Practical Application of Formal Verification Techniques on a Frame Mux/Demux Chip from Nortel Semiconductors

Y. Xu[1], E. Cerny[2], A. Silburt[1], A. Coady[1], Y. Liu[1], P. Pownall[1]

[1]Nortel Semiconductors, Ottawa, Ontario, Canada
{xuying, silburt, acoady, yingliu, philp}@nortelnetworks.com

[2]Dépt. IRO, Université de Montréal, Montréal, Québec, Canada
cerny@iro.umontreal.ca

Abstract. We describe the application of model checking using FormalCheck to an industrial RTL design. It was used as a complement to classical simulation on portions of the chip that involved complex interactions and were difficult to verify by simulation. We also identify certain circuit structures that for a certain type of queries lend themselves to manual model reductions which were not detected by the automatic reduction algorithm. These reductions were instrumental in allowing us to complete the formal verification of the design and to detect two design errors that would have been hard to detect by simulation. We also provide a technique to estimate the length of a random simulation needed to detect a particular design error with a given probability; this length can be used as a measure of its difficulty.

1 Introduction

The rapid increase in the complexity of microelectronics systems and the degree of integration of subsystems and systems on one silicon chip make their design and verification increasingly difficult. The amount of Verilog or VHDL RTL code written to describe a system for the synthesis tools is large, but the simulation test bench code written to verify that the chip satisfies its specification is much larger, and growing [1]. This is a direct consequence of the desire to achieve short time to market while producing an error free design, and due to the growing size of the chips to be verified. Many techniques and tools are being introduced on the market to improve the situation, to cut down on the amount of code and/or to improve the verification coverage of the design. The list includes:

Simulation oriented:

- Improved verification productivity through better test-bench construction environments [8].
- Reduced number of testbench cases by using random input sequences [6, 15].
- Improved observability through automatically generated property checkers [6].

L. Pierre and T. Kropf (Ed.): CHARME'99, LNCS 1703, pp. 110-124, 1999.

- Improved coverage evaluation through classical software code-coverage techniques [e.g., 9] or based on state / transition information of the network of finite-state machines underlying the design [6].

(Semi-)Formal methods:

- Symbolic model checking based on verifying a temporal logic specification of properties [13, 5] or using language inclusion tests of ω-automata [4, 10].
- Partial state-space exploration (simulation and symbolic exploration combined) [7].
- Theorem proving at higher levels of design abstraction [2, 14].

Although simulation remains the main verification workhorse, formal verification (model checking in particular) using commercially available tools is making slow inroads when verifying complex interactions between design modules.

In this paper, we describe our experience in formal verification using FormalCheck [4, 10] (a model checker based on language inclusion test of ω-automata) from Lucent Technologies as applied to a large ASIC design. We show that a judicious deployment of FormalCheck on a small subset of modules was efficient and highly beneficial to the improvement of the quality of a Frame Multiplexer/Demultiplexer chip (called FMD in what follows).

The principal problem in the application of a model checker to a large design is the so-called *state explosion*: the representation of the state space that has to be maintained by the tool during verification growing beyond the available computer memory. The result is that the verification cannot be completed. To counter this problem, one should concentrate on smaller subsystems that are difficult to verify by simulation. Usually these are complex control structures implemented using a number of cooperating state machines. In addition, one should consider scaling down any datapaths, register arrays, and / or constraining the input space. Both of these techniques may change the behavior of the design or limit the state-space exploration, thus making the verification result less useful or even invalid. A better solution is to apply model reduction or abstraction techniques. These can simplify the model representation by introducing complete nondeterminism on state variables in those parts of the design that should not have influence on the properties to be verified. If the property holds even after the reduction, then it is guaranteed to hold on the original design. If not, then either a lesser reduction must be used or, there is a design error which then should be confirmed by simulating the error trace (counter-example) produced by the model checker. In our verification work, we applied model checking to Context Switch, which is a critical subsystem of FMD. We scaled down the number of channels, and developed a reduction technique that allowed us to successfully complete the verification when combined with the automatic model reductions carried out by Formal Check. In the process, two important design errors were discovered. The contributions of this paper can thus be summarized as follows:

- A description of the verification approach: isolation of the modules, creation of a model of the use environment, and formulation of properties.

