Verification-Guided Soft Error Resilience

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

Wenchao Li
UC Berkeley
wenchao@berkeley.edu

Subhasish Mitra
Stanford University
subh@stanford.edu

Abstract

Algorithmic techniques for formal verification can be used not just for bug-finding, but also to estimate vulnerability to reliability problems and to reduce overheads of circuit mechanisms for error resilience. We demonstrate this idea of verification-guided error resilience in the context of soft errors in latches. We show how model checking can be used to identify latches in a circuit that must be protected in order that the circuit satisfies a formal specification. Experimental results on a Verilog implementation of the ESA SpaceWire communication protocol indicate that the power overhead of soft error protection can be reduced by a factor of 4.35 by using our approach rather than protecting all latches.

1. Introduction

Technology scaling to 65nm and below has caused reliability problems to become a dominant design challenge. In fact, design today can be seen as a process of achieving a trade-off between performance, power, and reliability. Problems arise due to soft (transient) errors, aging, environmental and device parameter variations, and aggressive deployment to reduce power and increase performance. In particular, soft errors can be significant contributors to system-level silent data corruption and have been the subject of much recent research [5, 16].

There is therefore a pressing need for error-resilient design as well as estimation of the system-level impact of circuit-level errors. For soft errors in latches and flip-flops, there are many protection techniques already available; we point the reader to recent papers [17, 16] for relevant references. However, circuit mechanisms for error resilience come at the price of increased power and area overheads, and possibly reduced performance. As an example, we cite recent fault injection experiments on a microprocessor design [20]. The authors report that protecting 60% of latches against soft errors sufficed to bring the chip-level soft error rate down to 9%. However, further bringing the error rate down to 0 incurred significant overheads, including increasing the power penalty to 18.2% from 10.6%. There is therefore a need to identify only those circuit resources that must be protected against reliability problems in order for the circuit to meet necessary specifications.

We present a *verification-guided approach to error resilience*, wherein algorithmic techniques for formal verification are used to estimate system vulnerability to device errors and reduce the overheads of circuit mechanisms for error resilience (fault tolerance). The underlying idea is that errors that do not affect circuit correctness, as given by a formal specification, can be safely ignored. We demonstrate our approach for dealing with soft errors in latches, using the *single-event upset* (SEU) error model. Combining a formal SEU model with a formal circuit model, we use the state-of-the-art Cadence SMV model checker [1] to identify latches that must be protected as well as those that don't. (Note that our approach can be used with any verification method and tool.) The problem of identifying latches that need not be protected is different from classical sequential redundancy with respect to permanent stuck-at faults in two ways. First, the error persists only for a single cycle. Second, the "redundancy" of a latch is with respect to a formal specification, generally captured by a set of assertions, rather than only checking equivalence of a faulty and fault-free circuit.

We present a case study of a publicly available Verilog implementation of end-nodes in the SpaceWire spacecraft communication protocol proposed as a standard by the European Space Agency (ESA) [2]. The results of our experiments show that most latches in the SpaceWire circuit can be left unprotected even for a comprehensive formal specification created from the ESA standards document [8], resulting in a reduction in power overhead from 58% for protecting all latches to just 13%.

Related Work. Our approach can also be viewed as an exhaustive way to perform fault injection, guided by a formal specification. The approach has both pros and cons compared to the alternative approaches of random fault injection [9, 19, 12, 11] and fault-free simulation (e.g., architectural vulnerability factor estimation) [18], and complements them and other techniques [4, 14]. A brief comparison with these alternative approaches with respect to five key factors is given in Table 1. In particular, the proposed formal approach is effective even for designs for which good workload estimates are unavailable. Krautz et al. [13] recently presented a formal approach for analyzing the effectiveness of error detection and correction logic; our work differs in many ways, including in that it is applied to an arbitrary circuit, before fault tolerance is employed, in order to identify which latches must be protected.

