, in turn, this would improve the quality of the trace controller guidance. In other words, the low activity signals capture the difference between the abstract behavior and the observed (simulated) behavior. The "quality" of an abstraction is judged primarily on how many distance bins it generates in the abstract state space: the finer the bin partitioning the more accurate the simulation guidance. During incremental refinement, if two distinct abstractions have the same number of distance bins, but the relative distribution of abstract states has changed, we consider it an improvement.

The following sections provide first an example of the operation of the refinement engine, then an algorithm to remove modules from the abstraction in order to allow new modules to be added without crossing the permissible size bound for the abstraction and a discussion of when a refinement should be triggered.

4.1 Example

We now present an example of abstraction refinement using the algorithm described. With reference to Figure 3.a, consider a design comprising three modules, *M1*, *M2* and *M3*. Figure 3.b shows the FSM diagram of all the three individual modules: $M1$ has two input signals, r and s and two output signals, p and q. *M2* has no input and one output, signal r. Finally *M3* has two inputs p and q, and one output, signal s. All the FSMs are Mealy machines, thus the outputs are a copy of the internal FSM's state. The initial state of the system is $pqrs = 0000$ and we want to reach one of the final states $pqrs = 11xx$ (x=don't care). Note how this design is a pathological case for sake of the example: since it does not take any external input, the trace controller does not have any impact on the progression of the simulation.

Let us assume that, at first, the abstraction includes only module *M1* of the design. Thus the abstraction engine creates four distance bins, each including one state, with the goal state at cost 0 and the initial state at cost 3. This is indeed an approximation since the goal state cannot be reached in just three simulation steps. After a few steps of simulation, say two or three, the refinement engine determines that the switching activity between the abstraction (module $M1$) and module *M2* is higher than between the abstraction and module *M3*. Indeed the only signal connecting *M1* and *M2* is r, and it switches at each simulation cycle, while signal s switches twice each time p and q differ. Thus it would recommend to include module *M3* in the abstraction, obtaining the FSM's shown in Figure 3.c. In this new abstraction, the goal state is at distance 5 from the initial state, as in the original given design. Thus this abstraction provides more accurate cost information along the path that the simulator is executing towards the goal state.

Note that, if the abstraction were to be refined using modules *M1* and *M2* instead, no improvement would be accomplished in refining the costs along the simulation path as shown in Figure 3.d. Finally, note also that the complete FSM of the design (comprising all modules) includes additional states, whose cost information is however irrelevant in attempting to reach the goal state.

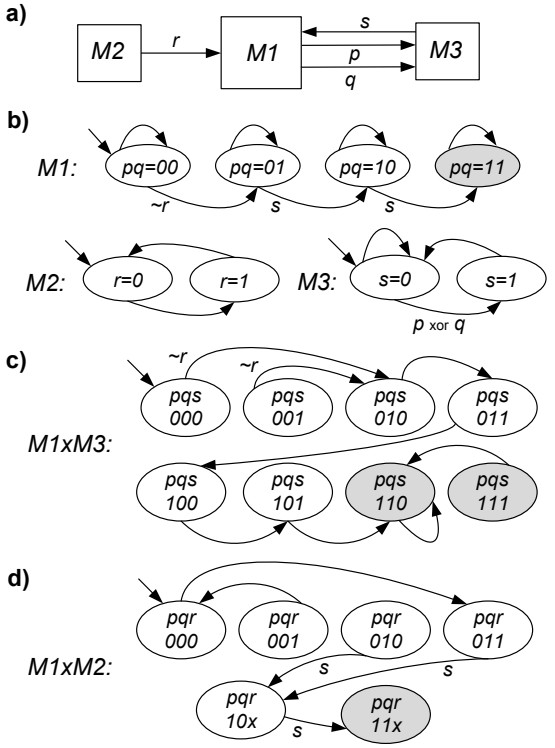

Figure 3: Example of abstraction refinement. a) The design for the example comprises three modules, *M1*, *M2* and *M3*. b) FSM diagrams for each of the design's modules. c) A good abstraction refinement by including module *M3*. d) A poor refinement choice by including module *M2*.

4.2 Refinement interval

Each time the abstraction is refined, a new set of distance bins must be created on the newly obtained FSM. This computation entails a fix-point backward reachability analysis and can be fairly time consuming, particularly with larger abstractions. One of the guidelines of abstraction-driven simulation is to have the simulator be the workhorse of the search and only use light-weight formal analysis. In practice, we found in our experimental evaluation that the time required by the abstraction engine to generate the distance bins corresponds often to the time it takes to execute several thousands simulation cycles.

