Hardware Dependability in the Presence of Soft Errors

Ashish Darbari and Bashir M. Al Hashimi
School of Electronics and Computer Science
University of Southampton, SO17 1BJ, England
ad06v,bmah@ecs.soton.ac.uk

Abstract

Using formal verification for designing hardware designs free from logic design bugs has been an active area of research since the last 15 years. Technology has matured and we have a choice of formal tools such as model checkers, equivalence checkers, and a range of theorem provers. Hardware reliability and fault tolerance has been studied for a long time as well, and some good solutions in the form of redundancy are available for making hardware resilient against faults. However, understanding the impact of a particular kind of fault known as a single-event-upset (SEU) or a transient fault especially in the context of low-power design is not well understood, and therefore achieving adequate tolerance for low-power processors against SEUs is still very much an open problem. A significant bottle-neck in this has been the traditional fault injection methodology whereby the impact of a fault is analysed whilst a processor is running a specific binary program image. Thus the true impact of the fault is limited by the shadow of the particular program. Another key problem has been the modification of the original design to incorporate fault injection hardware. Thus, the design being checked for faults is different from the original design. In this paper we report on our experiences on studying transient fault injection on a 32 bit multi-cycle RISC processor using the formal specification and verification framework of Symbolic Trajectory Evaluation (STE). Our approach offers the benefit of studying fault injection by not modifying the original design and doing it in a program independent way. The vulnerability of the processor is assessed in terms of its architecural features, which is possible due to symbolic model checking.

Keywords: Transient Fault Injection, Reliable Processors, Model Checking

1. INTRODUCTION

A transient fault or a single-event-upset (SEU) [1] poses a formidable challenge due to the multi-dimensional nature of the fault. Two dimensions identified earlier [2] have been $spatial$, and $temporal$. We have identified a third one – the $data$ dimension. The spatial dimension means, a transient can hit any component of a design, which means we need to analyse all components and all paths of a design. The second is temporal - a transient can hit at any time, thus one needs to cover all temporal possibilities during the running of a design. The third dimension - data means that a transient can hit a circuit whilst it is processing some data. This is especially true of processors with their own memory and they execute a program when a transient hits it. So one needs to analyse the effect of a transient fault on a design taking into account that the design could be executing any application program (data). A significant bottle-neck in this has been the traditional fault injection methodology [3–5] whereby the impact of a fault is analysed whilst a processor is running a specific binary program image. Thus the true impact of the fault is limited by the shadow of the particular program. Another key problem has been the modification of the original design to incorporate fault injection hardware. Thus, the design being checked for faults is different from the original design. The problem of SEUs is better addressed at the level of server grade processors and desktop processors, since one can incorporate different redundancy techniques. However, for a low-power processor that runs on battery, incorporating redundancy on the same lines as for desktop processors does not make sense from energy point of view. Therefore, one needs to strike a fine balance between tolerance requirements and energy [6, 7], which can only be achieved by having a precise and comprehensive understanding about the impact of SEUs on simple, low-power designs.

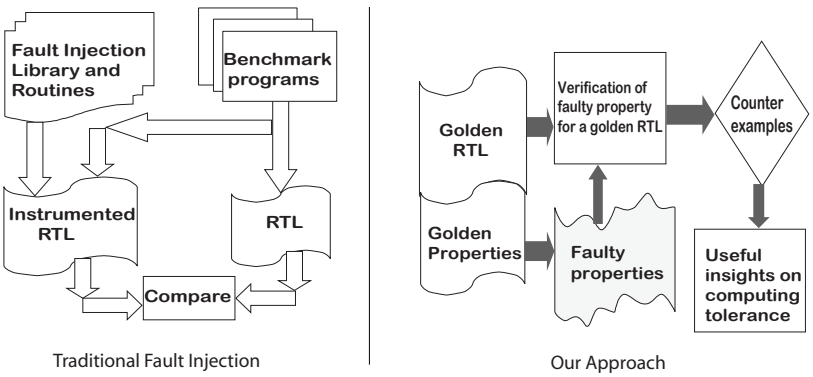

FIGURE 1: Traditional Fault Injection. Traditionally fault injection is carried out by comparing the behaviour of two different RTLs, one that has been instrumented to incorporate faults, and the other – golden RTL – which expresses the intention of the designer in a fault-free case. Both these RTLs are simulated whilst running specific binary program images. The comparison is usually done by comparing the output values of gates, registers, memory elements and so on. **Our Approach.** The golden properties are instrumented to capture transient faults, and are then verifed against the golden RTL. The outcome of this is a set of counter-examples that provide useful insights on how to obtain tolerance. The golden RTL and golden properties are obtained by iterative refinement of the given RTL, and an initial set of STE properties developed by the designer, through a formal verification step.

