

Detecting Malicious Logic Through Structural Checking

Scott C. Smith, Senior Member, IEEE
University of Missouri – Rolla
Department of Electrical and Computer Engineering
133 Emerson Electric Co. Hall
1870 Miner Circle, Rolla, MO 65409
Phone: (573) 341-4232, E-mail: smithsco@umr.edu

Jia Di, Member, IEEE
University of Arkansas
Department of Computer Science and Computer Engineering
323 Engineering Hall
Fayetteville, AR 72701
Phone: (479) 575-5728, E-mail: jdi@uark.edu

Abstract—Hardware is just as susceptible as software to “hacker attacks”, through inclusion of malicious logic; and the consequences of such an attack could be disastrous! The impact of software viruses has been felt, at one time or another, by the entire computerized world, through loss of productivity, loss of system resources or data, or mere inconvenience. However, the nature of malicious logic and defending against it is fundamentally different from its software counterpart. Malicious logic has the added dimension of not being removable once encapsulated in the system. This paper will identify hardware vulnerabilities and will outline an automated method, called *Structural Checking*, to detect and prevent malicious logic from becoming incorporated into an ASIC, which could cause catastrophic system failure, security breaches, or other dire consequences.

I. INTRODUCTION

Hardware can be compromised through the inclusion of malicious logic that causes the system to output incorrect data or output data to the wrong port or address, that changes the system’s internal timing or control, or that disables the system clock or bus. This can be done by changing or adding internal logic, in such a way that it is undetectable using traditional testing and verification tools and techniques. These days most complex systems are not designed from scratch; they instead use many 3rd party Intellectual Property (IP) blocks. Hence, one or more 3rd party IP blocks could contain malicious logic that may affect the entire system. Furthermore, all non-trivial digital designs rely heavily on Computer Aided Design (CAD) tools, such as Synopsys, Mentor Graphics, Cadence, etc. These CAD tools themselves could contain software viruses, causing them to insert malicious logic into hardware circuits.

Take for example the case of disabling the system bus by inserting minimal additional circuitry that would not be detected using traditional testing and verification procedures. This could be done by adding a tri-state buffer (TSB), an SR latch, and a few AND gates, as shown in Fig. 1. This logic would only disable the bus, forcing one bit to a constant value, when one specific data value is either written to or read from one specific memory address. Hence, this type of malicious logic would not be detectable using traditional testing and verification techniques, since this would require an exhaustive

test of all possible data values being written to every memory address (i.e., an N-bit address and M-bit data would require $2^N \times 2^M$ writes), which is unrealistic for today’s large memory sizes. Furthermore, the number of exhaustive test combinations to detect the malicious logic using traditional methods could be even further increased by adding additional inputs to the AND function; the number of bus bits disabled by the malicious logic could be easily increased by inserting additional TSBs; and the input to the TSB could be connected to any internal signal, instead of Vcc or ground, making the bus failure that much harder to diagnose.

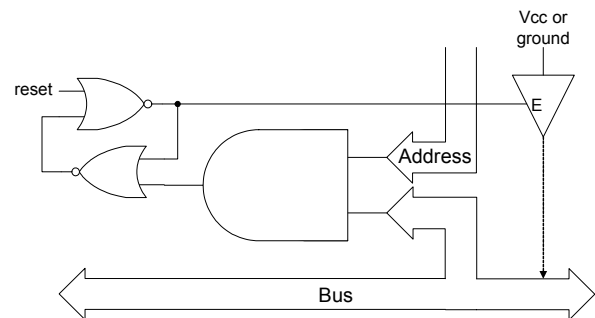


Fig. 1. Malicious hardware to disable the system bus.

This paper will expand upon this example, present other possible hardware vulnerabilities, and describe an automated method, called *Structural Checking*, to detect malicious logic during the ASIC design phase, before a chip has been fabricated.

II. RELATED WORK

A. Hardware Verification Methods and Their Limitations in Malicious-Logic Checking

Hardware verification is a crucial step to detect design errors prior to production. Verification becomes increasingly important as circuit complexity continues to grow. Various verification methods have been developed and implemented in CAD tools. Although it seems that these techniques can be applied for malicious-logic checking, they are not suitable for this purpose due to their limitations.