Thus, we recommend invoking an abstraction refinement only infrequently, after the simulator has spent sometime attempting to reach the goal state and failed. One possible technique would entail keeping track of how many times the trace controller backtracks in the search, or how long it has stalled in equidistant states. A more lightweight solution simply triggers the refinement engine when the goal is not reached after a pre-set number of simulation cycles. In setting this value, several factors should be taken into account: the complexity of the present abstraction, the estimated distance of the goal state and also the effort (in cycles) spent in the current search.

The complexity of the abstraction directly impacts the time required to generate the distance bins via backward reachability analysis: usually there is an exponential relation between the number of latches in the abstraction and this time. It is recommendable to compensate for large formal computation times with at least

equally lengthy simulation times, to keep the simulator as the lead engine and because large abstractions usually generate a richer set of information that the simulator can subsequently explore. In addition, the simulator should also be given sufficient time to leverage the information available from the formal engine: if the new abstraction provides many distance bins between the present and the goal states, the simulator should be given more time to attempt to traverse through those bins, than if there were just a couple of bins to cross between the present state and the goal.

4.3 Module removal

In order to enable the refinement engine to suggest new module additions, while not crossing the permissible size bound in the abstraction, it is necessary from time to time to remove modules from the abstraction. The combination of module addition and removal allow for the abstraction to morph over time, adapting itself to provide the best cost estimates in the surrounding of the present design state, while the system progresses towards the goal state.

Ideally, the module to be removed should be the one contributing the least to the current abstraction's quality. However, this evaluation would require computing the cost function for many variants of the abstraction, each including all modules but one, to evaluate which is the best residual abstraction. Since this is impractical, we developed a heuristic, which evaluates the contribution of a given module to the overall abstraction quality when the module is added. Note that this is not necessarily the same as the quality detraction suffered when the module is eliminated a few refinements later.

Thus, at each refinement step, upon module addition, the improvement brought to the abstraction is logged. The improvement is evaluated in terms of number of distance bins generated and how drastically states are re-partitioned among the bins. This evaluation is used when a removal is needed, by selecting the module providing the least improvement. In case of a tie, the most recently added module is removed, since this would probably cause minimal disruption to the connectivity of the logic currently in abstraction.

5. CASE STUDY

We present here a case study where we applied our refinement technique on a small MSI (modified-shared-invalid) cache coherence protocol design. The design includes two local cache controllers (pcacheA and pcacheB), an arbiter and two processor shells, to emulate the processor interaction with the local cache. In addition, the setup for our abstraction-guided simulation requires a top level module to instantiate all the components, a checker module specifying the property to be falsified and a testbench module generating legal inputs for the design from a handful of random stimuli. The schematic of the MSI design's structure is shown in Figure 4.

At the beginning of the abstraction-guided simulation, only the checker module is included in the abstraction, as shown in the first frame of the figure. This module includes three latches and the cost function generated divides the state space into 4 distance bins. After some time, the abstraction is refined using the simulation data collected at the I/O of the checker module and one of the cache controllers is found to be that with the least average activity, thus it is added to the abstraction, which now includes 11 latches, but still divides the states space in only 4 bins. Successive abstraction refinements will add more modules and attain a finer cost function granularity until the goal state is reached after a total of 863 simulation cycles and 3 refinement steps. If abstraction refinement were not available, the goal state could only be reached in over 20,000 simulation cycles when relying on a fixed abstraction including only the checker module.

As we discuss later in Section 6.3, we also evaluated a dual refinement criteria to compare against our solution. This criteria will give priority of inclusion to modules that have the highest switching activity with the current abstraction. Figure 5 shows how this alternate refinement heuristic will require more simulation and refinement steps in order to converge.

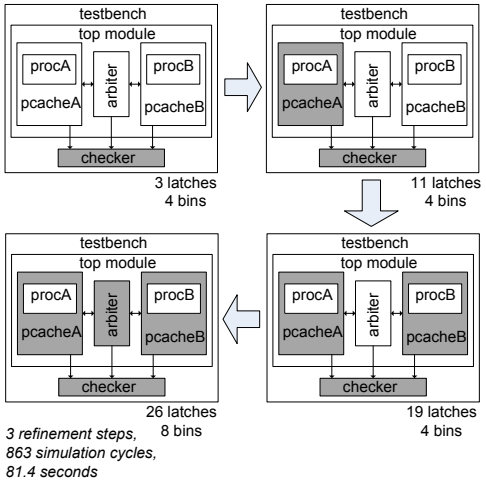

Figure 4: MSI design case study. The schematic illustrates the sequence of refinements required to reach the goal states using our refinement engine for the MSI design targeting property P2.

