Temporal Environmental Assumptions in Simulation, Formal and Hybrid Verification

Ed Cerny[†], Ashvin Dsouza[†], Kevin Harer[†], Pei-Hsin Ho[‡]

[†] Verification Group, Synopsys, Inc.; [‡] Advanced Technology Group, Synopsys, Inc.

ABSTRACT

We present a method that enables developing *environment models* or *assumptions* using properties in property languages like SVA, OVA and PSL, or using RTL monitors in design languages like Verilog and VHDL, for pseudo-random simulation, formal property verification and hybrid verification. Our method also includes automatic dead-end avoidance and enables assume-guarantee reasoning. We demonstrate the effectiveness of the method on four real-world designs and environment models.

1. INTRODUCTION

Functional verification verifies register-transfer-level (RTL) designs against their functional specifications. Most designs make certain assumptions about their environments. For example, in Figure 1, the DUV may be designed with the assumption that whenever it makes a grant to the environment (*grant=1*), the environment will always remove the request in the next clock cycle (*req=0*).

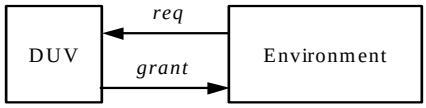

Figure 1 DUV and its environment

During the verification of the DUV, if we drive it with input stimuli that violate the assumption, then it needs no longer adhere to its functional specification. In that case, a subsequent erroneous behavior of the DUV is called a false negative, since it does not indicate a real bug of the DUV. To avoid distracting the user with false negatives, random simulation, formal property verification and hybrid methods [6] all require a model of the environment to verify the DUV.

```
Input clk, rst, req_input;
reg previous_grant; output req;

assign req = (previous_grant)? 0 : req_input;
always @ (posedge clk or negedge rst)
        if (!rst) previous_grant <= 0;
        else previous_grant <= grant;
```

Figure 2 Generator-style environment model

Traditionally, environment models are written as generator-style testbenches that generate random legal stimuli to drive the DUV. For the system in Figure 1, a generator-style testbench that drives the input signal *req* of the DUV is shown in Figure 2. The register *previous_grant* asserts if and only if the input *grant* asserted in the previous clock cycle. If *previous_grant* asserts, the signal *req* will be driven by the value 0; otherwise it will be driven by the random input *req_input*.

An alternative way to describe environment models is via checker-style constraints that specify the legal input stimuli for the DUV. Checker-style environment models can be written as properties in property languages like OVA [9], PSL [1] and SVA [15]. For example, the environment in Figure 1 can be specified as a property as follows.

```
if grant then #1 ! req;
```

Figure 3 Checker-style environment model as a property

The temporal operator "#1" means "one clock cycle later." Checker-style environment models can also be written as monitors in RTL Verilog or VHDL. A monitor is basically a design with an output signal that asserts if and only if the property is violated. A monitor for the environment in Figure 1 is shown in Figure 4. The output signal *fail* asserts if and only if the environmental assumption is violated. Note that the monitor does not drive the input *req* of the DUV (signal *req* is an input of the monitor) as in Figure 2. Verification tools have to understand the sequential monitor and drive the input *req* of the DUV in a way that does not assert the *fail* signal.

Today most simulators only support generator-style and most formal property verification tools only support checker-style environment models. As a result, verification engineers must specify the environment model twice in order to utilize both verification techniques. Therefore, supporting checker-style environment models for both random simulation and formal property verification will enable the user to specify the environment model once and use it "everywhere".

