Stimuli Generation with Late Binding of Values

Avi Ziv
IBM Research Laboratory in Haifa
Haifa, 31905, Israel
email: aziv@il.ibm.com

Abstract

Generating test-cases that reach corner cases in the design is one of the main challenges in the functional verification of complex designs. In this paper, we describe a new technique that increases the ability of test generators by delaying assignment of values in the generated stimuli, until these values are used in the design. This late-binding allows the generator to have a more accurate view of the state of the design, and thus it can better choose the correct values. Experimental results show that late-binding can significantly improve coverage, with a reasonable penalty in simulation time.

1. Introduction

Functional verification is widely acknowledged as the bottleneck of a hardware design cycle [4]. To date, up to 70% of the design development time and resources are spent on functional verification. The investment in expert time and computer resources is huge, as is the cost of delivering faulty products [3]. The increasing complexity of hardware designs raises the need to develop new techniques and methodologies that provide the verification team with the means to achieve its goals quickly and with limited resources.

In current industrial practice, most of the verification is done using generating a massive amount of tests by random test generators [2, 6, 11]. The use of advanced random test generators can increase the quality of generated tests, but it cannot ensure that all the "dark corners" in the design are exposed and that the entire test plan is implemented. Coverage analysis [7] helps detect non-covered events, but it does not help cover these events. Generation of interesting test-cases that reach the non-covered corner cases is one of the main challenges in functional verification. Often, these hard-to-reach corner cases happen when several events occur at the same time or in close proximity to each other.

A method commonly used to increase the capability of test generators examines the state of the design before generating the next input. In this method, often referred to as *dynamic generation* [4], the test generator uses the state of the design to determine which input has a better chances of reaching an interesting case. For example, if the goal of the test generator is to create resource interdependency in the pipelines of a processor, it generates instructions that use the same resources as the instructions already in the pipelines.

The problem with this approach is that in many cases, by the time the generated inputs reach the interesting area in the design, the state of the design changes and the reason that led to the generation of the specific pattern no longer exists. In the example above, it is possible that the generator, in an attempt to create interdependency in the execution pipelines, uses register `R1` because this register is currently in use in the execution pipeline. However, by the time the generated instruction reaches the execution pipeline, the instruction that used `R1` may have finished its execution. In this case, the requested interdependency event is not reached.

An alternative approach to improve coverage uses symbolic simulation [5] and Symbolic Trajectory Evaluation (STE) [9] techniques. In symbolic simulation, some of the inputs to the design are represented as symbols. These symbols propagate through the circuit to the outputs. The main advantage of symbolic simulation is that large spaces can be covered in parallel, with a single symbolic input. However, it suffers from explosion of the symbol representation, and thus it is limited to small designs. In addition, it is difficult for symbolic simulators to simulate complex functions and operators, such as multiplication.

A third approach builds test generators that have a precise model of the verified designs. Using this model, the test generator represents the requested event as a state-machine traversal problem [8] or a constraint satisfaction problem [1]. A solution of this problem yields a test-case that reaches the requested event. The main disadvantage of this approach is that it requires high effort for specification of the model and building the generation engine.

1530-1591/04 $20.00 (c) 2004 IEEE

In this paper, we propose a new technique that overcomes some of the disadvantages of dynamic generation without the complexities of symbolic simulation and generation sing a precise model of the design. The proposed technique delays the assignment of values by the test generator to some variables in the generated stimuli, until these values are actually used in the design. This delayed assignment (or late-binding) to parts of the stimuli allows the generator to have a more accurate view regarding the state of the design, and it can thus choose the correct values that more precisely lead to a desired event. Using the proposed method for the example above, the test generator leaves the value of the source register undefined in the primary input. When this value is needed, during the data fetch stage in the execution pipelines, the generator examines the state of the other pipeline stages and chooses a register that is already in use in other pipeline stage, thus creating the desired interdependency event.

The advantage of this method over dynamic generation is that it provides the test generator with a better view, and thus better control, over the generated patterns. The proposed method is much simpler than symbolic simulation, since it requires only a small overhead for each variable, and it does not require complicated symbolic arithmetic. Therefore, it is much more suitable for high-level models that contain complex operators.

