An Abstraction-Guided Simulation Approach using Markov Models for Microprocessor Verification

Tao Zhang†‡, Tao Lv§, Xiaowei Li†*
†Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China, 100190
‡Graduate University of Chinese Academy of Sciences, Beijing, P.R. China, 100039
{zhangtao, lvtao, lxw}@ict.ac.cn

Abstract—In order to combine the power of simulation-based and formal techniques, semi-formal methods have been widely explored. Among these methods, abstraction-guided simulation is a quite promising one. In this paper, we propose an abstraction-guided simulation approach aiming to cover hard-to-reach states in functional verification of microprocessors. A Markov model is constructed utilizing the high level functional specification, i.e. ISA. Such model integrates vector correlations. Furthermore, several strategies utilizing abstraction information are proposed as an effective guidance to the test generation. Experimental results on two complex microprocessors show that our approach is more efficient in covering hard-to-reach states than similar methods. Comparing with some work with other intelligent engines, our approach could guarantee higher hit ratio of target states without efficiency loss.

1. Introduction

Functional verification has been a critical factor in the design flow because of the rapidly growing scale and complexity of modern chips. This is especially true for microprocessors, considering their complex micro-architectures. Although the simulation-based approach remains the predominant verification method, it has limited performance in many cases, for example in covering hard-to-reach states (e.g. assertion violations and coverage holes) of designs. Formal techniques are competent to analyze and verify all corner cases, but it is hard for them to tackle the complexity of large designs.

In order to combine the advantages of these two methods and mitigate their weaknesses, researchers have proposed several semi-formal verification approaches [1-7]. Among them, the abstraction-guided simulation is a very promising one, in which partial design information is calculated on an abstract model, and then is used to guide the simulation towards a target state.

The work in [3] is pioneering in abstraction-guided simulation, followed by [4-7]. The main points of these techniques are the abstraction engine and the test generation engine with a guidance strategy. GUIDO presented in [3] is a fundamental framework, in which modules closely interacted with the target property are extracted as an abstract model, and then abstract distances are obtained by computing pre-images of the target state on the abstract model. Such abstract distance information is used in the guiding principle of test generation. But GUIDO employs a simple random test generation engine, so it is easy to get stuck at dead-end states (states seeming close to the target in the abstract model, but actually not) due to the abstraction deviation. The authors of [4] introduce abstraction refinements to alleviate the dead-end problem. The abstract model is iteratively refined to extract more design information until the target state is reached in simulation. However, the computational cost of iterative abstraction refinements is so expensive that this technique cannot be widely applied to verification of complex designs. Besides above efforts on abstraction strategies, an efficient test generation strategy is desired to effectively avoid getting stuck at dead-ends and hence to direct simulation to the target.

In [5], a more global guidance strategy for simulation is proposed, which achieves a better balance between searching and backtracking. Previously visited states are saved in “buckets” for each abstract distance (each onion ring). Before each clock cycle, a new "start state" is selected from one bucket with an exponentially decreasing probability, from the closest (the lowest distance value) bucket to the outermost one. However, covering hard-to-reach states still requires too much time in this approach due to its simple random test generator.

Clearly, a random test generator can be replaced with a more intelligent engine. The work in [7] incorporates cultural algorithms (CA) to improve test generation efficiency. Data mining is adopted to extract abstract partitions, and the obtained abstract distance information is used in the fitness function to evaluate “individuals” in the CA. This approach performs well, but it is not fit for functional verification, since vectors are mined at bit-level and functional invalid vectors may be generated. In addition, it is difficult for it to mine the complex relationship among bits in a vector, considering that the number of bit-combinations grows exponentially. For a microprocessor with 32-bit or 64-bit instructions, such bit-level optimization techniques are not appropriate.

1.1 Motivation of Our Approach

It is very important for functional verification to efficiently generate test cases for hard-to-reach states. It is especially difficult for microprocessor verification. Such test cases can be written manually, but it costs too much expert time and these cases are hard to be ported between different projects. Existing approaches incorporate either a random test generator with a guidance strategy [3-5], or an intelligent test generator such as
using evolutionary algorithms [6, 7]. The former is too simple to achieve high efficiency, while the latter is unfit for functional verification. We believe that machine learning based techniques, such as Markov model based ones, are quite suitable for this purpose.