Furthermore, supporting checker-style environment models for both random simulation and formal property verification enables hybrid verification methods to orchestrate random simulation and formal property verification to achieve better verification coverage.

```
Input clk, rst, req, grant;
reg previous_grant; output fail;

assign fail = (previous_grant && req)? 1 : 0;
always @ (posedge clk or negedge rst)
        if (!rst) previous_grant <= 0;
        else previous_grant <= grant;
```

Figure 4 Checker-style environment model as a monitor

One major advantage of checker-style environment models over generator-style environment models is that the properties and monitors can be used as either assumptions to model the environment or assertions to check the behavior of the DUV, which enables the re-use of verification IPs. For example, suppose that we have a system of two blocks A and B that interact with each other; i.e., each block is the other block's environment. During the hierarchical verification of the system, the assumptions that we made about block A during the verification of block B should be used as assertions for verifying block A (to guarantee that the assumptions are valid), which is called *assume-guarantee reasoning[10]*.

Recent result in [21] shows that generator-style environment models can also be made to support assume-guarantee reasoning but would require new verification techniques that no existing commercial formal property verification or random simulation tools support today.

The rest of the paper is organized as follows. In Section 2 we discuss the related work. We go through the terminology in Section 3 before we describe our method. In Section 4 and Section 5 we introduce the method for handling temporal properties and sequential monitors in formal property verification and random simulation, respectively. In Section 6, we discuss the dead-end state problem and an automatic method to avoid dead-end states. We present experimental results in Section 7 and conclude the paper in Section 8.

2. RELATED WORK

The Simgen framework was first presented in [11][17][18][19][20] for handling environment models specified as combinational properties (constraints) for random simulation. Simgen utilizes a Binary Decision Diagram (BDD)[4] based constraint solver that effectively converts the combinational properties into a BDD at compile time before random simulation. Each path from the root to the leaf node 1 of the BDD corresponds to a valid input vector for the DUV. At each clock cycle of random simulation, according to the current value of design output signals, Simgen randomly walks in the BDD to generate a valid input vector for the DUV. The random walk also respects user-specified biasing constraints for input signals of the DUV. In [13], the BDD is rebuilt at each cycle of random simulation. In our experience, this method greatly slows down random simulation and should be used only if the method in [11][17][18][19][20] cannot complete the buildup of the BDD at compile time. The above methods however do not directly support sequential checkers (temporal properties as in Figure 3 and sequential monitors as in Figure 4). To handle sequential checkers using the above methods, the user has to manually separate the combinational constraints from the state machines and feed the combinational constraints to the BDD-based constraint solver. Our method automates this process for sequential checkers for not only random simulation but also formal property verification and hybrid methods and uses a combinational constraint solver [8] that is an enhancement of the Simgen technology.

The methods in [7][10][16] convert combinational checkers into BDDs and then convert the BDDs into gate-level combinational generators for both random simulation and formal property verification. In our experience, the combinational generators converted from the BDDs usually are much more complex than the original combinational checkers, which makes formal property verification highly inefficient. Our method supports not only combinational checkers but also sequential checkers in formal property verification without suffering the same complexity blowup problem.

One can also build environment models that consist of both checker and generator style constraints. Commercial testbench automation tools Vera [5] and Specman [12] enable the users to write combinational checker-style environment models to supplement their sequential generator-style environment models. Our method should enable these commercial tools to support sequential checker-style environment models.

3. PRELIMINARIES

We introduce in this section the high-level flow and formalism that are required for describing our method in later sections.

3.1 High level flow that converts checkers into gate-level networks

Checker-style environment models can be specified as temporal properties in property languages like OVA [9] and PSL [1] or sequential monitors in RTL languages like Verilog [3] and VHDL. Each RTL monitor has an output signal that asserts if and only if the corresponding property is violated. The properties and monitors can be used as either assumptions or assertions.

We apply the following process to obtain the gate-level networks from the assertions and assumptions. We first convert OVA or PSL properties into RTL monitors using OVA or PSL compilers [1]. For example, the property in Figure 3 will be automatically converted into the monitor in Figure 4.

