

Increase Competitiveness by using Formal Verification Tool in design flow of wireless data system:

Author: Serge Lasserre Advanced System Architecture group
Luis Lopez-fernandez ISIEE

Abstract:

System On Chip (SOC) solution brings new challenges in integration and test. Multi-core approaches to solve MIPS, power requirement in new applications, like wireless multimedia, digital audio and video, requires new verification methodologies to validate traffic controllers which become the back bone of such systems.

Traditional test approaches just become inappropriate (huge numbers of test vectors), and too time-consuming with respect to design cycle time. Formal verification is a new approach, long stayed in the university lab, which can help design productivity and quality, and become key to the first-pass success of these designs.

This paper will illustrate this through the real example developed during the TI_ARM915 project for which FormalCheck™, a model checker from Lucent technology, has been successfully used to prove one of the most critical sub-module of the traffic controller.

We will discuss how FormalCheck™ has improved the design cycle time of the LINDA DMA controller and how it has also bridged the gap between specification and design. Through examples we explain the methodology which was developed to model the module via properties and constraints (queries).

We will conclude by looking at the current limitations of this method and what can reasonably be expected from these tools today and in the close future.

Introduction

DSPs and more generally system on chip (SOC) solutions bring new challenges in design integration and test. Applications such as Settop Box, Personal Digital Assistant or smart phone, only to name a few, always require higher integration and parallelism to meet MIPs and features requirements. Increasing the clock frequency is just not enough.

Today these systems already combine DSPs, micro-controller and specific hardware accelerators on the same silicon, all sharing common memory spaces to communicate. Complex traffic controllers are at the heart of such systems. Although they do not represent a high percentage of the design area they become the backbone of such systems and are also extremely difficult to verify with conventional test approaches. The number of test vectors to guarantee a full functional coverage of such modules becomes simply enormous and is not manageable within tight development schedules. In fact, despite the increase of complexity, development time for those projects stays stable (1-2 years) which means that many bugs are likely to be found after first silicon if new test methodologies are not put in place.

For TI_ARM915 project, we decided to take a different approach to test a SDRAM controller, one of the most critical modules of the traffic controller. It was clear from the beginning that this module could not be correctly tested by conventional test approaches. It would have required approximately 15×10^{10} test vectors to achieve full functional coverage. Instead we used FormalCheck™, a model checker tool from Lucent Technology to validate the module with formal verification.

Although Formalcheck™ has already been used within TI to verify specific properties of modules in both the C6x and the AV7110 projects where standard test approach failed, It has never been used as a validation tool from day one. We will explain how this was done on the SDRAM controller module and the methodology we used.

Design methodology overview

Formal verification tools give a unique opportunity to follow a clean design flow. Usually design errors come from insufficient specification or specification errors, misunderstandings of specifications, insufficient test coverage. Formal verification can help in all aspects. It can reduce misunderstanding if seen as a golden test given to the designer. It can provide a comprehensive test environment, which can be used as a real test bench all along the design effort reducing the design effort considerably (everybody knows that 3/4 of the time is

spent in validation). It can also reduce specification errors in some aspect by forcing people to be rigorous during the specification phase.

In the TI_ARM915 project, the two first points were really put in practice as one person independent from the design team created the formal verification environment (set of queries) from the design specification. The work was done in parallel and, in fact, completed earlier than the HDL code itself. Once one is used to the tool, describing the properties and constraints takes less time than writing the RTL HDL code.

