MUTATION-BASED VALIDATION OF HIGH-LEVEL MICROPROCESSOR IMPLEMENTATIONS

Jorge Campos and Hussain Al-Asaad
Department of Electrical and Computer Engineering
University of California, Davis, CA
E-mail: {jcampos, halasaad} @ece.ucdavis.edu

Abstract

In this paper we present a preliminary method of validating a high-level microprocessor implementation by generating a test sequence for a collection of abstract design error models that can be used to compare the responses of the implementation against the specification. We first introduce a general description of the abstract mutation-based design error models that can be tailored to span any coverage measure for microprocessor validation. Then we present the clustering-and-partitioning technique that single-handedly makes the concurrent design error simulation of a large set of design errors efficient and allows for the acquisition of statistical data on the distribution of design errors across the design space. We finally present a method of effectively using this statistical information to guide the ATPG efforts.

1. Introduction

Fierce competition in the computer industry imposes tight time-to-market requirements on chip makers. Thus as implementation complexities increase with each generation of microprocessors, engineers are forced to validate a larger design space in a shorter time frame. This task becomes even more difficult when the method of validation relies on human efforts to list the corner cases of the design space to which test vectors are later generated either manually or automatically. Unfortunately, virtually all automatic test pattern generation (ATPG) systems inefficiently use the identified set of corner cases when generating test programs because they analyze the corner cases in a sequential fashion. Also, many focus their efforts in validating a system's finite state machine or employing formal methods, but these approaches are impractical for complete microprocessor implementations. It is for these reasons that effort must be made to investigate a new validation paradigm that effectively and efficiently generates a test program for a given high-level microprocessor implementation and a corresponding set of error models that represent the desired coverage measures.

It is the goal of this research project to create such a versatile validation system where an already existing microprocessor hardware description can be inserted into the validation environment along with a collection of modeled design errors that guide the ATPG process. This validation environment will consider all error models concurrently during every simulation/ATPG iteration in order to distribute the ATPG efforts evenly throughout the design space. Our previous research has focused on creating a novel technique to concurrently simulate a collection of modeled errors on a high-level microprocessor implementation, and our results were encouraging. We are using the knowledge we gained on this subject to create an effective mutation-based validation system. The paper proceeds as follows. In Section 2 we discuss our background on the subject of concurrent simulations, and we use our findings to propose the validation system in Section 3. Section 4 introduces how the abstract design errors are to be modeled, and Section 5 presents a novel data structure for these design errors such that it provides for simulation efficiency and ATPG effectiveness. Section 6 describes how the simulation statistics can be used in order to generate effective tests. Finally, we conclude the paper and discuss future work in Section 7.

2. Background

In our previous research efforts, we have developed a method to simulate a collection of modeled design errors concurrently on a hardware description of a microprocessor implementation [1]. We have modified the way signals and condition statements are implemented to efficiently support concurrent propagation of a set of mutant values across a netlist. We have applied this simulation technique onto a Motorola 6800 microprocessor implementation by John E. Kent [opencores.org] by creating a software model of the processor that supports concurrent propagation of mutant values, and we have collected statistical data from a random simulation of 300,092 modeled errors.

The simulation results demonstrate that modeled design errors have a high probability of being dropped after affecting 10% of the internal signals or less for

