RobuCheck: A Robustness Checker for Digital Circuits

Stefan Frehse Görschwin Fey André Sülflow Rolf Drechsler

Institute of Computer Science
28359 Bremen, Germany
{sfrehse,fey,suelflow,drechsle}@informatik.uni-bremen.de

***Abstract*—Continuously shrinking feature sizes cause an increasing vulnerability of digital circuits. Manufacturing failures and transient faults may tamper the functionality. Automated support is required to analyze the fault tolerance of circuits.**

In this paper, ROBUCHECK is presented - a design tool to analyze the fault tolerance of digital circuits. Engines based on simulation and formal methods are integrated to identify components that require additional fault protection. Consequently, an overall estimation of fault tolerance of the circuit is determined.

I. INTRODUCTION

The increasing soft error rate caused by continuously shrinking feature sizes demands for fault tolerance in digital circuits. Transient or permanent faults may tamper the functionality of the circuit. Precautions against soft errors are taken at different design levels, e.g. architectural level, algorithmic level, or layout level [21], [18].

Design tools are required to verify and analyze the fault tolerance of circuits. Various techniques have been developed. On the one hand there are simulation- or emulation-based methods and on the other hand there are formal methods.

Methods based on simulation (e.g. [17], [3]) are fast but cannot cover the complete input space and state space in combination with all potential faults in reasonable time. In contrast, formal methods (e.g. [8], [9], [15], [13]) are complete by proving the fault tolerance with respect to the whole input space. The integration of simulation-based methods and formal methods is very powerful. Simulation yields results even for very large systems where formal methods reach their limits. But in contrast, formal methods guarantee completeness and by this find corner case problems that would be missed by simulation. Previous work does not describe a tool that integrates the different approaches.

In this paper, the ROBUCHECK–framework to analyze the fault tolerance of circuits is presented. ROBUCHECK uses the underlying model of [9] to measure fault tolerance with respect to transient faults. The fault tolerance is checked for any part in the circuit, e.g. combinational logic and state elements. For this purpose, simulation and formal analysis are integrated in one framework. Moreover, the paper compares the formal engines in terms of time and space complexity, respectively.

The presented framework can be embedded into the classical design-flow of digital circuits as shown in Figure 1.

This work has been funded in part by DFG grants DR 287/19-1 and FE 797/5-1.

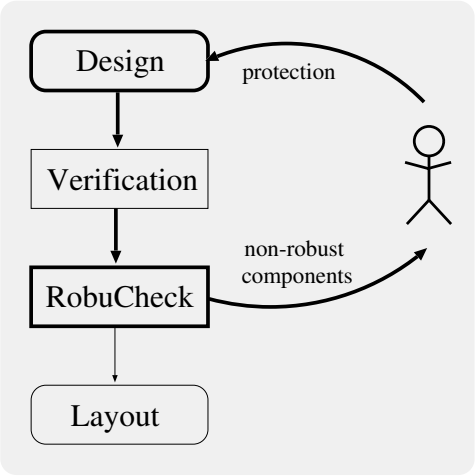


Fig. 1. Embedding in the design flow

When a *Register Transfer Level* (RTL) description is available ROBUCHECK is used to analyze the circuit. Given a circuit description, ROBUCHECK determines components that require additional protection. Now, a designer uses the provided information to improve the fault tolerance for non-robust components. By iterating with ROBUCHECK, the designer checks the implemented fault protection logic to achieve the expected level of fault tolerance. The iteration stops, if required level of fault tolerance is achieved.

ROBUCHECK integrates the previous work of [8], [9], [19], [11], [10] to a full framework for automatic analysis of fault tolerance. By this, additional methods for the classification of single components and the usage of application specific constraints are provided. Moreover, a visualization back-end supports the designer during further design steps to improve the fault tolerance.

The paper is structured as follows: The next section gives an overview over ROBUCHECK and introduces the model used during the analysis. Section III describes the underlying proof engines in more detail. Finally, some use cases illustrate the application of ROBUCHECK in Section IV. Section V presents conclusions.

II. MODELING AND SYSTEM OVERVIEW

The underlying model and analysis methods are introduced in Section II-A and Section II-B, respectively. Section II-C gives an overview of the whole system.