- Identification of model structures that can be considerably reduced by converting the majority of state variables into primary inputs. This makes the next-state transition function of these state variables completely non-deterministic. Yet, the automatic reduction algorithm as implemented in FormalCheck (Cospan [4]) were not able to detect this reduction possibility. By combining the power of our manual reductions with the automatic ones, the Context Switch subsystem was efficiently verified in 512Mb of memory.

We also describe a measure of "hardness" of a design error detected by a model checker. It consists of computing an estimate on the length of a random input sequence that would detect the particular error with a given probability (e.g., a measure of how long it would take to detect the error by random simulation).

The paper is organized as follows: In Section 2 we briefly describe our design and the verification environment. In Section 3 we introduce the FMD design and the overall verification strategy. In Section 4 we concentrate on the Context Switching subsystem that was deemed to require verification by model checking, the classes of properties verified, and the design errors detected. Then, in Section 5, we discuss a model reduction needed for the verification to complete. In Section 6 we conclude the paper. The appendix contains a description of a method for computing the length of a random simulation sequence to detect a design error previously identified by a model checker.

2 The design verification environment

The RTL design of a chip like the FMD is carried out by a team of designers according to the specification of the product. The verification of its component blocks is done by simulation by the respective designers before chip-level integration. The interaction of the blocks and the conformance of the design to the specification is verified by simulation of the whole chip once the RTL code is ready. This simulation follows a test plan that identifies all the features whose functionality must conform to the specification.

It is well known that features involving complex interactions between internal design modules are hard to verify by simulation, and that model checking can be effective in such cases. This was exactly the case of certain parts of the FMD design. We therefore deployed FormalCheck to those portions of the design that would have required long simulation sequences and yet with no guarantee on the completeness of verification.

In the following section we describe briefly the FMD design and the portion verified using FormalCheck.

3 The design of the Frame Multiplexer/Demultiplexer

The FMD chip is part of a system used in multiplexing/demultiplexing framed data between various channels and a SONET line. It consists of some 250k gates. The architecture contains a datapath over which the data frames travel and are processed, under the control of cooperating state machines. Since the frame and mux/demux parameters may depend on the specific connection and channel number (0 to 27), each

time a piece of information is to be processed, its "context" information is retrieved from a Context Memory, the data and the context are updated and then written back to memory. The accesses to the memory are thus shared by the datapath and by the controlling processor. The controlling processor sets up the initial context information and retrieves status information from internal registers and the memory asynchronously relative to the datapath operations. There are thus concurrent accesses to the Context Memory by the Datapath and the Microprocessor.

The arbitration between the accesses and the retrieval / storing of the context information is handled by a Context Switch subsystem. A Datapath access always takes precedence, and thus a Microprocessor access is held off until a free cycle exists in which the Datapath is not accessing the Context Memory. Furthermore, a Microprocessor access can piggyback on a Datapath access. For power consumption and efficiency concerns, the Context Memory is subdivided into 6 memory blocks according to the type of information and frequency of access, hence, there are 6 Context Memory controllers. There are also 28 Facility Data Link (FDL) FIFOs in the Context Switch (one FIFO for each Datapath channel). The control structure in the Context Switch (Figure 1) is complex, and it also contains queues for the requests and a 3-stage pipeline for the Read-Modify-Write processing sequence on the data. To minimize

