A static analysis for the minimization of voters in fault-tolerant circuits

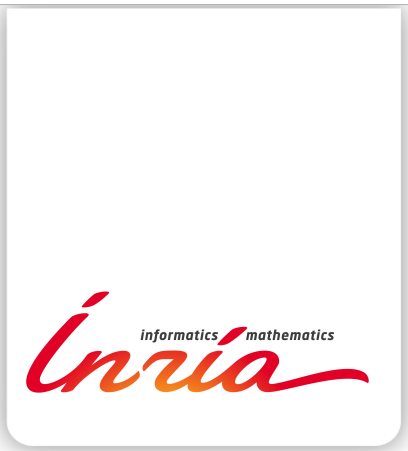
Dmitry Burlyaev, Pascal Fradet, Alain Girault

A static analysis for the minimization of voters in fault-tolerant circuits

Dmitry Burlyaev, Pascal Fradet, Alain Girault

RESEARCH REPORT
N° 9004
December 2016
Project-Team Spades

ISSN 0249-6399 ISRN INRIA/RR--9004--FR+ENG



A static analysis for the minimization of voters in fault-tolerant circuits

Dmitry Burlyaev, Pascal Fradet, Alain Girault

Project-Team Spades

Research Report n° 9004 — December 2016 — 27 pages

Abstract: We present a formal approach to minimize the number of voters in triple-modular redundant (TMR) sequential circuits. Our technique actually works on a single copy of the TMR circuit and considers a large class of fault models of the form "at most 1 Single-Event Upset (SEU) or Single-Event Transient (SET) every k clock cycles". Verification-based voter minimization guarantees that the resulting TMR circuit (i) is fault tolerant to the soft-errors defined by the fault model and (ii) is functionally equivalent to the initial TMR circuit. Our approach operates at the logic level and takes into account the input and output interface specifications of the circuit. Its implementation makes use of graph traversal algorithms, fixed-point iterations, and binary decision diagrams (BDD). Experimental results on the ITC'99 benchmark suite indicate that our method significantly decreases the number of inserted voters, yielding a hardware reduction of up to 55% and a clock frequency increase of up to 35% compared to full TMR.

Key-words: digital circuits, static analysis, fault-tolerance, triple modular redundancy, optimizations

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Analyse statique pour la minimisation de voteurs dans les circuits tolérants aux fautes

Résumé : Nous présentons une approche formelle pour minimiser le nombre de voteurs dans les circuits utilisant la redondance modulaire triple (TMR). Notre technique opère sur une seule copie du circuit TMR et considère une large classe de modèles de fautes de type «au plus 1 perturbation isolée (SEU) ou transitoire (SET) tous les k cycles ». Notre minimisation du nombre de voteurs garantit que le circuit TMR résultant (i) reste tolérant aux fautes définies par le modèle de fautes considéré et (ii) est fonctionnellement équivalent au circuit initial. Notre approche consiste en une analyse statique du circuit logique et peut prendre en compte des contraintes et/ou informations sur les entrées/sorties. Son implémentation repose sur des algorithmes de parcours de graphes, des itérations de point fixe et des diagrammes de décision binaire. Les résultats expérimentaux sur les jeux d'essai ITC'99 indiquent que notre méthode réduit significativement le nombre nécessaire de voteurs: elle permet une réduction (jusqu'à 55%) du nombre global de portes et une augmentation (jusqu'à 35%) de la fréquence d'horloge.

Mots-clés : circuits numériques, analyse statique, tolérance aux fautes, redondance modulaire triple, optimisations

1 Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors is an important research topic. As technology shrinks, the risk of system failures due to soft errors increases, which is especially dangerous in safety-critical industries (*e.g.,* space, transport, nuclear, *etc.*). Natural radiation, such as neutrons of cosmic rays and alpha particles of packing or solder materials, is a common source of soft errors [44, 5, 47, 26]. There are two main types of soft errors: Single-Event Upsets (SEUs) (*i.e.,* bit-flips in Flip-Flops (FFs)) and Single-Event Transients (SETs) (i.e. glitches propagating in the combinational circuit). Since an SET may potentially lead to several bit-flips, SETs are more general than SEUs.

Triple-Modular Redundancy (TMR) proposed by von Neumann [46] remains the most popular fault tolerance technique in Field-Programmable Gate Arrays (FPGAs) to mask both types of soft-errors. Yet, manual introduction of TMR [29] into a circuit design is often a tedious and error-prone process. Hence, several CAD tools automatically implement TMR for fault tolerant FPGA designs[11, 22, 39, 45, 25].

In a triplicated sequential circuit, adding voters at the primary outputs is not sufficient in general. Indeed, an error may remain in a memory cell long enough until another error corrupts a different redundant copy of the circuit. In that case, the final vote may produce an incorrect output. Voter insertion after each memory cell is sufficient to prevent errors from remaining in cells. However, it greatly increases both hardware overhead and the critical path, which directly influences the circuit performance. Thus, the overall TMR throughpout is degrading whereas it should be the main advantage of TMR over time-redundant fault-tolerance techniques.

From the functional point of view, introducing a voter per cell is excessive in most cases. Intuitively, this is because some voters are useless, either because faults at this stage will be captured by another voter "later" in the circuit, or because some faults are naturally masked by the logic. But, to the best of our knowledge, there is no tool dedicated to voter minimization in TMR that guarantees fault-tolerance according to a user-defined fault model. The main existing research trends in TMR have been providing probabilistic solutions and not absolute ones (see Section 9).

Our objective is to propose an *automatic*, *optimized*, and *certified* transformation process for TMR on digital circuits. In this paper, we focus on the optimization aspect of the automatic transformation: it should insert as few voters as possible, while guaranteeing to mask all errors of the considered fault-model.

We consider circuits described at the gate level (*i.e.,* netlists of AND, OR, NOT gates plus FFs – also called memory cells). This level has two main advantages:

- gate level netlists can be described by an elementary language, which simplifies correctness proofs;
- it is easier to prevent synthesis tools from optimizing (undoing) our transformation at this late design stage.