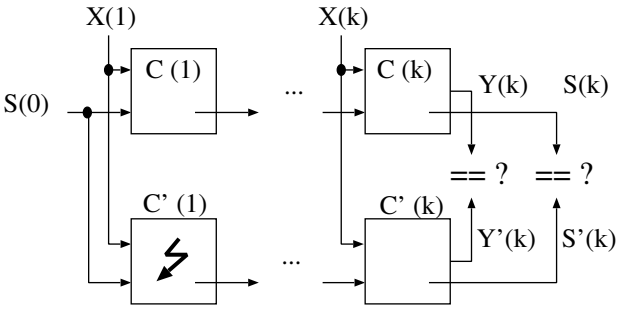


Fig. 2. Analysis

A. Model

ROBUCHECK supports the classification of fault tolerance with respect to transient faults in the circuit [9]. For this purpose, a circuit is divided into components, e.g. gates or hierarchical structures. A component is classified non-robust, if a fault at the component may tamper the output behavior. A transient fault at a component is modeled by changing the value at the component non-deterministically.

Using components as a base for the classification, allows to restrict the analysis to single faults of components that may correspond to multiple gate level faults in practice. For example, using the n-bit output of an adder as component covers all failures at the output and all failures of the internal gates of the adder. As a result ROBUCHECKreturns unprotected components as well as an overall estimation of the fault tolerance of the circuit.

To prevent system failures, appropriate actions have to be taken if a transient fault occurs in a circuit. In practice, mechanisms to achieve fault tolerance have to handle a fault within a short period of time after the occurrence. Otherwise the fault may propagate through the system. More precisely, the fault should be corrected or reported within a predefined number of clock cycles. Consequently, the analysis can be limited to an observation window of k cycles which – as a side effect – reduces the complexity of the analysis.

Based on this model ROBUCHECK partitions components into three classes:

- A component is **robust**, if its failure cannot affect the output behavior or the internal state of the circuit.
- A component is **non-robust**, if its failure may affect the output behavior within the predefined observation window.
- A component causes ***Silent Data Corruption*** (SDC), if its failure cannot affect the primary outputs within the predefined observation window but may corrupt the internal state. For brevity, we also say that "a component is classified as SDC".

Based on this classification, bounds of the fault tolerance of a circuit are defined. The **lower bound** is the percentage of robust components. The **upper bound** is given by the percentage of components classified as robust or SDC. Additionally, the model of [9] also exploits fault detection logic flagging a fault signal after detecting a fault. This extension is also included in ROBUCHECK.

B. Analysis

As suggested in [9], each component g is extended by fault injection logic that allows to replace the original value of g with a non-deterministic value. A transient fault is modeled by activating the fault injection logic at g.

In Figure 2 a schematic view of the model for the analysis is shown. The circuit under verification $\mathcal{C}$ and a copy $\mathcal{C}'$ with fault injection logic are compared. Both circuits start in the same arbitrary state denoted by $S(0)$ and are stimulated by the same input traces. According to the transient fault model a single fault is injected in the first time frame, indicated by $\lightning$. For each time frame, it is check whether a fault affects the outputs or state elements, respectively (indicated by $==$?). Based on this analysis components are classified robust, non-robust, and SDC.

The constraints on $S(0)$ significantly influence the computed fault tolerance. If $S(0)$ is too restrictive, i.e. not all reachable states are considered while analyzing the design, too many components may be classified as robust. On the other hand, if non-reachable states are allowed on $S(0)$ components may be classified non-robust with respect to that scenario that does not occur during normal operation. Computing the reachable state set based on *Binary Decision Diagrams* (BDDs) [2] is often infeasible in terms of memory and run time. A trade-off between accuracy and computational overhead is possible. ROBUCHECK integrates exact computation of reachable states based on BDDs [2], but also supports the computation of over- and under-approximations [9].

Furthermore, constraints can be added to compute fault tolerance with respect to a given specification [19]. For example, if a program runs on a fault tolerant CPU, the data of the program has to be taken into account. By constraining input traces and memory states, only relevant data of the desired program is considered, such that the fault tolerance with respect to the program is computed.