6. EXPERIMENTAL EVALUATION

We evaluated our solution by developing a baseline abstraction-guided simulator tool which relies only on static abstractions. Then we augmented this framework with our refinement engine. The static abstraction solution relies on Synopsys' VCS logic simulator [23], interfacing with a BDD-based abstraction engine that we developed in-house. The communication between the simulator and the BDD engine was via Verilog PLI callback routines. The abstraction engine includes an algorithm to select a few design modules, compute their product FSM, and then apply reachability analysis to the abstract design. In addition, it interacts with the trace controller by receiving the present simulation state and returning the associated cost. The trace controller incorporates also a number of search heuristics to guide the simulator based on the approximate cost function. The abstraction engine we developed collects switching activity dynamically from the simulator via a number of call-back routines and it selects the candidate with the lowest average switching activity at the periphery of the current abstraction upon request from the top-level tool's orchestration routine.

In the following sections, we present our testbenches and our experimental evaluation where we compare the performance of the baseline solution against the one equipped with the refinement engine. Finally, we present an analysis of an alternative refinement heuristic, in comparison to our solution.

6.1 Designs and properties

We evaluated the solution on a number of publicly available testbenches: a simple MSI cache coherence protocol from the VIS benchmark suite [25], discussed already in Section 5, and a number of design blocks derived from the Picojava processor by Sun Microsystems [22], including the full PicoJava design. For each design we crafted a number of properties to be proven false, and wrote a checker module describing them, or we used properties that were provided with the design. We also either used testbenches provided with the design or developed one in house to generate legal random stimuli for each design. Table 1 presents the characteristics of the

Design	in/out	FF	Gates	Properties
MSI	14/15	43	1,674	P1:wrongly evicting dirty cache line P2:not invalidating a shared cache line
BSI	84/62	108	4,778	P1:misaligned fill of I-cache P2:wrong handling of I-cache miss P3:wrong transition in I-cache fill FSM P4:wrong transition in SMU FSM
ICU	28/80	64	1,797	P1:misalligned fill of cache line P2:wrong fill of cache line on miss
PicoJava	44/76	3,646	189,895	P1:wrong transition in cache fill FSM P2:stall condition in SMU P3:misaligned fill in cache P4:stall in I-cache fill state machine

Table 1: Characteristics of the testbench designs and property descriptions

testbench designs and a brief description of each property. From the PicoJava processor, we derived the following testbenches: 1) ICU, including only the Instruction Cache Unit of the processor and 2) BSI, comprising the ICU, the Stack Management Unit (SMU) and the Bus Interface Unit(BIU). Finally we also used the full processor design as a testbench in itself.

6.2 Results

We applied abstraction-based simulation to all designs and properties and compared the performance of the solution in terms of 1) whether the tool could reach a goal state or not, 2) the number of simulation cycles required to hit the goal state and 3) the total wall clock time required to complete the analysis. For the refinement solution, we also report the time spent in refining the abstraction. The experiments run on a Pentium 4, Redhat linux system operating at 3.2Ghz and equipped with 2GB of memory.

Table 2 compares the performance and speed of convergence of the baseline solution, a static abstraction-based simulator, with our solution, which includes the refinement engine. The abstraction in the baseline solution includes only the checker module, while the refinement solution starts with only the checker in the abstraction, and then adds and/or removes additional modules over the course of the analysis. In all cases the experiment was repeated with 10 different random seeds, in the table we report number of simulation cycles required in the best case seed and the runtimes indicated are the average over all runs. Each row in the table corresponds to a single design/property pair. The second and third columns provide the simulation cycles and the total runtime required by the static analysis in seconds. It is worth noting that the static abstraction solution does not converge to the goal state for a number of testbenches, particularly those entailing more complex designs. We stopped the analysis after running 200,000 simulation cycles without success.

The following three columns report the same information and the time spent in our refinement engine. For these tests the refinement engine could only add modules to the abstraction until when the permissible size limit was reached. The three columns under the "Refinement with removal" header report the same information for a refinement engine that can both add and remove modules. Note how, for the full Picojava design testbenches, this feature was in some cases necessary in order to converge to the goal state. The remaining columns in the table will be discussed in the next section.