Since the main contributors to Soft-Error Rate (SER) at frequencies below 1 GHz are the FFs [32], we focus first on errors caused by SEUs (*i.e.,* bit-flips in FFs). We consider fault models of the form "at most one bit-flip within K cycles" denoted by $SEU(1, K)$. However, SETs in high-speed Integrated Circuits (ICs) have become a growing concern [30, 13, 37]. In response, we expend our approach to SET fault-model in the form "at most one transient fault within K clock cycles" denoted by $SET(1, K)$.

The proposed voter-minimization methodology is based on a static analysis that checks whether an error in a single copy of the TMR circuit may remain after K cycles. If not, protecting the primary outputs with voters is sufficient to mask the error. If, for instance, the circuit is

a pipeline without feedback loops, then any bit-flip will propagate to the outputs and will thus disappear before K cycles, where K is the length of the longest path. But if the state of the circuit is still erroneous after K cycles (in the form of an incorrect value stored in one of its memory cells), then there is a potential error accumulation since, according to the $SEU/SET(1,K)$ models, another soft-error may occur in another copy of the circuit. It may lead to two incorrect redundant modules of the TMR circuit and the loss of its fault-tolerance properties. In this case, additional voters are needed to prevent an error accumulation and mask all errors circulating inside one redundant module before the next soft-error may occur.

Our static analysis consists of four steps. The first step, described in Section 2, is purely syntactic and finds all loops in the circuit. Error accumulation can be prevented by keeping enough voters to cut all loops.

In many cases, a digital circuit resets (or overwrites) some memory cells, which may mask errors. Detecting such cases allows further useless voters to be removed. This second step is performed by a semantic analysis (Section 3) taking into account the logic of the circuit.

Circuits are also often supposed to be used in a specific context. For instance, a circuit specification may assume that a `start` signal occurs every x cycles and outputs are only read y cycles after each `start`. When such assumptions exist, taking them into account makes the semantic analysis more effective. Section 4 and Section 5 explain how to integrate such input and output specifications respectively.

Our analysis has been implemented based on graph algorithms and fixed point iterations using Binary Decision Diagrams (BDDs). We have tested several safe approximations and trade-offs between cost and precision. The implementation and experiments are presented in Section 7. Related work on TMR and voter insertion strategies are reviewed in Section 9. We summarize our contributions and sketch a few extensions in Section 10.

This article extends and revises the work presented in DATE'14 [15]. Section 6, presenting the extension of the approach to SET, is new. Sections 3 and 7 present and assess respectively a new abstract domain; explanations and examples have been added throughout.

2 Syntactic Analysis

We consider triplicated circuits with voters but we actually work on a *single copy* of the circuits. The effect of insertion or removal of voters can be represented and analyzed on a single copy of the TMR circuit. We model a sequential circuit C as a directed graph G_C where each vertex represents a FF (memory cell or latch) and an edge $x \to y$ exists whenever there is at least one combinational path between the two FFs x and y in C. An error in a cell x may propagate, in the next clock cycle, to all cells connected to x by an edge in this graph. Note that this is an over-approximation since the error may actually be masked by some logical operation.

Under the fault model $SEU(1,K)$, error accumulation is the situation where an error remains in the circuit K clock cycles after the SEU that caused it. Any circuit C without feedback loop will return, after an SEU, to a correct state before K clock cycles, provided that K is larger than the maximal length of the paths in G_C. In environments with high levels of ionizing radiations (*e.g.,* space, particle accelerators), K is bigger than 10^{10} [10]. For comparison, Soft-Error Rate (SER) can be as small as 10^{-10} bit-upset/day for Virtex FPGAs in terrestrial conditions [12]. So, even if our approach can deal with any K, we can assume that K is larger than the max length of all paths in G_C. It follows that error accumulation can only be caused by cycles in G_C, which must therefore be cut by removing vertices. Removing a vertex in G_C amounts to protecting the corresponding memory cells with a voter in the triplicated circuit.

The best solution to cut all cycles in G_C is to find the Minimum Vertex Feedback Set (MVFS),

Table 1: Voter Minimization, Syntactic Analysis Step

	Circuit	FFs	Syn.
Data Flow I.	Pipe.FP.Mult.8x8[1]	121	0
	Pipe.log.unit[1]	41	0
	Sh./A.Mult.8x8[36]	28	28
	ITC'99[21](subset)		
Control Flow Intensive	*b*01 Flows Compar.	5	3
	*b*02 BCD recogn.	4	3
	*b*03 Resourc.arbiter	30	29
	*b*06 Interrupt Hand.	9	3
	*b*08 Inclus.detect.	21	21
	*b*09 Serial Convert.	28	21

i.e., the smallest set of vertices whose removal leaves G_C without cycles. This standard graph problem is NP-hard [35]. While there exist good polynomial time approximations [27], the exact algorithm was efficient enough to be used in all our experiments with relatively small circuits (< 200 FFs).

Having a voter after each cell belonging to the MVFS prevents error accumulation. This simple graph-based analysis is very effective with some classes of circuits. In particular, it is sufficient to remove all internal voters in pipelined architectures such as logarithm units and floating-point multipliers (see Table 1).

However, this approach is not effective for many circuits due to the extensive use of loops in circuit synthesis from Mealy machine representation. In such circuits, most cells are in self-loops (e.g., D-type flip-flops with Enable input). This entails many voters if the syntactic analysis is used alone. However, if the circuit functionality is taken into account, we can discover that such memory cells may not lead to erroneous outputs. Detecting such cases requires to analyze the logic (semantics) of the circuit. We address this issue in the following section.

3 Semantic Analysis

The semantic analysis first computes the Reachable State Set (RSS) of the circuit with a voter inserted after each memory cell in the MVFS. Then, for each cell $m \in$ MVFS, it checks whether its voter is necessary: (i) First, the voter is removed and all possible errors (modeled by the chosen fault-model in each state of RSS) are considered; (ii) If such an error leads to error accumulation, then the voter is needed and kept.