2. STE MODEL CHECKING - A BRIEF INTRODUCTION

We wanted to explore features in our property specification language that would enable us to write properties about complex hardware designs with ease, that could be used for model checking efficiently. STE has been used successfully in large-scale datapath verification [8] of microprocessors.

STE gains its strength by combining the ideas of $ternary\ modelling$ (using 0,1 and X) with $symbolic\ simulation$ (using symbolic variables) over $time$. The presence of Xs provides abstraction necessary to handle the complexity of verifying symbolic properties during model checking. These characteristics allow STE to perform property checking more efficiently than conventional model checking algorithms, which operate over more expressive logics like CTL [9]. Of course, there is a trade-off; because STE's logic is simple, it may take more properties to verify the same functionality than in CTL. On the other hand, although the logic of STE seems rather weak, its expressive power is greatly extended by implementing a *symbolic* ternary simulation algorithm [10].

Symbolic ternary simulation uses Boolean variables and propositional formulas to represent whole classes of data values on circuit nodes. The ternary value associated with each node is given by a symbolic data structure whose variables act as parameters. With this representation, STE can combine many (ternary) simulation runs—one for each assignment of values to the variables—into a single symbolic simulation run covering them all. The usual implementation uses BDDs [11], but other representations are possible.

The formulas representing values at different circuit nodes can have variables in common, so they can also record interdependencies among node values. Symbolic values therefore greatly increase the expressive power of the limited temporal logic of STE. For example, input-output relations can be extracted from a circuit by using symbolic simulation to derive formulas for the values on output nodes as functions of variables standing for arbitrary values on input nodes. These can then be checked against a specification in the form of some reference formulas provided by the verification engineer.

Specifications in STE, take the form of what are known as $symbolic\ trajectory\ formulas$. Formally, we define the syntax of formulas [8, 12] as follows:

Definition 1. *Syntax of STE formulas*

$$
\begin{array}{rcll}
f & \triangleq & n \text{ is } 0 & \text{- node } n \text{ has value 0} \\
 & | & n \text{ is } 1 & \text{- node } n \text{ has value 1} \\
 & | & f_1 \text{ and } f_2 & \text{- conjunction of formulas} \\
 & | & f \text{ when } G & \text{- } f \text{ is asserted only when } G \text{ is true} \\
 & | & \mathsf{N}f & \text{- } f \text{ holds in the next time step}
\end{array}
$$

where f_1 and f_2 range over formulas, $n \in string$ ranges over the nodes of the circuit, and G is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a $guard$. The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent inter-dependencies among node values. For example, we can associate an arbitrary propositional formula G with a node using the construct 'n is G' defined by

$$n \text{ is } G \triangleq ((n \text{ is } 1) \text{ when } G) \text{ and } ((n \text{ is } 0) \text{ when } \neg G)$$

For implementation efficiency Boolean expressions are denoted by binary decision diagrams [13] (BDDs). Verification takes place by testing the validity of an $assertion$ or property against a given $model$. Circuit models in STE are defined on three-valued states, and are constructed on-the-fly during simulation, from the FSM (.exe) representation of the circuit. A property is of the form $(A \Rightarrow C)$, where both A and C are trajectory formulas. Intuitively, the antecedent A provides the stimuli to the circuit, and the consequent C expresses what the designer expects to see. In practice, every successful STE run i.e., a run that returns the value True, is a theorem that holds for all the Boolean variables mentioned in the property. However, when the outcome of an STE model checking run is a counter-example, it is an indication of a bug in the hardware, and in the context of STE, it means that if we can come up with a satisfying assignment of Boolean values True (logic 1) and False (logic 0) to the Boolean variables in the counter-example, one can explicitly reveal the trace (consiting of 0s and 1s) that would be responsible for the bug. Usually there is more than one way to satisfy the counter-example, and this means that in one symbolic model checking run, we can succintly capture all the possible traces.

STE model checking also encounters state space explosion problem [14], but in practice there is a deductive layer [8, 15–17] that wraps the model checking engine. This layer is built from a set of sound inference rules within the logic of STE. These rules are used [16], often repeatedly to decompose the given original property into smaller properties, and once the smaller properties are checked against the given model, the overall property verification run is composed back from the inference rules. We have made good use of the STE inference rules in our task, and have found that they greatly enhance the overall verification task both in terms of the speed of execution, and the time taken by the model checker to perform checking.