Overall we note from the table that for several complex designs, abstraction refinement is necessary in order to derive an abstraction of sufficient quality to converge to the goal state. In addition, in most situations, the use of a refinement engine allows to complete the analysis in many fewer simulation cycles than when using a static abstraction. For some of these situations, the overall wall clock time is worse when using the refinement engine because of the computation cost entailed by generating the new cost function.

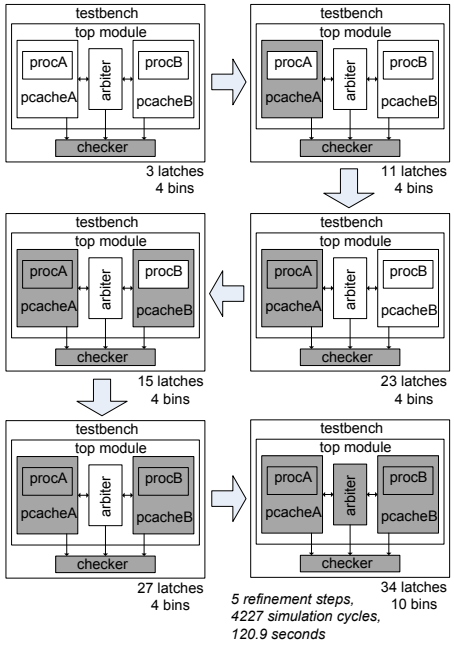

Figure 5: Refinement with the alternative heuristic. The schematic shows the refinement sequence for the same design and property as the case study, this time using the maximum-activity alternative refinement heuristic. Note how the systems requires more refinement steps to converge.

We derive from this observation that the refinement engine should be triggered less frequently in an abstraction-based run.

6.3 An alternative refinement heuristic

c In order to gain a qualitative sense of the soundness of our refinement heuristic, we implemented another variant of the refinement engine. In this variant, the selection of a module for inclusion in the abstraction is based on the maximum switching activity during simulation. That is, the module at the boundary of the refinement, which has the highest average switching activity with input signals of the abstraction, is the one to be added to the abstraction during a refinement stage. The progress of refinement for the MSI P2 testbench under this heuristic is shown in Figure 5.

The last six columns of Table 2 report the performance evaluation of this heuristic over all our testbench designs. As can be noted in the table, this heuristic provide markedly fewer benefits compared to our solution. Indeed, in all but one case, this refinement approach requires more simulation cycles and time than a minimum-activity heuristic. Note also that even when the module removal feature is included in the refinement engine, this solution cannot converge for the last two properties of the Picojava design.

7. CONCLUSIONS

In this paper we have presented a novel solution for refinement in abstraction-based simulation frameworks. Our solution determines how to refine an abstraction by analyzing observed switching activity at the abstraction boundary during simulation. Signals with low switching activity provide the main behavioral difference between the abstract and the original design, thus modules generating these signals are best candidate to be included in the abstraction.

Our experimental evaluation confirms that abstraction refinement greatly enhances the number of designs and properties that can be evaluated with an abstraction-based simulation framework. In addition, they show that our minimum-activity heuristic for refinement

Testbench	Static		Refinement add only			Refinement with removal			Alternative add only			Alternative with removal		
	cycles	runtime(s)	cycles	runtime(s)	refine.(s)	cycles	runtime(s)	refine.(s)	cycles	runtime(s)	refine.(s)	cycles	runtime(s)	refine.(s)
MSI P1	145	1.8	144	45.2	43.2	144	45.2	43.2	144	45.2	43.2	144	45.2	43.2
MSI P2	21717	26.7	**863**	**81.4**	**77.0**	863	81.4	77.0	4227	120.9	113.1	4227	120.9	113.1
BSI P1	11207	48.3	**1617**	**421.3**	**312.1**	1617	421.3	312.1	3191	456.5	373.2	6451	479.8	421.0
BSI P2	-	TO	17246	523.1	401.3	10945	551.7	456.3	**4693**	**401.3**	**342.1**	7893	446.3	391.6
BSI P3	404	2.98	138	10.45	9.1	138	10.45	9.1	138	10.45	9.1	138	10.45	9.1
BSI P4	59	1.44	59	1.44	1.02	59	1.44	1.02	59	1.44	1.02	59	1.44	1.02
ICU P1	13499	26.2	**7578**	**135.1**	**111.2**	17578	179.7	145.2	16109	157.3	112.4	39671	243.1	189.6
ICU P2	-	TO	176	10.1	8.9	176	10.1	8.9	176	10.1	8.9	176	10.1	8.9
PICO P1	-	TO	**2733**	**301.7**	**245.1**	3143	322.1	258.3	6733	345.4	278.3	7815	371.3	317.1
PICO P2	-	TO	**3789**	**311.3**	**267.1**	4562	337.5	278.2	21561	393.2	271.4	27567	423.7	334.4
PICO P3	-	TO	-	TO	-	**6148**	**894.5**	**789.1**	-	TO	-	-	TO	-
PICO P4	-	TO	-	TO	-	**41687**	**1674.5**	**1347.3**	-	TO	-	-	TO	-