The Markov model based strategies have shown success in several prior ATPG works [8, 9]. In this paper, we present a Markov model based abstraction-guided simulation approach. Our approach has several major advantages as follows.

1) The design specification is utilized to facilitate the establishment of the Markov model, which make it easier to produce valid functional vectors and to model test vector correlations.

2) Some heuristic strategies are proposed for simulation guidance, which consider rich information, including abstract distances, test vector correlations, and so on. These strategies dynamically adjust the Markov model during simulation, which greatly improves the test generation efficiency. In addition, some strategies are integrated to avoid getting stuck at dead-ends, which further speeds up the test generation engine.

3) As for the abstraction engine, we implement it in a fine granularity. Concretely speaking, abstraction is carried out on the data dependence graph around target state variables. When the extracted logic is too large to be processed, it is divided and conquered, like that in [7].

Experimental results show that the proposed approach is efficient in covering hard-to-reach states in microprocessors. Compared to methods with other intelligent engines, such as the CA algorithm in [7], our approach can achieve higher hit ratio of hard-to-reach states without efficiency loss.

The rest of this paper is organized as follows. Section 2 presents some background information of the abstraction-guided simulation and the Markov model. Section 3 describes the framework of our approach and its details. Experimental results and analysis are presented in Section 4. Finally, conclusions are drawn in Section 5.

2. Background

2.1. Abstraction-guaranteed Simulation

Formal verification techniques such as model checking are very efficient in evaluating all the corner cases of a design. But due to their limited processing capacity, they can hardly be directly deployed on verification of industrial designs. A practicable solution for this problem is just formally analyzing a simplified abstract model. Then acquired abstract information is used to direct the simulation towards a target [3-7].

An approach that utilizes an abstract model to guide the simulation is generally called an Abstraction-Guided Simulation method. Its working principle is demonstrated in Figure 1. The simulation starts from an initial state, and then it searches forward for several steps. Each reached state is evaluated by its abstract distance to the target, which is computed on an abstract model. If the state is closer to the target (with a smaller distance), it is selected as the next start state. This process loops until the target state is visited or a limit of clock cycles is reached.

As shown in Figure 1, the key problem of abstraction-guided simulation is the encounter of "dead-end" states caused by abstraction deviations. These states seem to be close to the target in abstract distances but actually not. If the simulation wastes too much time in exploring from dead-end states, the actual path to the target is very difficult to be found. In this work, we propose a powerful test generation method, which is very efficient in avoiding getting stuck at dead-ends and hence leading to the target.

2.2. Markov Model

In general, a Markov model is a directed graph with weighted edges. Each vertex in the model represents a specific transaction or a type of transactions. The weight on each edge measures corresponding probability of transition from the source vertex to the sink vertex. For a Markov model with \( N_v \) vertices, each vertex has \( N_e \) edges directing from itself to all vertices in the model. At the beginning, the weights of all the edges outgoing from a vertex are equal, which means transitions from this vertex are generated at random. The weights can be adjusted dynamically to generate specific transaction sequences. Figure 2 demonstrates an example of Markov model built upon a simple processor instruction set architecture (ISA).

As applied to abstraction-guided simulation, basic Markov model needs to be enhanced to work more efficiently. In our approach, each vertex in the Markov model is equipped with dependence variables, and an efficient adjustment strategy to the Markov model using simulation feedbacks is also proposed.
3. The Proposed Approach

The framework of our approach is shown schematically in Figure 3. It accepts, as inputs, the target state and the register-transfer level (RTL) description of the design. Its output is a sequence of test vectors leading to the target.

The whole workflow starts from an abstraction engine, which extracts an abstract model and computes pre-images on it to get the abstract distance information. Such information will be delivered to the simulation guidance controller. The data dependence graph of the logic surrounding the target state variables is utilized to assist the abstraction process. Next, the Markov model based test generator is created in the light of the design specification.