3. OUR APPROACH

We identified a 32-bit multi-cycle RISC processor to conduct our study of fault injection, using the formal specification and verification framework of Symbolic Trajectory Evaluation (STE). Our approach [18] offers the benefit of studying fault injection by not modifying the original design and doing it in a program independent way. We express faults in properties that are written using the syntax of STE.

We first obtain a golden RTL by verifying fault-free properties against the given RTL. Usually, the RTL may have logic design bugs and/or the properties themselves may have errors. In either case, the verification step would produce counter-examples. We use these counter-examples to refine the RTL and/or properties accordingly and we repeat this process till we stop getting anymore counter-examples. The RTL obtained at this stage is called a golden RTL and the resulting set of properties - golden set of properties. A high-level view is presented in Figure 1.

Once we obtain the golden pair of RTL and properties, we modify the golden set of properties, to capture the transient faults we would like to investigate. The modified set of properties with the fault,

is then verified against the golden RTL using STE model checking. When we verify a property-with-a-fault against the golden RTL, the fault itself is expressed by modifying A, whilst the expected behaviour of the circuit expressed in C remains unchanged. The outcome of this model checking run is a counter-example — a Boolean constraint that needs to be satisfied in order to make the property-with-a-fault work exactly as the property-without-the fault. This Boolean constraint denoted by a BDD, is an abstraction of all the possible traces involving 0s and 1s, which if fixed would make the circuit work correctly, for the faulty property. Since the counter-example is made from Boolean variables placed on the circuit nodes in the antecedent of the property, it gives architectural inter-dependency amongst the various circuit nodes in the presence of a fault. By using generic Boolean symbols (variables) and conducting symbolic simulation, instead of simulaton over 0s and 1s, one obtains a data independent way of analysing the impact of faults — a strategy we believe could be useful to the designers and testing community.

4. RESULTS

In this section we summarise, our findings on the fault injection in different parts of the processor shown in Figure 2. An important observation in our experiments is that many times in spite of a fault, its effect may not necessarily propagate to those points in the circuit that we are interested in observing. In this case we say that the fault has been $masked$. In our approach, since we construct a fault-free property and a faulty property, if the effect of a fault (expressed in the faulty property) is not propagated to the nodes in the consequent of the property then the fault's effect is said to be masked.