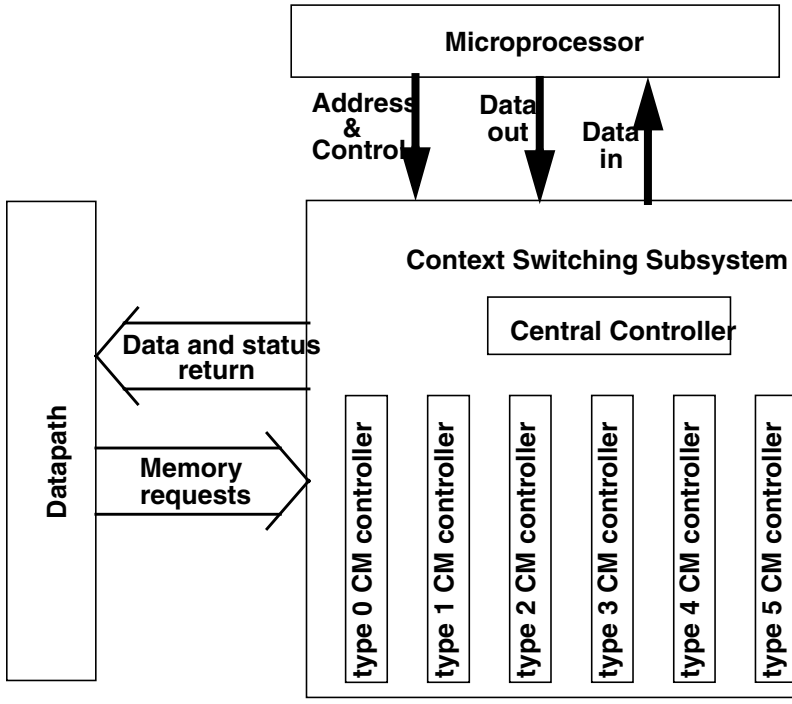

Fig. 1. Bock diagram of the Context Switch subsystem of FMD

power consumption, many of the pipeline registers are only virtual, i.e., they do not physically appear in the design. This is possible, because the data flow pattern is such that in some cases buffering over two cycles is sufficient. This overall complexity makes the verification of the Context Switch quite difficult by simulation, because it is hard to imagine all the input combinations that force different orders of events inside the Context Switch.

It follows that the design blocks of the Context Switching Subsystem were identified as the natural candidates for formal verification. We isolated them, defined their operating environment (non-deterministic state machines and constraints on the inputs) and then verified their operation using a set of properties derived from the global test plan, from the intimate understanding of how the Context Switch should operate based on the chip-level specification, and from RTL code reviews. In the following sections we describe the typical structure of the properties that were verified and then discuss the necessary model reductions for the verification to be completed on a Ultra Sparc 10 workstation with 512Mb of memory. The Context Switching blocks represent some 20k gates described in about 3500 lines of RTL Verilog code, not counting comments. There are some 840 state variable bits. Depending on the property and the reductions achieved, the reachable state space consists of 10^{12} to 10^{20} states (as reported by FormalCheck).

4 The properties and the design errors detected

A total of 110 queries were verified using FormalCheck on the Context Switch, some queries consisted of a number of properties. We primarily targeted the mutual exclusion of accesses to the context memories by the Datapath and the Microprocessor, and the proper emission of control signals to the memory. We did not verify the correctness of the contents stored in the memory, since the memory is (logically) correct by construction and the physical data connections to it were verified by traditional simulation test benches. In addition, the various information fields in the memory were also tested by simulation.

4.1 Properties

Due to the similarities among the memory controllers in the receive and transmit context switching blocks, the 110 queries can be subdivided into 4 classes discussed below. We also constructed (in Verilog) a non-deterministic model of the user that mimics the normal operating environment in which the Context Switch is utilized. In what follows, the underlined words are keywords of the FormalCheck Property specification user interface.

1. Unicity of the Microprocessor accesses to the receive-side Context Memory. This query consists of 9 properties which check that only one of the possible Microprocessor accesses is valid in any clock cycle. These are all safety properties of the form "*never condition*", where *condition* is a boolean formula over the control signals involved in the accesses. It describes the possible contention situations.

2. A Datapath write to the Context Memory must be preceded by a read 2 clock cycles earlier. The address value itself to the Context Memory is not verified. Again, this safety property has the form "*never condition*", except that *condition* now involves delayed control values. These delays were either already present in the design or they were obtained using state machines built from within the user interface of Formal Check using if-then-else constructs and state variables.
3. When the Microprocessor requests access to a data select register, then the 7th bit in the address must be 1. This safety property has the form[1] "*always condition => conclusion*", where *condition* characterizes the particular type of access and *conclusion* verifies that the 7th bit is actually 1.
4. All Microprocessor read or write requests to the context memories are eventually acknowledged. This is a liveness property verified under the fairness constraint that there is no Datapath accessing the memory for 4 consecutive clock cycles. It has the form "*After condition Eventually conclusion*", where *condition* describes the signal values representing a request, and *conclusion* describes the acknowledgment. The fairness constraint is constructed with the help of additional state variables implemented from the user interface of FormalCheck.

In the following we mention two properties that have a structure similar to those we have described, but which each had a major impact on the verification. Either because it lead us to the identification of circuit structures amenable to manual reductions (Property A), thus allowing us to complete the verification of many other properties, or because it detected important and difficult design errors (Property B).