We implemented the late-binding technique as part of the Odette verification environment [12]. The verification environment is implemented on top of SystemC [10]. The support for late-binding is part of the verification environment; it does not include changes to the simulation kernel of SystemC. Experimental results show that the late-binding technique can be used to reach hard-to-generate cases and increase the coverage rate over dynamic test generation.

The rest of the paper is organized as follows. In Section 2, we describe the late-binding technique and a simple algorithm for its implementation. In Section 3, we discuss several improvements to the basic method that are suitable for RTL simulation. In Section 4, we provide some experimental results. We conclude with few remarks and leads for future study in Section 5.

2. Late Binding of Values

Generation of interesting test cases for hardware designs that reach corner cases is a difficult task, even in dynamic generation, because often the state of the design changes before the generated stimuli reaches the interesting area in the design, and thus the reason that led to the generation of the specific pattern no longer exists. A possible solution to this problem is to delay assignment of values by the test generator to parts of the generated stimuli, until these values are actually used in the design. In other words, as long as the undefined value is copied or moved from place to another, there is no need to set its value. When an attribute with an undefined value is used for calculations or affects the control flow, the generator is called to set the value of the attribute before the access is actually performed. This delayed assignment (or late-binding) of values allows the generator to have a more accurate view regarding the state of the design, and thus it can choose the correct value that more precisely leads to a desired event.

Figure 1 illustrates the generation of an instruction with a write-read dependency, using the proposed method. In the first step, shown in Figure 1(a), the generator generates a new `add` command and sets one of its source registers to be undefined. Note that choosing the source register based on the current state of the execution pipelines will not lead to the desired results, as will be shown later.

After the instruction is fetched, it flows for several cycles in the fetch and decode units, where it is being processed (Figure 1(b)). Because the source register field in the instruction is only copied from place to place in these units, the undefined value is copied with the rest of the instruction and its value is not set by the generator.

In Figure 1(c), the instruction reaches the data fetch stage in the execution pipe. In this stage, the register file is accessed with the source register to fetch the operand of the instruction. Since this access is not a simple copy, it triggers the test generator that needs to assign a value to the source register.

The test generator examines the state of the pipelines and finds out that an instruction at the execution stage of pipe B is writing to `R17`. Therefore, it sets the undefined value of the source register in the instruction to `R17` and creates the required write-read dependency. The final state of the pipelines, after the value of the source register is defined, is shown in Figure 1(d). Note that none of the potential target registers that were at the execution stages of the pipelines when the instruction was generated can be used to create the requested dependency.

When late-binding is used, it is important to maintain the correct simulation semantics of these values. That is, it should be impossible to distinguish between the late binding of the values and assignment of the same values at the primary inputs of the design. For example, if an undefined value is copied inside the design, the generator should not be allowed to set different values to various copies of the same undefined value.

The following is a simple algorithm that implements late-binding and maintains the correct simulation semantics. Each variable (signal, register, or group of signals or registers) in the design is associated with a Boolean flag that indicates if its value is undefined and a pointer to a set that contains all the other variables that should have the same value as this one. Access to a variable with the undefined