3.1 The precise logic domain D_1

Correct and erroneous values are represented by the four-value logic domain D_1:

$$D_1 = \{0, 1, \overline{0}, \overline{1}\}$$

where $\overline{0}$ and $\overline{1}$ represent erroneous 0 and 1, respectively. The truth tables of standard operations in this four-value logic are given in Table 2. Note that AND and OR gates can mask errors:

$\overline{x} \vee 1 = 1$, $\overline{x} \wedge 0 = 0$, $\overline{0} \wedge \overline{1} = 0$, $\overline{1} \vee \overline{0} = 1$. The **err** function models bit-flips: *i.e.,* $\mathbf{err}(0)=\overline{1}$ and $\mathbf{err}(1)=\overline{0}$. The **vot** function models the effect of a voter on a single copy of the circuit and corrects an error: *i.e.,* $\mathbf{vot}(\overline{1})=0$ and $\mathbf{vot}(\overline{0})=1$. Finally, for any $x \in \{0,\ 1\}$, $\mathbf{vot}(\mathbf{err}(x))=x$.

Table 2: Operators for 4-value logic domain D_1

OR	**0**	**1**	$\overline{\mathbf{1}}$	$\overline{\mathbf{0}}$
0	0	1	$\overline{1}$	$\overline{0}$
1	1	1	1	1
$\overline{\mathbf{1}}$	$\overline{1}$	1	$\overline{1}$	1
$\overline{\mathbf{0}}$	$\overline{0}$	1	1	$\overline{0}$

AND	**0**	**1**	$\overline{\mathbf{1}}$	$\overline{\mathbf{0}}$
0	0	0	0	0
1	0	1	$\overline{1}$	$\overline{0}$
$\overline{\mathbf{1}}$	0	$\overline{1}$	$\overline{1}$	0
$\overline{\mathbf{0}}$	0	$\overline{0}$	0	$\overline{0}$

	NOT	**err**	**vot**
0	1	$\overline{1}$	0
1	0	$\overline{0}$	1
$\overline{\mathbf{1}}$	$\overline{0}$	0	0
$\overline{\mathbf{0}}$	$\overline{1}$	1	1

3.2 Semantic analysis with D_1

A sequential synchronous circuit with M memory cells and I primary inputs is formalized as a discrete-time transition system with the transition relation $\delta : \{0,1\}^M \times \{0,1\}^I \longmapsto \{0,1\}^M$. We abuse the notation and use M (resp. I) to denote both the number and the set of memory cells (resp. inputs) of the circuit. The state of a circuit is the values of its cells and the initial state s_0 is obtained after the circuit reset. $\Delta(S)$ denotes the function returning the set of states obtained from the set S after one clock cycle. Formally

$$\Delta(S) = \{s' \mid \exists i.\ \exists s \in S.\ \delta(s,i) = s'\}$$

Δ applies the transition function δ to all states of its argument set and all possible inputs. The set of reachable states RSS is defined by the following iteration:

$$S_0 = \{s_0\} \qquad S_{i+1} = S_i \cup \Delta(S_i) \tag{1}$$

Starting from the initial state, we compute the set of reachable states by accumulating states obtained by applying iteratively Δ. The set of possible states being finite, the iteration reaches a fixed point equal to the RSS and denoted[1] by $\{s_0\}^*_\Delta$.

The second phase is to check whether the suppression of voters may lead to an error accumulation under the chosen fault-model. Let δ_V be the transition relation of a circuit equipped with a voter after each cell in a given set V, and let Δ_V be its extension to sets. δ_V is defined as:

$$\begin{array}{l}\delta_V((m_1,\ldots,m_M),i) = \delta((m'_1,\ldots,m'_M),i)\\ \text{where}\ \ \forall\ 1 \le j \le M,\ m'_j\ = \begin{cases}\mathbf{vot}(m_j) & \text{if } m_j \in V\\ m_j & \text{otherwise}\end{cases}\end{array}$$

This checking process is described by Algorithm 1:

We start with the circuit equipped with a voter after each cell in the MVFS (line 1). For each such cell m, we check whether its voter suppression entails error accumulation. Bit-flips are

[1] We will use this notation with other initial states and transition functions.

Algorithm 1 Semantic Analysis – Main Loop

Input : $MVFS$; // The minimum vertex feedback set;
Δ; // The circuit transition function;
s_0; // The initial state;
Output : V; // The subset of vertices (*i.e.,* memory cells) after which a voter is needed

1: $V := MVFS$;
2: $RSS := \{s_0\}^*_\Delta$;
3: **forall** $m \in MVFS$
4: $V := V \backslash \{m\}$;
5: $S := \Delta^K_V(\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)])$;
6: **if** $ErrAcc(S)$ **then**
7: $V := V \cup \{m\}$;
8: *return* V

introduced in all possible cells and states of RSS according to the fault-model (line 5):

$$\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)]$$

The transition function corresponding to the circuit with the current set of voters (V) is applied K times (Δ^K_V), where K is the number of clock cycles in the fault model ($SEU(1, K)$). The resulting set of states shows error accumulation if there exists an erroneous cell in at least one state of this set, which we capture with the predicate *ErrAcc* in line 6. *ErrAcc* is defined as:

$$ErrAcc(S) \Leftrightarrow \exists s \in S.\ \exists m \in s.\ m = \overline{0} \vee m = \overline{1}$$

If the set S does not show error accumulation, the voter is useless and can indeed be suppressed. Otherwise the voter is re-introduced (line 7).

In practice, Δ is applied a small number of times dictated by the circuit functionality and available analysis time. It is always safe to stop the iterative computation before reaching K; the only drawback would be to infer an error accumulation when there is none. The number of Δ applications can be also adjusted to the available analysis time. In our experiments, the analysis time limit was set to 20 minutes and K to 50. Furthermore, the iteration is stopped:

- if the current set of states is errorless, then there cannot be error accumulation (no error can reappear);
- or, if the erroneous current set is the same as the previous one, a fixed point is reached and there is an error accumulation.