Property A: The Datapath FDL FIFO consumer (head) and producer (tail) indexes are updated correctly on each such queue. The size of each FIFO is 128 1-byte entries. For example, when the Microprocessor reads a byte from the FIFO of channel 3, then the consumer index should be increased by one. The component properties of this query have a similar structure as (4) above, however, it is mentioned here because:

a) The original property definition had to be broken up into two parts, so as to implement the verification over the modulo 128 incrementation of the indexes: We first consider the case when the index is in the range of 0-126, and then the case when the index is changing from 127 to 0.
b) As we mentioned earlier, there are 28 FIFOs accessed using a common path. The selection of which FIFO is to be incremented is based on information contained in the request. The verification could not complete because of the number and the size of the FIFOs. In spite of the fact that the property would select a specific queue, the automatic reductions could not identify a sufficiently large number of state variables to reduce. Yet, as will be seen in the next section, the structure of the model and the property allows more powerful reductions which, if carried out manually, allow completing the verification of this kind of property on the design.

---

1. Implication “A => B” is replaced by “(not A) or B” to specify the proposition to FormalCheck.

Property B: When a valid Microprocessor read request to the Context Memory is acknowledged, the read address (channel number) issued to the memory 2 cycles earlier is the same as the channel number specified by the Microprocessor.

Table 1 shows the CPU time, the memory size, the total state space, and the reachable states for the queries involving Properties A and B. Notice that Property A was verified after the reduction in only some 30 minutes of CPU time, while without the reductions it could not complete at all. The execution time of Property B is typical of the longest verification times we encountered. Either it would complete within about 90 minutes, or it would run out of physical memory and start heavily paging (thrashing), since model checking (like simulation) has relatively poor spacial locality in memory accesses. We would thus periodically observe the status of the verification process using the Unix *ps* command. Normally, on an unloaded workstation, the CPU utilization would be above 80%. When it runs out of memory and starts thrashing, the CPU utilization by the process drops below 5% and it is time to terminate the process (unless one wishes to wait almost indefinitely or until the process runs out of the allowed virtual address space).

Table 1. Verification Statistics

	CPU time (minutes)	Memory (MB)	total states	reachable states
Property A	34.4	27.40	8.44e14	1.95e12
Property B	85.7	167.72	1.77e21	8.70e19

4.2 Error detection

As mentioned earlier, due to the concurrent accesses to shared resources, the number of possible behaviors that must be verified in the Context Switch is very large. It is thus difficult to carry out the verification by simulation. FormalCheck implicitly enumerates all the possibilities which allowed us to detect two difficult design errors using Property B. These errors were found after all the planned simulations of the entire FMD RTL design had been executed. The errors were related to different issues in the implementation, but both exhibited the same behavior: when the Microprocessor reads the Context Memory, it could receive data from a wrong address.

More specifically, the counter-example[1] generated by Formal Check that leads to the first design error confirmed that the particular input sequence would have been difficult to foresee in a simulation scenario: When the Microprocessor requests to write to a read-only address or to access an invalid address, it is acknowledged which allows it to initiate a new request. At the same time, however, the Microprocessor-initiated

1. It is an input sequence computed by the model checker that shows what inputs must be applied to reach a point where the property is violated, i.e., leading to the design error.

memory read signal may incorrectly be activated in this case. Therefore, if there is a valid Microprocessor read to the Context Memory immediately following the first access, it would receive the (default) data obtained as a result of the preceding incorrect read.

The sequence leading to the second error was even more complex and difficult to stimulate. It now involved an interaction between the Datapath and the Microprocessor accesses to the Context Memory and the internal registers of the Context Switch. Again, incorrect data was returned to the Microprocessor, but this time it was due to an error in the complicated control circuitry that handles updating of the virtual pipeline registers: The registers for addresses and data as related to the Datapath requests were not updated in the same clock cycle. A Microprocessor access to these registers that would occur between the two parts of the update could read an incoherent pair of address-data values.

A question is often asked when a design error is detected using a new kind of a verification tool. How hard would it be to detect this error using simulation? We could measure the difficulty of detection by estimating the chance of hitting the "right" sequence in a simulation testbench. This is impossible with directed testbenches that depend on the ingenuity of the designers, but, as mentioned in Section 1, random simulation may be used to verify designs without having to devise specific input sequences. Therefore, we propose to measure the detection difficulty by estimating the length of the input sequence needed to stimulate the design error with a given probability. This is similar to the estimation of random test sequences for stuck-at-0/1 faults, except that we do not have a simple fault model here.