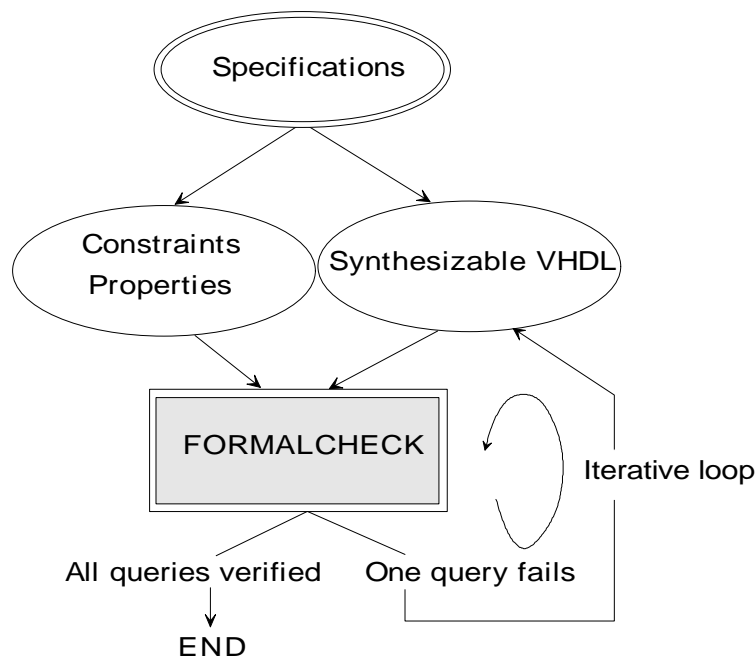


Figure 1: Design flow

As soon as the design got sufficiently debugged (basic problems related to HDL coding), we tested the functionality with FormalCheck™ and quickly found bugs (12 in total). We worked (figure 1: iterative loop) in an iterative manner using FormalCheck™ as a normal test bench. Most of them were corner case errors, which would have taken a very long time to highlight. It enables us to find design flaw very early and correct them in an iterative manner. Because they would have required a sophisticated test bench, these bugs would have been found late in the integration phase or even worse during the life of the components.

The set of queries was constructed by considering the target module like a black box (abstraction of all aspects related to design implementation). This approach has the disadvantage of generating more queries than probably needed. Because some proprieties are true by construction (HDL code) and would not

required a specific query. On the other hand, this approach keeps design and verification completely separate maximizing the chance to generate the best coverage.

Background on FormalCheck™

Before illustrating by concrete example the work done, it is important to give a brief overview on the tool. FormalCheck™ is a model checker. In simulation, designers write a test bench with test vectors to demonstrate that their module behaves correctly (meet set of properties). In model checking (MC), designers write assertions (or properties) to check by mathematical proof that there are no vectors producing bugs. Model checkers determine that properties pass or fail under all possible conditions and over any sequences. After reduction and analysis of the full state space, model checkers give a counter example if the property fails.

The reader who is non familiar with temporal logic can be surprised by the way of expressing properties in FormalCheck with the qualifiers *ALWAYS*, *NEVER* and *EVENTUALLY*. In fact these operators are introduced by the mathematical paradigm of temporal logic.

Properties fall into two categories: safety and liveness. In the same way that we use NOT or AND operators in conventional logic, we have a set of temporal operators *ALWAYS*, *NEVER* and *EVENTUALLY* in temporal logic [2]. Safety properties are expressed in two formats *ALWAYS* or *NEVER*. For Instance, a simple example of property in our SDRAM controller would be that two consecutive activate commands (ACTV) on a same bank never occur within less than 80 Ns.

Liveness properties describe an event that *EVENTUALLY* happens. For example in our system each request is eventually granted. A liveness property may not be verified within a finite simulation trace.

Properties may have an enabling condition and a discharging condition. That means that we can activate or deactivate a property when a particular trigger event occurs. In this way the structure of a property can be expressed as:

after <enabling condition>
[always, never, eventually] <full filling condition>
unless <discharging condition>

When dealing with events that take place in a sequence of time FormalCheck™ provides the possibility of using state variables (to simulate timers). A variable

representing a counter will allow one to verify that 80 Ns has elapsed since the last activate command.

Variables, however, are part of the state space and increase the verification time and complexity.