In the simulation, test vectors generated from the Markov model are fed to the design in each cycle. After one-cycle simulation, the current state of the design is collected and analyzed by the simulation guidance controller. The information obtained from such analyses assists the adjustment of the Markov model. Through dynamical adjustments during the simulation, the Markov model can learn from the knowledge obtained in the past, and direct the simulation to converge to the target more efficiently.

3.1. Abstraction Engine

The quality of the abstract model is important for the efficiency of the whole approach. The more accurate the abstract model is, the better it provides guidance for simulation. But considering the capacity of modern formal techniques, it is infeasible to construct a very complex abstract model.

Our approach utilizes the data dependence graph (DDG) for assistance. The DDG of a design is a graph, in which vertices represent registers in the design, and a directed edge from vertex A to vertex B denotes the data dependence of B upon A, that is to say, there exists combinational logic in the design stemming from register A to register B. The DDG of the logic around target state variables is drawn from the RTL description of the design. Then the abstract model is built from DDG with the following criteria:

1) Proximity to the target state variables: Registers are selected by a breadth-first algorithm from the inside out according to the distances to the target state variables in DDG.

2) Interaction between registers: In the register selection process, if the size of the abstract model exceeds a limit, they are partitioned into groups. Registers with tight interaction tend to be put into the same partition. Each abstract partition should contain at least one of the target state variables.

An example of the abstraction process on a DDG is illustrated in Figure 4.

After extracting the abstract model, the pre-images from the target state are computed on this model in a BDD-based method. If the abstract model has several partitions, the pre-image computation is carried out on each of them separately.

3.2. Markov-model based Test Generator

One key problem in the Markov model based abstraction-guided simulation is the construction of the Markov model, and the other is the guidance strategies utilized to direct the simulation.

In our approach, high level functional specification of the design is utilized to assist the construction of the Markov model. In other words, the Markov model is built according to the processor ISA (instruction set architecture). As a directed graph with weighted edges, each vertex of the Markov model stands for a specific instruction or a type of instructions, and the weights on the edges mean the corresponding probabilities of transitions from the sources to the sinks. In the simulation, a transition from vertex A to B means an A-type instruction is generated followed by a B-type one. In the beginning, the Markov model is initiated with equal weights on all edges, which means that each vertex is free to transfer to any other. And at every cycle, the weights of edges are adjusted by feedbacks in order to generate more appropriate instruction sequences.

3.2.1. Test Vector Correlations

By analysis of many hard-to-reach states in different designs, we find that it usually requires correlations among test vectors to excite these states. It is especially true for complex designs with many primary inputs. For example, exciting the forwarding logic in a processor requires strong correlations between consecutive instructions.

Our method set up the correlation among test vectors in two ways. One is the preferred operation sequences modeled
implicitly by the Markov model, the other is the dependence variables in instructions. We employ the concept of dependence variables, which is introduced in [8] for exercising user-specified activity points by a Markov-model based method. These dependence variables are used to transfer information between generated test vectors.

The structure of dependence variables is implemented with a cache of their historical values. There are three important parameters: (1) the size of the cache, which is implemented using a First-In-First-Out (FIFO) queue with a fixed capacity, (2) the probability to access the cache, and (3) the degree of dependency, which is implemented as an exponentially decreasing distribution with a parameter $\lambda$. As shown in Figure 5, the greater the parameter $\lambda$ is, the more likely a recent historical value is selected as the new value of the dependence variable. The historical information of dependence variables is shared among all the vertices of the Markov model. Refer to [8-9] for details.

Rather than using a special template language as in [8], we use the functional specification (i.e. ISA for microprocessors) to identify dependence variables in the Markov model. Concretely speaking, immediate operands and register numbers in different instructions are identified as dependence variables.

Figure 5 illustrates the workflow of generating a BEQZ (branch if equal to zero) instruction with the help of dependence variables in our approach. According to the ISA, a BEQZ instruction is composed of four fields: the operation code (opcode), the register numbers ($rs$, $rt$), and the immediate value (offset). If a field, such as opcode, is not a dependence variable, it is assigned a specified value generated by the Markov model. Otherwise, it has a chance to get its value from its historical values (such as $rs$, $rt$, offset).