C. System Overview

Figure 3 gives an overview of our framework. ROBUCHECK automatically computes the fault tolerance given the following input: *i*) the design under verification (in Verilog [12] or BLIF format [7]); *ii*) the component model for the analysis, e.g. register transfer level or gate level; *iii*) the length of the observation window (k time steps).

Non-formal and formal engines are integrated into ROBUCHECK to perform the analysis: a simulation-based engine (SIM), an engine based on *Sequential Equivalence Checking* (SEC), and an engine based on *Automatic Test Pattern Generation* (ATPG). All engines use the same analysis model (see Section II-A), but have specific advantages and disadvantages.

The simulation engine provides a rough estimate about the fault tolerance of the circuit. First components are classified non-robust which reduces the search space for the following formal analysis. A formal analysis based on SEC analyzes the remaining components using a formal model of the full circuit [9]. A model adapted from ATPG is capable to formally analyze the fault tolerance of single components. The SEC-engine and the ATPG-engine use a solver for *Boolean Satisfi-*

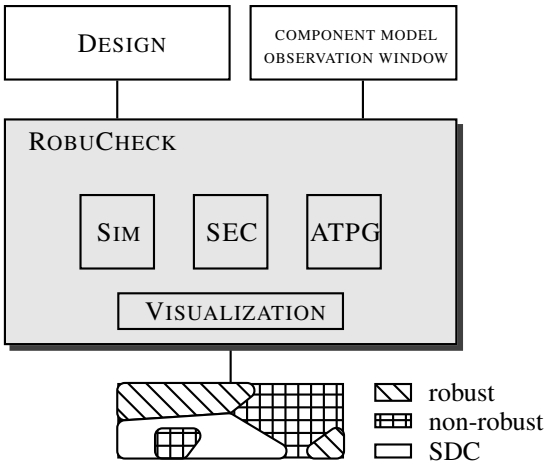


Fig. 3. ROBUCHECK

ability (SAT) as a back-end [5], [6]. In the last decades, SAT solvers have become very efficient due to sophisticated techniques like e.g. conflict-analysis [14], Boolean constraint propagation [16], and incremental learning [20]. The SEC-engine and the ATPG-engine are core features of ROBUCHECK.

The graphical back-end of ROBUCHECK supports the designer during subsequent analysis steps. Non-robust components and components that cause SDC are highlighted in the design and give an overview about parts that may be influenced by transient faults and thus require additional fault protection mechanisms. Non-robust components are visualized using the strong visualization capabilities of RTLvision^{TM} [4]. Some of the core features are: hierarchical schematic view, source code browsing, and cross-probing between schematic view, and source code.

III. ENGINES

This section briefly describes the engines and improvements to speed up the analysis. Features of the engines are discussed.

A. Simulation

Randomized fault injection computes an initial set of non-robust components. Faults are randomly injected and random stimuli are applied to analyze the influence on the circuit's behavior. Whenever an injected fault becomes visible at a primary output, the component is classified as non-robust. When the internal state is changed, SDC occurs. Simulation cannot determine robust components, because typically a complete simulation of all input stimuli is infeasible. The simulation always starts from the reset state to avoid reachability issues discussed above. The simulation stops, if *i*) a pre-defined number of traces has been simulated or *ii*) all components are classified as non-robust.

Fault simulation allows for a fast classification in case of non-robust circuits. In contrast, for very robust circuits the simulation causes an overhead as no components can be classified.

B. Sequential Equivalence Checking

The SEC-engine performs a formal equivalence check of the circuit under fault assumption and a fault free version of the circuit. Given a circuit $\mathcal{C}$, a copy $\mathcal{C}'$ containing fault injection logic for all non-classified components is created. Both circuits are unrolled up to a pre-defined number of time frames according to the length k of the observation window as specified by the user. Then, the behavior of the two circuits is analyzed assuming that at most one transient fault is injected into $\mathcal{C}'$ within the observation window. A cardinality constraint is applied to ensure that the fault injection logic at exactly one component is activated.