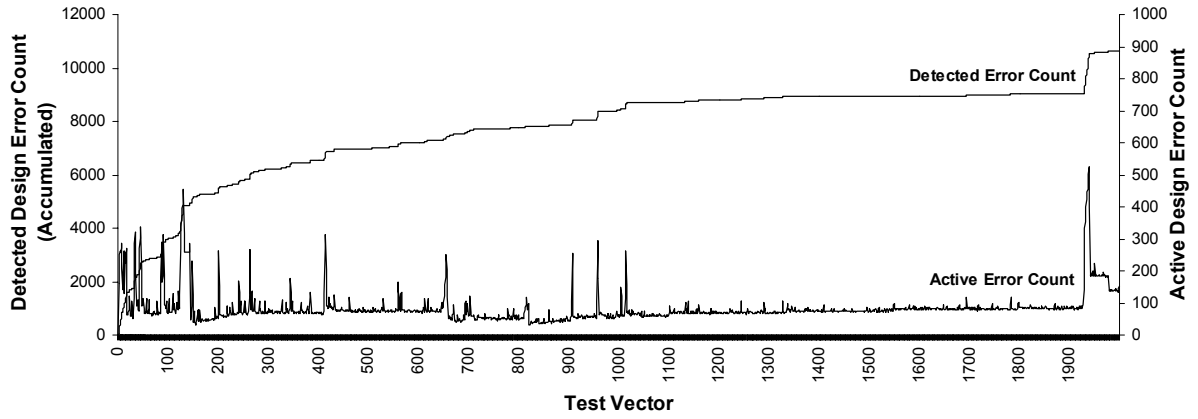

Figure 1: Detected/Active Error Relationship [1].

this processor implementation. The simulation results demonstrate that there is a significant correlation between the number of design errors that are detected per simulation iteration and the number of design errors that are active per simulation iteration (Figure 1); therefore establishing that it is fruitful to focus our ATPG efforts on maintaining the number of active design errors at its highest possible value. This trend obviously will not hold when a collection of hard-to-detect design errors is encountered, at which point a propagating sequence will need to be generated for each of these design errors. We have shaped our ATPG algorithm, which is described later, to have the ability to identify and address these scenarios as it defines the goals for each ATPG iteration.

3. Overview of Validation System

Our preliminary concurrent design error simulation environment performs simple event-driven simulations, but it is implemented in a compiler-based technique. We did this by creating a development library of constructs such as signals, processes, and some supporting objects that implement the event-driven simulation at run-time such as the FIFOs used for signal propagations and module executions. The run-time results are therefore being generated by the compiled implementation and not a simulator that dynamically creates an internal representation of the HDL implementation. Given that the TyVIS simulation environment is implemented in a similar fashion, our validation system is being modified to apply our concurrent simulation techniques onto the SAVANT VHDL analyzer and the TyVIS simulation kernel [8]. This approach provides us with a high-performance parser and simulation environment, and gives us access to a large array of existing microprocessor implementations.

The goal of this research project is to use the real-time simulation statistics to guide each ATPG iteration so it may produce a test sequence that potentially results in the detection of the greatest number of modeled design errors. A test sequence that effectively detects a wide range of modeled simple design errors has a high probability of implicitly detecting complex design errors in the implementation [6]. To achieve our goal, we need the following two items in our validation environment:

- A tool that simulates the complete set of modeled design errors concurrently on the HDL implementation.
- A tool that identifies the transitions that map the current state *S* to the target state *S'* of the microprocessor HDL implementation, and generates a test sequence that achieves these transitions.

By including both of these tools within the same compiled program, our validation environment becomes a compiled program where the simulation and ATPG environments are derived from the HDL source code.

A diagram depicting our projected validation process is provided in Figure 2. In the *description phase*, a microprocessor implementation and a collection of abstract design error models are created. In the *construction phase*, the SAVANT VHDL analyzer is used to convert the microprocessor implementation into a C++ simulation unit via the *publish_cc* runtime argument, and to convert the microprocessor implementation into an ATPG unit via the *publish_atpg* runtime argument. The target block of a dashed arrow denotes an object produced by the source block of the dashed arrow.

The strong point of the SAVANT analyzer is that we can extend it to convert its internal intermediate representation (IIR) of the microprocessor implementation into a graph composed of the implementation's basic blocks. This graph representation is our current research focus; it will be used during the constraint-solving steps of the ATPG system.