3.2.2. Evaluation of States

After each cycle in the simulation, the current state of the design is fed back to the simulation guidance controller, and then an analysis is performed to generate an appropriate feedback to the Markov model. The abstract distance information obtained previously is utilized in this process.

In our approach, the fitness function is calculated according to Equation (1) and Equation (2):

$$\text{fit}_i = \text{Max}_i - D_i$$  \hspace{1cm} (1)

$$\text{Fitness value} = \sum_{i=0}^{N_v} \frac{v_i}{v_{\text{total}}} \text{fit}_i$$  \hspace{1cm} (2)

where $N_v$ is the number of abstract partitions, $\text{fit}_i$ is the local fitness value in the $i$'th partition, $v_i$ is the number of the target state variables contained in the $i$'th partition, and $v_{\text{total}}$ represents the total number in all partitions. The local fitness value $\text{fit}_i$ in the $i$'th partition is calculated as the difference between the maximal abstract distance ($\text{MAX}_i$) and the abstract distance of the current state in the $i$'th partition ($D_i$).

The bigger the fitness value is, the better the current state is judged to be. The current state is identified as "excellent", if it has a fitness value bigger than that of the previous state. Once an "excellent" state is identified, a feedback with information of this state is sent to the Markov model, which is used to guide the adjustment to the model as is explained in Section 3.2.3. 3.2.3. Adjustment Strategy to the Markov Model

The Markov model based test generation engine accumulates useful historical information by the dynamical adjustments to the weights of edges in the Markov model. In other words, once the current state is identified as "excellent", the transition leading to it in the Markov model is strengthened and hence this specific type of instructions is more likely to be generated later.

The adjustment process to the Markov model is implemented in the following manner: firstly, the edge corresponding to the transition is identified; then, the weight on this edge is increased by $P_{\text{incr}}$; finally, weights on all other edges starting from the same source vertex are adjusted, so that the sum of weights on all edges from the same source vertex is normalized to 1.

We set the limits, i.e. the maximal value $P_{\text{max}}$ and the minimal value $P_{\text{min}}$, for the weight on an edge, in order to avoid getting stuck at some local optimum. As a result, even if a particular type of transaction is extremely "popular" at a time in the simulation, there are still minimal probabilities kept for other neighboring transactions.

Our strategy in the probability adjustment process can be described as "Easy to increase and hard to decrease for lower weights, and reversely for higher weights." In other words, for a weight to be increased, the larger its current value is, the smaller its incremental value $P_{\text{incr}}$ will be, as shown in Equation (3). In the normalization procedure, the larger the current weight is, the larger its decreased value $P_{\text{decr}}$ will be, as shown in Equation (4).

$$P_{\text{incr}} = \frac{P_{\text{max}} - P_i}{N_v}$$  \hspace{1cm} (3)

$$P_{\text{decr}} = k \times (P_i - P_{\text{min}})$$  \hspace{1cm} (4)

where $N_v$ is the number of vertices in the Markov model, $P_i$ is the current weight value, and $k$ is the coefficient which is set so
that all weights of the edges starting from a vertex add up to 1.

3.3. Simulation Guidance Algorithm

The overall algorithm of the simulation guidance is shown in Figure 6. The Markov model is built up and initialized with equal weights on all the edges from each vertex. The design is also reset to its initial state. Then at each cycle, an input vector is generated by the Markov model and fed to the design. After one-cycle simulation, the corresponding fitness value of the current state is calculated. If the current fitness value is greater than the previous one, the state is identified as “excellent” and the Markov model is adjusted accordingly. Otherwise, the simulation continues without any adjustments to the model. When there is no progress for a certain amount of clock cycles, the design is set to the next state in a priority queue (similar to buckets in [5]), which stores the visited “excellent” states in a non-descending order according to their fitness values. The Markov model is reset every 100,000 cycles to avoid local traps, i.e., being stuck at local optimum. The simulation continues until the target state is hit or a limit of clock cycles is reached.