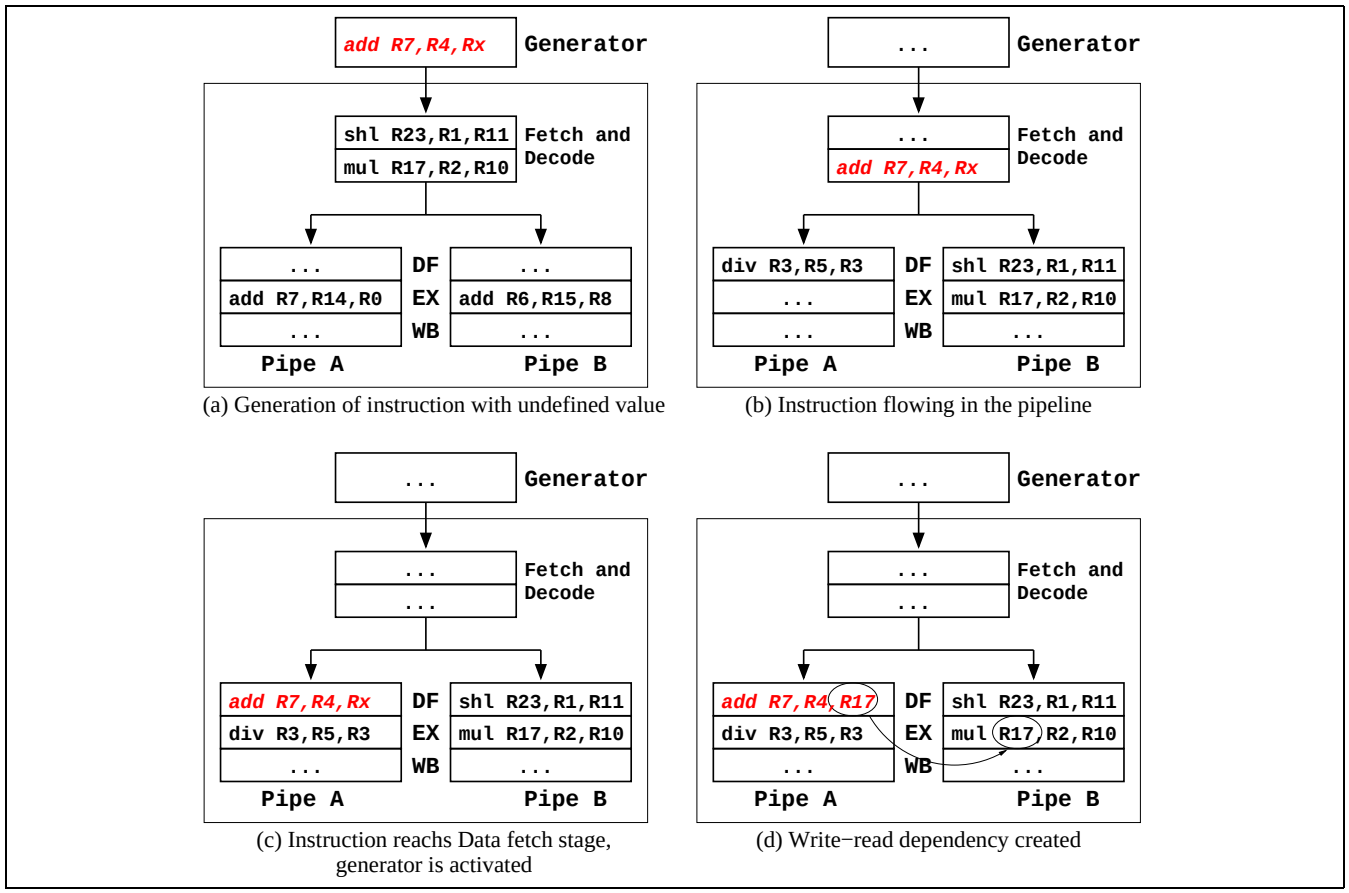

Figure 1: Generating write-read dependency with late-binding

flag set during simulation is handled in the following way:

1. If a variable with an undefined value is copied to another variable, the destination variable is added to the set pointed to by the source variable and the undefined flag of the destination is set. In addition, the destination variable points to the same set of undefined values as the source. That is, if `A := B` is executed and the undefined flag of `B` is set, then the undefined flag of `A` is set, `A` is added to the undefined value set of `B`, and `A` is set to point to the same undefined value set as `B`.

2. If a value is assigned to a variable `A`, the undefined flag of `A` is cleared. If the flag was previously set, `A` is removed from the undefined value set it is pointing to.

3. If a value is assigned to a variable `A` with an undefined flag set by an external tool, such as the test generator, all the variables in the undefined value set of this variable are set to the same value and their undefined flag is clear. In this case, the undefined value set is also cleared.

4. If a variable `A`, whose undefined flag is set, is accessed in the design not for copying purposes (e.g., the undefined flag of `A` is set and the statement `C := A + B` is executed), the test generator is activated and it assigns a value to `A` before the access takes place. Note, according to 3 above, all the copies of `A` are assigned the same value.

The above algorithm can be added to existing simulators by adding the support for the required data structures to the infrastructure of the simulator without making any changes to the actual implementation of the design itself.