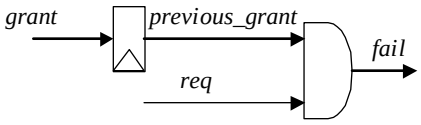

Figure 5 Gate-level monitor

Second we convert all the user-specified and automatically generated RTL monitors into gate-level networks using logic synthesis techniques. Figure 5 shows the gate-level network converted from the monitor in Figure 4, where the box represents a register with clock signal *clk* and reset signal *rst*. The register stores the last value of the signal *grant*. We will use this gate-level netlist to extract BDDs for random simulation. In addition, the gate-level networks of the assertions and the assumptions will be assembled to form a gate-level model for formal property verification.

3.2 Finite State Automata

The semantics of the gate-level network of a checker is a Finite State Automaton (FSA) that recognizes the sequences characterized by the checker. The automaton enters a rejecting state when the observed sequence becomes invalid. The rejecting

state for the example in Figure 5 represents the state where the *fail* signal asserts.

The FSA is a 6-tuple $A = (X, Y, S, T, s_0, S_r)$, where

1. X is the set of values of the input signals of the DUV that are to be constrained,
2. Y is the set of values of the signals of the DUV,
3. S is the set of states of the FSA,
4. T is the state transition relation $T \subseteq S \times (X \times Y) \times S$,
5. $s_0 \in S$ is the initial state, and
6. $S_r \subset S$ is the set of rejecting states (indicating the violation of the checker).

Both X and Y are input values of the FSA. Also, we shall refer to T by its characteristic function:
$T(s, (x, y), s'): S \times (X \times Y) \times S \rightarrow \{0, 1\}$.

To use the FSA as an assumption means that the DUV input values x of X are to be restricted in such a way that the DUV never enters a rejecting state s_r. This means that if the FSA is in state s that is not a rejecting state, then the input values of X that make the FSA transition to a rejecting state at the next clock tick must be prohibited. In other words, the input value x of X must satisfy the constraint $\neg T(s, x, y, s_r)$. A combinational constraint solver based on the Simgen technology [18] can randomly generate an input value x of X that satisfies the above condition.

More specifically, given the current value of s and y, the combinational constraint that has to be solved for the design input value x at each clock tick is obtained as

$$C(s, x, y) = \prod_{\sigma \in S - S_r, s_r \in S_r} \neg[(s = \sigma) \wedge T(\sigma, (x, y), s_r)]$$

where Π stands for Boolean product over all values σ in $S\text{-}S_r$ and rejecting states s_r in S_r. For gate-level networks, the above constraint is actually the negation of the combinational transitive fan-in cone of the fail signal. For the gate-level network in Figure 5, the combinational constraint C for the constraint solver is the logic inside the dotted box in Figure 6; i.e. $C = \neg(previous\_grant \wedge req)$.

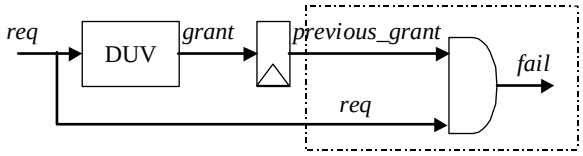

Figure 6 Checker-style assumption for random simulation

During simulation, the DUV and the sequential checker are simulated together as shown in Figure 6. The DUV signal value y is sampled sometime after the clock tick when y has stabilized. The input value x must then be applied so that the DUV has enough time to stabilize before the next clock tick. The sampled values of x and y by the FSA force it to make a transition. If there is no combinational loop created by the assumption monitors and the DUV, then it is sufficient to execute the combinational constraint solver and drive the input value x only once per clock cycle, because the value y cannot change as function of the value x before the next clock tick.

4. FORMAL PROPERTY VERIFICATION

Given the DUV, a set of assertions and a set of assumptions, we can build a formal verification model as follows. First, we feed all outputs of the assertion monitors to an OR gate called *assert_fail* and feed all outputs of the assumption monitors to another OR gate called *assume_fail*. As a result, the signal *assert_fail* (*assume_fail*) asserts if and only if an assertion (assumption) is violated. Second, we connect the *assert_fail* and *assume_fail* signals in the way shown in Figure 7 to generate the output signal *fail*.