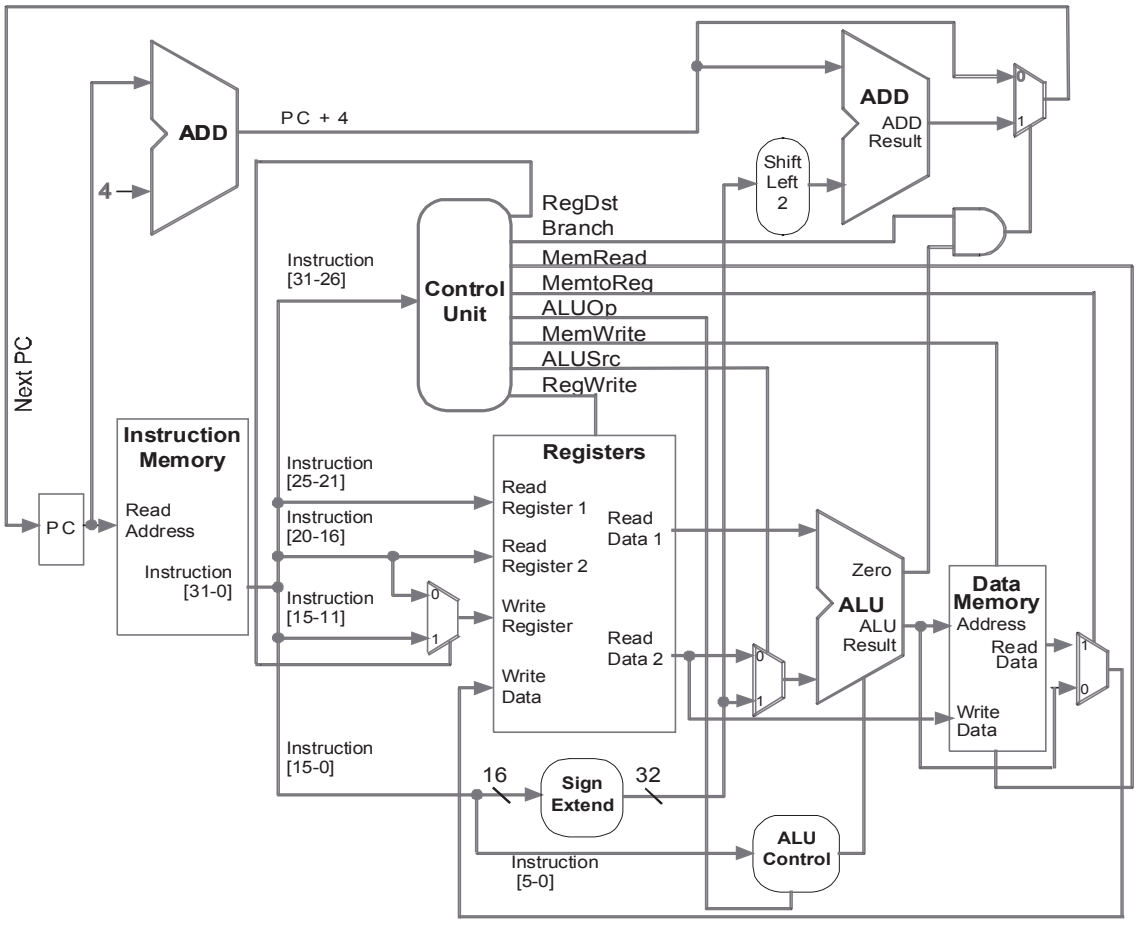

FIGURE 2: CPU datapath showing functional blocks (adapted from [19]).

Fetch

Properties needed to check the functionality of the fetch unit.

F_1 PC gets incremented correctly by 4 or, PC gets the Branch address.
F_2 The instruction memory works correctly.

Faults in the Fetch Unit		
Property	*Faulty Component*	*Fault Impact*
F_1	PC	Masked if the branch instruction is in the pipeline.
	Branch Address	Masked if the next instruction is not a branch one.
F_2	Instr Mem	Masked if faulty location is not read from or else over-written with new data.

Decode

Properties required to verify that the decode works correctly.

D_1 Verifying that the ReadData1 port of the register bank always gets the data from the register indexed by the address placed on ReadRegister1 port of the bank.
D_2 Verifying that the ReadData2 port of the register bank always gets the data from the register indexed by the address placed on the ReadRegister2 port of the bank.
D_3 WriteAddress port of the register bank takes on an address value selected by the RegDst.
D_4 WriteData port of the register bank takes on the data value selected by MemtoReg.
D_5 Registers get updated correctly on a write.
D_6 Sign-extension is correct.

Faults in the Decode Unit		
Property	*Faulty Component*	*Fault Impact*
D_1	Register Bank	Masked if faulty location is not read from or else over-written with new data.
	Instr[25:21]	Less if data is not read from the faulty combination of address location.
D_2	Register Bank	similar to D_1
	Instr[20:16]	
D_3	Instr[20:16]	Masked if RegDst is enabled.
	Instr[15:11]	Masked if RegDst is disbaled.
	RegDst	Masked only when Instr[20:16]=Instr[15:11]
D_4	ReadData port of DataMemory	Masked if MemtoReg is disabled.
	ALUResult port of the ALU	Masked if MemtoReg is enabled.
	MemtoReg	Masked only when ALUResult[31:0]=ReadData[31:0].
D_5	Register bank	Masked if RegWrite is enabled.
	MemtoReg	None
	RegDst	None
	WriteRegister Address of the register Bank	None
	ALUResult port of the ALU	None
	ReadData port of the DataMemory	None
D_6	Instr[15:0]	Severe, can be fixed only if the error in Instr[15:0] is corrected.

Control

The following control signals take on the appropriate values

C_1 RegDst $= \neg$Instr[31] $\wedge \neg$Instr[30] $\wedge \neg$Instr[29] $\wedge \neg$Instr[28] $\wedge \neg$Instr[27] $\wedge \neg$Instr[26]
C_2 Lw $=$ Instr[31] $\wedge \neg$Instr[30] $\wedge \neg$Instr[29] $\wedge \neg$Instr[28] $\wedge$ Instr[27] $\wedge$ Instr[26]