The Markov model based test generation engine has several effective mechanisms to reach target states quickly:

1) For simulation efficiency: The information of historical sequences leading to states with lower abstract distances is effectively extracted and saved in the Markov model, resulting that “excellent” states are easy to be visited later. It is quite different from random simulation engines in most existing approaches, in which states with lower abstract distances are reached by luck.

2) For avoidance of dead-ends: Two factors contribute to this. One is the priority queue explained in Figure 6, which saves visited states and makes long-time local search recoverable to other good states. The other is the minimal limitation of weights in the Markov model, i.e. $P_{\text{min}}$, which avoids input vectors getting stuck at local optimum.

Markov_model.state = initial uniform distribution; design state = initial_state; while !(target_reached) or !(timeout) do
  for 100000 clocks do
    Markov_model.gen_a_stimulus();
    simulate the stimulus and record current_state;
    calculate current_fitness of current_state;
    if current_fitness>previous_fitness then
      push current_state into the priority queue;
      generate a feedback to the Markov_model;
      Markov_model.adjust();
    end if
    if no_progress_for_N_clocks then
      design state = next state in the priority queue;
    end if
  end for
  Markov_model.reset();
end while

Figure 6: Pseudocode of the simulation guidance algorithm.

4. Experiments

We implemented our Markov-model based semi-formal approach in C++ through the PLI interface of Verilog. The benchmarks are a DLX processor [10] and a MIPS32 compatible processor got from www.opencores.org. All the experiments are run on a Linux machine with four 3.0 GHz Intel Xeon CPUs and 6G bytes RAM.

In our experiments, all possible combinations of register values constitute the state space. Target states are selected from some critical parts of the designs, such as the forwarding logic and the exception handling logic. In the experiments, the test generation for each target is repeated by 20 times with different random seeds to record the average time.

4.1. Comparison with Other Methods

Table 1 shows the results on the DLX processor. The simulation time in each row is the average of 20 times test generation for one hard-to-reach state. The column “Random” presents the simulation time of random tests. We have also implemented a previous Markov-model based ATPG method [9] and an abstraction-guided simulation method with random test generation engine [5]. The results of these methods are presented in the columns “StressTest” and “Guided”, separately. Since different abstract models have distinct effects on the abstraction-guided simulation, for the sake of fairness, the “Guided” method uses the same abstract model as obtained in our approach. In the last column, the results of our approach are reported.

The bottom line of Table 1 reports the hit ratio of target states, i.e. the percentage of visited target states vs. all ones. The hit ratios of the random method and the StressTest technique are 20% and 60% separately, which means that the two methods time out in tests of many target states. The “Guided” method also fails to reach the target state of $t_5$, in which we find close test vector correlations, e.g. data dependency relations between instructions, are required to reach the target. Through further analysis of the results, it is found that the “StressTest” method can cope with some hard-to-reach states, especially those requiring close test vector correlations, e.g. $t_5$. But its simple guidance strategy is quite insufficient and has low efficiency in abstraction-guided simulation. Data in the bottom line demonstrate that our approach provides higher hit ratio of