The order in which the cells in the MVFS are analyzed (line 2, in Algorithm 1) may influence the number of removed voters. We use the following heuristic to chose the ordering of voter selection: starting from the MVFS of memory cells with voters, we sort it first according to the number of successive memory cells that each cell has in the circuit netlist (the number of successors in G_C). Then, we consider primarily the removal of voters that lead to the corruption of the smallest number of cells in the next clock cycle. The voters whose removal may lead to a large number of corrupted cells are considered last. We found out that following this ordering, we are able to suppress more voters than with a random ordering or the ordering relying on the number of preceding memory cells in the netlist.

3.3 More Abstract Logic Domains

The aforementioned method is precise but costly since it considers all possible inputs. In general, keeping track of the relations between indeterminate inputs is not very useful. Fortunately, our technique can be used as it is with other, more abstract, logic domains. There are several domains that retain enough precision and allow larger circuits to be analyzed.

The 4-value logic domain D_2 decreases the state space explosion that occurs with D_1:

$$D_2 = \{0, 1, \mathrm{U}, \overline{\mathrm{U}}\}$$

The abstract value U represents a correct value (either 0 or 1) and $\overline{\mathrm{U}}$ represents any (possibly erroneous) value (*i.e.,* 0, 1, $\overline{0}$ or $\overline{1}$). A vector of n inputs is represented as a unique vector $(\mathrm{U}, \ldots, \mathrm{U})$ with D_2 whereas 2^n vectors had to be considered with D_1. The truth tables of standard operations in D_2 are given in Table 3.

Table 3: Operators for 4-value logic domain D_2

OR	**0**	**1**	**U**	$\overline{\mathbf{U}}$
0	0	1	U	$\overline{\mathrm{U}}$
1	1	1	1	1
U	U	1	U	$\overline{\mathrm{U}}$
$\overline{\mathbf{U}}$	$\overline{\mathrm{U}}$	1	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$

AND	**0**	**1**	**U**	$\overline{\mathbf{U}}$
0	0	0	0	0
1	0	1	U	$\overline{\mathrm{U}}$
U	0	U	U	$\overline{\mathrm{U}}$
$\overline{\mathbf{U}}$	0	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$

x	NOT	**err(x)**	**vot(x)**
0	1	$\overline{\mathrm{U}}$	0
1	0	$\overline{\mathrm{U}}$	1
U	U	$\overline{\mathrm{U}}$	U
$\overline{\mathbf{U}}$	$\overline{\mathrm{U}}$	$\overline{\mathrm{U}}$	U

In contrast with D_1, a gate with two erroneous values cannot produce a correct one. Logical masking of errors can only occur with two operations: $0 \wedge \overline{\mathrm{U}}$ and $1 \vee \overline{\mathrm{U}}$. This is sufficient to take into account the masking performed by explicit signals (*e.g.,* resets).

Typical examples where the semantic analysis with D_2 is more effective are circuits that use D-type FFs with an `enable` input driven by a Finite State Machine (FSM) encoded in the circuit. The syntactic approach would keep a voter for each such cell (they are in self-loops). The semantic analysis can detect that such cells are regularly overwritten by fresh inputs. For example, the resource arbiter *b*03 in Section 7 is such a circuit. After initialization, its finite state machine forces 12 cells ($fu1$-$fu4$, $ru1$-$ru4$, $grant\_o$[3:0]) to be overwritten with fresh values every other cycle. The semantic analysis (using D_1 or D_2) is able to show that those cells, although in self loops, do not need to be protected by voters.

Another approximate logic domain is the 16-values logic domain D_3, where a memory cell is encoded as a subset of its four possible values. It is defined as the powerset of D_1:

$$D_3 = \mathcal{P}(\{0, 1, \overline{0}, \overline{1}\})$$

A value A in D_3 is the set of all possible values that its memory cell can take at this stage of the analysis. For example, a fully determinate value is represented by a singleton (*e.g.,* $\{0\}$ for a correct 0 or $\{\overline{0}\}$ for a bit-flipped 1), an unknown but uncorrupted value by the set $\{0, 1\}$, and a completely unknown value by the set $\{0, 1, \overline{0}, \overline{1}\}$.

The operators of D_3 are the power set extensions of the operators of D_1.

$$\begin{array}{lcl} A \ \wedge_3 \ B & = & \{x \mid x = a \wedge_1 b,\ a \in A,\ b \in B\} \\ A \ \vee_3 \ B & = & \{x \mid x = a \vee_1 b,\ a \in A,\ b \in B\} \\ \neg_3 A & = & \{x \mid x = \neg_1 a,\ a \in A\} \\ err_3(A) & = & \{x \mid x = err_1(a),\ a \in A\} \\ vot_3(A) & = & \{x \mid x = vot_1(a),\ a \in A\} \end{array}$$

where $\wedge_1$ $\vee_1$, $\neg_1$, err_1, and vot_1 denote the *and*, *or*, *not*, *err*, and *vot* operators of D_1 as defined in Table 2.

That domain is a trade-off in terms of precision between D_1 and D_2. The main advantage of D_3 over D_1 is the prevention of state explosion, since a vector of n unknown and uncorrupted inputs is represented as a unique vector $(\{0,1\},\ldots,\{0,1\})$. Contrary to D_2, D_3 remains able to represent logical masking such as $\{\overline{0}\} \ \wedge_3 \ \{0,\overline{1}\} = \{0\}$ or $\{\overline{1}\} \ \vee_3 \ \{1,\overline{0}\} = \{1\}$. Domain D_3 can be seen as retaining precise information about the possible values and corruptions but ignoring the relationships between different inputs.

4 Inputs Specification

Circuits are often designed to be used in a specific context where some input signals must occur at definite timings. Taking into account assumptions about the context may make the semantical analysis much more precise, in particular, when the logical masking of corrupted cells depends on specific inputs (*e.g.,* a `start` control signal). Our approach is to translate these specifications into an interface circuit feeding the original circuit with the specified inputs. The analysis of the previous section can be applied to the resulting combined circuit. As a consequence, error accumulation is checked with the method described in Section 3.2, but under the constraints specified by the interface. The only small adjustment needed in Algorithm 1 is to make sure that errors are introduced only in the cells of the original circuit and not in the cells of the interface circuit.