Starting with a single time frame, the classification starts by forcing the outputs of $\mathcal{C}$ and $\mathcal{C}'$ to be different. If there exists a fault, there exists an assignment to primary inputs and to the initial state such that the primary outputs differ; the component where the fault occurs is classified as non-robust. In the same way components are classified as SDC by forcing differing states between $\mathcal{C}$ and $\mathcal{C}'$. While components remain that are classified as SDC, the classification proceeds. The formal model is iteratively extended to additional time frames. The classification stops, if *i*) all components are classified as either robust or non-robust, or *ii*) the user-specified length of the observation window is reached.

This formal model is similar to the one of *Bounded Model Checking* (BMC) [1]. As an advantage all components can be classified using a single formal model. But memory and run time limits may be reached even for circuits of moderate size. For this reason an additional ATPG-based engine is integrated in ROBUCHECK.

C. Automatic Test Pattern Generation

The ATPG-engine classifies a single component at a time. By this, the formal analysis on the ATPG-like model leads to smaller instances and single components can be classified with respect to a larger number of time frames. While the model for SEC always contains the whole circuit, the model for ATPG only contains the output cone of the component under consideration and the transitive input cone of all primary outputs included. Moreover, fault injection logic is only required for the single component under consideration and fault free parts of $\mathcal{C}$ and $\mathcal{C}'$ are shared. As in SEC the analysis proceeds by iteratively extending the analysis starting from a single time frame up to the length of the observation window.

The classification stops, if *i*) the component is classified as non-robust or robust, or *ii*) the model reached the length of the observation window.

D. Comparison

Table I shows a comparison of both formal engines. SAT-instances are denoted by Φ in the following and $|\Phi|$ denotes the size of the SAT instance given as the number of clauses. The size $|\mathcal{C}|$ of a circuit $\mathcal{C}$ is given by the number of basic gates (AND, OR, etc.) required for the realization. The length of the observation window is denoted by k.

Given a sequential circuit $\mathcal{C}$, the ATPG-engine creates a SAT-instance Φ_{ATPG} only for the relevant parts with respect to a single component. However, in the worst case the engine may have to consider a fault free version and a faulty version of the whole circuit. Thus, the size of the SAT instance is in $O(|\mathcal{C}| \cdot k)$ for analyzing k time frames.

TABLE I
COMPARISON OF FORMAL ENGINES

	ATPG	**SEC**
Size of SAT instance	$O(\|\mathcal{C}\| \cdot k)$ two copies of $\mathcal{C}$	$\Theta(\|\mathcal{C}\| \cdot k)$ two copies of $\mathcal{C}$ plus fault injection logic per comp.
Number of SAT instances	$\|\mathcal{C}\|$	1
Reuse of learned information	for a single component; for all time frames	for all components; for all time frames
SAT-calls $r =$ #robust comp., $n =$#non-robust comp., $u =$#classified as SDC	$O((r+n+u) \cdot k) = O(\|\mathcal{C}\| \cdot k)$	$O((n+u) \cdot k)$
Size of the search space	$2^{\|\mathrm{PI}\|k+\|\mathrm{FF}\|+l}$	$2^{\|\mathrm{PI}\|k+\|\mathrm{FF}\|+\|\mathcal{C}\|\cdot l}$

The SEC-engine always considers the whole circuit in the SAT instance Φ_{SEC}, i.e. the size of Φ_{SEC} is in $\Theta(|\mathcal{C}| \cdot k)$ when considering k time frames. The size of the instances and the memory usage of ATPG is smaller than for the SEC approach, i.e. $|\Phi_{\mathrm{ATPG}}| \leq |\Phi_{\mathrm{SEC}}|$.

The second row gives the number of SAT-instances typically created by the engines. ATPG requires up to one SAT instance per component that is iteratively extended to the full observation window. The SEC approach only uses a single SAT instance to classify all components.

This also explains the utilization of learned information in the third row. ATPG learns information for a single component, drops the SAT instance afterwards and starts from scratch for the next component. The SEC engine accumulates learned information for all components.

The number of SAT calls depends on the robustness of the circuit. The ATPG engine handles each component individually and thus requires one call per component and per time frame. The SEC engine is more efficient on circuits with a large number of robust components. Here, only one SAT call is required to determine a large set of robust components simultaneously.

