Formal Verification of FIRE: A Case Study

Jae-Young Jang
Dept. of ECE
University of Colorado
Boulder, CO 80309
jjang@duke.colorado.edu

Shaz Qadeer
Dept. of EECS
University of California
Berkeley, CA 94720
shaz@eecs.berkeley.edu

Matt Kaufmann **Carl Pixley**
Motorola Inc.
7600A Capital of Texas Highway
Austin, TX 78731
{matt_kaufmann,carl_pixley}@email.mot.com

Abstract

We present our experiences with the formal verification of an automotive chip used to control the safety features in a car. We used a BDD based model checker in our work. We describe our verification methodology for verifying a very complicated property on a relatively large design. We also describe the bugs that were found and present our views on how to make model checking an effective integrated part of the design flow for complex hardware systems.

1 Introduction

Verification is increasingly becoming the bottleneck in the design flow of electronic systems. Simulation of designs is very expensive in terms of time and exhaustive simulation is virtually impossible. As a result, designers have turned to formal methods for verification. Symbolic model checking is becoming a popular method for verifying commercial sequential designs.

In this work, we used Verdict, a BDD based CTL (Computational Tree Logic [1]) model checker under development at Motorola, to verify the correctness of an automotive chip we will refer to as FIRE. Verdict has been extensively used to verify commercial designs [2]. In the past few years, as model checking has become a popular method for verifying sequential designs, several case studies of verification of commercial designs have been published [3, 4, 5]. This work is intended to be a case study in which several aspects of formal verification of commercial hardware designs are highlighted. In this work, close interaction with the designer resulted in the identification of a very complex property to be verified on a relatively large design. A number of important issues came up during the process. First of all, unclear and imprecise specifications of the design proved to be a big hurdle. We have realized now after talking to other people in the industry that this is a very common and neglected problem. Second, the property that the designer wanted us to verify was so complicated that it was not possible to express it as a CTL formula on just the interface variables of the design. Intimate knowledge of the design was required and properties had to be formulated on internal signals. Third, we faced the very well known problem of state explosion. To overcome this problem, we tried to break the proof into simpler proof obligations on the modules of the design. This was done by manually creating an abstraction of the environment for each module and proving suitable local properties on them so that the global property that we wanted to verify could be deduced from the local properties. It is a limitation of current formal verification tools that we were dependent on our intuition for verifying the correctness of the abstractions and for deducing the global property from the proved local properties.

FIRE is a complex automotive chip used to implement safety features in a car. FIRE interfaces with a micro-controller and some analog devices controlling the safety features like seat-belts and airbags. The micro-controller controls these safety analog devices by giving appropriate instructions to FIRE. The micro-controller gets data from an electro-mechanical device, an accelerometer for instance, that detects an impending collision. The internal register file of FIRE is accessible to the micro-controller to be read and written to. The micro-controller communicates with FIRE by reading and writing to its internal registers according to a complicated protocol that will be described later. If the car crashes, the micro-controller and another module called *TzDriver* (not a part of FIRE) can act together to have FIRE assert a signal called *crash* which is an input to the safety devices. The safety devices can perform one or more of the following four actions as a result of the *crash* signal: pop the driver or the passenger side airbag or tighten the driver or passenger side seat-belt.

The micro-controller interacts with FIRE according to a protocol. To write data to a particular register in FIRE, the controller puts the address and data on the bus in one write cycle and puts the complemented address and data on the bus in the next write cycle. Only then does FIRE accept the write as valid and commit the data to the addressed register. Since address bus and data bus transactions are multiplexed, each write cycle is composed of two sub-cycles, the address and the data being put on the bus on successive sub-cycles. Similarly, for reading data from some register, the controller puts the address on the bus and FIRE puts the data in the addressed register on the bus in the first read cycle. In the next cycle, the controller puts the complemented address and FIRE puts the complemented data on the bus. Only then does the micro-controller accept the data as valid.