FormalCheck™ explores all possible scenarios when checking a particular behavior. In this way, for example, in a synchronous circuit FormalCheck will explore in a synchronous design the case in which the clock signal is always 0, making all properties fail. The designer must indicate the input space he is interested in by adding constraints on the inputs. The general structure of a constraint is the same than the one of a property. There is a set of predefined constraints that can simplify the work of the designer (Clock, Reset, Constant, etc.). A set of properties and a set of constraints form a query. [3]

One can easily understand from this short overview, that formal verification like others means of verifications relies on the correct set of queries that fully characterize a module. If input signals are too heavily constrained or if some queries are missing design errors might not be detected. But the set of queries is more manageable than the set of test vectors.

Overview on the SDRAM controller

The SDRAM controller is part of a traffic controller [4] scheduling accesses from multiple users such DSP, micro-controller graphic controller or even I/O peripheral through DMAs. Sharing memory also implies loosing time in memory access overhead and arbitration. To minimize this lost of performance a SDRAM controller has been designed to support pipelined requests to benefit from local and spatial properties of our system.

The SDRAM controller (figure 2) supports 4 types of request. These requests can be pipelined to reduce or suppress gaps between requests as often as possible. These 4 types of requests (Req = 0,1,2,4,8) combined with the size (3 bits) and the direction R/W are:

- Burst request (read or write) of n words (n = 1 to 8)
- Sequential burst request (read or write) of n words (n = 1 to 8)
This type of burst is used to concatenate several n words burst together and create a longer burst (>8) without additional overhead. For this special type of burst the start address correspond to the address already present in the SDRAM controller (last data of previous burst + 1)
- Auto refresh request: generate a single auto refresh command.
- MRS request to program the SDRAM.

In addition to these access request signals, the SDRAM controller is controlled by configuration signals such as 16M/64M, 2B/4B, freq., etc.... These

configuration signals allow the SDRAM controller to manage 16Mbits SDRAM and 64M bits. The latter can be organized in either 2 or 4 banks. The SDRAM access time is optimized for several frequencies (13 MHz, 26 MHz, 52 MHz and 104 MHz) to meet the minimum latency required by these memories (12 possible configurations in total).

The SDRAM controller also supports systems that require bursts, crossing page or bank boundary either inside a single burst or across multiple sequential bursts.