Our technique has a dual use: the results can be re-used for computing mutation-based coverage metrics for formal

Factor	Random Fault Injection	Fault-Free Simulation	Verification-Guided
Input coverage	Not exhaustive	Not exhaustive	Exhaustive
Fault coverage	Which signals to inject faults in? When?	Non-issue	Exhaustive
Applicability	General	Application-specific	General
Confidence	Needs long simulation runs; which outputs to compare?	High confidence can yield pessimistic results	Very high, but needs comprehensive formal spec.
Scalability	Fair (accuracy can degrade)	Variable	Problem for large designs (needs compositional reasoning)

Table 1. Comparison of verification-guided approach to others

verification [10, 6]. Thus, our approach can be seen as not only meshing with an existing verification flow, but in fact generating result used both to compute coverage and reduce overheads of error resilience.

2. Background

We give some background on the formal verification technique of model checking that is used in this paper.

Model checking is a highly automated formal verification technique, that uses algorithmic methods to exhaustively explore all states reachable from the initial circuit states. Model checking algorithms form the basis of many recent industry tools for *assertion-based verification*, *sequential equivalence checking*, and *property checking*. Further details may be found in the book by Clarke et al. [7].

Formal specifications can be provided in a variety of ways. For sequential equivalence checking, the specification is captured by a simpler version of the circuit that is considered correct; e.g., a simple, non-pipelined processor can serve as a specification for a complex, pipelined, superscalar version. However, for arbitrary control logic, ASICs, and whole systems, design requirements are often most easily captured as partial specifications, and formalized using *temporal logic*. Temporal logic forms the basis for the recent IEEE standard 1850 property specification language (PSL) [3].

The experiments reported in this paper use Cadence SMV, a state-of-the-art model checker based on the technique of *symbolic model checking* [15]. Formal specifications for the main case study were written in *linear temporal logic* [7].

3. Approach

We describe our approach in this section, including the overall flow (Sec. 3.1), underlying formal notions (Sec. 3.2), and relation to coverage metrics for verification (Sec. 3.3).

3.1. Tool Flow

The overall flow of our approach is depicted in Figure 1. Although we have shown this flow for SEUs, it is also applicable for other classes of circuit reliability problems.

The inputs to the process are the RTL description of the

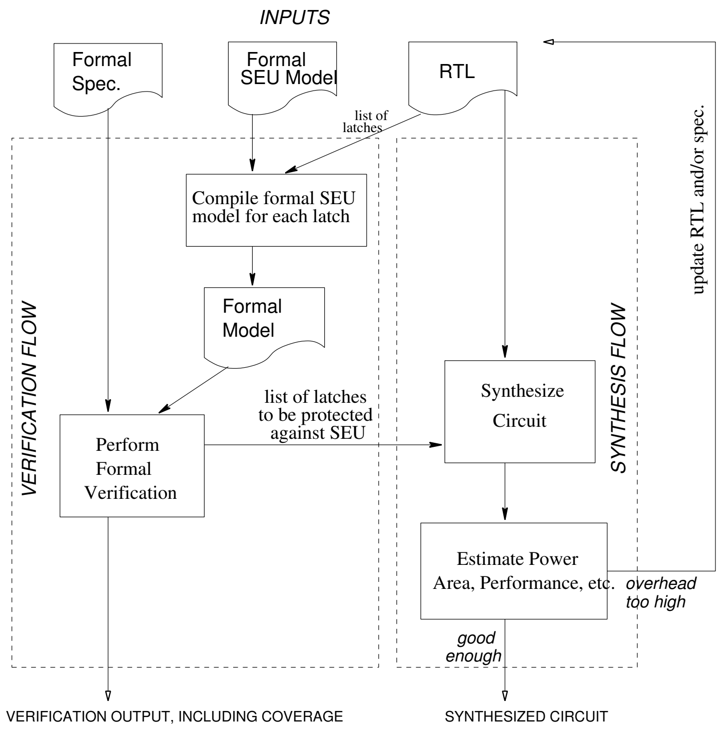
INPUTS
Formal Spec.
Formal SEU Model
RTL
list of latches
update RTL and/or spec.
Compile formal SEU model for each latch
Formal Model
VERIFICATION FLOW
SYNTHESIS FLOW
list of latches to be protected against SEU
Perform Formal Verification
Synthesize Circuit
Estimate Power Area, Performance, etc.
overhead too high
good enough
VERIFICATION OUTPUT, INCLUDING COVERAGE
SYNTHESIZED CIRCUIT

Figure 1. Flow of verification-guided soft error resilience

circuit, a formal specification (possibly comprising many properties/assertions), and a formal model of how the circuit reliability problem affects its system-level behavior. We elaborate on the latter point, for SEUs, in Section 3.2. The formal SEU model is automatically compiled into n finite-state machines, each one corresponding to the effect of an SEU on a latch (state variable).