3. Improvements to the Basic Algorithm

In the basic algorithm for supporting late-binding, while maintaining correct simulation semantics, an undefined value can exist as long as the value is just copied or moved from place to place. If an attribute with an undefined value is involved in any other operation, the generator is called and the value of the attribute is resolved. This algorithm is useful in high-level descriptions, such as transaction level models, where many assignments exist and the operations on attributes may be complex. As the level of description gets lower, the number of pure copy operations gets smaller. Therefore, the possibility that an undefined value will survive and reach areas deep in the design is reduced. As a result, the chances that the ex-

Inputs		Basic Algorithm			"Smart" resolution		
A	B	A	B	Y	A	B	Y
0	0	0	0	0	0	0	0
0	1	0	1	0	0	1	0
0	U_B	0	R_B	0	0	U_B	0
1	0	1	0	0	1	0	0
1	1	1	1	1	1	1	1
1	U_B	1	R_B	R_B	1	U_B	U_B
U_A	0	R_A	0	0	U_A	0	0
U_A	1	R_A	1	R_A	U_A	1	U_A
U_A	U_B	R_A	R_B	$R_A \cdot R_B$	R_A	R_B	$R_A \cdot R_B$

Table 1. Evaluation of a simple AND gate

ample in Figure 1 will work in an RTL description of a processor are practically zero. On the other hand, many operation in RTL are simple Boolean functions such as AND, OR, NOT, and XOR. Therefore, efficient handling of late-binding in these functions can the technique useful at lower levels of abstraction, such as RTL.

Table 1 shows the truth table of a simple AND gate with two inputs, A and B, and an output Y. The two left columns in the table show the inputs to the gate before the output is evaluated. The other columns show the inputs and output of the gate after the output is evaluated. U_x represents an undefined value, and R_x represents an undefined value that was resolved (assigned a value by the generator). The middle of the table shows the evaluation of the gate when the basic algorithm is used. Since the AND gate is not a simple copy, each time the output is evaluated, the values of the inputs are resolved before the actual evaluation occurs. For example, the third row in the table shows the case when the value of A is 0, while the value of B is undefined. In this case, according to the basic algorithm, the value of B is resolved before the output is evaluated, although the value of B does not affect the value of Y.

The problem illustrated in the example above can be solved by using lazy resolution; that is, delaying the resolution of undefined values until they are actually needed to determine the value of the evaluated output. In the example above, the value 0 of A determines the value of the output, and therefore, with lazy resolution there is no need to resolve the undefined value of B.

When one of the inputs to the AND gate is 1, the gate just copies the value of the other input to its output. Following the spirit of the basic algorithm, if the value of the other input is undefined, it should be copied to the output of the gate. This "smart" resolution scheme is shown in the right side of Table 1. With this scheme, undefined values at the inputs of an AND gate need to be resolved only if both inputs have different undefined values. This scheme can be

$C \leftarrow$ all inputs with undefined value that the output is sensitive to;
while $|C| > 1$ **do**
  Choose an input c from C;
  Resolve the value of c;
  Remove from C any inputs to which the output is no longer sensitive;
if $|C| = 1$ **then**
  The value of the output is undefined;
  The output is added to the undefined value set of the input in C;
else
  The value of the output is determined;

Figure 2: Generic smart resolution algorithm

further improved by resolving just one of the inputs to the gate when both inputs are undefined.

To handle inverters and other inverting gates, we add an *inverted flag* to each attribute. This flag indicates whether or not the undefined value in the attribute is inverted, compared to the head of the set of attributes with the same undefined value. When an inverter with undefined input is evaluated, the output is added to the set of the input and the inverted flag of the output is set to the negation of the inverted flag of the input.

Using the "smart" resolution scheme and the inverter handling, we can handle all Boolean functions with two inputs without any resolution, unless both inputs are undefined. We can also handle more complex operators that are often used in RTL description, such as multiplexers. A more generic algorithm that handles arbitrary Boolean functions is described in Figure 2. The algorithm starts by constructing a set C of the inputs with an undefined value, to which the output is sensitive (i.e., the value of the output may be affected by the values of these inputs)[1]. Next, the algorithm chooses an input c in C and calls the test generator to resolve its value. There are many possible heuristics for choosing the input c. For example, choosing the input that creates the greatest reduction in C, choosing the input that is less needed to reach a required event, etc. After c is resolved, it is removed from C. In addition, all other inputs in C, to which the output is no longer sensitive, are also removed from C. These steps are continued until at most one input remains in C. If C is empty, the value of the output is determined. If one input is left in C, then the output is either a copy or a negation of the remaining input. In this case, the undefined flag of the output is set and the output is added to the undefined value set of the remaining input.