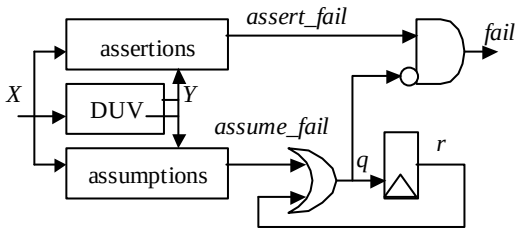

Figure 7 Formal verification model

In Figure 7 the signals X are DUV inputs and the signals Y are DUV signals. The assertions and assumptions monitor both the X and Y signals. The register output r latches the output q of the OR gate and thus will stay high once it goes high; i.e., the output q asserts if and only if an assumption has been violated. The *fail* signal is the conjunction of *assert_fail* and the negation of q, so it asserts if and only if an assertion is violated and no assumptions has been violated. Therefore, the assertions are formally proven if and only if the *fail* signal is formally proven to be a constant zero.

We simply present the formal verification model to formal property verification or hybrid verification tools. Formal property verification engines will try to either prove that the fail signal will never assert (all assertions are proven) or find a counter example that asserts the *fail* signal (some assertions are violated).

5. RANDOM SIMULATION

As indicated in Section 3, when the DUV stabilizes after the most recent active clock edge, we (1) sample the value y of the DUV signals and the value s of the assumption FSA, (2) solve the constraint C with the values y and s to obtain a random value x, and (3) drive the DUV inputs with the random value x.

For example, suppose that the active clock edge for all registers is "*posedge clk*" and the DUV stabilizes after P/4 where P is the clock period. First, we can sample the values of y and s at P/4 after each "*posedge clk.*" Second, if there exists a solution that satisfies the constraint C, we can obtain a random solution x from the constraint solver. Third, we can drive the value x to DUV inputs any time that leaves enough time for DUV to stabilize after the input stimulus and before the arrival of the next "*posedge clk.*" For example, we can drive the inputs of the DUV at "*negedge clk.*"

At the following *posedge clk*, the values of *y* and *s* are again sampled, and the assumption FSA advances to the next state as determined by the state transition relation *T*. Since we solved for *x* using the stable values of *y* and *s*, the assumption FSA cannot be in a rejecting state. This process repeats until the random simulation is complete.

Notice that although we are dealing with sequential checkers, we only need to solve a combinational constraint system *C* at every active clock edge. But because the combinational constraint solver cannot predict the future, there is a possibility that at a given active clock edge, the constraint system *C* does not have any solution. In that case we say that the simulation is in a *dead-end state*. We address this problem in the next section.

6. DEAD-END STATES

To illustrate the problem of dead-end states in sequential checkers, consider the following set of assumptions defined in OVA, where *p, q* and *r* are inputs to a DUV clocked by *clk*.

```
clock posedge clk {
  event evA: if p then #2 ~r;
  event evB: if q then #1 r;
}
```

Figure 8 Assumption with dead-end states

The assumptions say that if signal *p* is set to 1 then two cycles later signal *r* must be set to 0, and if signal *q* is set to 1, then one cycle later signal *r* must be set to 1.

Now, given the assumptions, the combinational constraint solver can legitimately set *p* to 1 in a certain cycle, and then set *q* to 1 in the next cycle. However, in the following cycle, the solver is unable to solve for *r*, and the system has reached a dead-end state. In that case, we can generate a warning message, reset the design and resume random simulation from the reset state. If we get into dead-end states very often during the random simulation, the verification coverage and simulation time may suffer due to repeated resets.

The interesting aspect of this dead-end state is that it occurs because of the solutions chosen by the solver in earlier cycles. If the solver chose different solutions earlier, the dead-end state would not have occurred. For example, after setting *p* to 1, the solver had not set *q* to 1 in the next cycle, it would have avoided the dead-end states described.