The RTL is translated into a formal model of the circuit, either manually or using automated tools. For the work in this paper, this is a description in the input language of a model checker. (The model checker Cadence SMV includes an automated translator from Verilog to its input format.) However, in general it would depend on the formal verification (FV) tool that is used. Then, for each latch v_i, the corresponding formal SEU model is composed with the formal circuit model, to obtain the *combined model* for the FV tool for v_i. The combined model is fed to the FV tool along with the formal specification.

A total of $n + 1$ runs of the FV tool are performed: one for each of n latches, as well as a single run to verify that the formal specification holds for the original circuit model in the absence of an SEU. If the combined model for latch v_i fails to satisfy the formal specification, we conclude that v_i must be protected against an SEU; otherwise, not. Thus, the output of these n runs is a list of latches that must be protected against SEUs.

The RTL is then synthesized, using the appropriate cell libraries for the list of latches that need to be protected. Power, performance, area, and other circuit parameters are estimated. If the overhead of circuit protection is acceptable, the process stops. Otherwise, the designer must adjust the circuit and/or the specification to meet design objectives.

Although Figure 1 illustrates our approach for SEUs, the flow is likely to be largely unchanged for dealing with other kinds of errors. It is independent of the formal verification technique used, and involves few changes to existing synthesis and verification flows. In fact, as we note in Section 3.3, it can be viewed as a way to compute mutation-based coverage metrics for verification, so it can even augment existing verification flows.

Note further that the result of this process is highly dependent on the formal specification. A specification that places no constraints on the design will generate results indicating that all latches can be left unprotected! Thus, the results of our approach have a dual use: one of computing the coverage of a formal specification. If all (or almost all) latches can be left unprotected, it indicates that we might need to strengthen the specification. We discuss this point further in Section 3.3.

3.2. Formal Model

As is standard in formal verification, a sequential circuit is formally modeled as a triple (V, δ, S_0), where V is the set of Boolean state variables (latches) $\{v_1, v_2, \ldots, v_n\}$, δ is the transition relation of the system defining how the system evolves over time, and S_0 is a Boolean formula over V denoting the set of initial states in which the circuit can begin operation. The transition relation specifies, for each state variable v_i, how its value in the next cycle v_i' is obtained. Note that since δ is a relation, not just a function, non-determinism in the system can be easily modeled. Formally, there is a next-state assignment $v_i' := f_i(V)$ for each i, where f_i is a set-valued function.

Given an RTL-level circuit description, such as in Verilog, we manually create a formal model of it as given above. Timing-related details in the RTL are modeled using non-determinism, so that the resulting formal model exhibits a superset of the actual system behaviors. Any verification performed on the formal model will then be conservative.

In the SEU fault model, there is a single bit flip during an arbitrary cycle of circuit operation. Figure 2 shows the formal model of an SEU in latch v_i.

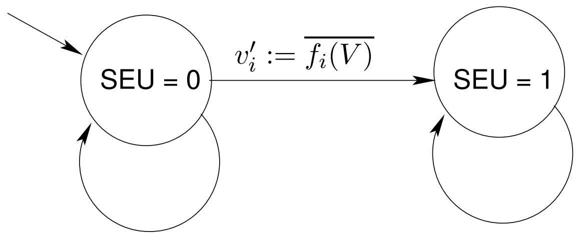

Figure 2. Formal model of impact of an SEU on Boolean state variable v_i

This model comprises a state variable SEU that records whether an SEU has occurred. If SEU is 0, the model non-deterministically chooses whether to stay in the same state, or to flip the value of latch v_i in the next cycle as shown by the label on the transition $v_i' := \overline{f_i(V)}$. The use of *non-determinism* allows us to check in a single model checking run the impact of a soft error in v_i at *an arbitrary cycle* of operation. Thus, the two-state automaton shown in Figure 2 models an SEU in v_i.