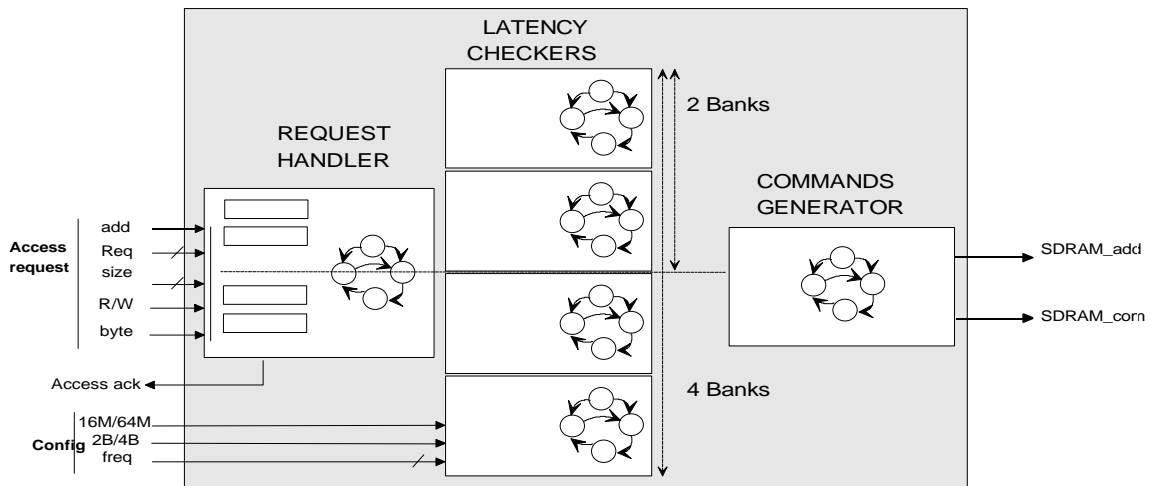


Figure 2: Overview of SDRAM controller achitecture

From all these signals the SDRAM controller generates the expected command signal (SDRAM_com: RAS, CAS, W, DQML, DQMU, CKE) with its associated column or row address and returns an acknowledgement (with Data after a Read command). The SDRAM controller has approximately 2K gates and is made of several synchronized state machines. It contains 113 registers, half of which are used to store addresses.

How FormalCheck™ was used to test this module

In order to define a set of properties and constraints to test the circuit, we are going to see the SDRAM controller as a protocol translator (figure 3). The SDRAM controller does not have the initiative for generating commands, its only function is to take a request coming from a traffic controller and to send to the SDRAM a set of commands corresponding to this request. So it can be seen as an interpreter between two units that speak different languages.

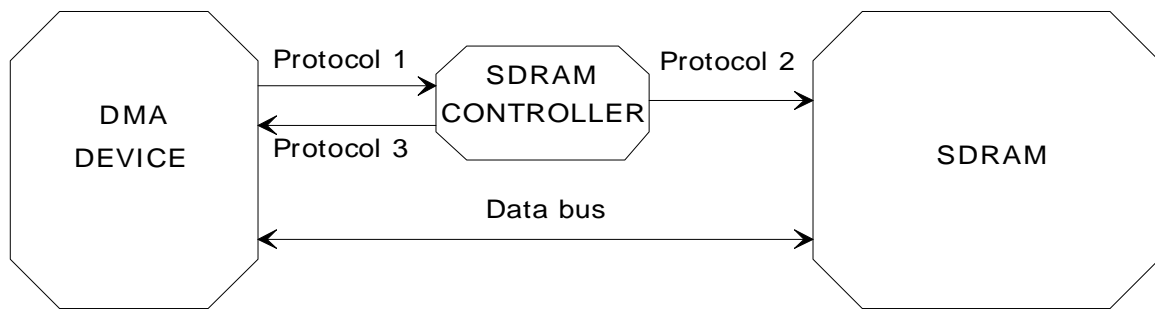


Figure 3: SDRAM controller viewed as a communication protocol

The SDRAM controller interface has been in three groups of signals that we called arbitrary protocol 1, protocol 2 and protocol3:

- 1- Protocol 1 is used to send the request from the Traffic controller (TC) to the SDRAM controller
- 2- Protocol 2 is used to send the commands from the controller to the SDRAM.
- 3- Protocol 3 is used to give back the information to the TC.

These protocols are going to be defined through a set of constraints and properties.

Protocol 1 is made only of input signals. Its correctness depends only on constraints that we choose. This is critical for the quality of the verification. If the tool is over-constrained, the algorithm will reduce the state space too much hiding some potential errors. On the contrary, if insufficient constraints are put, the state space will remain too large (state explosion) making verification impossible or the verification will fail unnecessary.

The correctness of protocol 2 is checked through a set of properties that we can group in two types. The first verifies the “Syntax Correctness”, the second verifies the “Semantic Correctness”.

The correctness of Protocol 3 consists of checking that a given request receives the correct answer.

Many of these statement might look trivial to the reader but many time their translation in term of properties are not; as we are going to see in some of the following examples.

Some examples of properties and constraints

First of all, some general constraints like clock and reset must be given. If no constraint is made on the clock, for instance FormalCheck™ is going to keep the clock always low as one possible scenario and the verification will fail.