It is possible for the constraint solver to look ahead some number of cycles when solving the constraints. In this case, if the solver always looked ahead one cycle, then it would know not to set *q* to 1 one cycle after setting *p* to 1. However, this slows down simulation by making the solver slower, and is also susceptible to the horizon effect: no matter what look-ahead was used, a dead-end state might occur just outside the limit.

A better solution is to consider strengthening the assumptions themselves to prevent solutions that lead to a dead-end state. For example, we can add an assumption to the assumptions in Figure 8 to produce the assumptions in Figure 9:

```
clock posedge clk {
  event evA: if p then #2 ~r;
  event evB: if q then #1 r;
  event enC: if p then #1 ~q;
}
```

Figure 9 Assumptions without dead-end states

The extra assumption says that if *p* is set to 1 then *q* must not be set to 1 in the next cycle. This prevents the dead-end states from occurring. In fact, there are no dead-end states possible with the assumptions in Figure 9.

The technique that we developed for dead-end state avoidance builds on this observation to automatically compute the weakest assumption necessary to avoid dead-end states due to the assumptions.

We also extend our technique to avoid, where possible, dead-end states due to the DUV. To illustrate this, consider what might happen if *p* and *r* in Figure 8 were DUV inputs, but *q* was a DUV output. Then we could not prevent the DUV from setting *q* to 1 one cycle after the constraint solver set *p* to 1. The only way for the constraint solver to guarantee no dead-end states in this case would be to never set *p* to 1. The assumptions of Figure 8 augmented with this assumption are shown in Figure 10. This is a stronger set of assumptions than that generated in Figure 9.

```
clock posedge clk {
  event evA: if p then #2 ~r;
  event evB: if q then #1 r;
  event evC: p==0;
}
```

Figure 10 Strong assumptions without dead-end states

The danger of using knowledge of future design output values in the dead-end avoidance procedure is that it could avoid detecting real design errors. Therefore, this feature has to be used with care as explained in Section 6.3.

6.1 Avoidance algorithm for deadend states in assumptions

Let $A = (X, Y, S, T, s_0, S_r)$ be an assumption checker FSA.

We first compute a fixed point that will represent all states that inevitably lead to a dead-end state in later cycles. Since this is a fixed-point computation, there is no bound on how much later the dead-end state may occur.

$$
\begin{aligned}
D_0 &= S_r \\
D_{k+1} &= D_k \cup \{ s \mid \forall y. \forall x. \forall s' T(s,(x,y),s') \rightarrow s' \in D_k \} \\
D &= \bigcup_i D_i
\end{aligned}
$$

Since *X, Y* and *S* are finite, the sequence must converge in a finite number of iterations to some *D*.

We next compute the set of reachable states, *R*, of *A* using a fixed-point calculation:

$$R_0 = \{s_0\}$$
$$R_{k+1} = R_k \cup \{s \mid \exists x.\exists y.\exists s'T(s',(x,y),s) \wedge s' \in R_k\}$$
$$R = \bigcup_i R_i$$

Finally, we define a new FSA, *A' = (X, Y, S, T, s_0, R∩D)*, and use *A'* in place of *A* as the assumption. If *D* includes s_0, then there is no way to avoid dead-end states. If *R∩D* is a subset of S_r, then the assumption is free of reachable dead-end states. Both of the above algorithms are implemented using the CUDD BDD package [14]. All the set-theoretic operators in the above algorithms can be implemented as Boolean operators in BDDs. Given the set of assumptions in Figure 8, this algorithm would automatically extend it to produce the assumptions in Figure 9.

6.2 Avoidance algorithm for deadend states in assumptions and DUV

The algorithm we have described in the previous section is guaranteed to avoid any dead-end states inherent in the assumptions. However, as described earlier, the DUV can also interact with the assumptions to cause dead-end states. We can add an even stronger assumption to prevent this from happening by using the following fixed-point computation for D:

We assume that there is no combinational path from any of the DUV inputs to any of the DUV outputs. In this case, the computation of the fixed point, *D*, is modified as follows:

$$D_0 = S_r$$
$$D_{k+1} = D_k \cup \{s \mid \exists y.\forall x.\forall s'T(s,(x,y),s') \rightarrow s' \in D_k\}$$

The rest of the procedure remains the same. However, this may result in the dead-end-state voidance assumption being too strong. For example, given the assumptions in Figure 8, where *q* was a DUV output, this algorithm would produce the set of assumptions in Figure 10. Again, the above algorithm is implemented using the BDD data structure.