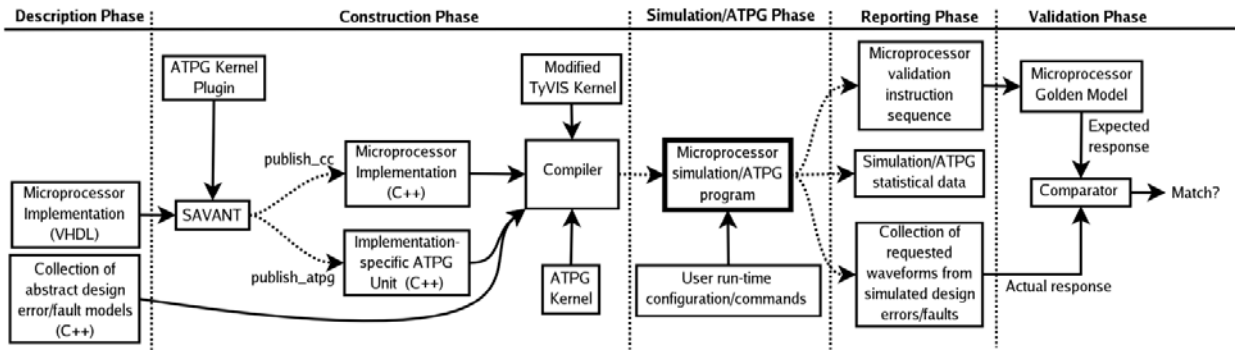

Figure 2: Validation process overview.

The *simulation/ATPG phase* executes the simulation/ATPG program that was generated during the construction phase where a collection of user run-time configurations and commands act as inputs. The user specifies what outputs to produce and what modeled errors to consider during the simulation and ATPG efforts.

The simulation/ATPG program creates the output files during the *reporting phase*. These outputs include the instruction sequence that detects the optimal set of modeled errors, the collection of simulation/ATPG statistical data, and the actual response of the microprocessor implementation as well as a collection of responses that correspond to a user-specified set of modeled errors. The instruction sequence is fed into the golden model (specification) during the *validation phase* to generate the expected response, which is compared with the actual response. Any detected discrepancy needs to be analyzed to determine if it is a result of an actual design error, or if it corresponds to an invalid input sequence.

4. Abstract Design-Error Models

Superscalar microprocessor implementations adopt many special techniques to achieve a high performance including register renaming, branch prediction, out-of-order execution, and simultaneous multithreading. These microprocessor implementations are therefore inherently a collection of many disjoint functional units. Extensive tests are typically applied directly to each of these units, but any attempt to validate their functionality when inside the microprocessor implementation requires that the microprocessor's coverage measure affect each of these unit's control-based corner cases. Implementing a set of design error models that span the coverage measure is highly dependent on the implementation style of the circuit under test, therefore forcing us to develop a versatile validation system where the design error model description can be tailored to any coverage metric.

A design error results in the generation of an erroneous value under a specific state of the system. Similar to some fault injection campaigns [4][5], we are basing our description of design error models on three basic characteristics: *i*) the activation criteria, *ii*) the consequence of activation, and *iii*) error injection. Multiple design errors are simulated concurrently; therefore each design error has a unique identification number.

A design error's activation criteria specify a set of signals and the conditions that they must meet before the design error is activated. Once the activation criteria is met, the code segment particular to a design error is executed which generates a set of mutations for a corresponding set of injection points. A mutant value is injected into the specified signal within the netlist immediately after it is generated.

Given that a design error on an implementation of a modular component will appear on every instantiation of that component, all design error models have to obey the hierarchical error model [3] where every instantiation of a modular component will have the same set of design errors with corresponding identification numbers. This is important because it allows a design error to simultaneously appear at multiple instantiations of a component and it correctly models aliasing in the case where these mutant values mask each others' propagation across the netlist.