Clock constraint : clk_constraint
Signal dma_clk_l

<i>Start: Low</i>
<i>1st duration: 1 crank</i>
<i>2nd duration: 1 crank</i>

Another example of constraint on input signals is given for the requests themselves. Signals qualifying a request stay stable until the request is granted (access ack). All signals associated to a request are going to be constrained in the same way and we will show here only one of them. The request can take only the following value 0,1,2,4,8 which indicate respectively no request, burst access, sequential burst access, auto refresh and MRS. The request value can change only if the value is 0 or if the acknowledge is received.

```

(Macro)      Clk_is_rising == dma_clk_l = rising
(Macro)      Request_can_change == req_l = 0 or access_ack = 1 and clk_is_rising

(state_variable) request_value range 0 to 8
              initial value := 4
              if request_can_change then value :=0,1,2,4,8;
assignment constraint : stable_correct_input_request
              signal : req_l
              value : request_value

```

Similarly other rules deduced from the specification will enable one to further constrain protocol 1 allowing a reduction of the state space.

One other important domain of constraint is the configuration signals. A full verification is going to be done for one of the 12 possible configurations listed previously. Then, for the other, only queries, which differ will be run saving a bit of time. The chosen configuration was 104 MHz, 16 M bits memory with 2 banks.

Now that we have seen how to constraint the input signals (protocol 1), let's focus on protocol 2 and the notion of syntax and semantic correctness. The notion of syntax covers all aspect related to the SDRAM commands. A command must exist(all combinations of output signals are not allowed in SDRAM), happen in a given order and respect latency with respect to other commands.

For example two activate commands on the same bank must not occur within less than 80 Ns (9 cycles at 104MHz).

```

(Macro)      bank0_2_16_selected == add(11) = 0
(Macro)      command_is_actv == last_CKE = 1 AND RAS = 0 AND CAS = 1 AND
              WE = 1
(state_variable) last_CKE range 0 to 1
              initial value := 0
              if clk_is_rising then
                last_CKE := CKE;
              endif;
(state_variable) timer_ac_ac_2_16_b0 range 0 to 9
              initial value := 9
              if command_is_actv AND bank0_2_16_selected AND Clk_is_rising then
                timer_ac_ac_2_16_b0 := 1;
              elsif timer_ac_ac_2_16_b0 = 9 then
                timer_ac_ac_2_16_b0 := 9;
              elsif Clk_is_rising then
                timer_ac_ac_2_16_b0 := timer_ac_ac_2_16_b0 + 1;
              endif;
(property) ac_to_ac_b0
Never timer_ac_ac_2_16_b0 < 9 AND command_is_actv AND bank0_2_16_selected

```

This property, with its associated constraints, checks that no ACTV command is applied on bank 0 while timer of bank 0 is less than 9. All latencies are similarly checked [1].

This other example below illustrates how FormalCheck™ verifies that only commands, which are defined in the memory specification, are generated by the SDRAM controller.

```
(Macro)      Command_is_actv == ....
             Command_is_deac == ....
             ...
(property) same_vocabulary
Always Command_is_actv OR Command_is_deac OR Command_is_read OR...
```

The notion of semantic correctness consists of verifying that a request is correctly translated into a command. In other words, that a read request generate a READ command and not a write command. That's the SDRAM controller read only two data for a burst of two or that the address of the request correspond effectively to the address send to the SDRAM. This can be summarized by the three questions: what to do?
where?
how much time?

Although this can be stated very simply (see below) one must understand that several state variables must be defined to emulate requested_address [1] as in previous examples.

```
(property) where_to_access
Always Requested_address = SDRAM_add
```

Finally, we must check protocol 3. The verification of protocol 3 consists of checking that the correct answer is always sent back. For instance, a request must always be acknowledged for instance. No valid data must be put on the bus without having been requested. The correct address must be saved (last data + 1)

```
(property) all_req_are_granted
after ((req != 0) AND (access_ack = 0)) eventually (access_ack = 1)
```

A complete description of the set of queries required to test the SDRAM controller is available in a detailed report [1].