6.3 Methodology

Note that dead-end states might be the result of a bug in the design. In addition, the input stimuli that lead the design into dead-end states might be the only input stimuli that can reveal some tough bugs of the design. In both cases, we may miss the detection of the bug if we always turn on automatic dead-end avoidance during random simulation.

Therefore, we propose the methodology that the user should first try random simulation without automatic dead-end avoidance. When dead-end state happens, the user should examine the cause of the dead-end states. If the cause is most likely due to incomplete assumptions, the user has the option to manually strengthen the assumptions to remove the dead-end states or turn on the automatic dead-end avoidance feature to save engineer's time. Note that formal property verification does not require dead-end avoidance and is not impacted by dead-end states.

7. EXPERIMENTAL RESULTS

For formal property verification, our method builds a formal verification model by simply hooking up the assumptions, the assertions and the DUV together with a few extra logic gates. As a result, check-style environment model introduces little overhead to formal property verification engines.

The more interesting question is whether our method's constraint solving and automatic dead-end avoidance operations would introduce great overhead to random simulation. To answer this question, we performed experiments on four real-world designs and environment models: USB2, PCI Express, and two other designs with coded names. Table 1 shows the statistics of the designs.

Table 1 Design statistics

Design	#gates	#registers	#inputs	#outputs
USB2	262K	7000	284	329
PCIE	60K	2352	250	28
C-1	39K	1784	252	355
C-2	64K	1452	319	232

The environment models were specified in both checker and generator styles. For the checker-style environment model, we performed the experiments with and without automatic dead-end avoidance. The sizes of the environment models were measured in terms of the sizes of the gate-level networks as well as the numbers of the BDD nodes in the combinational constraint solver. Simulation run time was measured for 250K clock cycles of random simulation with a commercial simulator on a 750MHz SPARC processor.

Table 2 Comparison of the run times

Design	Env. models	#BDD nodes	#registers	#gates	CPU Time
USB2	generator	N.A.	303	62K	624
	checker	3423	281	52K	745
	dead-end	4040	281	52K	550
PCIE	generator	N.A.	252	1.6K	351
	checker	207	32	600	349
C-1	generator	N.A.	14	6135	1567
	checker	250	17	6163	1548
C-2	generator	N.A.	20	641	2173
	checker	353	12	922	1894
	dead-end	380	12	922	2005

Table 2 presents the experimental results. The second column shows the number of BDD nodes in the combinational constraint solver. With dead-end avoidance, extra constraints may result in an increase on the number of BDD nodes in the constraint solver. The checker-style environment models for PCIE and C-1 are dead-end free, so we do not report their dead-end avoidance data.

The fifth column shows the CPU seconds for running 250K cycles of random simulation. For the USB2 design, with dead-end avoidance, the checker-style environment model becomes faster than the generator-style environment model. This is possible because the time for the constraint solver to generate a random solution is proportional to the average path length in the BDD and not the total number of nodes in the BDD. We believe, however, that the performance improvement achieved here by dead-end avoidance is more accidental than normal. The compile time for building the BDDs in the constraint solver and deadend avoidance computation is less than 1 CPU minute for these four examples with the exception that the deadend avoidance computation took approximately 10 CPU minutes for the USB2 design.

We would like to draw the conclusion that for these four examples our method of handling checker-style environment models, with and without automatic dead-end state avoidance, does not incur much performance overhead over generator-style environment models in random simulation.