A schematic diagram of FIRE is given in Figure 1. The principal components of FIRE are a set of temporary registers (*ADD, ˜ADD, DATA*), a register file (*R0-R7*), a De-multiplexer (*ST-DEMUX*), and two modules called *Watchdog* and *TzTest*. It has 64 I/O pins and 172

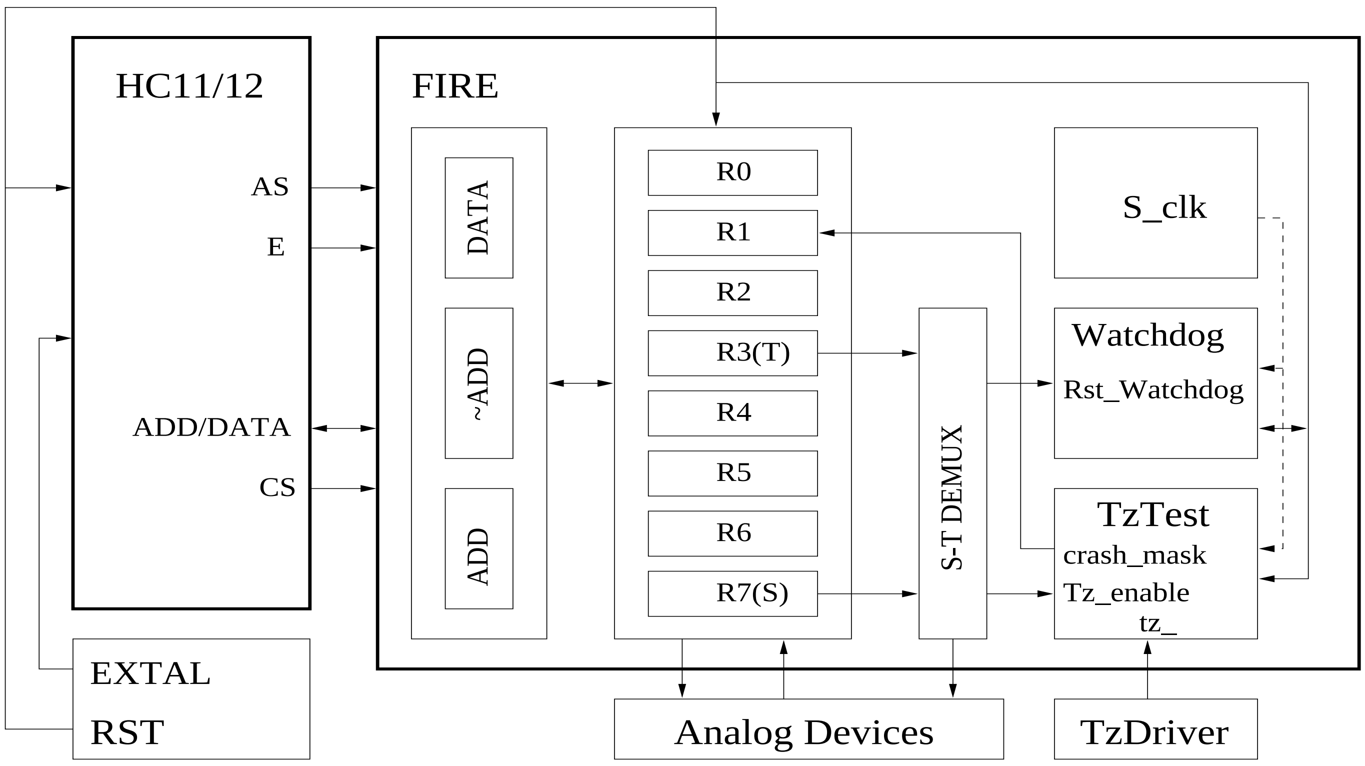

Figure 1: Schematic of FIRE.

