Supplementing Simulation of a Microcontroller Flash Memory Subsystem with Formal Verification

Othmane Bahlous, Infineon Technologies AG and
Abdelouahab Ayari, Mentor Graphics

February 7, 2012

Abstract

This paper describes our efforts to supplement dynamic simulation of an Infineon microcontroller flash memory subsystem with formal verification. We verify the functional correctness of the error correction circuits (ECC) and the flash-array redundancy structures. We use a methodology involving specific steps, including capturing requirements, formulating assertions, reducing complexity, decomposing properties and executing a tool. We present in detail each of these steps and describe how the steps are integrated with a simulation-based coverage flow.

1 Introduction

In recent years, formal verification [5, 7] has emerged as means to address the limitations of simulation: the ability to check only a small fraction of the behaviors of non-trivial hardware designs. Formal verification can be employed to verify that a given design satisfies a given requirement, usually captured using industry standard assertion languages [1, 2] for all legal input patterns. Use of this approach is on the rise and a substantial amount of work [8, 9, 11, 12] has been done to characterize the verification problems that are more appropriate for formal verification. As a result, in recent years, more and more companies are deploying formal verification for components of designs that are difficult to verify sufficiently with dynamic simulation [6, 10].

In this paper we discuss a practical approach to applying formal verification to supplement dynamic simulation of a flash-memory subsystem, specifically to check the functional correctness of the error correction circuits (ECC) and the flash-array redundancy structures.

The paper is organized as follows: Section 2 describes the design under verification and defines the overall context of our work. Section 3 and Section 4 describe the sub-blocks subject to formal analysis. Section 5 discusses the achieved results and draws some conclusions. We close in Section 6 with a short outlook for future deployment of formal verification at Infineon.

2 Design Under Verification

The design under verification (DUV) is a flash subsystem in the Infineon AURIX automotive microcontroller (see Figure 1). It consists of a flash analog macro and control logic, the Flash Standard Interface, or FSI. The FSI consists mainly of an 8-bit RISC CPU with its own RAM/ROM, control and configuration registers, error correction circuits, and redundancy blocks. The design is complex and hugely data intensive.

A UVM testbench environment exists for the DUV. The testbench includes most standard UVM components like scoreboards, monitors, sequences, and drivers. The verification plan includes the different functional requirements of all sub-blocks. We are using Questa Sim [4] to run the dynamic verification, capture the coverage results in its Unified Coverage Database (UCDB), analyze the results, and track coverage of our verification plan. We are using Questa Formal [3] for formal verification of the DUV.

In the next sections, we will focus on the error correction circuits (ECC's) and redundancy blocks. We will describe their behaviors, traditional functional verification and its limitation and present our formal verification approach.

3 Error Correction Circuit

ECC's detect and correct errors introduced during storage or transmission of data. A given ECC has two modes of operation: a generation mode, where it computes the so called error detection code; and a correction mode, where it uses the error detection code to detect and correct the error bits.