Limitations and compromises using FormalCheck™

Once the set of queries has been created, we launched what seemed to be an easy query "ac_to_ac_b0". At the beginning of the verification, FormalCheck™ gave the following message:

Design Data:
2.61e3 Combinational variables
99 State variables: 1.81e39 reachable states
4.29e9 inputs/state

After 17 hours, a total of 950MB of memory were used and the verification was not yet finished. The number of possible reachable states was too big for the calculating power and the available amount of memory on our machine (ultra-2 sparc with 1GB of memory).

This was by far above the maximum figures recommended for the usage of this tool ($10e^{15}$ reachable state) for circuit of this kind.

We found out that our circuit with its 113 registers could not be verified as it stands. We had to find ways to reduce the complexity without removing or modifying anything in the functionality. 31 of these registers are used by state machines and control, while 82 registers are used to store addresses. We could not constrain addresses too much without removing some important reachable states (page crossing function).

First, we decided to change the memory size from a 16Mbit SDRAM to a 16Kbit SDRAM (2 banks of 2 pages) reducing the number of registers dedicated for the address to only 27 (figure 4). This did not change at all the functionality that we wanted to prove.

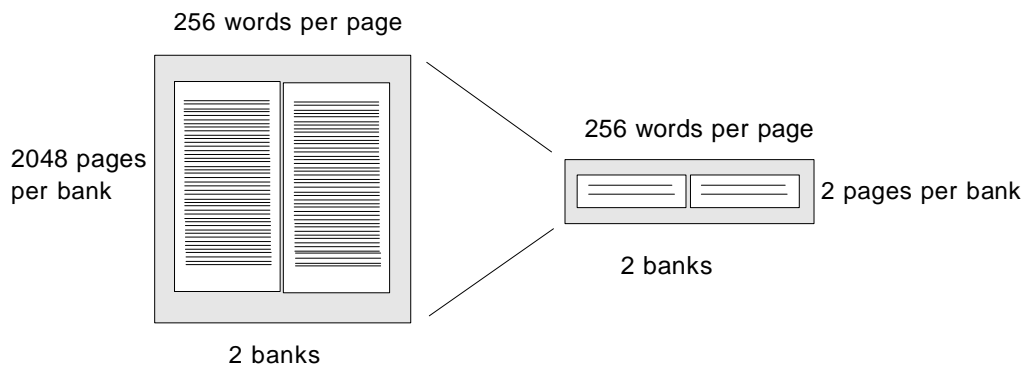


Figure 4: Complexity reduction

This modification in the HDL code was done with a script PERL to make it repeatable very easily as long as the code was being corrected. In fact, looking back at the design methodology generics should have been used in the HDL code to avoid any code modification and facilitate formal Verification. The problem is that FormalCheck™ does not support generic in the top level of the HDL code so generics must be hidden below.

We also found out that the reduction algorithm did not work well when the SDRAM controller was configured in 2 banks. Registers memorizing the address or registers related to the management of the 3rd and 4th banks were not removed when the SDRAM controller was configured in 2 bank mode. Some part of the logic related to the 3rd and 4th banks were not gated by the input signal 2B/4B mainly to improve speed performance. While writing HDL code, designers must be particularly careful of this to facilitate the work of reduction algorithms.

To reduce even further the state space and close the gap with the recommended figure ($10e^{15}$), additional constraints on the address range were looked at. In fact the full range (12 bits) was not required to validate the functionality of the control. Most of the address in the middle of a page can be withdrawn from the verification because they do not cause any specific behavior of the controller. Bits 8,7,6 and 5 were constrained to the constant value 1.

By adding this constraint, the correct connection of all address bits is not checked by the formal verification. But this can be quickly checked by a very limited number of standard test vectors.