latches. A feature of FIRE is that it can interface with either one of two micro-controllers – HC11 or HC12. It can automatically detect which micro-controller is in the environment. Two different clocks that are not synchronized with each other are used to drive the different components of FIRE. *EXTAL* clocks the micro-controller (4MHz for HC11 and 2MHz for HC12) and a slower clock *S_clk* (100KHz) drives the *Watchdog* and *TzTest* modules. The micro-controller generates signals AS and E which are inputs to FIRE. E is the clock signal for FIRE. The temporary registers *ADD*, *˜ADD* and *DATA* are used by FIRE to implement the protocol described above. The micro-controller can also ask FIRE to perform some special control tasks by writing appropriate values to distinguished registers *S* and *T*. The De-multiplexer logic deciphers these values and generates the correct signals to perform the desired tasks. The *RST* signal is the reset which if asserted resets both the micro-controller and FIRE. The module *Watchdog* is a safety feature built into the design for detecting a malfunctioning micro-controller. It has a counter and its function is to assert *RST* if the counter expires. Therefore, the micro-controller periodically asks FIRE to assert *Rst_Watchdog* which has the effect of resetting the counter in *Watchdog*.

It is expected of safety devices that they come into operation only when the need arises. Otherwise, they may cause a lot of irritation and might even result in an accident. Consider the case of airbags in cars. If the safety device malfunctions and the airbag pops out while the car is on a freeway, it might very well cause an accident. A module like *Watchdog* and the complicated protocol for reading and writing data were put in place for exactly this reason. Therefore, a very important property that we would like to verify is that the FIRE chip gives the *crash* signal to the analog device if and only if the micro-controller has given it the appropriate instructions. In the next section, we present our method for proving this property on the design.

This paper is structured as follows. In Section 2, we describe the verification problem and our methodology. We also discuss some of the important issues that cropped up during our work. We hope that this discussion will motivate further research in computer-aided formal verification. In Section 3, we describe the results of our experiments. In Section 4, we describe the bugs we found. In Section 5, we conclude by presenting an overview of the difficulties we faced and some directions for future work needed to make verification more integrated in the design flow.

2 Methodology

We started with the Verilog model of FIRE and its specification document. The property that concerned the designer of the chip most was that an airbag should be popped only when the correct firing sequence is observed by FIRE. But the precise definition of the "correct firing sequence" in terms of actual signals in the Verilog model was not clear at all from the specification document. To get a precise CTL property on the signals in the chip we were forced to study the Verilog model in detail. We finally realized that the correct operation of FIRE depended on recognizing a complex sequence of operations. Only after this sequence of operations had been performed should FIRE assert the *crash* signal to the analog device. In order to recognize the complex sequence of actions which should lead to the assertion of the *crash* signal, the correct operation of the register file and the *S-T DEMUX* is essential. The *S-T DEMUX* is a complex piece of logic which detects consecutive writes to registers S and T and performs some appropriate controlling function depending on what was written in the two registers. We found the "correct firing sequence" to be as follows. First, the micro-controller has to activate FIRE by setting a register *QRS*(The third bit of R3) inside FIRE to 1. It does this by writing appropriate values to S and T in consecutive write cycles. If a crash is detected, *TzDriver* (not a part of FIRE) asserts the signal $tz\_$ which is an input to *TzTest*. A small finite state machine inside *TzTest* detects this, enables a counter and asserts a signal $crash\_mask$ which remains high for the duration of the counter. The micro-controller should write appropriate data values to registers $R5$ and $R6$ in that duration. Only some bits of the registers $R5$ and $R6$ are relevant. Denoting these bits by i, the signal *crash* is given by $QRS \wedge crash\_mask \wedge R5_i \wedge R6_i$. The property to be verified is that *crash* is asserted if and only if the above described sequence of events happen. Since the property to be verified was so complex and involved all the components of FIRE, we decided to break it into a set of local properties such that if these properties are satisfied by the model then the original property is also satisfied by the model. Although we did not formally prove that the local properties suffice, our method was validated by enabling us to discover three bugs (see Section 4).