In this application, the complexity of the block

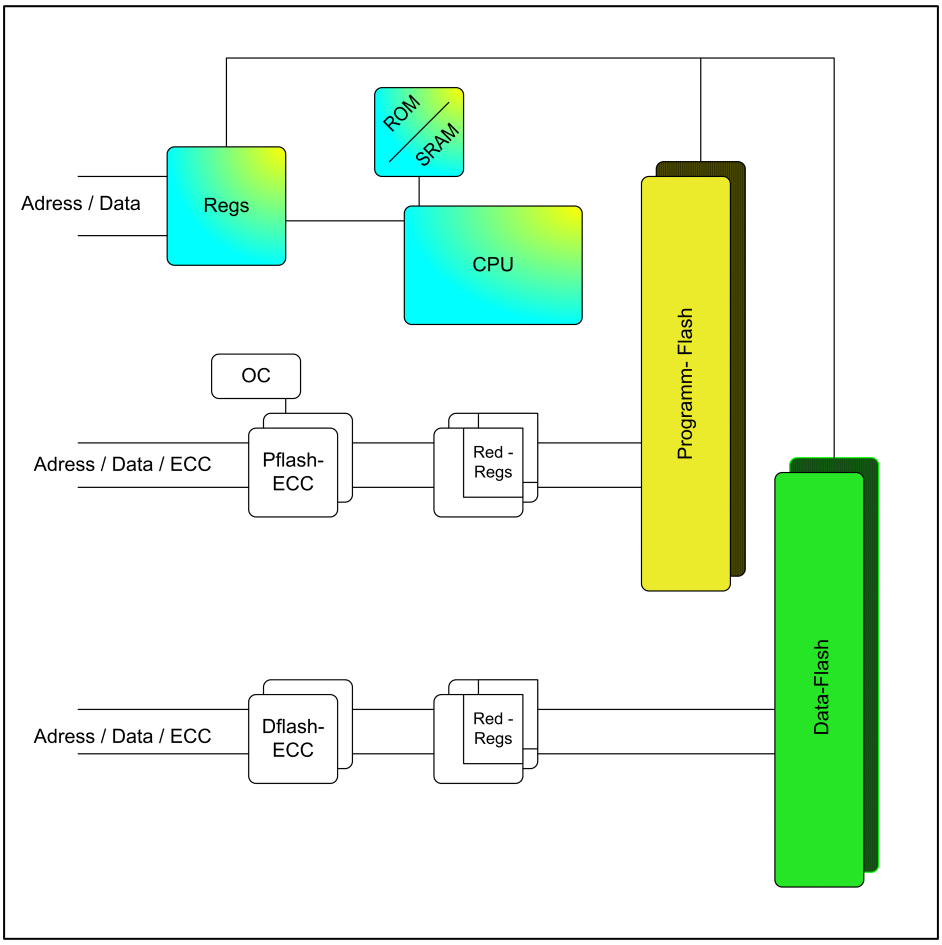

Figure 1: Flash Subsystem

is the major hurdle we are facing in functional verification using traditional simulation-based approaches. We have two sources of complexity here: the first factor is the magnitude of the input data (over 200 bits wide) and the second factor is the large number of possibilities where bit corruption could occur. These factors lead to a huge number of reachable design states. With traditional simulation-based approach using directed tests, only a negligible portion of the state space is covered and the usage of constrained-random simulation gives only marginally better results. To achieve complete coverage here, the number of tests should coincide with the number of all legal input patterns and thus all these tests could never be generated or run unless time and money were of no concern.

This limitation, along with the need to have the ECC's fully verified, created a strong incentive for the use of formal verification. In the remaining section, we will elaborate on this approach in more detail.

3.1 Formal Verification of ECC

In our DUV, there are two kinds of error correction circuits: one circuit is employed for the program flash (PF-ECC) and a second circuit for the data flash (DF-ECC). The two ECCs differ in the data-width, hamming distance and number of error bits to be detected and corrected. The two perform the same functionality, but are based on different algorithms and implementations.

Verifying an ECC is a two-stage process. In the generation mode, a verification engineer must confirm that the computed error detection code coincides with the expected code. In the correction mode, the engineer must confirm that the error detection and correction operations are correct. In practice, this means verifying that the right bit error flags are set and that data correction is performed correctly.

Here we focus only on PF-ECC; the approach is similar for DF-ECC. Setting up the formal verification environment was an easy task. We created a simple SystemVerilog ECC-wrapper that instantiated the ECC, which is in VHDL, and contains the checkers and modeling layer code. For Questa Formal no setup was needed as the tool automatically recognizes all clocks, resets, and port domains without directives. In the next two subsections, we will describe how we model and prove the verification requirements for both modes.

3.2 Verification of ECC Generation Mode

The implemented ECC generation mode, where we check for correct generation of error detection code (*ecc_code*), is based on a reference matrix M that specifies the expected error detection code. Simplifying small details away, the *ecc_code* satisfies the following equation

$$\{ecc\_code, d\} = M \cdot d,$$

where M is a $m \times n$-matrix, and d is an n-width input data.

The equation above is easily implemented in our wrapper:

```
function logic [n-1:0] ref_model (logic [m-1:0] d);
  integer i;
  ref_model = 0;
   for (i=0; i <= m; i++) begin
     ref_model = ref_model xor (d[i]? matrix[i]:0);
   end
  return  ref_model;
endfunction
assign {ref_ecc, ref_data_o} = ref_model(data_i);
```

The correctness check means proving that the error detection code ref_ecc computed by the matrix coincides with that computed by the ECC. Moreover, as the ECC generates error flags that determine whether and where bits are corrupted, it is also important to establish that all error flags are deasserted. Because the ECC forwards the input data to the next block without modification in this mode, we should check that data input and output are the same.