We use ω-regular expressions to specify circuit interfaces. An ω-regular expression specifies constraints using vectors of $\{0,1,\star\}$, which replace primary inputs by 0, 1, or leave them unchanged ($\star$ being the wild card). Consider, for instance, a circuit with two primary inputs $[i_1,i_2]$, then the expression $([1,0]+[0,1]).[\star,\star]^\omega$ specifies that the circuit first reads either $i_1 = 0$ and $i_2 = 1$, or $i_1 = 0$ and $i_2 = 1$, and then proceeds with no further constraints.

In general, specifications need non-determinism to describe a partially specified or a non-deterministic context. Hence, the aforementioned ω-regular expression can also be seen as a Non-deterministic Büchi Automaton (NBA) that reads inputs and replaces them by 0, 1, or leaves them unchanged ($\star$).

For instance, the expression $([1,0]+[0,1]).[\star,\star]^\omega$ can be represented as the two-state NBA of Figure 1 (a): in the first state, it reads inputs and returns either the outputs $[1,0]$ or $[0,1]$ (regardless of the inputs). Then, the automaton goes (and stays) in the second state where inputs are read and produced as outputs. The indices in $\star_1$ and $\star_2$ allow to identify the inputs according to their position.

To generate a circuit from an ω-regular expression, we first convert the corresponding NBA into a deterministic automaton as follows. Each nondeterministic edge is made deterministic using new inputs (sometimes referred to as oracles). If a vertex has n nondeterministic outgoing edges, adding $log_2(n)$ new inputs is sufficient. For example, the specification $([1,0]+[0,1]).[\star,\star]^\omega$ can be made deterministic by adding a single additional input i. The automaton (see Figure 1 (b)) now reads three inputs: if i is 0 (resp. 1) it produces $[1,0]$ (resp. $[0,1]$). The resulting deterministic automaton is then translated into an interface circuit using standard logic synthesis

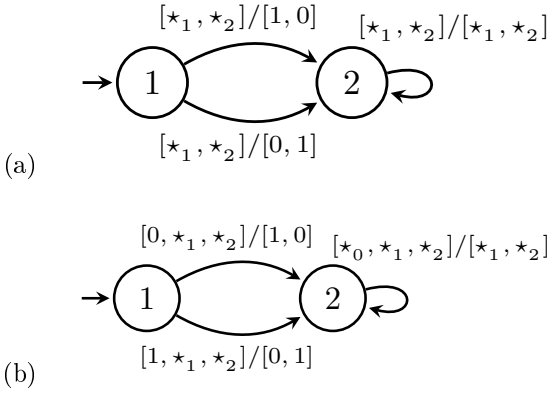


Figure 1: Input interface as a NBA (a) and its deterministic version (b)

techniques [23, p.118]. If the original circuit has I inputs, the resulting interface circuit will have $I + a$ (a new inputs to make it deterministic) inputs and I outputs. It is then plugged by connecting its outputs to the inputs of the circuit to be analyzed.

A typical example where an input specification is useful is the circuit $b08$ of Section 7. Such a circuit has a `start` input signal and 8-bit data input. Its input interface specification can be expressed as the following ω-regular expression:

$$([1,\star,\star,\star,\star,\star,\star,\star,\star].[0,\star,\star,\star,\star,\star,\star,\star,\star]^{17})^{\omega} \tag{2}$$

A `start` signal is first raised and the input data is read. For the next 17 cycles, data is processed and `start` is kept to 0. This process is repeated over and over. Since `start` is raised every 18 clock cycles, the internal data registers are rewritten periodically with new data, as they can keep erroneous data only until the next `start` signal. The circuit also has an internal FSM which can be corrupted but the periodic `start` ensures that it returns to its initial state every 18 cycles. Consequently, error accumulation is impossible for any $K > 18$, and no voters (except implicit voters at primary outputs) need to be inserted.

5 Outputs Specification

Consider another example, similar to the previous one, with 2 inputs, 1 output, and where some waiting can occur before raising the `start` signal. Formally, the input interface would be:

$$([0,\star]^{*}.[1,\star].[0,\star]^{17})^{\omega} \tag{3}$$

This interface does not guarantee that `start` will be raised before K clock cycles. Since the analysis must consider the case where `start` is not raised, it may detect error accumulation even though `start` would ensure logical masking. However, if it is known that the primary outputs are not read before some useful computation triggered by the `start` signal completes, a better analysis can be performed.

We specify the output interface by adding to each vector of the input interface a vector of $\{0, 1\}$ indicating whether the corresponding outputs are read (1) or not read (0). For instance, the output interface of the previous example, where the single bit output is read only after `start`

is raised, can be specified as

$$(([0,\star] : [0])^*.([1,\star] : [0]).([0,\star] : [1])^{17})^\omega \tag{4}$$

It states that the output is not read ([0]) until the `start` signal is raised. Then, the output is read ([1]) during 17 cycles.

The extended ω-regular expression is translated into an NBA as in Section 4, then made deterministic, and finally translated into a sequential circuit. The corresponding interface circuit will additionally produce 0 or 1 signals to filter the useless and needed outputs respectively. Each such signal is connected using an AND gate to the corresponding primary output of the original circuit. The final configuration with the surrounding interface circuit is shown in Figure 2.

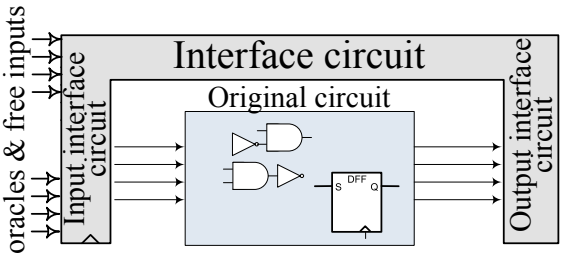


Figure 2: Original circuit with the surrounding interface circuit.

The property to check must now be refined to allow error accumulation as long as no error propagates to the filtered primary outputs. Recall that when an error occurs, it is allowed to propagate to outputs (or final voters) within the next K clock cycles since no additional soft-error can occur during that time. If there is an error accumulation, the analysis must further ensure that no error can propagate to outputs after the K cycles *i.e.,* when additional errors occur which could not be masked by final voters.