Consequently, both engines are working on slightly different search spaces. In principle, the SAT solver searches for full assignments for all variables in the SAT instance. But given the structural knowledge, after assigning values to variables describing the initial state, the values of primary inputs and the faulty values any other variable receives an assignment by simulation (or Boolean constraint propagation in a SAT solver, respectively). As shown in the last row, the ATPG engine determines values for all flip-flops (FF) for the initial state, for the primary inputs PI for all k time frames considered and for l bits defining the output value of the faulty component, leading to a search space of $2^{|\mathrm{PI}|k+|\mathrm{FF}|+l}$ assignments. The SEC engine additionally classifies which component is faulty, described by a search space of $2^{|\mathrm{PI}|k+|\mathrm{FF}|+|\mathcal{C}|\cdot l}$ assignments.

The formal engines create copies of the circuit in the problem instances. Compared to this, the overhead to run the simulation is very small. The simulation runs directly on the internal circuit graph of ROBUCHECK. But only a single input trace is considered at a time. Moreover, only a single fault can be injected at a time to perform a correct analysis. Only if the fault is injected at a component sensitized under the trace and in the given state, erroneous output values or SDC can be observed. As a result, simulation is relatively weak even in identifying non-robust components but it is very fast and handles large circuits.

IV. USE CASES

In this section, use cases for different circuits and different configurations of ROBUCHECK are presented.

A. Detecting Implementation Weaknesses

The aim of ROBUCHECK is in verifying the fault tolerance of a digital circuit and providing an overall measurement of the robustness. The following use case demonstrates the effectiveness of ROBUCHECK.

Assuming a designer has implemented a circuit with precautions against transient faults using *Triple Modular Redundancy* (TMR) of the functional units with voter logic. All signals in the TMR modules are protected against transient faults given a correct implementation of the voter logic. This correctness can be checked using ROBUCHECK.

Exemplarily, the plain ITC'99 circuit *b01* was extended to a TMR implementation. The resulting circuit has two inputs and outputs with overall 226 signals and 198 gates.

Due to the TMR implementation, the circuit masks single faults and contains 98% robust components. Only primary inputs and the logic that drives the primary output may be affected by transient faults.

The formal engines are quite effective. Both formal engines compute the exact fault tolerance within 20 seconds (SEC:

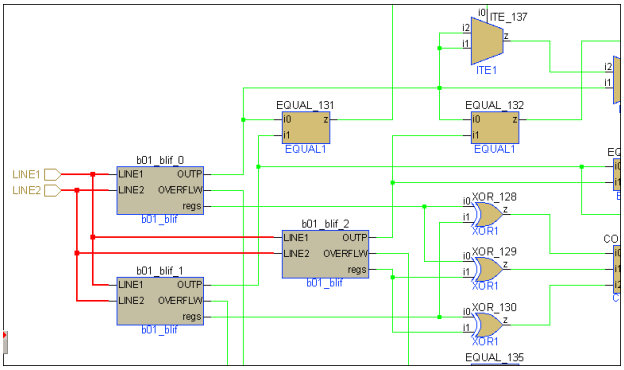


Fig. 4. Visualization

TABLE II
INFLUENCE OF STATES

Name	Lower Bound		Upper Bound	
	Approx.	Exact		Approx.
b10	0.0%	0.4%	1.7%	1.7%
b10-par	81.3%	83.2%	83.2%	83.2%
b10-tmr	1.5%	8.9%	97.8%	97.8%
b10-tmrflt	97.8%	97.8%	97.8%	97.8%

12s, ATPG: 19s). In comparison to this, the engine based on simulation required 50 seconds to simulate over two million traces and 10.000 cycles. In this setting simulation and formal analysis determined the same robustness values. However, no proof can be given by the simulation engine whether there are more non-robust components contained in the circuit or not. Here, the formal engines provide a proof within a short run time.

Figure 4 visualizes the results using the graphical back-end of ROBUCHECK. Most components are robust (denoted by output signal in green color or, if printed in greyscale, in light grey). Only faults occurring directly at the primary inputs or at the primary output cause erroneous output. The corresponding non-robust components are marked in red or dark grey. The circuit does not contain components that, if failing, only cause SDCs without corrupting the output data.