A study on bug occurrences in pipelined and superscalar microprocessor implementations [7] shows that over two-thirds of design errors are related to the control logic. It is because of this tendency that we are employing a form of the mutation control error (MCE) model [1][2]. For the MCE model in this paper, the processor state (*s*) and a correct value (*vc*) of the control signal (*c*) act as the activation criteria, signal *c* is mutated from *vc* to a an erroneous value (*ve*) as a consequence, and injected back into signal *c*. The complete set of MCE instantiations on the Motorola 6800 implementation consists of 300,092 modeled errors. The task of generating a test sequence for each modeled error sequentially would require a significant amount

of redundant work, therefore justifying the development of our concurrent approach.

5. Clustering and Partitioning

High-level validation through the use of concurrent mutation-based simulation techniques provides the best test sequences when the set of error models completely span the coverage measures. One reason for this is that concurrent simulation of a complete set of modeled errors allows a test sequence to drop all detected errors, thus reducing the number of modeled errors for which subsequent test sequences need to be generated. A second reason is that aiming each ATPG iteration at detecting the maximal number of modeled errors results in a highly-effective test sequence with high probabilities of detecting complex design errors.

Unfortunately, supplying a validation system with a complete set of modeled errors affects the simulation performance because the simulator is forced to analyze a larger set of modeled errors per simulation iteration as it searches for all errors that must be activated. It is possible to reduce the complexity of the error activation cycle by reducing the algorithm's search space. If we carefully organize the data-structure that contains the live set of modeled errors, our reward can be twofold: (*i*) we minimize the error-activation cycle, thus optimizing the simulation, and (*ii*) we give the ATPG system information on the distribution of undetected modeled errors among the design space, thus providing it with the ability to aim its error-activating efforts at the activation criteria with the highest density of undetected errors.

We can achieve these stated goals by "clustering and partitioning" the complete set of modeled design errors into groups that are organized by their activation criteria. Before discussing the clustering technique, let us explore the various possible techniques for generating mutant values. A mutant generator is a unit within a simulation environment in charge of activating any design error when its activation criteria are met by generating the corresponding mutant value and injecting it into the netlist. Therefore for each mutant generator, the collection of signals in the netlist that acts as activation criteria to any of its modeled design errors needs to propagate any change in value to this mutant generator. Also, whenever an activation criterion propagates into a mutant generator, the mutant generator needs to search through its set of design errors and identify every design error that needs to be activated. This method of generating mutant values provides us with three implementation alternatives:

- *Centralized mutant generator*: Only one unit in the simulation environment is in charge of generating mutant values. This results in the lowest propagation complexity because the propagation overhead imposed by concurrent error-model simulation is at most one extra propagation step per internal signal. Unfortunately, the design error search space is linear with respect to the number of undetected design errors.
- *Distributed mutant generators*: One mutant generator is assigned to each modeled design error. Even though this reduces the search space per mutant generator to its minimum, it results in a maximum propagation complexity because the propagation overhead imposed by concurrent error-model simulation on a signal is proportional to the number of design errors to which it acts as activation criteria.
- *Hybrid (clustered) mutant generators*: Design errors are "clustered" into groups that have common activation criteria, therefore maintaining the propagation complexity and search space per mutant generator at feasible levels.

In a validation system where multiple error models are being used, the hybrid mutant generation technique gives us the flexibility of keeping design errors of disjoint activation criteria in separate clusters and allows us to optimize the search algorithm of each mutant generator by introducing a "partitioning" technique. Our partitioning scheme reduces the search space per mutant generator by selecting the signal that acts as the most common activation criteria in that cluster and designating this signal as the partitioning point. Once we have chosen a partitioning point per cluster, we can organize each cluster as a hash table where the value of the partitioning point is used as the hashing key.