In the Appendix we outline a method that allows us to make such estimation in some cases, based on the counter-example generated by the model checker. Using that method, we have calculated that to detect the first error with the probability of 99.99%, we would need some 24199 input patterns. This does not look like a formidable length of a sequence to simulate, however, it requires setting up the random testbench and formulating the property checkers in the form of finite-state machines implemented as Verilog modules. Last and not least, the random simulation could only detect this error with a certain probability, but not with absolute certainty, while the model checker guarantees that the design does indeed satisfy the property, for any input sequence.

The second error is even more complex and harder to detect by simulation, as it requires a particular coordination between the Datapath and the Microprocessor accesses. Unfortunately, here the counter-example sequence is not unique and we could not compute the estimation (we discuss possible ways around this problem in the appendix). Still, due to the more complex coordination of the events that lead to the error, it is likely that the length of a random input sequence would be much longer than in the first case, and, in addition the test bench would have to reliably predict the correct results at this level in the design environment. This would be difficult based on a more abstract behavioral model of the whole chip as the reference for comparison. A model checker reduces this investment in test-bench development at the block level.

5 Model reductions

Due to the state explosion problem, FormalCheck could not complete the verification of properties that refer to channel numbers and compare the resulting values (like Property A) even when the blocks related to the Context Switch were isolated from the chip design, the number of channels was scaled down from 28 to 8, and the automatic iterated reduction algorithm was used. We found, however, that effective manual reductions[1] could be carried out on certain state variables, after which the verification of the properties took only about 30 minutes. Naturally, in the case of a violation of a stated property, like the one which identified the design errors mentioned above, we had to make certain that it was not a false negative answer to the query. We achieved that by examining the counter-example and the Verilog code, and then confirming the error by a Verilog simulation of the counter-example sequence. In both cases where we obtained a negative answer, the simulation confirmed the presence of a design error, i.e., no false negative.

The type of reductions that were not detected by the automatic reduction algorithm and which we manually applied were similar in many of the designs that we verified. The structure of the designs is quite common in dataprocessing circuits, in which the appropriate processing context (set of registers, memory data, etc.) of an arriving datum is selected based on a data descriptor, processing is carried out, and the datum is sent out, while possibly updating the context information.

The situation can be summarized by the structure shown in Figure 2 that corresponds to the following Verilog code extracted from the Context Switch model:

```
module littleFifoDesign(fifo_num, fifo_rd, rst, clock, fifocsmr_old);

  input fifo_num; // select context by FIFO number entered on this input; this is
     the data descriptor (0 or 1)
  input fifo_rd; // increment data if 1; represents the data processing operation
  input rst; // reset
  input clock;
  output fifocsmr_old; // output FIFO index corresponding to the fifo number

  reg [6:0] fifocsmr_R0;    // 7-bit FIFO consumer index 0
  reg [6:0] fifocsmr_R1; // 7-bit FIFO consumer index 1
  wire [6:0] fifocsmr_new, fifocsmr_old; // interconnections

 always @(posedge clock)
 begin
```

1. The manual reduction consists of replacing the state variable by a free (unconstrained) primary input that in effect converts the next-state transition function into a completely non-deterministic one, meaning that the next-state value can be any. The reachable state-space remains the same or is even larger, however its symbolic representation and those of the next-state functions become trivial, thus reducing the amount of memory needed by the model checker.

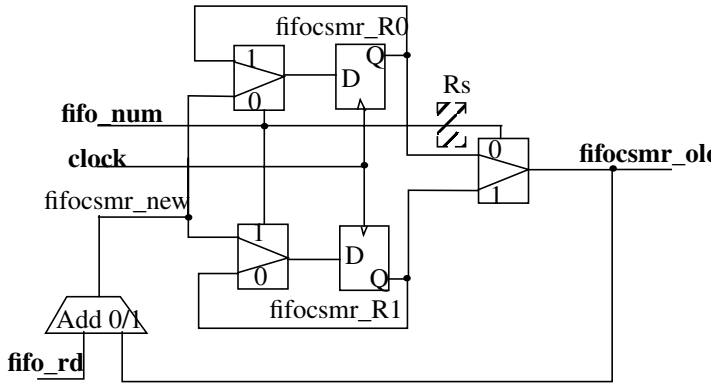

Fig. 2. Example of FIFO circuit structure for manual reduction

```
   if (rst == 1'b0)
    begin
    {fifocsmr_R0,  fifocsmr_R1} <= {2{7'b0000000}};
    end
    else
    begin
     case (fifo_num)
         1'b0: fifocsmr_R0 <= fifocsmr_new;
         1'b1: fifocsmr_R1 <= fifocsmr_new;
     endcase
   end
  end
assign  fifocsmr_old   =  fifo_num == 1'b0 ?
                  fifocsmr_R0 : fifocsmr_R1;
assign fifocsmr_new = fifocsmr_old + {6'b000000, fifo_rd};
endmodule
```