B. Requirements of Malicious-Logic Checking

Checking for malicious logic must be performed at all levels of abstraction, from RTL to physical level netlist, since malicious logic could be added at any stage in the design cycle. Furthermore, a malicious-logic checker must be able to handle today's very large designs; therefore, an automated tool that requires minimal user intervention, and efficiently implements the developed malicious-logic checking algorithms in terms of execution time is needed. In addition, it is highly desirable that the algorithm/tool is easy to understand/use so that the end users are able to customize it to their needs.

C. Simulation

Historically, simulation was the most widely used method to verify the functional behavior of a circuit. Simulation can be performed at all levels of abstraction; and the result of each run is trustable. However, the problem with simulation is that the quality of the check is only as good as the quality of the test cases. Also, gate-level simulations are notoriously slow to execute, which is a major problem as the size of digital designs continues to grow. Therefore, simulation has been proven more effective in very early stages of hardware debugging, when the design is still infested with multiple bugs, rather than in later stages when only subtle bugs remain uncovered [1]. Since malicious logic is considered to be a subtle bug, simulation is neither effective nor efficient for malicious-logic checking.

D. Formal Verification

Given two behavioral descriptions of a hardware design, with the higher level description as "specification" and the lower level one as "implementation", formal verification consists of proving formally that the implementation is "properly related" to the specification [2].

1) Equivalence Checking

Equivalence checking is commonly used to formally prove that two representations of a circuit design exhibit exactly the same behavior [3]. The RTL behavior of a digital circuit is usually described with a hardware description language, such as Verilog or VHDL. This description is considered to be the golden reference model of the design. Once the RTL description has been verified by other methods, the design is usually converted into a netlist by a logic synthesis tool, which in theory guarantees that the netlist is logically equivalent to the RTL source code. In practice, programs have bugs and it would be a major risk to assume that the synthesis has been performed without error. Also, in real life it is not uncommon for designers to make manual changes to a netlist, thereby introducing a major additional error factor.

Therefore, a verification step is needed to check the logical equivalence of the netlist to the golden reference model. In addition to simulation, an alternative way to solve this is to formally prove that the RTL code and the netlist synthesized from it have exactly the same behavior in all "relevant" cases. This process is called formal equivalence checking and is a

problem that is studied under the broader area of formal verification. Recent research in equivalence checking has advanced various aspects, e.g., robust Boolean reasoning [4], improved sequential checking [5-9], application for test [10], RTL static sign-off [11], and microprocessor validation [12-13]. In these approaches, the RTL description is always considered to be error-free, and is used to verify gate-level netlists.

Although theoretically equivalence checking is able to find inconsistencies in the netlist compared to the RTL model, it cannot detect malicious logic in the RTL code itself, since the RTL model is assumed to be error-free. From this point of view, equivalence checking targets implementation errors instead of design errors. Therefore, equivalence checking cannot guarantee a circuit's trustability.

2) Model Checking

Computation Tree Logic (CTL) based model checking or Temporal Logic model checking is a method of verification that ensures a system satisfies its requirement specification by (i) representing the system as a finite Kripke structure, (ii) writing the specification in a suitable temporal logic, and (iii) algorithmically checking that the Kripke is a model of the specification, meaning that the specification holds or is true with respect to the finite Kripke structure [1]. Due to the finiteness of the Kripke structure and algorithmic process, model checking automates the verification of finite state systems. This method has been applied successfully to verify both combinational and sequential circuit designs [14-18]. In addition to the advantage of being fully automatic, model checking always produces a counter example, which provides invaluable insight to realize the real reason for the failure and crucial clues for fixing the problem.

Because of its reliance on an explicit state transition graph (STG) representation of the hardware system to be verified, the main technical challenge of model checking is the state explosion problem, i.e., the number of states in a global graph increases exponentially with the number of gates/processes/elements in the system [19]. Many approaches have been proposed to address this issue, including the symbolic method that represents transition relation implicitly with ordered binary decision diagrams (OBDDs) [20-21], and state reduction techniques such as symmetry reduction [22-26], partial order reduction [27-28], and abstraction [29-35].

However, these approaches are either not effective enough or have not been demonstrated on larger circuits. Besides, some of them, e.g., extraction of FSM controllers and abstraction of the design, tend to cause costly false errors [36]. Considering the efficiency requirement of malicious-logic checking, especially for run-time reconfigurable FPGAs, the computation complexity of the model checking method is not acceptable. In addition, model checking typically uses a state-based hardware description that is oriented towards expressing its behavior rather than its structure, which makes it difficult to apply hierarchical verification. It is therefore incompatible with the proposed Structural Checking approach for detecting malicious logic.