Now that we have defined the data structure and discussed the run-time implementation of our clustering-and-partitioning technique, we need to introduce the method by which we organize the design errors into clusters and partitions. Given that our underlying goal is to create partitions where the included design errors have at least one common signal in their activation criteria, we can implement clustering-and-partitioning using the following steps:

1) Generate a table where the rows represent the signals that act as activating criteria in the system and the columns represent the set of modeled design errors under consideration. For any modeled error, the set of intersecting points specifies the set of signals that collectively denote that modeled error's activation criteria. Initially, each of these signals and modeled errors are marked *unselected*.
2) Identify the unselected signal that intersects with the greatest number of unselected modeled errors, and mark this signal as *selected*.
3) Group all unselected modeled design errors that intersect with the selected signal into a

	α_1	α_2	α_3	α_4	α_5	α_6	
S_1	X					X	
S_2			X				
S_3							
S_4		X		**X**		**X**	←Iteration 2
S_5	**X**	**X**	**X**		**X**		←Iteration 1
S_6	X						
S_7				X			
S_8					X		

Figure 3: Clustering-and-partitioning example.

cluster, and mark each of these modeled errors as *selected*.

4) For this new cluster, set the selected signal as the partitioning point and partition the cluster into a hash table where the activation criterion of the partitioning point is used as the hashing key.
5) If unselected design errors exist, then return to step 2 to perform another iteration.

An example for clustering and partitioning is provided under Figure 3 where S_i denotes a signal and α_i denotes a modeled design error. In this example, the first cluster consists of the set of modeled errors $\{\alpha_1, \alpha_2, \alpha_3, \alpha_5\}$ with partitioning point S_5, and the second cluster consists of the set of design errors $\{\alpha_4, \alpha_6\}$ with partitioning point S_4.

In practice, each possible value for the partitioning point usually has a nonempty set of corresponding design errors. An example of this is the mutation control error (MCE) model [1][2] where the state signal is chosen as the partitioning point. The state signal occurs in every MCE design error, therefore selecting it as the partitioning point results in an efficient data structure when the key is used directly as the index. This is because the performance of searching for the design errors to be activated remains independent of the size of the state-space. The only factors that grow along with the state space are the hash table size and the total number of design errors.

6. Automatic Test Pattern Generation

We can take advantage of the clustering-and-partitioning data structure by using an ATPG algorithm that gives priority to any partition whose activating test sequence has the highest probability of detecting the most design errors. We introduced in Section 2 that it is fruitful to focus our ATPG efforts on maintaining the number of active design errors at its highest possible value, which correlates to generating a test sequence for the partition with the greatest number of undetected design errors.

This results in the algorithm provided under Figure 4 which has the role of aiming the ATPG efforts at the activation criteria with the highest density of undetected design errors (deterministic-activation), and only performing deterministic-propagation in the case where probabilistic-propagation is insufficiently effective. Line 1 sorts the list of partitions into the order of descending member size to ensure that any unsuccessful attempt to generate a test for a partition P is followed by an attempt on the next best partition during the subsequent iteration of the *while* loop. Line 6 attempts to generate a test sequence that activates an inactive design error in P, and any failed attempt results in the removal of that design error from P (fault dropping). These dropped design errors are marked as *unexcitable*. Line 10 handles the case where the activation criteria for the partition P is already met, which is expected to happen whenever probabilistic propagation on the set of active modeled errors from P is insufficiently effective. Therefore line 10 is used to generate a test sequence that propagates an active design error in P to a primary output, and any failed attempt results in the removal of that design error from P. These dropped design errors are marked as *undetectable*.

At the start of the ATPG effort, it is expected that deterministic activation on the dominant partition will be effective in causing the detection of enough design

```
Precondition:
    Lp = list of all partitions from every cluster

ATPG-ITERATION(Lp)
1.  Sort Lp into descending order of member size
2.  P ← first partition in Lp
3.  SUCCESS ← false
4.  while P exists and SUCCESS = false
5.    if activation criteria for P is not met
6.      then TP ← generate activation pattern(s)
                  for any inactive error in P
                  while dropping errors from
                  unsuccessful ATPG attempts
7.           if activation is successful
8.             then SUCCESS ← true
9.             else P ← next partition in Lp
10.     else TP ← generate propagation pattern(s)
                  for any active design error in
                  P while dropping errors from
                  unsuccessful ATPG attempts
11.          if propagation is successful
12.            then SUCCESS ← true
13.            else P ← next partition in Lp
14. if SUCCESS
15.   then return TP
16.     else fail
```