1 If determining whether the output is sensitive to an input is too difficult, C is initialized with all the inputs having undefined values

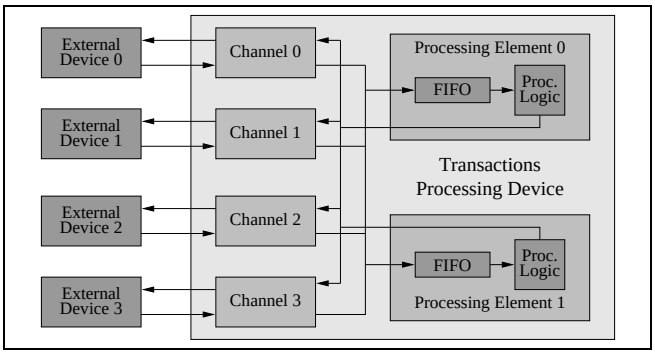

Figure 3: Block diagram of the I/O device

4. Experimental Results

We implemented the basic algorithm for late-binding support as part of the Odette Verification Environment [12]. The Odette Verification Environment provides verification support for object-oriented designs. It is built on top of the SystemC class library [10]. The current implementation is done outside the simulation kernel of SystemC. Therefore, it supports late-binding only for a small set of types, namely objects and data-members of objects that are built-in C++ types.

To check the effectiveness of the late-binding technique and its ability to improve coverage, we used it in the verification of a high-level model of an I/O device, whose block diagram is shown in Figure 3. The device receives requests through four channels and processes the requests using two processing elements. The processing element that processes each request is determined by the type of the request. Each channel, upon receiving a request, checks its validity and sends it to the proper processing element, where they are stored in a queue until the processing element can handle them. The processing elements read the requests in their queue one at a time and process them. Upon completion of the service, a response is sent to the requesting channel.

Because the processing elements service requests independently, and possibly at different rates, it is possible that requests are serviced out-of-order (i.e., two requests received in the same channel are serviced in the reverse order from which they were received).

The goal of the experiment was to cover all possible out-of-order events in the device. We compared three types of generation schemes: static generation, dynamic generation, and dynamic generation with late-binding. The generation strategy in all the schemes was to send a request to the processing element with the fuller queue, followed by a request to the other processing element. Since there is high probability that the first request will be delayed longer at the fuller unit, the chances of an out-of-order event increases. The implementation of this strategy in the three generation schemes was as follows:

- In static generation, the generator generates a sequence of requests to one of the processing elements in an attempt to fill its queue. It then generates a request to the other processing element.
- In dynamic generation, the generator examines the state of the two queues before a request is generated and sets the type of the request to one that is handled by the processing element with the fuller queue. The next request is sent to the other processing element.
- When late-binding is used, the type of the request, which determines the processing element in which it is served, is set to be undefined. When the input channel finishes its local processing of the request, it determines the processing element to which it will send the request by examining its type. Since the value of the type is undefined, the generator is called to resolve it. At this time, the generator can have a more accurate view of the state of the queues, and it sets the type to one that is serviced by the fuller unit. The next request is sent to the other processing element.

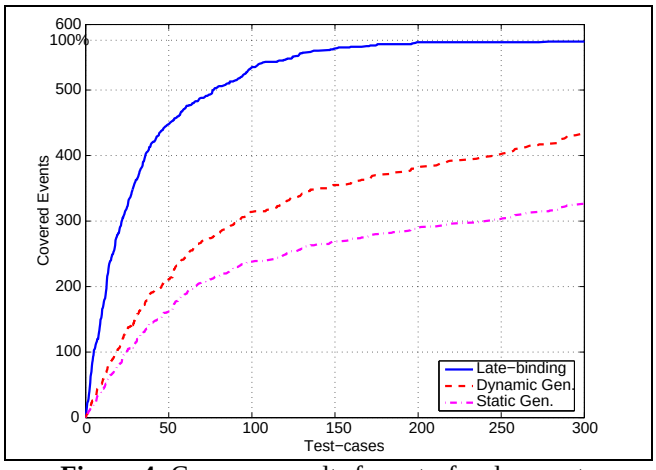

Figure 4: Coverage results for out-of-order events

Figure 4 compares the coverage of out-of-order events for the three generation schemes. The figure shows that 100% of the out-of-order events were covered by dynamic generation with late-binding, in less than 300 test-cases, while dynamic and static generation were able to reach only 75% and 60% of the events, respectively.