This is performed by lines 6-15 of Algorithm 2. If an error accumulation is detected in the reached state set S, K cycles after a fault occurrence (line 6), then we calculate all states $S^*_{\Delta_V}$ that can be reached after these K cycles (line 7). Then, we iteratively simulate the occurrences of additional errors (line 9-12) separated by at least K steps. E_0 (line 7) represents the circuit reachable state space with only one fault. E_i represents the reachable state space after at most $i+1$ errors separated from one another by at least K clock cycles. The global fixpoint E_i (line 13) represents the set of all possible states that can be reached after all possible sequence of errors allowed by the fault model. It can now be checked that none of these states leads to the propagation of an error to the (filtered) primary outputs (line 13).

Since this computation is done assuming that voters operate correctly, we must ensure that no error accumulate in a cell followed by a voter. Indeed, in that case, if a similar error occurs in a second copy of the circuit, the voter would fail to mask it. The function *ErrProp* (line 13) detects if there is a reachable state where a memory cell with a voter or a primary output is corrupted and prevents the voter under consideration (m) to be removed. We assume that each primary output is represented by a new memory cell. Let *out*, *vot* and *cor* be predicates denoting whether a cell represents an output, a cell protected by a voter or is corrupted respectively, then *ErrProp* is defined as:

$$\begin{aligned} \mathit{ErrProp}(E_i) \;\Leftrightarrow\; \exists s \in E_i.\, \exists m \in s.\, (\mathit{out}(m) \;\vee\; \mathit{vot}(m)) \\ \wedge\, (\mathit{cor}(m)) \end{aligned}$$

These criteria are safe but sometimes too strict. Consider, for instance, a circuit with a sequence of two enabled flip-flops (*i.e.,* with self loops) that produce significant output only two

cycles after the enable signal is set. Both cells may be protected by voters to break self loops and prevent error accumulation. However, no voter is necessary since error accumulation can occur only when no significant output is produced. Indeed, when the enable signal is set, new input and intermediate results will overwrite the (possibly corrupted) cells and a correct output will be produced. If we first try to remove the first voter, our algorithm will detect that an error can remain in the first cell after K steps. That cell will in turn corrupt the second one still protected by a voter. Hence, the condition *ErrProp* will prevent removing the first voter whereas starting with the second or removing both voters would have been possible. Therefore, a useful refinement of Algorithm 2 is, whenever *ErrProp* is true only because of error accumulation before some voters (and no error propagates to the output), to iterate and check whether all these voters can be removed.

Algorithm 2 Semantic Analysis with Output Specification

Input : $MVFS$; // The minimum vertex feedback set;
Δ; // The circuit transition function;
s_0; // The initial state;
Output : V; // The subset of vertices (*i.e.,* memory cells) after which a voter is needed

1: $V := MVFS$;
2: $RSS := \{s_0\}^*_{\Delta}$;
3: **forall** $m \in MVFS$
4: $V := V \backslash \{m\}$;
5: $S := \Delta^K_V(\bigcup_{m_i \in M} RSS[m_i \leftarrow \mathbf{err}(m_i)])$;
6: **if** $ErrAcc(S)$ **then**
7: $E_0 := \{S\}^*_{\Delta_V}$;
8: $i := 0$;
9: **repeat**
10: $i++$;
11: $E_i := E_{i-1} \cup (\Delta^K(\bigcup_{m_i \in M} E_{i-1}[m_i \leftarrow \mathbf{err}(m_i)]))^*_{\Delta_V}$;
12: **until** $E_i = E_{i-1}$
13: **if** $ErrProp(E_i)$ **then**
14: $V := V \cup \{m\}$;
15: *return* V

Output interfaces are especially useful for circuits whose outputs are not read before some input signal is raised and some computation is completed. For instance, the shift/add multiplier (see Sec 7) waits for a `start` signal. During that time, errors may accumulate in internal registers and propagate to the outputs, which are not read. When `start` occurs, fresh input data is read and written to internal registers (which are thus reset). The outputs are read only after the multiplication is completed and a `done` signal is raised.

Note that output interfaces can model Transient Error Tolerance (TET) where all errors at outputs are not necessarily critical. For instance, if erroneous outputs are considered non-critical within a specified number of cycles, output interfaces may express it and allow further optimizations. In this case, the optimized TMR configuration is tuned to particular system requirements. Such quality-guided optimizations are investigated on MPEG decoding in [31, 38] to select gates whose hardening maximize fault-tolerance.

6 Extension to Single-Event Transients

In the previous sections, we considered single event upsets and the corresponding fault-models $SEU(1, K)$, corresponding to "at most one bit-flip every K cycles". Hereafter, we extend our approach to single event transients, in particular, the fault model $SET(1, K)$ which can be read as "at most one SET within K clock cycles".

An SET occurs when an energetic subatomic particle strikes a combinational logic element [30]. Such particle causes a transient voltage disturbance, which can propagate on wires and possibly be latched by several memory cells. Consequently, an SET can potentially lead to several bit-flips (*i.e.,* several SEUs). In this section, we present the extension of our previous analysis to SET.

6.1 Precise modeling of SETs

As opposed to an SEU, the effect of an SET depends on the logical propagation (and possible logical masking) of the signal perturbation through the combinational part. Such signal perturbation or glitch is latched in a non-deterministic manner. From now on, a signal can take 3 values: a logical one, a logical zero, or a glitch written $\lightning$.

$$Signal := 0 \mid 1 \mid \lightning$$

A glitch can be masked in a combinatorial circuit by $\text{OR}(\lightning, 1) = 1$ or $\text{AND}(\lightning, 0) = 0$. The precise modelling of a glitched signal in a TMR circuit requires the knowledge of its correct value (present in the corresponding signals of the two other redundant modules). Consequently, the precise domain D_1 is extended as D_t to model a glitch propagation in a combinatorial circuit of one redundant module:

$$D_t = \{0, 1, \overline{0}, \overline{1}, 0^{\lightning}, 1^{\lightning}\}$$

where $0^{\lightning}$ and $1^{\lightning}$ represent respectively a glitched 0 and 1. That is, $0^{\lightning}$ represents a glitch at one point of the circuit such that the value in the two other redundant copies is 0. A glitch on an incorrect signal with the value $\overline{0}$ (resp. $\overline{1}$) will be represented by the signal value $1^{\lightning}$ (resp. $0^{\lightning}$). One example that illustrates the difference between a glitch and a corrupted value is:

$$D_1 : \ \overline{0} \ \vee_1 \ \overline{1} = 1 \qquad D_t : \ 0^{\lightning} \ \vee_t \ 1^{\lightning} = 1^{\lightning}$$

While in the first case, an *or* gate with corrupted but stable signals returns a correct value, in the second case, the glitch propagates.