3) Theorem Proving

Automated theorem proving (ATP) is the proving of mathematical theorems by a computer program. Depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible. For the frequent case of propositional logic, the problem is decidable but NP-complete, and hence only exponential-time algorithms are believed to exist for general proof tasks. For first-order logic, identifying theorems is recursively enumerable, i.e., given unbounded resources, any valid theorem can eventually be proven. Invalid statements, i.e., formulas that are not entailed by a given theory, cannot always be recognized. In these cases, a first-order theorem prover may fail to terminate while searching for a proof. More expressive logics, such as higher order and modal logics, allow for convenient expression of a wider range of problems than first order logic, but theorem proving for these logics is less developed [37]. In general, theorem proving is useful for verifying algorithms and integrating verification results [36]. This has been demonstrated in several previous works [38-39].

The major limitation of theorem proving is that it is a deductive process by its very nature, which raises both theoretical and practical concerns regarding management of its complexity. Automation must be provided to some degree. However, most of the theorem provers available today are semi-automated at best, in that they require some form of human interaction to guide the proof searching process [19], which has prevented this from being a general solution. Moreover, since first-order theorem proving uses Boyer-Moore logic, its ability to represent and handle time, e.g., asynchronous inputs like interrupts, is unsatisfactory [19]. These limitations make theorem proving not suitable for malicious-logic checking.

E. Semi-Formal Methods

As a compromise between simulation and formal verification techniques, semi-formal methods, including coverage measurement, test generation, symbolic simulation, and model checking for bugs [36], have been developed. While these methods have demonstrated effectiveness on certain applications, they have their limitations as well, e.g., missing bugs, BDD explosion, published examples are too small, models are hard to justify, etc. These problems need to be solved before semi-formal methods can be used to detect malicious logic in circuits.

III. STRUCTURAL CHECKING TO DETECT AND REMOVE MALICIOUS LOGIC

A. Structural Checking Algorithm

After potential attacks have been enumerated through the Hardware Threat Modeling process [40], their corresponding malicious logic constructs would be incorporated into a Structural Checking tool, such that the tool could identify similar constructs in an IC as potentially malicious. To do so,

Structural Checking would first construct a bi-directional linked-list structure of the entire ASIC, which would then be searched for known malicious logic constructs. The linked-list would be bi-directional so that it could be searched starting from the inputs to detect internal data tampering or denial of service malicious logic, or starting from the outputs to detect external data tampering or information leakage attacks using a “trace back” approach. The tool would generate a report containing the location and type of the suspected malicious logic, and prompt the user to verify if the suspected logic is indeed malicious. If so, the tool would automatically remove the malicious logic.

This method is applicable to all levels of abstraction, from RTL description to physical layout (i.e., GDSII format). However, application to transistor-level or physical-level netlists is much more difficult than to RTL or gate-level netlists, due to the increasing complexity of the required linked-list structure of the circuit and the greater difficulty in representing malicious logic constructs at these lower levels of abstraction.

An RTL description will first be synthesized to a library of known components before creating the structural linked-list model, while a gate-level netlist can either be resynthesized or can use an intermediate library to map the gate-level netlist components to a set of known components (i.e., the functionality of each component in the structural model must be known by the Structural Checking tool). The linked-list structure will then be searched for malicious logic constructs, such as extra bus drivers, extra latches, gated clocks, etc. When detected, the malicious logic will be removed from both the linked-list structure and the original circuit description, pending user approval. This can be done by mapping the malicious circuit structure to its corresponding portion of the original circuit description, and modifying the infected circuit description segment in a predetermined manner. Therefore, the mapping between the RTL code or original gate-level netlist and the linked-list structural model must be retained, such that malicious logic constructs can be mapped back to their origin and removed.