Modeling these requirements is done using SystemVerilog Assertions (SVA). Figure 2 shows the complete set of assertions used here. Our approach is functionally complete. That is, all functional aspects of this mode are captured as assertions and formally proved. Moreover, in contrast to simulation, which will never be able to fully cover these requirements, formal verification completes this verification task within a few seconds. From a cost-benefit perspective, this application shows that with minimal setup and resource consumption, we obtain optimal results.

3.3 Verification of ECC Correction Mode

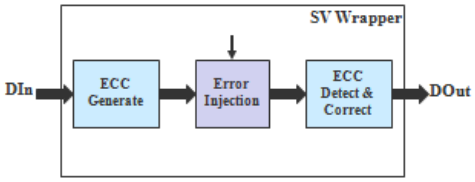

Figure 3: ECC Correction Mode

```
//Check Error Flags
integer i;
for (i=0; i <= c; i++) begin
  check_error_flag_is_low : assert property  (!error_flag[i]);
end
//Check ECC Code and Data I/O
check_correct_ecc_gen       : assert property (ecc_code == ref_ecc);
check_data_o_is_ref_data_o  : assert property (ref_data_o == data_o);
check_data_i_is_gen_data_o  : assert property (data_o == data_i);
check_err_detect_o_is_low   : assert property (! err_detect_o);
```

Figure 2: Assertions for ECC in Generation Mode

```
assign data_ecc_code       = {data, ecc_code};
assign buggy_data_ecc_code = {buggy_data, buggy_ecc_code};
assume_inj:assume property ($countones(data_ecc_code xor buggy_data_ecc_code) <= n);
```

Figure 4: Constraining Number of Bit Errors

In the correction mode, the ECC should detect and flag corrupted bits, and then perform a correction of the affected bits. In this mode, our wrapper contains two instances of ECC and a further module to model data corruption and error injection. Figure 3 displays the structure of our wrapper. The ECC on the left-hand side is used to generate the error detection code. The ECC placed on the right-hand side is used to detect and correct data corruption. The error injection block is used to modify the data coming from the first ECC and forward it in modified form to the next ECC block. In the error injection module, we use a constraint ensuring that its output data is the same as its input data, except for some flipped bits. This constraint is implemented with the assumption given in Figure 4.

The verification requirements in this mode are very similar to those in the generation mode. The correction mode has an additional requirement to check that the ECC is properly correcting the corrupted bits, but the assertions used to verify this requirement are very similar to those used in the generation mode.

Capturing the requirements as SystemVerilog assertions is relatively easy, but verifying them via formal techniques turns out to be difficult. This should not be surprising as ECCs are not optimal for model checking. Indeed, though ECCs generally do a good job of data manipulation, they lack the regular structure that is usually an indication of a good candidate for model checking. To achieve our goal, we decompose large verification tasks into smaller ones, aiming for sub-tasks with a limited level of complexity that is manageable for formal verification.

We used a simple SVA assumption to model error injection in the ECC. This assumption captures all the error scenarios: no error is occurring, one error is occurring, etc. It also implicitly models all bit positions in data where errors can be injected. One way to decompose this problem is to consider each error case separately. Below is the decomposition scheme for the case where two errors are injected.

```
for (p=0; p++; p < DATA_WIDTH)
 begin
  assign buggy_data_ecc_code[p] =
     (p == pos_1 || p == pos_2)?
      not  data_ecc_code[p]
     :     data_ecc_code[p]
 end
```

where pos_1 and pos_2 are parameters indicating the positions where the errors are injected. This decomposition can be seen as explicit assignment of the number of errors and positions where the errors have to occur. From a technical point of view, the decomposition reduces a large source of nondeterminism introduced by the assumption used above.

When formal verification tools run into complexity limitations, users tend to reduce the complexity of the verification task by analyzing only parts of the original state space and abstracting a large part of the state space. In contrast to this approach, our decomposition strategy was complete in the sense that we have handled (1) all the error cases, (2) all possible error positions, and (3) all verification requirements.