Notice from the description above that satisfaction of the property is contingent on the correct implementation of the protocol for writing the register file, the *S-T DEMUX* logic and the FSM inside the module *TzTest*. Therefore, we focused the verification effort on these three components. First, we verified the protocol for writing the register file of FIRE. We verified the property that a register is updated if and only if the micro-controller puts the correct sequence of address and data on the address/data bus, i.e., correct address, correct data, complement address and complement data appear on consecutive cycles on the bus. Second, we verified the operation of the *S-T DEMUX* logic. Specifically, we verified the operation of the finite state machine which detects consecutive writes to S and T. We also verified the control behavior of the De-multiplexer relevant in the case of a crash. Third, we verified the finite state machine inside *TzTest* module which controls the signal *crash_mask*.

We discuss below some of the important issues that came up during our work.

2.1 Environment for the micro-controller

FIRE's interface with the micro-controller consists of the signals AS (address strobe), E, CS (chip select) and the address/data bus. To transfer data from the micro-controller to FIRE chip, two write cycle are needed and FIRE has to be selected by asserting CS in both of them. Note that other transactions could be performed on the bus between the two write cycles in which FIRE is selected, i.e. there could be an arbitrary number of clock cycles in which CS is not asserted between two write cycles. If we let the interface signals vary nondeterministically, we could write a CTL property that says whenever $write\_cycle_1$ is followed by $write\_cycle_2$ with FIRE being selected in both cycles, then in the very next cycle the register gets updated. The property, using the *Weak Until*[1] operator $\mathbf{U}_w$, would look something like the following.

$$\begin{aligned}&\mathbf{AG}((CS \wedge write\_cycle_1)\\ &\quad \Rightarrow \mathbf{AX}(\mathbf{A}\,[\neg CS\ \mathbf{U}_w\ (CS \wedge\\ &\quad ((write\_cycle_2 \wedge \mathbf{AX}(reg\_updated)) \vee\\ &\quad (\neg write\_cycle_2 \wedge \mathbf{AX}(\neg reg\_updated))))]));\end{aligned}$$

The properties $write\_cycle_1$ and $write\_cycle_2$ have been used as macros in the above property. They would be complicated properties themselves, and would somehow have to state that the addresses and data bits in the two cycles were complementary. Clearly, this approach would make the property to be verified very long and complicated. Notice that this approach would result in complicated properties for any other part of the design, S-T DEMUX logic for instance, whose correct behavior is contingent on receiving the correct writing sequence from the micro-controller.

In our approach, we modeled the micro-controller environment as a nondeterministic finite state machine. Assertion of CS to select FIRE and generation of a read or a write operation is done nondeterministically in each cycle. We implemented a monitor inside the environment and it asserts a particular variable *monitor_valid* whenever the micro-controller issues a correct writing sequence. Having such a variable makes CTL properties very easy to express. Of course, the trade off is that the state space of the design increases. But we have observed that a complicated property with a lot of nested sub-formulae can sometimes be much more efficiently model checked by creating a simple monitor and simplifying the property, even though the state space increases. We verified the correctness of the environment and the monitor by checking CTL properties on them without considering the FIRE module. This was easy to do because the environment and monitor were small and relatively uncomplicated.

2.2 Multiple Clocks

Currently, model checkers can work only on designs with a single global clock. If multiple clocks are present, the design can be verified by model checking only if they are synchronized. In this case, the fastest clock is made the global clock and the other clocks are derived from it. It is a limitation of the model (Kripke structure [1]), i.e., state transition graph of global states, on which the model checking algorithms work, that asynchronous clocks cannot be handled more directly. In this model, the design is visualized as being in some global state which changes at every clock tick according to the state transition graph. In the case of FIRE, there are two asynchronous clocks in the system. Therefore, we do not have a global clock which controls transitions between states. The interface between the micro-controller and FIRE is controlled by signal *E* driven by the faster clock called *EXTAL*. The other clock *S_clk* which is 40 times slower than *EXTAL* is used only inside the *Watchdog* and *TzTest* modules. Of course, we could just forget about asynchrony and synchronize the two clocks. This reduces the behavior in the system and could actually lead to erroneous results. In this case, we actually discussed the problem with the designer of FIRE and synchronized the clocks on his suggestion[2].