B. Structural Checking Example

Take for example the case of disabling the system bus, as shown in Fig. 1 and explained in the introduction. This malicious logic was added to a simple 4-bit RISC microprocessor, such that bit 1 of the data bus was disabled only when data value 1011_2 was either written to or read from address $1EC_{16}$. This malicious logic only required a few lines of VHDL code and one additional internal input, E_f , as shown in Fig. 2. The microprocessor, including the malicious logic, worked correctly when executing numerous simulations, running a variety of different benchmark programs, showing that traditional testing techniques would not find this problem. When an extra instruction to write 1011_2 to address $1EC_{16}$ was inserted, bit 1 of the bus was disabled, as expected. Even this simple microprocessor with a very small $11\text{-bit} \times 4\text{-bit}$ memory has $2^{11} \times 2^4 = 32,768$ different address/data

combinations.

```
TSBex: tri_state_buff port map(Ef, logic1, D_data(1));  
Ef <= ((not D_addr(10) and not D_addr(9) and D_addr(8) and D_addr(7)  
and D_addr(6) and D_addr(5) and not D_addr(4) and D_addr(3) and  
D_addr(2) and not D_addr(1) and not D_addr(0) and D_data(3) and not  
D_data(2) and D_data(1) and D_data(0)) nor Ef) nor reset;
```

Fig. 2. VHDL code to disable the system bus.

This malicious logic can however be detected by simply scanning the synthesized netlist. By performing a rudimentary analysis of the circuit, the expected number of TSBs could be calculated. If the circuit does not contain the expected number of TSBs, the circuit is compromised, otherwise the circuit is deemed ready for functional verification, which would detect any TSBs replaced by malicious ones (i.e., for the circuit to pass functional verification, the malicious TSBs would need to be extra TSBs).

We wrote a C-program that scans the synthesized netlist in one pass to detect any malicious bus drivers. The program currently runs external to the synthesis tool and is setup to work with circuits synthesized to the Mentor Graphics SCL05u synthesis library. This tool could also be used from within the Mentor Graphics synthesis tool, Leonardo Spectrum, by using a Tcl (Tool Command Language) script. The automated Tcl flow would first synthesize the circuit to the SCL05u synthesis library, write the synthesized netlist to a file, and then call our C-program to scan the netlist to detect and remove any malicious TSBs. This would require minimal deviation from the regular synthesis flow, requiring the user to execute one additional Tcl script from the pull-down menu. Furthermore, the Tcl script can be easily ported to other CAD platforms (e.g., Synopsys, Cadence, etc.).

The program scans the SCL05u synthesized netlist to count the number of TSBs, and compares the expected number of TSBs to the actual number, outputting “no malicious bus drivers detected” if the two are the same, otherwise the number of TSBs is output along with the message “check expected number of tri-state buffers,” meaning that either the expected number of TSBs is incorrect or the design contains one or more malicious TSBs. This approach was applied to the microprocessor described above, successfully detecting the malicious TSB.

IV. COMPREHENSIVE EXAMPLE

This example consists of a dedicated failsafe system to control fuel rod positioning in a nuclear reactor to ensure that the reactor operates in the safe range, thus avoiding a nuclear meltdown. Since this is a safety-critical system, it would be designed using failsafe techniques, and would consist of two identical but independent processing units that would both compute the necessary actions to be taken, and would trigger an alarm and eventually shut down the reactor, whenever the processing units’ results differed (i.e., redundancy checking), or when one or both of the units stopped functioning. Furthermore, the system would be isolated from external

communication to prevent network-based software attacks. The processing units could be commercial microcontrollers/DSPs, but would more likely be designed as radiation-hardened ASICs to make the system more secure and robust. This system is very similar to the failsafe dual-DSP system developed by the author in conjunction with QuEST, LLC, as part of a redesign of the obsolete Bay Area Rapid Transit (BART) train control system.

A. Application of Hardware Threat Modeling

Applying the Hardware Threat Modeling concept [40] to this design would determine: 1) that information leakage is not likely to occur, since the system will be isolated from the outside world; 2) that denial of service could occur, but is not likely to cause serious consequences, since this would trigger an alarm indicating that human intervention is required, and would automatically shut down the reactor if the problem persisted; and 3) that the most critical hardware vulnerability is data tampering, since false data could cause both ASIC processing units (i.e., controllers) to incorrectly position the fuel rods, causing the reactor to operate beyond the safe range, which could eventually lead to a nuclear meltdown.

Focusing on this type of threat and the ASIC controller architecture, potential attacks would be identified, such as modifying sensor data inputs, altering fuel rod control positioning calculations, modifying fuel rod positioning outputs, etc. In order for malicious logic to successfully perform one of these attacks, it must be inserted in such a way that the system operates as expected, except under a very specific unique set of circumstances, which would likely not be discovered during testing and verification. Furthermore, since this is a safety-critical system utilizing dual ASIC controllers, both ASICs would need to tamper with the data in exactly the same way and at the same time in order to successfully defeat the failsafe redundancy checking.