B. Robustness Computation

In order to compute the exact robustness of a digital circuit, the set of reachable states must be known. But the computation of reachable states is expensive. Our approach is also able to use an approximation of the set of reachable states to approximate the robustness of a circuit [9] as explained in Section II-B. To demonstrate the outcome, the ITC'99 circuit *b10* has been extended with: i) a parity checker for fault detection (*b10-par*), ii) TMR without fault detection (*b10-tmr*), and iii) TMR with fault detection (*b10-tmrflt*). Table II lists results of the robustness computation for the exact and the approximate approach. As explained in Section II-A a lower and an upper bound are calculated. If all components are classified the lower and the upper bound meet at a precise value. Whether all components can be classified within the observation window of 10 time steps depends on the circuit.

Consider the exact values at first. In case of *b10-tmr* silent data corruption is not detected. Therefore a large set of components remains non-classified resulting in a wide gap between the lower bound of 8.9% and the upper bound of 97.8%. For the other circuits most but not all components are classified. The approximate values are close to the exact values for both bounds showing a good quality for the approximate approach.

The advantage of the approximate approach are shorter run times in all cases. The exact approach was 2 to 1000 times slower. In particular, the approximate approach can even handle those circuits where the exact set of reachable states is not available.

C. Detailed Example

In this section a detailed example is presented. The circuit consists of three counters with voter logic. Figure 5 shows an overview. All state elements are initialized to 0. When the signal `count` is one, the internal counters are incremented by one. Otherwise the internal value does not change. The output `value` gives the current value of the counters. Moreover, the three counter modules are synchronized at every sixth clock cycle. The number of cycles that passed after the last synchronization is given by the output `cycle`. After five clock cycles majority voting is used to determine the value loaded into each of the three submodules.

In Figure 6 the result of ROBUCHECK is presented. Results for the exact set of reachable states are shown in Figure 6(a). While the observation window is extended (along the x-axis) more and more components are classified as being either robust or non-robust. The upper bound decreases quite fast to the final value of 82% – propagation of fault effects can be done very fast in this circuit. The lower bound reaches the final value only after analyzing six time frames. This is the number of clock cycles required to re-synchronize the counters, i.e. after six time frames SDC corruption is corrected.

Figure 6(b) considers an approximate check. The under-approximation of reachable states leads to an upper bound for the robustness. Non-robust components are classified effectively for this circuit. In contrast an overapproximation of reachable states which yields a lower bound for the robustness remains relatively inaccurate. Here, a better approximation of the reachable states can be applied.

This example shows how some insight into the circuit helps to determine the observation window required to fully analyze the circuit. Moreover, while ROBUCHECK is running, intermediate results of the analysis are available until the full observation window is reached.

V. CONCLUSION

The tool ROBUCHECK analyzes whether a circuit can be corrupted by transient faults like single event upsets, etc. Multiple engines are integrated to facilitate a high coverage of the functional space of the circuit. The formal engines even ensure that the full functional space under any potential fault is analyzed. But the formal approach cannot be applied to very large circuits. In this case a simulation based engine helps to identify non-robust components in the circuit.