Figure 4 shows that late-binding can be used to improve coverage, but the question is at what cost? Handling the data structures needed to support undefined values can slow down simulation. Therefore, it is important to show that the gain from the increased capabilities of late-binding is greater than the cost of slower simulation. We measured the simulation time of test-cases on two models of the I/O device that differ in their level of abstraction. The measurement was done for three cases:

1. Without late-binding. That is, the late binding option is turned off.

	without late-binding	late-binding not used		late-binding used	
Model 1	20.5	22.1	(8%)	27.3	(33%)
Model 2	43.6	47.8	(10%)	54.3	(25%)

Table 2. Simulation time (in seconds) for the I/O device

2. Late-binding is turned on, but it is not used. That is, some attributes in the requests can have undefined values, but the generator never sets the values of the attributes to be undefined.
3. Late-binding is used. The generator follows the generation strategy for late-binding described above.

Table 2 compares the simulation time of these three cases, for the two models of the I/O device. The table shows that just the activation of late-binding slows the simulation by 10%. Using the late-binding capabilities increased the slow down to 30%. We looked for the reasons for this increase, and found that most of them can be attributed to time spent in the generator itself, and only a small part is spent in maintenance of the sets of attributes with the same undefined values. Therefore, we expect this slow down to be less dominant when the simulation rate decreases and simulation performance becomes more important. For example, while simulation time more than doubled in the second model (for the same test-cases), the overhead of using late binding increased from 5.2 seconds to 6.5 seconds, or by just 25%.

5. Conclusions

In this paper, we presented a technique to increase the ability of test generators to reach corner cases in the design by delaying the assignment of values in the generated stimuli until they are needed by the design. This late-binding technique allows the test generator to have a more accurate view regarding the state of the design, and thus it can more precisely choose the correct values that lead to a desired event. We described two algorithms that maintain the correct simulation semantics when late-binding is used: a basic algorithm that is suitable for high-level models, and a smart resolution algorithm for lower levels of description, such as RTL. Experimental results indicate that late-binding can significantly improve coverage with a reasonable penalty in simulation time.

One of the main challenges in using late-binding, and an area for future research, is combining late-binding with monitoring and checking tools. Naive monitoring of an attribute with an undefined value can lead to a premature resolution of the undefined value, and thus reduce the efficiency of the late-binding technique. On the other hand, waiting for a "natural" resolution of the undefined value can significantly complicate the work of the monitor, specifically if it is looking for a complicated scenario. Another challenge is an efficient implementation for late-binding in a design simulator, with minimal effect on the simulation speed and integration of the late-binding capabilities into test-generators and verification environments.

References

[1] A. Adir, E. Bin, and A. Ziv. Piparazzi: A test generator for micro-architecture flow verification. In *Proceedings of the High-Level Design Validation and Test Workshop*, November 2003.

[2] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of the 32nd Design Automation Conference*, pages 279–285, June 1995.

[3] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

[4] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[5] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In *Proceedings of the International Conference on Computer Design*, pages 580–583, November 1999.

[6] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-Gen: A random test-case generator for systems and SoCs. In *IEEE International High Level Design Validation and Test Workshop*, pages 145–150, October 2002.

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[8] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *Proceedings of the 38th Design Automation Conference*, pages 816–821, June 2001.

[9] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.

[10] SystemC 2.0 user's guide. http://www.SystemC.org/.

[11] J.-T. Yen and Q. R. Yin. Multiprocessing design verification methodology for Motorola MPC74XX PowerPC microprocessor. In *Proceedings of the 37th Design Automation Conference*, pages 718–723, June 2000.

[12] A. Ziv. Functional verification environment for object-oriented hardware designs. In *Forum on Design Languages (FDL)*, September 2003.