B. Example Attack

Assume that one of the main inputs to the fuel rod positioning calculation is reactor temperature, which is determined through an averaging process of multiple readings of various temperature sensors, which are input to both ASICs as 8-bit unsigned integers, as depicted in Fig. 3(a). The fuel rod positioning outputs are then fed into a redundancy checking system to ensure that both ASICs agree with the proposed fuel rod movement, or lack thereof. When the temperature is within the safe range, the system lowers the rods to increase the reactor fuel in order to produce additional energy; and when the temperature rises above a given threshold, the fuel rods are raised to ensure that the reactor continues to operate within the safe range. However, if the ASIC controller was modified, as shown in Fig. 3(b), to include malicious logic that would make the temperature input to the fuel rod positioning controller seem much lower than it actually is, and this attack could be simultaneously triggered on both failsafe ASIC controllers, the system would not be able to detect when the temperature rose above the safe range,

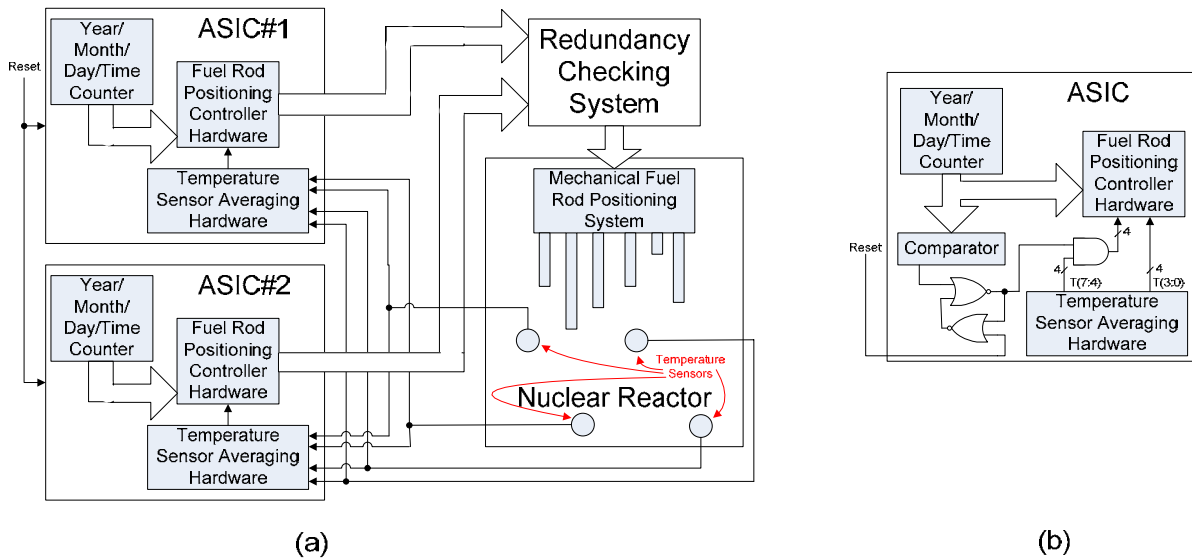


Fig. 3. (a) Nuclear reactor fuel rod positioning control system (b) ASIC with malicious logic inserted.

so the fuel rods would not be raised and the reactor would eventually become unstable, causing a catastrophic nuclear meltdown.

This attack could easily be done with the addition of very few gates, as shown in Figure 3(b). This approach would take advantage of the Year/Month/Day/Time Counter to simultaneously attack both ASICs in order to defeat the Redundancy Checking System. It would utilize a comparator to check for a specific time, day, month, and year (e.g., 9/11/09 at 11:00 p.m.); and when detected, would reset the cross coupled NOR gates, which would cause the most significant 4 bits of the controller's temperature input to become stuck at zero, thus signifying a safe temperature input to the controller. The comparator logic can be implemented using very few NAND gates; and the rest of the malicious logic consists of 2 NOR gates and 4 AND gates to hold the most significant 4 bits of the controller temperature input at zero after the comparator detects the specified hardwired time, day, month, and year.

A. Detection using Traditional Techniques

This is a very simple example, but even so, it would be very unlikely that the malicious logic would be detected using standard testing and verification techniques. Assuming the counter output was in BCD format (i.e., 8 bits for year, 5 bits for month, 6 bits for day, 6 bits for hour, 7 bits for minute, and 7 bits for second), the counter would need to operate for 2^{39} seconds to cycle through all combinations, which would take more than 17,000 years. Furthermore, additional inputs could be added to the comparator to ensure that even an exhaustive test of the counter would not necessarily detect the malicious logic.