The decomposition results in a larger number of verification tasks of substantially smaller complexity. We used Questa Verification Run Manager (VRM) [4] to organize and manage all these formal

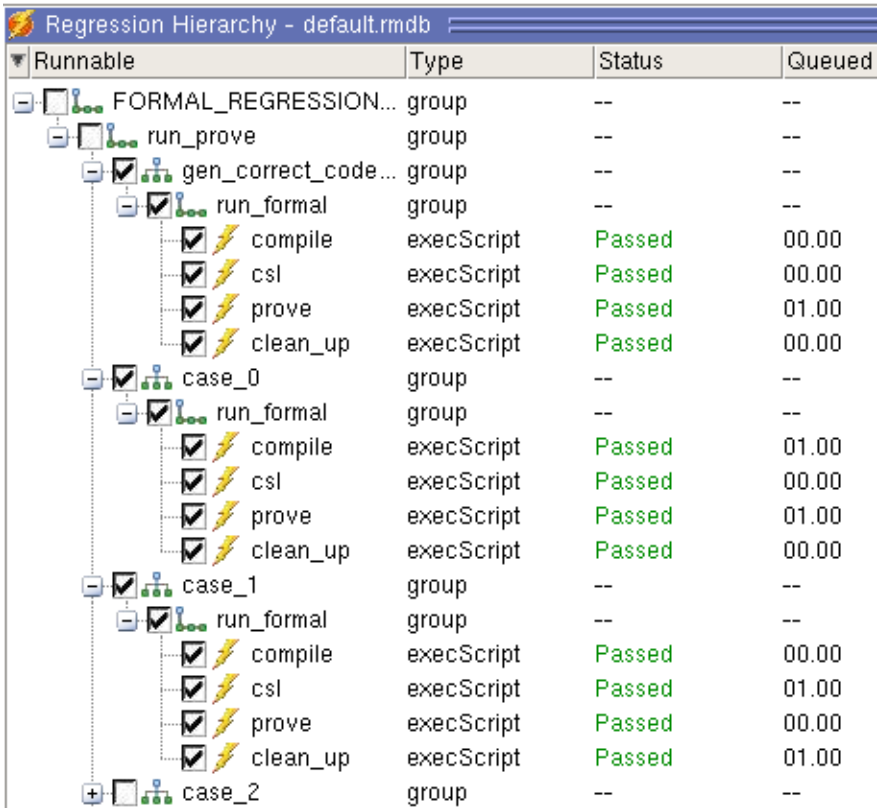

Figure 5: Verification Run Manager for Running Formal Multiple Runs

verification tasks. The greatest benefit from using VRM is the ability to launch multiple runs in parallel on a grid system, monitor progress, and automatically collect the results with minimal setup. Figure 5 shows the selected tasks to run in VRM: *FORMAL_REGRESSION* is the regression name and consists of sub-tasks: *gen_correct_code* stands for the generation mode, *case_0* stands for correction mode where no errors are injected, *case_1* stands for correction mode where one error is injected, etc.

4 Bitline Redundancy

Redundancy is used to improve the yield of the flash component. With additional redundant bitlines, whole bitlines can be replaced if they are defective. There is a configuration register in the redundancy bank (redbank) for each redundant bitline per sector. Each configuration register contains information about (1) whether the bitline is already defective, (2) whether it is in use, and (3) the address of the defective bitline. Depending on the value of the configuration register, a redundant bitline can replace any defective bitline using a redundancy multiplexer. When the flash is accessed, the data bits are correctly assembled using the redundancy multiplexer and taken from initial positions or from redundant bitlines depending on the configuration in the redbank.

The register bank information can be written only through the CPU, which negatively impacts the simulation of the redundancy structure. Normally, the value of redundancy registers is set only once during start-up. For the sake of verification, we should consider different register values, thus the importance of being able to change these values on the fly during simulation. To do so, special CPU microcode is needed to reach high levels of coverage for the registers. Moreover, verifying all writes to the flash subsystem requires additional simulation time, since writing to flash memory is generally time consuming.

4.1 Formal Verification of Redundancy Structure

Verifying redundancy structures requires checking that the redundancy mapping is centrally disabled, that all configuration registers are accessible for read and write, and that the scrambling is happening according to the specification (i.e., that output data is a scrambled form of the input data depending on the register values). Our challenge, which we addressed using a customized assertion generator,

was to independently prove the correct behavior for each redundant structure.

For disabling the redundancy mapping, we have to ensure that if the disable signal is active, then the flash data input and output are identical. This is realized with following simple assertion:

```
check_disabling_red:
  assert property(disable_i |->
                  fdata_i == fdata_o);
```

For register write operation, we have to ensure that the information in the redbank (defective address, usage, fail) is correctly updated within one clock cycle. Conversely, we should ensure that information update is always preceded by a writing operation. For the read operation, we have to ensure that the output data contains the same information as the redbank. Figure 6 displays the assertions used for these two operations.