2.3 Modular Verification and Abstraction

State space explosion is a well-known problem. In hardware designs, it manifests itself in an exponential growth of the state space with the number of latches in the design. Since model checking algorithms are based on state space exploration, their efficacy is also limited by this phenomenon. The introduction of BDDs [6] as a symbolic representation of a set of states increased the number of states that could be handled[7]. But, even BDDs have their limit and cannot handle designs with a lot of state holding elements.

We faced the state explosion problem in the case of FIRE. Our tool synthesized more than 200 latches from the Verilog description of FIRE and our model of its environment. Consequently, the tool was taking a lot of time to check the properties and could not even finish on some of them. To get around this problem, we tried to partition the design into more or less independent components. Most industrial designs are done in a modular fashion and FIRE is no exception. The register file, the *S-T DEMUX* logic and the *TzTest* module formed more or less independent pieces of logic with some interfacing signals. We realized that the properties we wanted to check

[1]$\mathbf{A}\,[p\ \mathbf{U}_w\ q] \equiv \neg\, \mathbf{E}\,[\neg q\ \mathbf{U}\ (\neg p \wedge \neg q)]$. A state s satisfies $\mathbf{A}\,[p\ \mathbf{U}_w\ q]$ if on all paths from s either p is always true or p is true until q becomes true.

[2]The designer felt that synchronizing the two clocks is a safe assumption.

were mostly local in nature. As an example, consider the local property of the register file that a register is updated correctly if and only if the correct writing sequence (according to the protocol) is output by the micro-controller, which does not depend at all on the other parts of the circuit. To check this property, we can create abstract models of the other pieces in the design and compose the model for the register file with these abstract pieces to get an abstract model of the whole design. Generally, this can reduce the state space in the abstract design dramatically. The verification task is thus broken down into two hopefully simpler tasks – verifying the properties on the abstract design and proving that if the abstract design satisfies a property the actual design will satisfy it too. In the case of verifying a property on the register file, the most natural abstraction for all pieces interacting with it is a nondeterministic FSM with a single state and a self loop. In other words, we let the inputs to the register file from the other modules in FIRE vary nondeterministically.

For verifying some properties, detailed models of more than one subcircuit had to be put together because of complex interactions among them. In such cases, verification was taking a lot of time. We used abstraction in a number of cases to simplify the problem. In the register file, there were 64 latches (8 registers each containing 8 bits). A very natural abstraction is to neglect the other registers while verifying that any particular register is updated correctly. In this case, the correctness of the abstraction is justified by the observation that the operation of a register does not depend on any other register. In the *TzTest* module, there was a 12-bit counter. The purpose of this counter is to keep some signal asserted for some duration. The *crash* signal is asserted if the micro-controller accesses the *S-T DEMUX* correctly in this duration. We believe that the actual duration of the counter does not matter as long as it is much longer than the time it takes for the micro-controller to write S and T. We made use of this observation to abstract away some of the bits of the counter thereby reducing the state space. Reducing the length of the counter was very useful in reducing the number of reachability steps[3]. Related work can be found in [8]. Note that in making this abstraction, intimate knowledge of the design was required. We will come back to this point in Section 5.

We observe here that getting the properties to go through the model checker required a lot of manual abstraction and that we had to rely on our intuition for the correctness of these abstractions. Our experiences described above have led us to believe that there is an urgent need for a higher degree of automation of compositional verification and abstraction in current verification tools. Researchers have been looking at compositional proof techniques for a number of years but we have yet to see a practical implementation of these ideas. Recently, an attempt at automating a limited kind of abstraction methodology using the model checker in VIS [9] as a decision procedure has been made in [10].

3 Verification

In this section, we describe in more detail specific examples of important properties that we model checked on the design. We verified a total of 76 CTL properties dealing with the different pieces of the design. Note that a lot of effort was spent in the formulation of a correct set of CTL formulae that expressed the properties we wanted.