Figure 4: Algorithm for each ATPG iteration.

errors from this partition so as to demote it from its dominant status. The probabilistic-propagation technique will continue to be effective for as long as there are enough design errors with simple propagation requirements. Whenever the ATPG-ITERATION algorithm encounters a partition that has an insufficient number of design errors with simple propagation requirements, the deterministic activation iteration will be followed by a deterministic propagation iteration on the same dominant partition.

7. Conclusions and Future Work

In this paper we have presented a method of validating a high-level microprocessor implementation by generating a test sequence for a collection of abstract design error models that can be used to compare the response of the implementation with the response of the specification. We first introduced a general description of the abstract mutation-based design error model that can be tailored to span any coverage measure for microprocessor validation. Then we presented the clustering-and-partitioning technique that single-handedly makes the concurrent design error simulation of a large set of design errors efficient and allows for the acquisition of statistical data on the distribution of modeled design errors across the design space during the concurrent simulation of modeled errors. We then presented a method of effectively using this statistical information to guide the ATPG efforts whose results are expected to achieve a high design error detection rate based on the observations made in [1]. An overview of the validation process has also been provided where the microprocessor implementation and the collection of modeled design errors are used to create a test sequence whose ability to detect the modeled simple design errors correlates to a high probability of implicitly detecting complex design errors [6].

Our immediate future goals constitute of finalizing the framework for converting a microprocessor's VHDL implementation into a stand-alone ATPG unit, and implementing it by extending the SAVANT VHDL analyzer [8]. Given that mutation-based error modeling can lead to an overwhelming number of possible modeled errors, we will continue our research by using this mutation-based validation system to perform an extensive study on microprocessor coverage measures and their corresponding effective error models. As a result, our ultimate goal is to provide a novel microprocessor validation system that is driven by a versatile VHDL simulation/ATPG system, and is empowered by effective microprocessor-specific design error models.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

References

1. Jorge Campos and Hussain Al-Asaad, "Concurrent Design Error Simulation for High-Level Microprocessor Implementations," *Proc. AUTOTESTCON*, 2004, pp. 382-388.
2. Hussain Al-Asaad, *Lifetime Validation of Digital Systems via Modeling and Test Generation*, Ph.D. Dissertation, University of Michigan, Ann Arbor, 1998.
3. Li-C. Wang, Magdy S. Abadir, and Jing Zeng, "On Logic and Transistor Level Design Error Detection of Various Validation Approaches for PowerPC Microprocessor Arrays," *Proc. VLSI Test Symposium*, 1998, pp. 260-265.
4. Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," *Digest of Papers: International Symposium on Fault-Tolerant Computing*, 1994, pp. 66-75.
5. L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New Techniques for Speeding-up Fault-Injection Campaigns," *Proc. Design Automation and Test in Europe*, 2002, pp. 847-852.
6. Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, pp. 34-41, April 1978.
7. Miroslav N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," *Proc. International Test Conference*, 2003, pp. 138-147.
8. Philip A. Wilsey, Dale E. Martin, and Krishnan Subramani , "SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDL," *VHDL Users' Group Spring Conference*, 1998, pp. 195-201.
9. Chia-C. Yen, Jing-Y. Jou, and Kuang-C. Chen, "A Divide-and-Conquer-Based Algorithm for Automatic Simulation Vector Generation," *IEEE Design and Test of Computers*, Vol. 21, No. 2, pp. 111-120, March/April 2004.
10. Ghassan Al-Hayek and Chantal Robach, "From Design Validation to Hardware Testing: A Unified Approach," *Journal of Electronic Testing: Theory and Applications*, Vol. 14, pp. 133-140, February-April 1999.