The property to be checked verifies that the FIFO consumer indexes are updated correctly:

```
Property:fifo_index_Without_Rs
Always:(was_fifo_num == 1 && was_fifo_rd == 1 &&
        was_fifocsmr_R1 <= 126) => (fifocsmr_R1 == was_fifocsmr_R1 + 1)
```

The signals was_fifo_num, was_fifo_rd and was_fifocsmr_R1 are the signals fifo_num, fifo_rd and fifocsmr_R1 delayed by one clock cycle using environment state machines defined either in the Verilog code or through the FormalCheck user interface as follows:

```
if (clock == rising && rst != 0)   was_fifo_num = fifo_num;
```

```
if (clock == rising && rst != 0)      was_fifo_rd = fifo_rd;
if (clock == rising && rst != 0)      was_fifocsmr_R1 = fifocsmr_R1;
```

To update the fifocsmr_R1 correctly, the contents of the other FIFO consumer index register (fifocsmr_R0) should not matter, that is, the state variable(s) related to fifocsmr_R0 can be replaced by free primary inputs. Yet, because this reduction depends on the way the property is formulated and on the circuit structure including the additional state variables for remembering the past values, it appears that every register value may depend on all the other ones and this possible model reduction does not seem to be detected by the automatic reduction algorithm. A manual reduction was carried out with the help of the FormalCheck reduction manager and user interface on 49 state variable bits - the 7 channels FIFO indexes (of 7 bits each) not related to the selection to be verified.

Note that to decrease the size of the model without the above reductions, we could have constrained the primary inputs such that fifo_num = 1'b1. However, this would have eliminated the presence of the other selection(s) from the input stream, thus potentially hiding a design error that manifests itself only in the presence of different consecutive operations. For example, consider the case where a register **Rs** (as shown shaded in Figure 2) is added by mistake in the design. If we constrain fifo_num = 1'b1 and initialize Rs to 1 during reset, then the property would still pass without detecting the error. However, it is detected under the manual reduction of R0.

Finally, the property stated above can also be reformulated without the use of the explicit state variables was_fifo_num, was_fifo_rd and was_fifocsmr_R1 to memorize the preceding values as follows:

```
After:    ( fifo_num == 1 &&
            fifo_rd == 1 &&
            fifocsmr_R1 == 0 &&
            rst == 1 &&
            clock == rising )
Always: ( fifocsmr_R1 == 1 )
Unless after:( clock == rising)
```

and enumerated over all values (0 .. 127) of fifocsmr_R1.

Using this formulation which is much longer to manipulate due to the large number of instances of the property (enumeration) the automatic reduction algorithm finds the proper reductions automatically. This seems to indicate that the presence of the additional state variables can introduce false dependencies between state variables as far as the reduction algorithm is concerned, and thus disallows it from abstracting fifocsmr_R0. Since the use of the state variables makes the formulation of properties much easier in many cases, we are investigating how to improve reduction algorithms to take situations like this into account. In the meantime we complement the automatic reductions by manual ones, based on our knowledge of the design, as indicated earlier.

6 Conclusions

We described the verification of a critical portion of a large ASIC design using a

commercial model checker. Model checking was applied to a subsystem that is difficult to verify by simulation. It allowed us to detect two very hard design errors that were not detected by the simulation suites. To complete the model checking in the available workstation memory, we had to apply manual model reductions. In the process we identified circuit structures that lend themselves to manual reductions for certain specific properties. Once such reductions were carried out, all our queries were efficiently verified in 512Mb of memory.

To illustrate how difficult the detected errors are, we computed estimates on the length of a random input sequence needed to detect the error with a certain probability. The computation requires constructing a recognizer of a detection sequence which can be in some cases obtained from the counter-example generated by the model checker.

One of the difficult tasks of carrying out the formal verification was to isolate the RTL code related to the subsystem of interest and to construct an appropriate model of the environment, that is, to identify the minimal input constraints. With what we have learned about model reductions, we hope that it might be actually possible, with some improvements in the model-checking technology and the design/verification methodology, to verify important queries on the entire chip model. This would lead to higher verification quality and better understanding of the division of coverage of the chip-level test plan features by simulation and by model checking.