3.3. Relation to Coverage Metrics

Several coverage metrics have been proposed for formal verification [10, 6]. The main ones, some of which are in industrial use, are based on mutating the circuit model and checking whether the specification continues to be satisfied. If yes, then the specification is not exhaustive enough, and the verification engineer must extend it. The intuition is that an exhaustive formal specification should closely characterize the set of correct circuit behaviors.

Consider our formal model of an SEU in latch v_i, shown in Figure 2. An SEU is a form of mutation, in fact very similar to those considered in the literature [10, 6]. Thus, if the list of latches to be protected against an SEU (indicated in Figure 1 as an input to synthesis) is empty, it is cause for suspecting the coverage of the specification.

Thus, our approach is not only useful for analyzing the impact of soft errors, but the results can also be re-used for computing coverage metrics. In fact, extending our approach beyond soft errors might even help define and compute new kinds of coverage metrics for formal verification.

4. Case Study: SpaceWire

Our main case study, to date, is a third-party Verilog implementation of a node in the SpaceWire network. *SpaceWire* [8] is a network for space applications composed of nodes and routers interconnected through bi-directional high speed data links. According to the SpaceWire website hosted by the ESA, it has been used in missions of the ESA as well as space agencies NASA and JAXA.

The SpaceWire standard [8] describes 6 protocol levels – physical, signal, character, exchange, packet, and network. In this paper, we are concerned with the exchange level that defines the protocol for link initialization, flow control, and link error detection and recovery (similar to the more widely known Transmission Control Protocol, TCP). We downloaded a specific Verilog implementation of a SpaceWire end node from opencores.org [2] which was not written by our group. The Verilog was manually translated into the input language for the Cadence SMV model checker. English language specifications from the standards document [8] were translated into formal specifications in linear temporal logic and inserted into the SMV file as assertions to be checked. The Verilog (and the corresponding SMV model) had to be fixed in a few places as a result of our initial experiments to formally verify it. All results discussed below are with respect to this fixed SMV model.

Overview and Model

For purposes of reasoning about the exchange layer control protocol, a SpaceWire end node comprises three modules: a transmitter (TX), a receiver (RX), and a state machine that sends control signals to them (FSM). Generating a SMV model from Verilog involved straightforward transliteration for the most part, retaining the control structure, and only abstracting away some data and timing. The end node includes logic for parity error detection and correction on the data (which can be used for communication channel errors as well as SEUs), so it is the control which is left unprotected and is of particular interest to our analysis of SEUs.

We briefly describe below the operation of the FSM, TX, and RX modules, indicating where state was abstracted away in going to a SMV model. Further details may be found in the standards document [8].

The FSM controls the overall operation of the end node. Its operation is shown in Figure 3. The sequence of ErrorReset, ErrorWait, and Ready provides a mechanism of initializing the SpaceWire node, either coming from a whole system reset or triggered by an error. During this sequence of operation, RX is enabled to receive, but TX is prohibited from sending. In the Started state, TX can send NULL signals to the other end, to establish a connection. Next, the FSM enters the Connecting state where TX is enabled to send flow control tokens (FCTs). When RX receives FCTs, it indicates that the other end has space in its receive buffer for data. The Run state is the state for normal operation where packets flow freely in both directions across the link. The node remains in the Run state until an error occurs or until the link is disabled.

The end nodes communicate over a channel that was modeled in SMV to be capable of dropping or creating parity errors in both control and data packets. (Appropriate "fairness" constraints [7] were imposed on the channel to ensure that a packet would eventually get to its destination, even if dropped several times.)

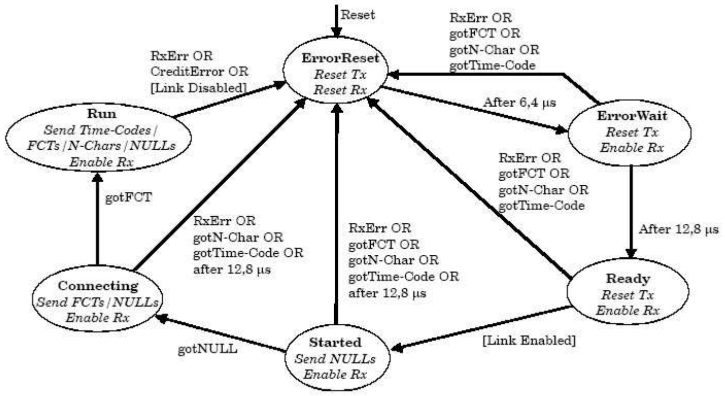

Figure 3. Operation of control finite-state machine in a SpaceWire end node. Reproduced from page 60 of [8].