Table 2: Performance evaluation. The table compares the performance of a static abstraction-based solution against our solution equipped with a refinement engine, with and without the capability of removing modules from the abstraction. The rightmost two sections of the table provide results using a dual heuristic in the refinement engine. Results indicate that the inclusion of a refinement engine enable many more test-cases to converge to the goal state and that a minimum-activity heuristic for refinement is almost always more successful than a maximum-activity one.

is promising for simulation-based refinement. Finally, our most complex testbenches benefit from the ability to remove components from an abstraction, to enable continued refinement without reaching an abstraction too complex for the analysis.

Currently we are working on evaluating our solution on more complex designs and analyzing simulation data to derive improved refinement heuristics.

8. REFERENCES

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

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

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. TACAS*, pages 193–207, London, UK, 1999. Springer-Verlag.

[4] P. Bjesse and J. Kukula. Using counter example guided abstraction refinement to find complex bugs. In *Proc. Design Automation and Test in Europe*, pages 156–161, Mar. 2004.

[5] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *Proc. Design Automation Conference*, pages 46–51, June 1990.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10e20 states and beyond. In *Proc. Logic in Computer Science*, pages 428–439, Jun 1990.

[7] E. Clarke and Y. Lu. Counterexample-guided abstraction refinement. In *Proc. Computer Aided Verification*, pages 154–169. Springer, 2000.

[8] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[9] F. De Paula and A. Hu. An effective guidance strategy for abstraction-guided simulation. In *Proc. Design Automation Conference*, pages 63–68, New York, NY, USA, 2007. ACM.

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

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

[12] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *Proc. ICCAD*, pages 120–126, Nov. 2000.

[13] International Technology Roadmap for Semiconductors. *http://www.itrs.net/*, 2007 edition.

[14] C. Kern and M. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[15] A. Kölbl, J. Kukula, and R. Damiano. Symbolic rtl simulation. In *Proc. Design Automation Conference*, pages 47–52, 2001.

[16] A. Kuehlmann, K. McMillan, and R. Brayton. Probabilistic state space search. In *Proc. ICCAD*, pages 574–580, Nov. 1999.

[17] K. Mcmillan and N. Amla. Automatic abstraction without counterexamples. In *Proc. TACAS*, pages 2–17. Springer, 2003.

[18] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *Proc. FMCAD*, pages 159–173. Springer, 2004.

[19] A. Parikh and M. Hsiao. On dynamic switching of navigation for semi-formal design validation. In *Proc. HLDVT*, pages 41–48, 2008.

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

[21] S. Shyam and V. Bertacco. Distance-guided hybrid verification with guido. In *Proc. Design Automation and Test in Europe*, pages 1211–1216, 2006.

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

[23] Synopsys, VCS. *http://www.synopsys.com/tools/verification/functionalverification/pages/vcs.aspx*.

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

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

[26] I. Wagner, V. Bertacco, and T. Austin. StressTest: an automatic approach to test generation via activity monitors. In *Proc. Design Automation Conference*, pages 783–788, 2005.

[27] D. Wang, P.-H. Jiang, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proc. Design Automation Conference*, pages 35–40, 2001.

[28] W. Wu and M. Hsiao. Efficient design validation based on cultural algorithms. In *Proc. Design Automation and Test in Europe*, pages 402–407, March 2008.

[29] P. Yalagandula, V. Singhal, and A. Aziz. Automatic lighthouse generation for directed state space search. In *Proc. Design Automation and Test in Europe*, pages 237–242, Mar. 2000.

[30] C. Yang and D. Dill. Validation with guided search of the state space. In *Proc. Design Automation Conference*, pages 599–604, June 1998.