In our experiments, we found two features of Verdict especially useful. First, the property-dependent scaling option to the model checker automatically reduces the state space to only those variables that influence the model checking property under consideration. This significantly reduces the time spent in model checking the property. Second, the dynamic variable reordering was very useful in reducing the intermediate BDD sizes which again reflected in a reduction in the time for doing model checking.

[3] If the counter is initialized, the number of reachability steps is at least 2^l, where l is the length of the counter.

We now describe some examples of properties on different parts of the design.

3.1 Data Path

The data path is constructed with read and write operation from the micro-controller to the register file in the FIRE chip. Data transfer is valid if the following sequence of operations is done in order.

$$OriginalAddress(OA) \rightarrow OriginalData(OD) \rightarrow$$
$$ComplementedAddress(CA) \rightarrow ComplementedData(CD)$$

Because some instructions for other chips may be issued between these operations (when CS is not asserted), this valid data transfer protocol is not easily written in CTL. As explained before, we used a monitor in the environment of FIRE to detect the correct writing sequence. The monitor sets a latch called *monitor_valid* whenever it sees the correct writing sequence. Then the CTL properties for data transfer are very simple.

$$\mathbf{AG}(monitor\_valid \rightarrow register\_is\_updated);$$
$$\mathbf{AG}(\neg monitor\_valid \rightarrow \neg register\_is\_updated);$$

3.2 FSM behavior

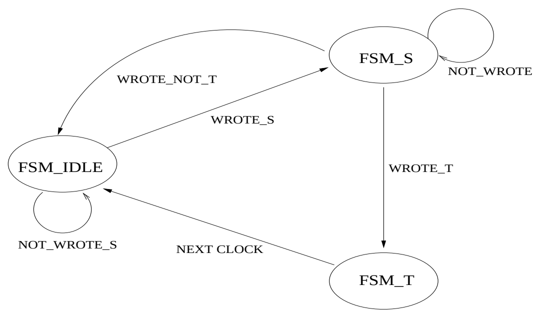

Figure 2: An example of a verified FSM

We had to verify properties on critical state transitions of a number of FSMs. Figure 2 is the one example that we verified. This FSM is part of the module called *S-T DEMUX*. The De-multiplexer in this module is enabled only in the state FSM_T and performs the proper action based on the values of registers S and T. We used the *Weak Until* operator to come up with the following property.

$$\mathbf{AG}(FSM\_IDLE \rightarrow \mathbf{A}[(\neg WROTE\_S \wedge FSM\_IDLE)\,\mathbf{U}_w(WROTE\_S \wedge \mathbf{AX}\,(FSM\_S))]);$$

A second property we checked on this FSM was false. This property is described in Section 4. Analogous properties were checked for other FSMs too.

4 Bugs

We found two bugs of which the first one is critical.

1. A register in the register file should be updated only by a valid writing sequence as earlier. But the temporary registers of FIRE which are used to maintain the history of the data transfer protocol were not initialized properly. As a result, data could be written to a register at the beginning of the complement write cycle. This bug could cause unexpected firing of safety devices. The property that yielded the debug trace is the following.

   $$\mathbf{AG}(\neg monitor\_valid \rightarrow \neg reg\_is\_updated);$$

2. If a valid write to register S is immediately followed by a valid write to register T, the De-multiplexer in *S-T DEMUX* is enabled. On the other hand, if another write operation is issued between these two write operations, the De-multiplexer should be disabled. This turned out to be false and a debug trace was generated by the model checker on the following property.

   $$\begin{aligned}&\mathbf{AG}(FSM\_S \rightarrow \mathbf{A}[(\neg WROTE \wedge FSM\_S)\\ &\quad \mathbf{U}_w((WROTE\_T \wedge \mathbf{AX}(FSM\_T)) \vee\\ &\quad (WROTE \wedge \neg WROTE\_T \wedge \mathbf{AX}\,(FSM\_IDLE)))]);\end{aligned}$$

5 Conclusion

In this study, we tried to verify the correctness of a design where the criterion for correctness was very complicated. In the process, we found two bugs of which one was critical. The case study illustrated a lot of problems that are faced while verifying commercial hardware designs using model checking. We think that these problems provide pointers to areas that need more work to make model checking, and more generally formal verification tools, an integrated part of a more effective design process.