References

1. A. Silburt Invited Lecture: ASIC/System Hardware Verification at Nortel: A View from the Trenches. *Proceeding of the 9th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, October,1997, Montreal Canada.
2. A.J. Camilleri. A role for Theorem Proving in Multi-processor Design. Proceeding of *the 10th International Conference on Computer Aided Verification (CAV'98)*,Vancouver, BC Canada, June/July 1998.
3. D.L.Dill. What's Between Simulation and Formal Verification. Proceeding of *the 35th Design Automation Conference (DAC'98)*, San Francisco, CA, USA, June 1998
4. FormalCheck User's Guide. Bell labs Design Automation, Lucent Technologies, V1.1, 1997
5. K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. Ph.D. thesis, SCS, Carnegie Mellon University, 1992. (See also Cadence Design Systems, Inc. www pages.)
6. 0-In Design Automation, Inc., "Whitebox verification," http://www.0-in.com/tools.
7. 0-In Design Automation, Inc., "0-In search," http://www.0-in.com/tools.
8. System Science, Inc. (Synopsys, Inc.), "The VERA verification system," http://www.systems.com/products/vera.
9. TransEDA, Inc., "VeriSure- Verilog code coverage tool," http://www.transeda.com/products.

10. Lucent Technologies,"FormalCheck model checker," http:/www.bell-labs.com/org/blda/product.formal.html.
11. P. Bratley, B.L. Fox, L.E. Schrage, *A Guide to Simulation*, 2nd Edition, Springer-Verlag, New York, 1986.
12. L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.
13. CheckOff User Guide, Siemens Nixdorf Informations Systemen AG & Abstract Hardware Limited, January, 1996.
14. Lambda user's manual, Abstract Hardware limited, 1995.
15. Verisity Design, Inc. Specman Data Sheet, http://www.verisity.com.

A. APPENDIX: Estimating the length of a random input sequence

We shall show how one can quantify the hardness of a design error by estimating the length of a random input sequence that would detect the error. The problem could be stated as follows:

Suppose that a synchronous RTL design has input ports on which one of $n \geq 1$ symbols can occur in each clock cycle, independently of the choice made in the preceding cycles. Assume also that the probability distribution function of the symbols on the input in a clock cycle is uniform, i.e., a symbol is chosen with probability $p = 1/n$. Let a design error E be detected by Formal Check such that the only input sequence that detects it consists of m consecutive symbols on the inputs, regardless what preceded this Detection Sequence (DS). The question is: given that the error is to be detected with probability P_d, what is the minimal length L of a random simulation sequence (independent choice at each clock cycles, uniform choice from n symbols) such that the detection sequence appears on the input at least once, i.e, that the error is detected.

A.1 An analytical solution

The proposed method can actually be used in a more complex setting than the one stated in the problem definition above, but we shall illustrate it on the simplified case. Construct a finite state automaton that recognizes the first occurrence of the DS in a sequence. For the problem above, such a recognizer is shown in Figure 3. The symbol U represents the set of the n symbols $|U| = n$, and A_i is the subset of symbols that can occur (one of them) in position i in the DS. Starting in state *0*, the automaton will reach state m if it detects DS. If now we replace the symbols labeling the transitions by the probabilities of occurrence of these symbols, we shall obtain a discrete time Markov chain, with initial state *0*. We can then compute the probability of reaching state m in at most $L > n$ transitions. Let p_i be the probability that one of the symbols from A_i occurs. The Markov chain corresponding to the automaton in Figure 3 is shown in Figure 4.
The probability state transition matrix of this chain is $\boldsymbol{T} = [p_{ij}]$, where p_{ij} is the probability of reaching state i from state j in one step (transition). For example, entry *(0,0)* is $p_{00} = 1\text{-}p_1$. The sum of the elements in any one column must be equal to one, since there is a transition from any state for all the input symbols. Let $S_0 = [1, 0, ..., 0]^T$, $|\boldsymbol{S_0}| = m+1$, be the probability of being in one of the states at time *0*, i.e., in this case, we start from state *0*, hence its entry in S_0 is *1*, and the other m entries are *0*. The