C_3 Sw = Instr[31] $\wedge$ $\neg$Instr[30] $\wedge$ Instr[29] $\wedge$ $\neg$Instr[28] $\wedge$ Instr[27] $\wedge$ Instr[26]
C_4 Beq = $\neg$Instr[31] $\wedge$ $\neg$Instr[30] $\wedge$ $\neg$Instr[29] $\wedge$ Instr[28] $\wedge$ $\neg$Instr[27] $\wedge$ $\neg$Instr[26]
C_5 ALUSrc = lw $\vee$ sw
C_6 RegWrite = lw $\vee$ Rformat
C_7 MemRead = lw
C_8 MemWrite = sw
C_9 Branch = Branch
C_{10} ALUOp1 = Rformat
C_{11} ALUOp0 = Branch

For properties $C_1 - C_4$, a fault in a single bit can be masked by coming up with an assignment of values - logic 0 or 1, to the non-faulty bits, such that the overall value of the control pin for example RegDst in case of C_1 stays as logic 0. Thus the control signals in $C_1 - C_4$ would continue to take on correct values under a subset of operating conditions. This is due to the fact that all the control signals in $C_1 - C_4$ are logic AND of the inputs.

For properties $C_5 - C_6$, for a fault in one of the input bits, we can mitigate the effect by asserting a logic 1 at the other non-faulty bit. This is due to the fact that ALUSrc and RegWrite are logic OR of the inputs.

For properties $C_7 - C_{11}$ the effect of fault is severe, since there is no set of operating conditions which can alleviate the effect of the fault, unless it is fixed at the source.

Execute

Properties required to verify that the Execute works correctly are:

E_1 ReadData1 output from the Decode stage is copied onto the Ainput.
E_2 ReadData2 output from the Decode stage is copied onto the Binput.
E_3 Zero gets a logic 1 on ALUResult being 0.
E_4 Branch Adder works corectly.
E_5 The output of the branch adder gets transferred to AddResult of the Fetch unit.
E_6 ALU operations - Add, Sub, AND, OR and NOT work correctly.

Faults in the Execute Unit		
Property	*Faulty Component*	*Fault Impact*
E_1	ReadData1 port of Register Bank	Severe, no mitigation possible until the fault is fixed at the source.
E_2	ReadData2 port of Register Bank	Masked if ALUSrc is enabled.
	Extend	Masked if ALUSrc is disabled.
	ALUSrc	Masked if ReadData2[31:0]=Extend[31:0]
E_3	ALUResult[31:0]	Masked if one of the bits of ALUResult[31:0] is a logic 1.
E_4	Extend	Severe, there is no set of operating conditions that can offset the impact of fault in either of these inputs.
	PC	
E_5	AddResult	Same as E_4.
E_6	Ainput[31:0]	The underlying principle is very similar to the case of the Control unit as explained earlier.
	Binput[31:0]	
	ALUctl[2:0]	

Data Memory

The case of Data Memory is exactly similar to the Instruction Memory for our case study since the two memories have been designed identical to each other.

5. RELATED WORK

Hazelhurst et al. [20] were the first to note that one can use formal property checking to specify and verify the effects of stuck-at faults. They proposed using STE to accomplish this, and for this reason their work is most similar to ours. But there are notable differences between our approach and the one Hazelhurst et al. proposed. In their method, faults were modelled by decomposing the finite-state machine representation into smaller components, and each smaller model is modified to incorporate faults that they investigated. The authors deal with three different FSMs, one that represents the RTL under investigation, one that injects the faults, and the third that monitors the effect of faults. However, manipulating low-level FSMs is *compute intensive*, is *less abstract* and *less intuitive*, and is therefore not desirable. Another shortcoming of their approach was that they hand-converted the VHDL code into an FSM representation, which obviously limits the applicability of their approach.

Leveugle [21] was the first to show how IBM's Property Specification Language (PSL) based property checking can be used together with controlled mutations, for transient fault injection. The basic idea is that the RTL that has to be investigated for faults is instrumented with extra hardware to model faulty behaviour. The author uses mutants [22] for this precise instrumentation technique. Much care was taken in developing these mutants, since having too many of them, will result in state-space explosion. The author noted that using PSL rules in practice was difficult, and often the rules were long and complex. In addition the tools that are used for model checking were not able to say much in case of a contradiction, besides saying "over-constrained". Also the tool Leveugle used was unsound, meaning the tool okayed the properties whilst there were known counter-examples.