The transmitter TX is responsible for encoding data (abstracted away in SMV) and transmitting it across the link. Packets are sent according to the following priority, from highest to lowest - Time-Code (system time information), FCT, N-Char (normal characters including data, EOP and EEP) and Null. If there is nothing to send, the transmitter sends NULL to maintain an active link. The transmitter also keeps a credit count of the number of characters that it has been given permission to send. The credit count roughly indicates the space of the opposite receiver buffer.

The receiver RX is responsible for buffering data and passing it on to the host system (abstracted away). It is also responsible for detecting disconnect errors, parity errors, escape errors and credit errors, and reporting these errors to the FSM. When an FCT is received, RX informs TX so that TX can update its credit count accordingly. RX also keeps an outstanding count of the number of characters that it expects to receive.

Formal Specifications

We wrote a total of 39 SMV assertions in linear temporal logic, each corresponding to an English-language specification in the standards document [8].

Table 2 lists representative assertions. Specifications we wrote fall into five categories, with each category represented in the table. The first set of specifications (row nos. 1 and 2) is on the FSM operation, indicating how and when the system can move between FSM states, as shown in Figure 3. The second is on the interaction between FSM, TX, and RX, exemplified by row 3 in the table that deals with error handling.[1] The next two sets, exemplified by rows 4

[1]Note that row 3 refers to both internal FSM state and the inputs it receives from TX and RX, ending in _i.

No.	Reference in [8]	Assertion
1	Sec. 8.5.2.2(b)	*LTL:* $\mathbf{G}(\texttt{FSM.state} = \texttt{ErrorReset} \implies \mathbf{X}(\texttt{RX.stateRX} = \texttt{RXRESET} \wedge \texttt{TX.state} = \texttt{Reset}))$ *English:* In the ErrorReset state the Transmitter and Receiver shall both be reset.
2	Sec. 8.5.2.5 (e)	*LTL:* $\mathbf{G}((\texttt{FSM.state} = \texttt{Started} \wedge \texttt{FSM.gotNULL\_i} \wedge \mathbf{X}(\texttt{FSM.state} \neq \texttt{ErrorReset})) \implies \mathbf{X}(\texttt{FSM.state} = \texttt{Connecting}))$ *English:* The state machine shall move on into the Connecting state from the Started state, if RX indicates that a NULL was received and no other condition forces the state machine to go back to the ErrorReset state.
3	Sec. 8.5.2.3 (e)	*LTL:* $\mathbf{G}(\texttt{FSM.state} = \texttt{ErrorWait} \wedge (\texttt{FSM.Lnk\_dsc\_i} \vee \texttt{FSM.HASgotNULL} \wedge (\texttt{FSM.err\_par\_i} \vee \texttt{FSM.err\_esc\_i} \vee \texttt{FSM.gotFCT\_i} \vee \texttt{FSM.gotNchar\_i} \vee \texttt{FSM.gotTime\_i})) \implies \mathbf{X}\ (\texttt{FSM.state} = \texttt{ErrorReset}))$ *English:* If, while in the ErrorWait state, a disconnection error is detected, or if after the gotNULL condition is set (HASgotNULL), a parity error or escape error or any character other than a NULL is received, then the state machine shall move back to the ErrorReset state.
4	Sec. 8.5.2.6 (c) Sec. 8.4.2	*LTL:* $\mathbf{G}(\texttt{TX.state} = \texttt{Send\_Null} \wedge \texttt{TX.state\_connecting} \wedge \texttt{TX.nedsFCT} \wedge \neg\texttt{TX.TXReset} \implies \mathbf{X}\ \texttt{TX.state} = \texttt{Send\_FCT})$ *English:* If TX is enabled to send NULLs, FSM is in the Connecting state, and TX is not getting reset, it will send out FCT upon a request to send FCT (nedsFCT) from the Receiver.
5	Sec. 8.4.4 Sec. 8.8	*LTL:* $\mathbf{G}(\texttt{RX.C\_Send\_FCT\_i} \wedge \texttt{RX.osd\_cnt} < 49 \wedge \neg\texttt{RX.reset} \wedge \neg\texttt{RX.Lnk\_dsc\_o} \implies \mathbf{X}\texttt{RX.osd\_cnt} = \texttt{RX.n\_osd\_cnt1})$ *English:* RX's outstanding counter (`osd_cnt`) represents the number of N-Chars that it expects to receive. An outgoing FCT represents a request for 8 more N-Chars from the opposite side. Hence, if the current `osd_cnt` indicates enough space left, and the system is not getting reset, and the link between the two nodes is not disconnected, then `osd_cnt` should update to `n_osd_cnt1` which is an increment by 8.
6	Sec. 8.7, Table 8, Figure 23	*LTL:* $\neg\mathbf{F}((\texttt{FSM1.state} = \texttt{ErrorReset} \wedge \texttt{FSM2.state} = \texttt{ErrorReset}) \wedge ((\texttt{FSM2.state} \in \{\texttt{ErrorReset}, \texttt{ErrorWait}, \texttt{Ready}\})\mathbf{U}((\texttt{FSM2.state} \in \{\texttt{ErrorReset}, \texttt{ErrorWait}, \texttt{Ready}\}) \wedge \texttt{FSM1.state} = \texttt{Connecting})))$ *English:* The following condition should never occur: With both nodes starting from the Error-Reset state, Node 1's FSM should not move into the Connecting state if Node 2's FSM is still in {ErrorReset, ErrorWait, Ready}. (A symmetric condition holds with 1 and 2 switched.)