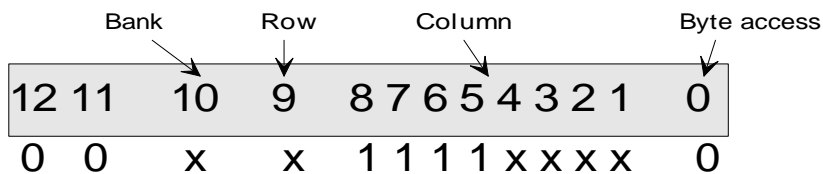


Figure 5: Constraints on address range

With this approach, we were able to verify successfully the SDRAM controller in 16M – 2 banks configuration at different frequencies. All queries passed even the “liveness” such as All_req_are_granted.

Full validation of this module has required 33 queries per configuration. It took approximately 70 hours to run this set of queries (33) on a work-station running at 248MHz (SPARC Ultra-entreprise or SPARC ultra-2). Not all configurations were verified because it would have taken a significant CPU time. Nevertheless, this could be seen like an acceptable checking-time for the final verification as these can be run in batch mode in parallel on several machines. Besides, once a full set has passed successfully in one configuration, some queries, which are not dependant on frequency, do not need to be checked again.

As one can understand, properties and constraints are not always straightforward to define. If a property has not been correctly defined, the corresponding query fails and the designer gets a waveform with the counter example. The probability of incorrectly defining a property, which is verified, is extremely low, except for an error in the specification itself. An error in the

constraints is much more likely to remain unnoticed. If the error produces an under-constraint, the verification will most likely fail because the verification will drive the circuit into a state, which should not be reachable. On the contrary, if one produces an over-constraint, queries can be verified without having explored all the reachable states.

In order for the designer to see if the constraints have been correctly described, one should introduce a wrong property into a query to make it fail. The counter-example given by the tool allows the designer to see if the constraints work correctly.

Conclusion

Formal verification is based on mathematical proof. Nevertheless, One must remember that this is not a guarantee for the designer to generate a module free of errors. The quality of the verification still relies on the completeness of the properties and on the ability to translate the specification in a set of queries. The verification can only be as good than the set of queries!

Having said that, we have demonstrated with this complex module, that it is possible to fully validate a critical part of the design that would have been impossible to check with conventional approach.

Despite the fact that all the configurations have not been tested (they could have been), we ended up with a sufficient level of confidence. So far, no bug has yet been found in this module.

FormalCheck™ enabled to catch errors earlier in the design, reduce overall system test considerably and improve development cycle time. By reducing the test time, it has also enabled the designer to focus on real design issues such as speed, power consumption, etc...

Our work also confirmed that with the current technology, the tool quickly reaches saturation if the module contains more than around 60 registers (this includes all additional states due to state variables!). Sixty is an approximate number above which computation time is hard to predict (state explosion). This is not a "hard" rules and highly dependent on the logic itself.

As we have seen, tricks must be put in place to reduce this number of registers and avoid the state explosion. This, of course, is not really compatible with formal verification as the modified circuit differs from the final one.

Model Checkers have been available in the industry only very recently. It is a domain, which is changing rapidly following the market pressure and the semiconductor industry demand. Progress must be expected specially concerning the interface of such tools (beta release). Progress in algorithms can also be expected enabling bigger module to be verified but this might not come so soon.

Therefore, formal verification tools will have to be used to validate critical sub-module functions more than the full circuit yet for some time. But this is already a breakthrough in the test approach.