8. CONCLUSIONS

We have described a method for handling checker-style environment models specified as either sequential RTL monitors or temporal assumptions for random simulation, formal property verification and hybrid verification. This method therefore enables the user to (1) specify the environment model once and use it with multiple verification techniques, (2) reuse temporal properties as both assertions and assumptions and (3) employ assume-guarantee based verification methodology.

Unlike the previous work, our method can automatically handle sequential RTL monitors or temporal assumptions; the user does not need to manually extract combinational constraints for the combinational constraint solver. In addition, we also present a method that automatically avoids dead-end states during random simulation.

We tested the implemented algorithm on four real-world designs and environment models. The experimental results show that our method of handling checker-style environment models does not incur much performance overhead over generator-style environment models in random simulation. In the future, the dead-end avoidance algorithm can be improved in many ways, including partitioning the assumptions, using BDD approximation techniques, and exploiting don't-care optimizations during the computation of the fixpoints *D* and *R*.

9. ACKNOWLEDGEMENTS

We would like to thank William H. Nicholls and Carl Pixley for providing us the combinational constraint solver used in our method; Steven R. McMaster and Yunshan Zhu for providing us the test cases; Oliver Kozber, Vishwa Raman and Jerry Taylor for implementing the working prototype of this method; and Stephen Meier for his support of this work.

10. REFERENCES

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, Y. Wolfsthal. FoCs – Automatic generation of simulation checkers from formal specifications, In Proceedings of CAV, 2000.

[2] Accellera Property Specification Language (PSL), Reference Manual, Version 1.01, April 2003.

[3] Accellera Open Verification Library (OVL), Assertion Monitor Reference Manual v 03.06.06, June 2003.

[4] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transaction on Computers, C-35(8):677-691, 1986.

[5] F.I. Haque, K.A. Khan and J. Michelson. The art of verification with VERA. Verification Central, 2001.

[6] 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 Proceedings of ICCAD 2000.

[7] J.H. Kukula and T.R. Shiple. Building circuits from relations. In Proceedings of CAV 2000, pp.112-123.

[8] W.H. Nicholls and C. Pixley. SoCKit combinational constraint solver, private communication, 2003.

[9] Open Vera Assertions (OVA), v1.3, Synopsys, January 2003.

[10] C. Pixley. Integrating model checking into the semiconductor design flow. Computer Design's Electronic System Journal, pp.67-74, March, 1999.

[11] C. Pixley, K. Shultz, J. Yuan. Integrated formal and informal design verification of commercial circuits. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA), pp. 1061-1067, 1999.

[12] S. Regimbal, J.-F. Lemire, Y. Savaria, G. Bois, E.-M. Aboulhamid, and A. Baron. Applying aspect-oriented programming to hardware verification with ***e***. Proceedings of HDLCON, 2002.

[13] K. Shimizu and D. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In Proceedings of DAC 2002, pp.801-806.

[14] F. Somenzi. CUDD: CU Decision Diagram Package. ftp://vlsi.colorado.edu/pub/.

[15] SystemVerilog 3.1, Accellera's Extensions to Verilog, June 2003.

[16] J. Yuan, A. Aziz, K. Albin, and C. Pixley. Constraint synthesis for environment modeling in functional verification. In Proceedings of DAC 2003.

[17] J. Yuan, A. Aziz, K. Albin, and C. Pixley. Simplifying Boolean constraint solving for random simulation-vector generation. In Proceedings of ICCAD 2002, pp.123-127.

[18] J. Yuan, C. Pixley, A. Aziz, K. Albin. A framework for constrained functional verification. In Proceedings of ICCAD 2003, pp.142-145.

[19] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using BDDs. In Proceedings of ICCAD 1999, pp. 584-589.

[20] J. Yuan, K. Shultz, C. Pixley, and H. Miller. SimGen: A tool for automatically generating simulation environments from constraints. Proceedings of the ITC Microprocessor Test and Verification Workshop, 1998.

[21] Y. Zhu and J.H. Kukula, Generator-based verification, In Proceedings of ICCAD 2003, pp.146-153.