While the precise domain D_1 requires the aforementioned extension to D_t, the domains D_2 and D_3 can overapproximate such glitch behavior with no extension. In particular, a glitched signal, as well as any possibly wrong stable signal, takes the value $\overline{\text{U}}$ in D_2. A glitched 1 (resp. 0) can be represented as $\{1, \overline{0}\}$ (resp. $\{0, \overline{1}\}$) in D_3.

A glitch propagated to a memory cell is non-deterministically latched as true or false. It follows that the precise glitch modelling in D_t implies that any glitched signal $0^{\lightning}$ (resp. $1^{\lightning}$) is non-deterministically latched as a correct 0 or as an incorrect $\overline{1}$ (resp. as a correct 1 or as an incorrect $\overline{0}$). This non determinism may lead to a significant state space growth in D_1. The domains D_2 and D_3 avoid this inconvenience since glitched signals are expressed in the same logic as the latched values.

To take into consideration all possible effects of an SET, it is necessary to calculate the set of reachable states for all cases of SET injections. These cases include a fault injection either at the output of a logical gate/a memory cell or the mutually exclusive corruption of branches of a wire split. The union of the state spaces that can be reached in each of these corruption cases forms the reachable state set.

The precise SET modeling in D_t imposes significant computational overhead. Its two important bottlenecks are the need to consider all possible SET injection points and all possible non deterministic choices when a glitch is latched. Both points can been taken into account by a transition function that expresses a circuit state change during a clock cycle with an SET and returns a set of possibly corrupted states. In the next Section, we propose a safe approximation of the precise SET modeling in domains D_1, D_2, and D_3.

6.2 Safe SET over-approximation

If a memory cell is connected by a combinational path to a component (wire or gate) where an SET occurs, this cell may be corrupted. We should find all sets of cells that can be corrupted at the same clock cycle to find the worst case. Each of these sets has a common combinational sub-circuit, in other words, a common combinational cone. The apex of such a cone is either the output of a memory cell or a primary input. A cone apex fully identifies a cone and the memory cells belonging to this cone.

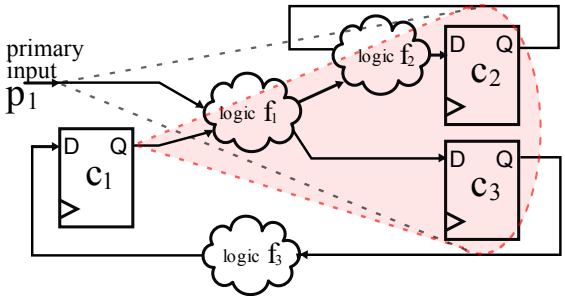


Figure 3: Combinational cones for SET modeling.

In Figure 3, the cone with apex at c_1 includes both cells c_2 and c_3. The cone with apex at p_1 also includes $\{c_1, c_2\}$. The cones with apexes at c_3 and c_2 contain $\{c_1\}$ and $\{c_2\}$ respectively.

As a result, the worst case scenario of any SET that happens inside a cone j is the union of all possible simultaneous corruptions of the memory cells $ms(j)$ in this cone. The power set $P(ms(j))$ is the set of all possible memory cell corruption configurations.

As soon as all corruption configurations are found, a new error injection procedure can be defined and used in both Algorithms 1 and 2. In particular, instead of mutually exclusive bit-flips injection to a state space S, expressed for SEU as $(\bigcup_{m_i \in M} S[m_i \leftarrow \mathbf{err}(m_i)])$, the corruption of the RSS by an SET is computed as the disjunction of possible simultaneous memory cells corruptions of the sets included in the cones after memory cells M or primary inputs I:

$$\bigcup_{j \in (M \cup I)} \left(\bigcup_{p \in P(ms(j))} S \left[\bigcap_{m_i \in p} m_i \leftarrow \mathbf{err}(m_i) \right] \right)$$

where $ms(j)$ is the subset of memory cells located in the cone with an apex at a memory cell or a primary input j.

Such corruption procedure is a safe over-approximation in the precise (D_t) and approximate (D_2, D_3) domains. The complexity bottleneck of the approach is the power-set computation with a large number of memory cells in a single cone. However, in the case of the approximate logic domains D_2 and D_3, we can consider only the worst case scenario: the simultaneous corruption

of all memory cells in a cone (without calculation of its powerset), computed as:

$$\bigcup_{j\in(M\cup I)} S\left[\bigcap_{m_i\in ms(j)} m_i \leftarrow \mathbf{err}(m_i)\right]$$

It may happen that the result of such SET insertion includes corrupted states that are not reachable because it does not take into consideration the internal error-masking capabilities of the combinational circuit. Nevertheless, we will see in the experiments that, for the analysis presented in this paper, such over-approximation is an appropriate choice.

7 Experimental results

The presented voter minimization technique has been implemented in Ocaml using the BDD library CUDD [2] and the Ocaml interface MLCuddIDL [3]. Transition systems and set of states are expressed as BDD formulae [20].