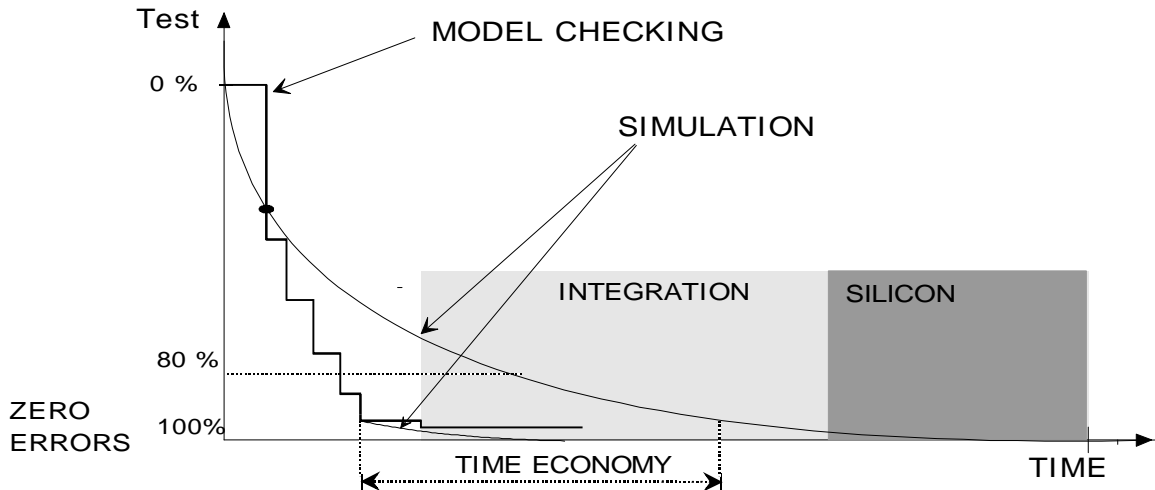


Figure 6: Model Checking versus Simulation

As the above figure shows, the zero-error remains difficult to reach in all cases. Some properties might take too much time or memory to be verified. The important fact is that with formal verification tools we get to level of verification that we could only reach much later (or even not) during the design with conventional approaches. Mixed validation with formal verification and simulation is the probably the appropriate answer for the time being. In this first steps it is more interesting to use simulation than model checking to quickly screen out the basic errors. When the first errors have been detected and corrected simulation becomes less efficient. With model checkers we will spend more time in the beginning for defining properties and constraints, but once it has been done finding errors is as easy as pressing the “run” button for each query. The test process becomes more or less linear. In some cases, like ours, the zero-error can not be reached, then the formal verification must be completed by a small set of well targeted simulation to reduce even further the probability of undetected errors.

References or information sources

- [0] FormalCheck™ Formal verification tool form Lucent Technology.
- [1] Formal verification of a SDRAM controller . Report on the work done by Luis Lopez during its internship at Texas Instruments France in the Advance System Architecture group.

This Report is accessible on the web.

http://www.asic.sc.ti.com/ses/core_eda/fv (FV team web page)

or http://web.tif.ti.com/adv_arch/formalck

This document is Texas Instruments private information.

- [2] Janssen, Geert L.J.M., 'Hardware Verification using Temporal Logic: A Practical View" in Formal VLSI Correctness Verification VLSI Design methods-II, ed. L.J.M. Claesen pp. 159-168, Elsevier Science Publishers B.V (North-Holland) IFIP, 1990.

- [3] FormalCheck™ User's Guide. Bell Labs Design Automation Lucent Technologies. June 1997.

- [4] Traffic controller Specification. Advanced System Architecture Group. Originator: Serge Lasserre, Dominique D'inverno, Gerard Chauvel. 1997 This document is Texas Instruments private information.

Randal E. Bryant, Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. July 1992 CMU-CS-92-160. School of Computer Science Carnegie Mellon University. Pittsburg, PA 15213.

E. Allen Emerson. 1990. Temporal Logic and modal logic. In Handbook of Theoretical Computer Science, (J van Leeuwen, ed.), Elsevier Science Pub. B. V./ MIT press, 1990, pp. 995-1072.

"Symbolic Model Checking: 10^{20} States and Beyond" J.R Burch, E. M. Clarke, K.L. McMillan. School of computer Science, Carnegie Mellon University. D.L. Dill L.J. Hwang Stanford University.

<http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bryant/www/bdd.html>

Introduction notes on COSPAN and FormalCheck. Kathi Fisler and Bob Kurshan (Lucent).

<http://www.cs.rice.edu:80/~kfisler/>

Chrysalis web site:

<http://chrysalis.com/>

Lucent technology web site:

<http://lucent.com/>