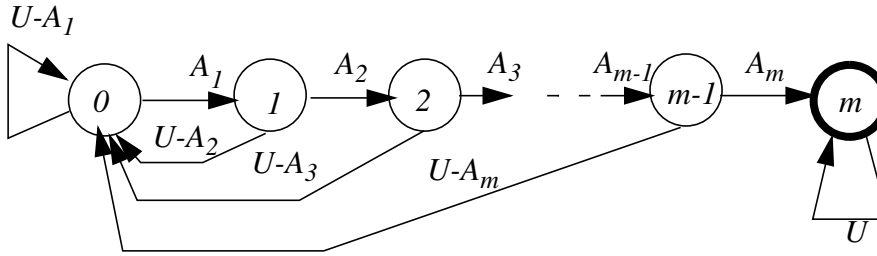

Fig. 3. A simple recognizer automaton, initial state is 0, state m is acceptin

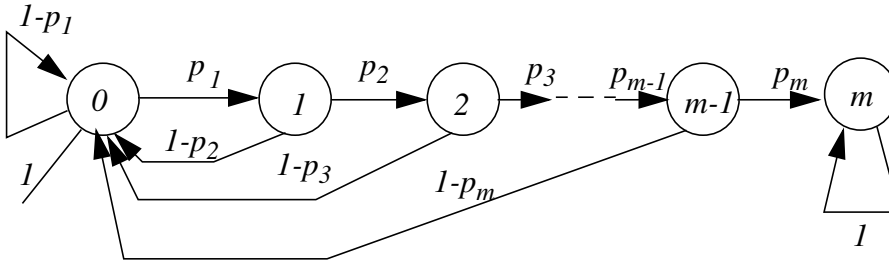

Fig. 4. Discrete time Markov chain for automaton in Figure 3

vector of probabilities of being in one of the $m+1$ states after L transitions is given by $\boldsymbol{P}_L = \boldsymbol{T}^L \boldsymbol{S_0}$. However, the accepting state m (the one in which the detection sequence has been observed somewhere in past) has a self-loop transition, i.e., once it is reached in $k < L$ transitions it will stay there; therefore, if we take the last element (m) of $\boldsymbol{P}_\mathrm{L}$ then this element contains the probability of having reached state m in up to L transitions. That is, the probability of detecting the error in a random sequence of length L is given by

$$\boldsymbol{P}_L(m) = (\boldsymbol{T}^L \boldsymbol{S_0})_m \, .$$

For a regular structure (e.g., when $p_i = p_j$), there may be a closed form solution for $\boldsymbol{P}_L(m)$, but in general, given a P_d, we can compute $\boldsymbol{T}^k \boldsymbol{S_0}$ numerically until $\boldsymbol{P}_L(m) \geq$

P_d. The number of iterations k then is the minimum length L of a sequence to produce DS on the input of the design with probability P_d.

In the case of the first design error reported under Property B in Section 4.2, it was determined that the 4-cycle sequence reported in the counter-example generated by FormalCheck is the only DS that can provoke the error, and that it can be preceded by any other input. It thus fitted the above simple form of a recognizer. The computed lengths for two detection probabilities are as follows:

Detection Probability	Nb. of clock cycles
0.999	18492
0.9999	24199

This assumes that the vectors were applied at the inputs of the blocks; longer sequences would be needed if applied from the primary inputs of the chip as discussed in Section 4.2.

In more complex situations where there are potentially many different detection sequences, the construction of the appropriate Markov chain may be difficult, since only one such sequence is reported in the counter-example. However, FormalCheck being based on language inclusion test of ω-automata, computes implicitly the complete DS recognizer, hence, at least in theory, one could compute the estimate on the length of the sequence. Unfortunately, for most practical cases this may not be of much use, because the size of the corresponding explicit Markov chain would be overwhelming.

A.2 Estimation of *L* by random simulation

Another way of estimating L is to actually perform random simulation experiments with different (uncorrelated) starting seeds for the random number generator used in producing the inputs. There are two problems with this approach:

- We wanted to avoid random simulation runs in the first place.
- The simulation runs can be quite long.

These problems could potentially be alleviated by using special techniques to bias the input sequences (knowing the error) to produce good estimators even with short random simulation sequences [11, 12]. If, however, the sequence recognizer is complex (i.e., not just a single detection sequence) then determining the appropriate bias formula may be difficult.

Fortunately, the need for such measures of error "hardness" may only be required until formal methods find their firm place in the normal design flow. This will happen when we can easily identify features in the test plan that can be verified most efficiently using formal methods, and when designers become more aware of such novel verification techniques.