B. Detection with Structural Checking

Once this type of data tampering attack is identified using the Hardware Threat Modeling approach, it can be incorporated into the Structural Checking tool, which can then

efficiently detect the malicious logic. In the Structural Checking approach, a linked list structure of the entire ASIC would be constructed, which would then be searched for known malicious logic constructs (i.e., similar in concept to the way that software virus detection tools operate). In this case, the tool would output the location and type of all latches and the number of flip-flop components. The designer would then read the automatically generated report from the Structural Checking tool and determine if the total number of flip-flops matches the expected number and if there are any unexpected latches. If the ASIC does not contain any additional memory elements, then it is deemed ready for functional verification, which would detect any memory elements replaced by malicious ones (i.e., for the circuit to pass functional verification, the malicious memory elements would need to be extra, otherwise the system would not function as expected during normal testing).

Applying this technique to the nuclear reactor ASIC controller example would result in a report containing the number of flip-flops in the design and the location of an SR latch (i.e., the cross-coupled NOR gates) on a path from the BCD counter to the Fuel Rod Positioning Controller Hardware. The designer would then read the report and determine that the number of flip-flops in the design is correct, but that there should not be an SR latch between the BCD counter and the Fuel Rod Positioning Controller Hardware. Upon closer inspection of this latch and the surrounding hardware, the designer would determine that it is indeed malicious logic; and it would then be removed from the circuit.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have outlined a method, called Structural Checking, for detecting malicious logic within hardware circuits, which if left unchecked, could have catastrophic

consequences, just like their software virus counterparts. This method has been automated and applied to a simple RISC microprocessor, and was able to detect malicious bus-disabling logic, which went undetected throughout numerous simulations, running a variety of different benchmark programs.

We then explained how malicious logic could be cleverly inserted into a nuclear reactor control system in order to cause the reactor to meltdown; and we showed how our Structural Checking approach could be used to detect the malicious logic, but have not yet incorporated this type of threat into our automated Structural Checking tool. This example pertained to checking for one specific type of threat for one particular application; however, similar examples can easily be realized for other types of critical infrastructure (e.g., control of the electric power grid, subway and train controllers, aviation control systems, etc.). Furthermore, in this example, Structural Checking was applied to a gate-level netlist; however, it is also applicable to RTL code, transistor-level netlists, and physical-level netlists.

We are currently working on identifying additional potential malicious logic constructs through Hardware Threat Modeling [40], and then incorporating these into our Structural Checking tool.