Table 2. Selected formal specifications. *LTL* indicates a specification in linear temporal logic.

and 5, are on transmitter and receiver operation. The final set of specifications (e.g., row 6) are on the end-to-end communication between two nodes in the SpaceWire network.

Our formal specification is as comprehensive as the corresponding English language specifications in the standards documents. Moreover, note that we have assertions that place safety conditions on the system's behavior as well as state progress conditions indicating that the system is "doing what it should".

Results

An SMV model 987 lines long (including assertions and fairness constraints) was generated from 1393 lines of Verilog. The synthesized circuit contains 130 latches. Using the formal specifications created from the standards document, we found that all but 28 of the latches could be left unprotected. These latches correspond to 14 state variables in the SMV model. (Some variables, for example, corresponding to counters and FSM state, generate multiple latches in the synthesized circuit.)

An example of a state variable that must be protected is `FSM.state`. On a bit flip, this can arbitrarily change the state of the FSM, leading to failure of many assertions, including row 6 in Table 2.

An example of a state variable that need not be protected is `FSM.HASgotBit` which is an internal FSM flag that indicates that the end node has received a bit. This flag is used in the FSM logic for state transitions and error handling, so it was initially somewhat surprising to us that an SEU in it allowed the end node to continue to satisfy all its assertions. It appears that the correlations between values of signals in the error handling logic are responsible for this inherent robustness to an SEU in `FSM.HASgotBit`.

Our experiments were performed using the Cadence SMV model checker [1]. For scalability, all categories of specifications except for the end-to-end assertions were verified on a model of a single node communicating with a channel that could generate any message (a conservative check). The total time for our SMV runs for these specifications was 27 minutes with a maximum of 4.5 minutes for a single run (SMV caches results, thus optimizing overall runtime). The end-to-end assertions were verified on a model comprising two nodes communicating over a lossy channel. This experiment took much longer due to the larger state space – 166 minutes with a maximum time of 109 minutes for a single run.