We mentioned that a very big hurdle we faced was a lack of clear specifications on signals in the design. We actually had to study the details of the design to figure out the specification. This approach has the inherent danger of writing an incorrect specification because of a bug in the design. Nevertheless, we feel that since designers have the best knowledge of the design they should write specifications themselves. Thus, writing specifications should be made as integrated a part of hardware design as documentation is a part of software design.

The study also illustrates an effective use of the concept of monitors. By building an appropriate environment and monitor for the design, we can simplify the construction of CTL properties to express specifications and thereby reduce the chance of an incorrect specification. This strategy can sometimes reduce the actual model checking time in the case when the property without the monitor is very complicated.

In view of the large number of states in current hardware designs, it is important that verification tools should provide support for techniques like compositional verification and abstraction. Multiple asynchronous clocks are very common in hardware design. The model of a global state transition graph which is used in current model checkers cannot model such designs. Other models which can do this should be investigated.

We believe that model checking is an effective way of verifying sequential designs. The level of expertise needed to use a model checking tool is not very high because it is highly automated. But, to use the tool effectively the user should have some knowledge of BDD representations and the notion of variable reordering. We found that dyanamic variable reordering was very helpful in reducing the model checking time. We believe that the most effective users of model checking are the designers themselves. We have seen that a thorough knowledge of the design may be needed to check properties even with the state-of-the-art in model checking. The design cannot always be viewed as a black box by the verifier. Therefore, the designer himself is often in the best position to verify his design.

Acknowledgments

Bernard Plessier, the designer of FIRE, was of immense help in formalizing the specifications of the design.

REFERENCES

[1] E. A. Emerson, “Temporal and Modal Logic,” in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072, Elsevier Science, 1990.

[2] B. Plessier and C. Pixley, “Formal Verification of a Commercial Serial Bus Interface,” in *International Conference on Computers and Communications*, (Phoenix, U.S.A.), pp. 378–382, 1995.

[3] E. M. Clarke and O. Grümberg and H. Hiraishi and S. Jha and D. E. Long and K. L. McMillan and L. A. Ness, “Verification of the Futurebus+ Cache Coherence Protocol,” in *Proc. 11th Intl. Symp. on Comput. Hardware Description Lang. and their Applications*, 1993.

[4] A. T. Eiríksson and K. L. McMillan, “Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study,” in *Proc. 7th International Conference on Computer Aided Verif ication*, LNCS, pp. 367–380, 1995.

[5] I. Beer, S. Ben-David, C. Eisner, and A. Landver, “RuleBase: An Industry-Oriented Formal Verification Tool,” in *Proc. of the Design Automation Conf.*, (Las Vegas, NV), pp. 655–660, June 1996.

[6] R. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[7] C. Pixley, “Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence,” in *Proc. of the Conf. on Computer-Aided Verification* (E. M. Clarke and R. P. Kurshan, eds.), vol. 531 of *Lecture Notes in Computer Science*, pp. 54–64, June 1990.

[8] E. Macii, B. Plessier, and F. Somenzi, “Verification of systems containing counters,” in *Proc. Intl. Conf. on Computer-Aided Design*, (Santa Clara, CA), pp. 179–182, Nov. 1992.

[9] R. K. Brayton *et al.*, “VIS: A system for verification and synthesis,” in *Eigth Conference on Computer Aided Verification (CAV’96)* (T. Henzinger and R. Alur, eds.), pp. 428–432, Rutgers University: Springer-Verlag, 1996. LNCS 1102.

[10] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, “Tearing Based Automatic Abstraction for CTL Model Checking,” in *Proc. Intl. Conf. on Computer-Aided Design*, (San Jose, CA), pp. 76–81, Nov. 1996.