REFERENCES

- [1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, The MIT Press, 1999.
- [2] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey," *ACM Transactions on Design Automation of E. Systems*, Vol. 4, pp. 123-193, April 1999.
- [3] Wikipedia: http://en.wikipedia.org/wiki/Formal_equivalence_checking (available February 2007).
- [4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21/12, pp. 1377-1394, Dec. 2002.
- [5] J. R. Jiang and R. K. Brayton, "On the verification of sequential equivalence," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22/6, Jun. 2003.
- [6] S. Huang, K. Cheng, K. Chen, C. Huang, and F. Brewer, "AQUILA: An Equivalence Checking System for Large Sequential Designs," *IEEE Transactions on Computers*, Vol. 49/5, May 2000.
- [7] C. A. J. van Eijk, "Sequential Equivalence Checking Based on Structural Similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19/7, Jul. 2000.
- [8] M. N. Mneimneh and K. A. Sakallah, "Principles of Sequential-Equivalence Verification," *IEEE Design & Test of Computers*, Vol. 22/3, May 2005.
- [9] M. Syal and M. S. Hsiao, "VERISEC: Verifying Equivalence of Sequential Circuits using SAT," *10th IEEE International High-Level Design Validation and Test Workshop*, 2005.
- [10] J. Bhadra, N. Krishnamurthy, and M. S. Abadir, "Enhanced Equivalence Checking," *IEEE Design & Test of Computers*, Vol. 21/6, Nov. 2004.
- [11] H. Foster, "Applied Boolean Equivalence Verification and RTL Static Sign-off," *IEEE Design & Test of Computers*, Vol. 18/4, Jul. 2002.
- [12] P. Mishra, N. Dutt, "A Methodology for Validation of Microprocessors using Equivalence Checking," *4th International Workshop on Microprocessor Test and Verification Common Challenges and Solutions (MTV'03)*, 2003.
- [13] F. Corella, "Automated Verification of Behavioral Equivalence for Microprocessors," *IEEE Transactions on Computers*, Vol. 43/1, Jan. 1994.
- [14] J. Sawada, "Formal Verification of an Advanced Pipeline Machine," Ph.D. Dissertation, University of Texas at Austin, Dec. 1999.
- [15] W. A. Hunt and J. Sawada, "Verifying the FM9801 Microarchitecture," *IEEE Micro*, Vol. 19/3, May 1999.
- [16] J. Rushby, "Model Checking and Other Ways of Automating Formal Methods," *Model Checking for Concurrent Programs Software Quality Week*, San Francisco, May 1995.
- [17] M. Musuvathi, "CMC: A Model Checker for Network Protocol Implementations," Ph.D. Dissertation, Stanford University, Feb. 2004.
- [18] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," *5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.
- [19] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, Vol. 1, pp. 151-238, 1992.
- [20] K. L. McMillan, *Symbolic Model Checking*, Boston: Kluwer Academic Publishers, 1993.
- [21] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. 35/8, pp. 677-691, 1986.
- [22] E. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Formal Methods in System Design 9*, Vol. 1/2, pp. 41-76, Dec. 1996.
- [23] E. Emerson, and A. Sistla, "Symmetry and model checking," *Formal Methods in System Design 9*, Vol. 1/2, pp. 105-130, 1996.
- [24] E. Emerson, and R. Treffer, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking," *Correct Hardware Design and Verification Methods (CHARME), Lecture Notes in Computer Science*, Vol. 1703, Springer-Verlag, New York, pp. 142-156, 1999.
- [25] C. Ip, and D. Dill, "Better verification through symmetry," *9*, Vol. 1/2, pp. 41-76, 1996.
- [26] K. Jensen, "Condensed state spaces for symmetrical colored petri nets," *Formal Methods in System Design 9*, Vol. 1/2, pp. 7-40, 1996.
- [27] P. Godefroid, D. Peled, and M. Staskauskas, "Using partial order methods in the formal verification of industrial concurrent programs," *ISSTA'96 International Symposium on Software Testing and Analysis*, pp. 261-269, 1996.
- [28] D. Peled, "All from one, one from all: on model checking using representatives," *5th International Conference on Computer Aided Verification, Lecture Notes in Computer Science*, Vol. 697, Springer-Verlag, New York, pp. 409-423, 1993.
- [29] P. Cousot, and R. Cousot, "Refining model checking by abstract interpretation," *Automated Software Engineering*, Vol. 6, pp. 69-95, 1999.
- [30] D. E. Long, "Model checking, abstraction and compositional verification," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa. CMU-CS-93-178, 1993.
- [31] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Prog. Lang. Syst. (TOPLAS)*, pp. 1512-1542, 1994.
- [32] R. P. Kurshan, "Computer-Aided Verification of Coordinating Processes," Princeton University Press, Princeton, NJ, 1994.
- [33] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, "Tearing based abstraction for CTL model checking," *International Conference of Computer-Aided Design (ICCAD)*, 1996.
- [34] A. Pardo, and G. Hachtel, "Incremental CTL model checking using BDD subsetting," *Design Automation Conference (DAC)*, 1998.
- [35] Z. Manna, M. C. Coln, B. Finkbeiner, H. Sipma, and T. E. Uribe, "Abstraction and modular verification of infinite-state reactive systems," *Requirements Targeting Software and Systems Engineering (RTSE)*, 1998.
- [36] D. Dill, "What's Between Simulation and Formal Verification?," slides from a presentation by Prof. Dill, Stanford University at DAC'98.
- [37] Wikipedia: http://en.wikipedia.org/wiki/Theorem_proving (available February 2007).
- [38] W. A. Hunt, "FM8501: A verified microprocessor," Ph.D. Dissertation, Technical report ICSCA-CMP-47, University of Texas at Austin, 1985.
- [39] W. A. Hunt, "The mechanical verification of a microprocessor design," In D. Borrione, editor, *From HDL Description to Guaranteed Correct Circuit Designs*, pp. 89-129, North Holland, Amsterdam, 1987.
- [40] J. Di and S. C. Smith, "A Hardware Threat Modeling Concept for Trustable Integrated Circuits," to appear in *IEEE Region 5 Technical Conference*, April 2007.