In checking the scrambling, we use a nested "for" loop to generate the assertions addressing the correctness for each page. Five assertions are generated for each page, describing the scrambling behavior in both directions. The variables in these assertions are the page number and the physical sector number. Figure 7 summarizes these assertions written in *pseudo* SVA code.

We write our assertions in a separate SystemVerilog checker that is bound via SystemVerilog *bind* command to the design, which is in VHDL. Binding SystemVerilog modules to VHDL designs is well supported in Questa Formal. Using bind, we are able to access all internal signals including those signals with complex user-defined types. The VHDL packages were made available to SystemVerilog via the shared package support that our formal verification tool provides.

Questa Formal ran on all redundancy checking assertions (340 in total) and completed its run within five hours using about 8G memory space on a 64-bit 2.7 GHz RHEL machine.

Block-level functional verification can be exported to the same unified coverage database used by simulation for tracking the coverage results. At Infineon one of our objectives is to merge results including coverage across all verification engines used for IP/Block level verification and then track our overall coverage automatically against our verification plan. The UCDB in the Questa platform provides such a capability.

4.2 Back-annotation of Formal Results in the Verification Plan

The redundancy verification requirements are included in the flash subsystem verification plan, which contains a section for the three redundancy verification items previously discussed. Each item is linked to a set of appropriate assertions. The verification plan itself resides in a collection of word processing and spreadsheet documents, and once created it can be imported into Questa Sim's Unified Coverage Database (UCDB). In addition to the verification plan, the UCDB is used to save all coverage-related metrics (including code coverage, functional coverage, and assertion data) as well as formal coverage (created using Questa Formal) and additional test and user metrics. Once imported, the verification plan UCDB can be merged with the simulation coverage data to allow the user to see their verification plan items alongside their actual coverage data. Our formal verification tool provides a utility that exports its results into a UCDB file that can later be merged with other UCDBs created by other tools/methodologies.

As stated before, all the redundancy checking assertions are formally proved. We then run the Formal export utility and obtained a UCDB which we merge with our verification plan. The content of the resulting UCDB is shown in Figure 8 and there we can see that all the three verification requirements are completely covered. Note that the resulting UCDB does not contain simulation coverage data, which can be incrementally merged once it is available.

Note that an important remark concerning the validity of combining verification results from different tools should be made here. In our case, the assertions of the redundancy block are proven in a constraint-free context. That is, no input assumptions are used. Therefore, these assertions will remain valid if we add constraints to the proof context. That means constraints coming from the flash-subsystem and also added through wiring of this block in the flash-subsystem will not change the validity of these assertions. However, the proofs of these assertions can become vacuous. Thus, we can skip the verification of the functionality of the redundancy block at the flash-subsystem but we still must check in some sense that the redundancy block is correctly driven.

5 Conclusion

The formal verification of the ECC showed that today's model checking tools make it easier than ever to use this technique, which in the past was seen as needlessly complex. We completely captured the functional requirements with assertions and completed full proofs, a major step towards higher design quality. Using a simulation-based approach for the ECC will cover only a small portion

```
property prop1 (a);
int x, y;
 (wr_red_i, x=addr_red, y=dta_red_is) |=> a[x]==y;
endproperty

check_rw_A_1:assert property (prop1(addr_array));
check_rw_A_2:assert property (prop1(fails));
check_rw_A_3:assert property (prop1(used));


check_rw_B_1:assert property (##1 !$stable(redb)
    |->   redb[$past(redb_addr)].fail_e) == $past(redb_i[J])
       && redb[$past(redb_addr)].used_e) == $past(redb_i[I])
       && redb[$past(redb_addr)].addr_e) == $past(redb_i[K:0]));
check_rw_B_2:  assert property (! wr_red_i |=> $stable(redb));

check_rw_C_1:  assert property (redb_o[K:0] == redb[redb_addr].addr_e);
check_rw_C_2:  assert property (redb_o[J]   == redb[redb_addr].fail_e);
check_rw_C_3:  assert property (redb_o[I]   == redb[redb_addr].used_e);
```

Figure 6: Assertion Set for Register Read and Write Operations

```
for every sector
 for every page in a wordline
  Assert:
   IF (RD_REG[page].used and not RD_REG[page].fail)
      and (Redundancy structure used and not defect)
      and (other structures are defect or not used)
   THEN replace RD_REG[page].addr by the redundant line and all others are passed
        through

  Assert:
   IF (no red used or red defect)
   THEN (data passed through)

  Assert:
   IF (more than one red used and addr to change different)
   THEN replace RD_REG[page].addr by the redundant line and all others are passed
        through

  Assert:
   IF (more than one red used and same addr to change)
   THEN replace RD_REG[prio_high][page].addr by the redundant line and all others are
        passed through
```