Krautz et al. [23] also used PSL based property checking to evaluate the effect of transients on the coverage of error-detection logic. The basic approach used by them is very similar to Leveugle — instrument the VHDL with faulty hardware, and then use PSL property checking to evaluate the behaviour of the RTL while it executes specific benchmark programs. The authors prefer dealing with two separate RTLs — one the golden RTL which is fault-free and other the fault injected model. Tolerance in the form of error-detection-and-correction is encoded in the RTL and PSL is used to express properties that will check if the tolerance is adequate to mask the faults.

6. CONCLUSION

Property checking and symbolic simulation based methodology provides a systematic method of conducting fault injection. Though this technique cannot provide tolerance to the effect of fault in all parts of the CPU, it provides insights into sets of operating conditions under which faults in certain components can be masked. This information would be useful to an architect for designing optimal redundant solutions for tolerance. For example if a processor is likely to execute a set of programs without many Branch instructions, it may not be necessary to provide redundancy against faults in the Branch side of the fetch unit. Many of the findings discovered as a result of our study seem intuitive, but it would be extremely difficult if not impossible to unravel these using a traditional fault injection approach. Due to symbolic simulation, exponential test space is cut down to linear, symbolic, verification cases. By using a property checking approach, one is able to express the spatio-data-temporal nature of the bit-flip effectively in the form of small number of properties. The use of STE as a specification language for properties turned out to be very useful since STE has a very simple syntax, although restricted when compared to PSL. STE also offers verification efficiency, the interested reader is referred to [12] for further details. Fault being captured in properties rather than in the RTL [2, 4, 21, 23] allows efficient model checking, and the RTL being checked for faults remains identical to the original RTL.

7. ACKNOWLEDGEMENT

This work was carried out as part of the EPSRC project (Grant No: EP/D057663/1) on "Reliable Low Power Embedded Computing System". The authors would like to thank Petru Eles and Zebo Peng at Linköping University, Sweden, and Peter Harrod, David Flynn, and Daryl Bradley of ARM, Cambridge, UK for their feedback.

REFERENCES

[1] E. Normand, "Single Event Upset at Ground Level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, December 1996.

[2] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 172–183.

[3] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems," 1995.

[4] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Exploiting FPGA for Accelerating Fault Injection Experiments," in *IOLTW 2001*. IEEE Comp. Soc., 2001, p. 9.

[5] Mei-Chen Hsueh and Timothy K. Tsai and Ravishankar K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[6] D. Zhu, R. Melhem, and D. Mosse, "The Effects of Energy Management on Reliability in Real-Time Embedded Systems," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 35–40.

[7] A. Maheshwari, W. Burleson, and R. Tessier, "Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 3, pp. 299–311, March 2004.

[8] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification," *IEEE TCAD*, vol. 24, no. 9, pp. 1381–1405, 2005.

[9] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*, 1991, pp. 297–402.

[11] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[12] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Journal of FMSD*, vol. 6, no. 2, pp. 147–189, 1995.

[13] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[14] K.L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[15] S. Hazelhurst and C.-J. H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs," *IEEE Tran. on CAD of Integrated Circuits*, vol. 14, no. 4, pp. 413–422, 1995.

[16] A. Darbari, "Symmetry Reduction for STE Model Checking," in 6^{th} *FMCAD Conference*, 2006, pp. 97–105.

[17] A. Darbari, "Formalization and Execution of STE in HOL," in *Supplementary Proceedings of the 16th International Conference on Theorem Proving in Higher-Order Logics*, ser. LNCS 2758, D. Basin and B. Wolff, Eds. Springer, 2003.

[18] A. Darbari, B. A. Hashimi, P. Harrod, and D. Bradley, "A New Approach for Transient Fault Injection Using Symbolic Simulation," *14th IEEE International On-Line Testing Symposium (IOLTS)*, vol. 0, pp. 93–98, 2008.

[19] James O. Hamblen and Michael Furman, *Rapid Prototyping of Digital Systems: A Tutorial Approach*, 2nd ed. Springer, 2001.

[20] S. Hazelhurst and J. Arlat, "Specifying and Verifying Fault-Tolerant Systems," in *Proceedings of DCC'02*, M. Sheeran and T. Melham, Eds., 2002.

[21] R. Leveugle, "A New Approach for Early Dependability Evaluation Based on Formal Property Checking and Controlled Mutations," *IOLTS 2005*, vol. 00, pp. 260–265, 2005.

[22] Jeffrey A. Clark and Dhiraj K. Pradhan, "Fault Injection: A Method for Validating Computer-System Dependability," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[23] 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 *Proceedings of the DATE 2006 conference*, 2006, pp. 176–181.