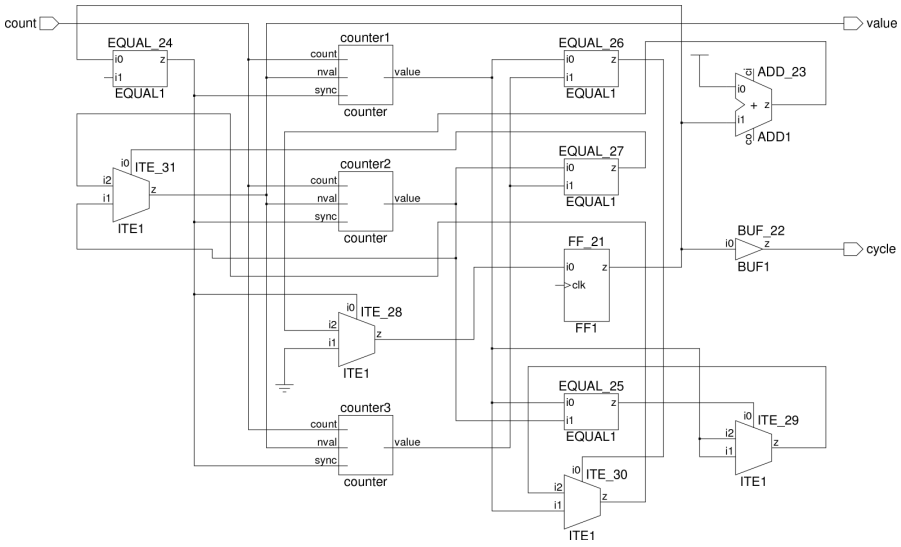
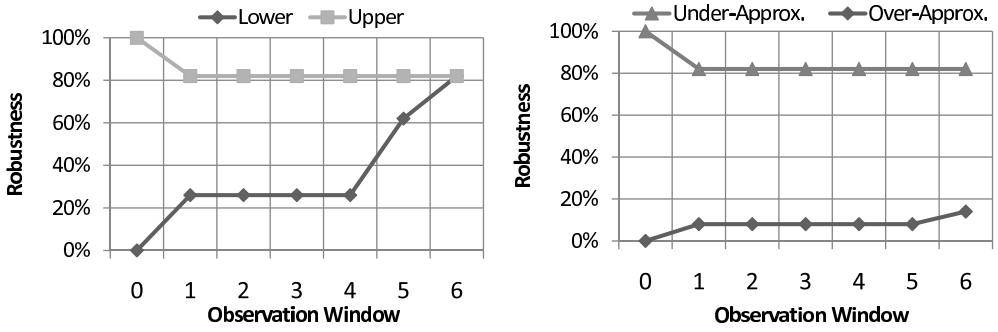


Fig. 5. Detailed example



(a) Exact (b) Approximation

Fig. 6. Robustness Analysis for the fault tolerant counter

REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[3] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. *Jour. of Electronic Testing: Theory and Applications*, 18(3):261–271, 2002.

[4] Concept Engineering GmbH. *RTLvision PRO*. http://www.concept.de, 2009.

[5] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5:394–397, 1962.

[6] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[7] Electronics Research Laboratory, University of California at Berkeley. *OCTTOOLS-5.2 Part II Reference Manual*, Mar. 1993.

[8] G. Fey and R. Drechsler. A basis for formal robustness checking. In *Int'l Symp. on Quality Electronic Design*, pages 784–789, 2008.

[9] G. Fey, A. Sülflow, and R. Drechsler. Computing bounds for fault tolerance using formal techniques. In *Design Automation Conf.*, pages 190–195, 2009.

[10] S. Frehse, G. Fey, and R. Drechsler. A better-than-worst-case robustness measure. In *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2009.

[11] S. Frehse, G. Fey, A. Sülflow, and R. Drechsler. Robustness check for multiple faults using formal techniques. In *Euromicro Conference on Digital System Design (DSD)*, pages 85–90, 2009.

[12] I. V. S. Group. *IEEE Standard Verilog Hardware Description Language*. http://www.verilog.com/IEEEVerilog.html.

[13] M. Hunger, S. Hellebrand, A. Czutro, I. Polian, and B. Becker. ATPG-Based grading of strong fault-secureness. In *IEEE International On-Line Testing Symposium*, pages 269–274, 2009.

[14] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[15] M. Miskov-Zivanov and D. Marculescu. Circuit reliability analysis using symbolic techniques. *IEEE Trans. on CAD*, 25(12):2638–2649, 2006.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[17] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin. CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework. In *Int'l Conf. on Comp. Design*, pages 363–370, 2008.

[18] O. Ruano, J. Maestro, and P. Reviriego. A methodology for automatic insertion of selective TMR in digital circuits affected by SEUs. *Nuclear Science, IEEE Transactions on*, 56(4):2091–2102, 2009.

[19] A. Sülflow, S. Frehse, G. Fey, and R. Drechsler. Anwendungsbezogene Analyse der Robustheit von digitalen Schaltungen. In *GMM/GI/ITG-Fachtagung Zuverlässigkeit und Entwurf*, pages 45–52, 2009.

[20] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conf.*, pages 542–545, 2001.

[21] C. Zhao and S. Dey. Improving transient error tolerance of digital VLSI circuits using RObustness COmpiler (ROCO). In *Int'l Symp. on Quality Electronic Design*, pages 133–140, 2006.