Figure 7: Assertion Set for Correct scrambling

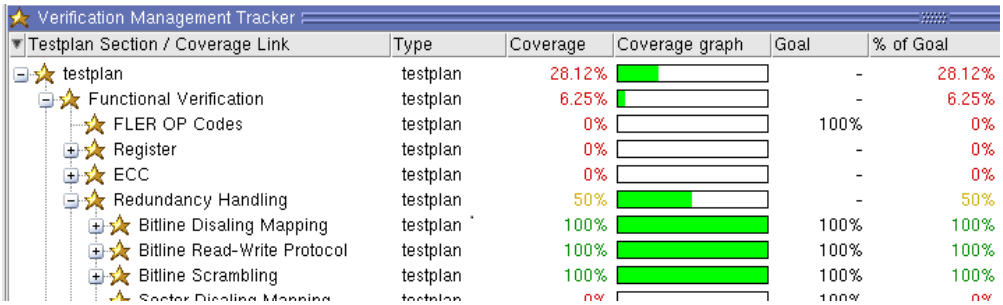

Figure 8: Formal Results Merged with Verification Plan in Questa's Verification Tracker Window

of the state space and consume a lot of run time.

One other important benefit is that during the proof process the tool generated counter-examples several times. Our debugging activities and close work with the designer showed that the assertion failures were due to specification ambiguities. That is, the ECC proof process was also a means to clarify and eliminate ambiguities from the specification and enable a dialog between the design and verification teams to better understand system functionality.

There are three main conclusions from this formal verification pilot: First, formal verification enabled complete verification of a block, whereas simulation would only have yielded a fraction of the coverage. Second, we saved time not only in running simulations but also in developing and debugging specific assembler codes used by the CPU to test the redundancy structures. Third, we showed how formal verification and simulation can be seen as complementary methodologies. Formal results can be exported to a coverage database and such results can be back-annotated to the verification plan. The same verification plan is used by the simulator to back-annotate the simulation results. Ultimately, coverage computed by different methodologies is merged in one single coverage database.

6 Future Work

The formal verification approach presented in this paper showed very promising results. This motivated us to plan further formal verification tasks in our department. In particular, we are planning to formally analyze a sensitive CPU-interrupt handler. We also are planning to extend our application to SoC interconnectivity verification.

Acknowledgments. We thank the designers Thomas Rabenalt and Mathew Neal for the fruitful discussions about the design. We thank Mark Handover, Gabriel Chidolue, Ping Yeung, Mark Eslinger, and Geoffry Koch, Thomas Ellis, Guido Clemens, Chris Rockwood, and Bruno Hanßler for comments on this paper.

References

[1] IEEE Standard for Property Specification Language (PSL), IEEE Std 1850-2005.

[2] IEEE Standard for SystemVerilog, IEEE Std 1800-2009.

[3] Questa® Formal User Guide, version 10.0d. Mentor Graphics Corporation, 2011.

[4] Questa® SIM User's Manual, version 10.0d. Mentor Graphics Corporation, 2011.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[6] Richard Boulton and Mark Handover. Cost Evaluation for Adopting Formal Property Checking: A Detailed Case Study. In *Proceedings of DVCon 2009*, pages 200–205, San Jose, CA, USA, February 2009.

[7] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. In *CHDL*, pages 15–30, 1993.

[8] Harry Foster, Adam Krolnik, and David Lacey. *Assertion-based design (2. ed.)*. Kluwer, 2004.

[9] Harry Foster, Lawrence Loh, Bahman Rabii, and Vigyan Singhal. Guidelines for creating a formal verification testplan. In *Proceedings of DVCon 2006*, San Jose, CA, USA, February 2006.

[10] Roger Sabbagh and Jim O'Connor. Have I Placed all the Right Assertions in all the Right Places? -Boosting Assertion Quality With Visual Flow Diagrams. In *Proceedings of DVCon 2007*, San Jose, CA, USA, February 2007.

[11] Ping Yeung. Five hot spots for assertion-based verification. In *Proceedings of DVCon 2005*, San Jose, CA, USA, February 2005.

[12] Ping Yeung. How to instrument your design with simple systemverilog assertions. In *EE Times*, 2011.