<table>
<thead>
<tr>
<th>Target states</th>
<th>Random Simulation time(s)</th>
<th>StressTest [9] Simulation time(s)</th>
<th>Guided [5] Simulation time(s)</th>
<th>Our approach Simulation time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>12.99</td>
<td>2.12</td>
<td>15.79</td>
<td>0.7</td>
</tr>
<tr>
<td>t2</td>
<td>459.96</td>
<td>163.46</td>
<td>31.19</td>
<td>15.4</td>
</tr>
<tr>
<td>t3</td>
<td>TO</td>
<td>TO</td>
<td>51.47</td>
<td>11.6</td>
</tr>
<tr>
<td>t4</td>
<td>TO</td>
<td>TO</td>
<td>77.27</td>
<td>20.5</td>
</tr>
<tr>
<td>t5</td>
<td>TO</td>
<td>2420.2</td>
<td>TO</td>
<td>14.8</td>
</tr>
<tr>
<td>t6</td>
<td>TO</td>
<td>TO</td>
<td>521.37</td>
<td>97.04</td>
</tr>
<tr>
<td>t7</td>
<td>TO</td>
<td>834.41</td>
<td>1410.40</td>
<td>55.3</td>
</tr>
<tr>
<td>t8</td>
<td>TO</td>
<td>627.95</td>
<td>194.53</td>
<td>12.5</td>
</tr>
<tr>
<td>t9</td>
<td>TO</td>
<td>TO</td>
<td>1446.6</td>
<td>46.1</td>
</tr>
<tr>
<td>t10</td>
<td>TO</td>
<td>1703.2</td>
<td>650.67</td>
<td>13.2</td>
</tr>
</tbody>
</table>

Table 1: Experimental results on a DLX processor.

<table>
<thead>
<tr>
<th>Hit</th>
<th>20%</th>
<th>60%</th>
<th>90%</th>
<th>-</th>
<th>100%</th>
</tr>
</thead>
</table>

Time-out (TO) is set to 7200 seconds.
target states, thus guarantee the quality of functional verification.

In the cases that target states are visited, it is clear from Table 1 that the proposed approach has a better performance. The “Guided” method performs well in most instances, but our approach gains higher efficiency in covering hard-to-reach states, taking advantage of a more intelligent test generation engine.

We carried out an additional set of experiments on another complex design, without any specific adjustments to our approach. It is a MIPS32 compatible processor with 3,450 flip-flops, and the results are presented in Table 2.

As shown in Table 2, the “StressTest” method fails in 3 out of 4 instances. The abstraction guided semi-formal methods (the “Guided” and our approach) both figure out all the instances. But the proposed approach has a better performance. On the strength of an efficient test generation engine, our approach gains higher efficiency than previous methods do, even for large, complex designs.

The work in [7] reports similar improvement of efficiency over the “Guided” method [5]. However, the hit ratio of target states of [7] is same with that of the “Guided” method, both 100%, in the experiments in [7], while our approach covers one more hard-to-reach state than the “Guided” method does. Therefore in some sense, our approach could provide higher verification quality than [7].

### 4.2. Adjustment Strategies in Markov-models

In this section, different adjustment strategies in Markov-models are analyzed.

Figure 7 shows the probability distribution of different Markov model based methods at the end of simulation. Simulation is carried out 3 times using different random seeds. In the distribution map, each row represents the weights of outgoing edges of one vertex in the model, and each column represents the weights of incoming edges. For example, the lattice in row 2, column 5 represents the weight of the edge from vertex 2 to vertex 5, the darker color the lattice holds, the larger the weight is.

As shown in the Figure 7, there is distinct regularity in the distribution maps of our approach, while it is almost random distributed in the “StressTest” method. Although the “StressTest” method is effective in testing user-specified points, it is unfit for covering hard-to-reach states. Because the “Activity Monitor” in the “StressTest” just observes the value changes of signals, it may be triggered too frequently, resulting in over adjustments to the Markov model. It is clear from Table 1, 2 and Figure 7 that our strategies in the Markov model are quite suitable for the abstraction-guided simulation.

### 5. Conclusion

In this paper, an abstraction-guided simulation approach based on the Markov model is presented to cover hard-to-reach states in microprocessor designs. An abstract model is extracted from the RTL design, and no abstraction refinement is needed. The Markov model with dependence variables is constructed, utilizing the high level functional specification of the design, which makes it easier to generate functional vectors and to extract vector correlations. Effective strategies about state evaluation and dynamical adjustment to the Markov model are also proposed, which greatly help the simulation converge to the targets. Experimental results on microprocessors show that with the help of these mechanisms, our approach is efficient in covering hard-to-reach states, even for large, complex designs. Compared to similar work with other intelligent engines, such as [7], our approach can achieve higher hit ratio without efficiency loss.

In the future, we are going to explore more powerful abstraction techniques and guidance mechanisms.

### References