Synopsys Design Compiler was used to generate the final circuit. Latches that did not map to any state variable in Verilog were protected or not based on a structural dependency analysis. Three power consumption numbers were then estimated: for the synthesized circuit without any SEU protection at all, with the BISER protection [20] for all latches, and with BISER protection using our verification-guided classification. The following results were obtained:

Technique	Power(mW)	Overhead
No SEU protection	1.160	–
SEU protection for all latches	1.832	57.9%
Verification-guided SEU protection	1.314	13.3%

Thus, using a verification-guided approach one can obtain a 4.35 X reduction in power overhead of protecting from SEUs using the BISER technique. We believe similar results can be obtained for other SEU protection methods as well, since the fraction of latches to be protected is small.

5. Conclusions and Future Work

We have proposed a verification-guided approach to estimating and reducing the overheads of circuit mechanisms for soft error resilience. Our approach has been demonstrated on a real case study of a third-party Verilog implementation of a component of the ESA SpaceWire network with specifications covering the specified behavior in the standards document [8]. The resulting power savings demonstrate the utility of our approach.

This paper has only taken a first step. Scalability of model checking (and formal verification, in general) is a concern, which we plan to address by using a modular (compositional) approach to verification. Our approach can also be combined with complementary methods such as random fault injection. Finally, our work has direct connections to the problem of computing coverage metrics for formal verification, which we plan to explore further.

Acknowledgments. We are grateful to K. McMillan for help with Cadence SMV. T. Loo and L. Wang created an initial version of the SpaceWire SMV model, and S. Chu helped with synthesis and power estimation. S. Malik, T. Austin, V. Bertacco, and K. Sakallah provided valuable feedback. This research was supported in part by the MARCO Gigascale Systems Research Center.

References

[1] Cadence SMV model checker. http://www.kenmcmil.com/smv.html.

[2] SpaceWire Verilog. http://www.opencores.org/projects.cgi/web/spacewire/overview, July 2005.

[3] IEEE P1850 - standard for PSL - property specification language. http://www.eda.org/ieee-1850/, URL circa Sep.'06.

[4] H. Asadi and M. B. Tahoori. Soft error modeling and protection for sequential elements. In *Proc. of the IEEE Intl. Symp. On Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 463–471, October 2005.

[5] R. C. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficiency of error correction. In *Proc. IEDM*, pages 329–332, 2002.

[6] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. In *Proc. CHARME*, LNCS 2860, pages 111–125, 2003.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[8] European Cooperation for Space Standardization. Space engineering – SpaceWire – links, nodes, routers, and networks (draft ECSS-E-50-12A). http://www.spacewire.esa.int/tech/spacewire/standards/, November 2002.

[9] K. Goswami, R. Iyer, and L. Young. DEPEND: a simulation-based environment for system-level dependability analysis. *IEEE Trans. Computers*, pages 60–74, Jan. 1997.

[10] Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conference (DAC)*, pages 300–305, 1999.

[11] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, pages 75–82, April 1997.

[12] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. on Computers*, 44(2):248–260, 1995.

[13] U. Krautz, M. Pflanz, C. Jacobi, H. W. Tast, K. Weber, and H. T. Vierhaus. Evaluating coverage of error detection logic for soft errors using formal methods. In *Proc. DATE 2006*, pages 176–181, 2006.

[14] S. Krishnaswamy, G. F. Viamontes, I. L. Markov, and J. P. Hayes. Accurate reliablity evaluation and enhancement via probabilistic transfer matrices. In *Proc. Design Automation and Test in Europe (DATE)*, pages 282–287, 2005.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1992.

[16] S. Mitra, T. Karnik, N. Seifert, and M. Zhang. Logic soft errors in sub-65nm technologies design and CAD challenges. In *Design Automation Conference (DAC)*, pages 2–4, 2005.

[17] M. Nicolaidis. Design for soft error mitigation. *IEEE Trans. Device and Matl. Reliability*, 5(3):405–418, Sept. 2005.

[18] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. Int'l Symp. Microarchitecture (MICRO)*, pages 29–40, 2003.

[19] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proc. Int'l Conf. Dependable Systems and Networks (DSN)*, pages 61–70. IEEE Press, 2004.

[20] M. Zhang, S. Mitra, T. M. Mak, N. Seifert, Q. Shi, K. Kim, N. Shanbhag, N. Wang, and S. Patel. Sequential element design with built-in soft error resilience. *IEEE Transactions on VLSI*, Dec. 2006.