The introduced logic domains (D_1, D_2, D_3) are encoded with multiple bits (two for D_1 and D_2; four for D_3) and the associated operators (*e.g.,* Tables 2 and 3) are expressed as logic formulae over those bits. For instance, the values of D_1 can be encoded with two bits (a, b) as:

$$\begin{array}{lll} 1 & \text{as} & (1,1) \\ 0 & \text{as} & (1,0) \\ \overline{0} & \text{as} & (0,0) \\ \overline{1} & \text{as} & (0,1) \end{array}$$

In this encoding, the first bit a is the correctness bit, and the second one b is the value bit. The NOT operator of D_1 can be represented by the function:

$$\neg_1(a,\ b) = (a,\ \neg b)$$

We used the Quine-McCluskey algorithm to simplify the boolean functions corresponding to the AND and OR operators of D_1. The AND operator is encoded as:

$$\wedge_1((a_1,b_1),(a_2,b_2)) = (a_3,b_3)$$

$$\begin{array}{lrcl} \text{where} & a_3 & = & ((a_1 \wedge a_2) \vee (a_1 \wedge \neg b_1) \vee (a_2 \wedge \neg b_2) \vee \\ & & & (\neg a_2 \wedge (\neg b_1 \wedge b_2)) \vee (\neg a_1 \wedge (\neg b_2 \wedge b_1)) \\ & b_3 & = & b_1 \wedge b_2 \end{array}$$

And the OR operator is encoded as:

$$\vee_1((a_1,b_1),(a_2,b_2)) = (a_3,b_3)$$

$$\begin{array}{lrcl} \text{where} & a_3 & = & ((a_1 \wedge a_2) \vee (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \\ & & & (\neg a_1 \wedge (\neg b_1 \wedge b_2)) \vee (\neg a_2 \wedge (\neg b_2 \wedge b_1)) \\ & b_3 & = & b_1 \vee b_2 \end{array}$$

BDDs proved to be quite efficient to express the data structures and the processing required by our technique. We made use of Rudell's sifting reordering [41] while building and applying the transition function. It allowed the semantic analysis of circuits up to 100 memory cells on a standard PC (Intel Core i5-2430M/2Gb-DDR3). For comparison, without reordering, the negative impact of big BDD structures on the algorithm performance was observed already for

Table 4: Voter Minimization, SEU model, Boolean domains $D_1 \mid D_2 \mid D_3$.

	Circuit	FFs	Syn.	Semantic			Sem.Inp.			Sem.Out.		
				D_1	D_2	D_3	D_1	D_2	D_3	D_1	D_2	D_3
Data Flow Int.	FP Multiplier 8x8[1]	121	0	0	0	0	0	0	0	0	0	0
	log.unit[1]	41	0	0	0	0	0	0	0	0	0	0
	Multiplier 8x8[36]	28	28	19	19	19	19	19	19	8	8	8
	ITC'99[21](subset)											
Control Flow Intensive	*b*01 Cmp. serial flows	5	3	3	3	3	3	3	3	3	3	3
	*b*02 BCD recognizer	4	3	2	3	3	2	3	3	2	3	3
	*b*03 Resource arbiter	30	29	17	29	17	17	29	17	17	29	17
	*b*06 Interrupt handler	9	3	3	3	3	3	3	3	3	3	3
	*b*08 Inclusion detector	21	21	21	21	21	0	21	0	0	21	0
	*b*09 Serial converter	28	21	20	20	–	20	20	–	20	20	–

A '–' denotes an out of time termination of the analysis (>20 mins).

circuits with 20-30 memory cells. We did not put much efforts in the optimization but we believe that there remain much opportunities for improvement.

We used both fault-models $SEU(1,K)$ and $SET(1,K)$ with $K = 50$, which allows K cycles/transitions to be computed effectively (Δ^K). The obtained results are *a fortiori* valid for any $K \geq 50$. However, for non-restrictive trivial input/output specification and small circuits, it is not worth to choose higher K values since all reachable states might be visited within a small number of execution steps K, and no further optimization will be achieved even if we continue the execution. When all reachable states are visited the execution can be stopped even if K steps have not been fully performed. Thanks to the encoding of input/output specification into the circuit structure (Section 5), the reachable states also contain the information about the values of input signals and the relevance of primary outputs (for the error-propagation analysis). The number of steps K needed to explore the whole state space varies depending on the specification and circuit complexity. For small circuit (*e.g.,* *b*02, *b*01) with simple input/output specification (*e.g.,* only the reset at the very beginning), we visit all reachable states in $K < 10$ steps. On the other hand, for larger circuits (shift/add multipliers or the circuit *b*08) with explicit complex input/output interface specifications (FSMs with 10 and more states), a higher value of K is rewarding and allows us to catch error masking behaviors that happen regularly (*e.g.,* circuit restarts or returns to the initial state in cyclic FSMs within every 30-40 cycles).

Our analysis has been applied to common arithmetic units taken from the *OpenCores* project [1] and from the *ITC'99* benchmark suite [21]. For each circuit, we defined non-restrictive input-output specification for the sake of generality. For the majority of the circuits, the input pattern specifies only synchronous reset at its initialization phase and no further reset (*b*01, *b*02, *b*03, *b*04, *b*06, *b*09). Such non-restrictive patterns may reduce achievable optimizations, which could be significantly increased if more details about the behavior of the surrounding circuit were provided. However, for the shift/add multiplier [36] the input-output specification is dictated by its functionality. The produced output is relevant only two cycle after the `start` signal has been raised (one cycle to fetch new data plus at least one cycle to process it). Since we should not

assume when the output is read out, we suppose that the data output may be read at any time two cycles after the last `start` and until the next `start`. As a result, our semantic analysis with this output specification shows that only the 8 product bits should be protected by voters.

Circuit $b08$ represents a group of self-stabilizing circuits that return to their initial state (and wait for the next `start`) within a bounded number of cycles (for $b08$, this period is 8 cycles). Additionally, by functionality, the circuit is supposed to be restarted periodically. The corresponding input and output specification allowed us to suppress all voters. We would like to highlight that any circuit with internal counters has a similar behavior of self-stabilization (the shift/add multiplier is another example).

Table 4 summarizes the results of the analysis on those circuits in D_1, D_2, and D_3, with the fault-model $SEU(1, K)$. The column FFs shows the total number of memory cells in the original circuit, while the other columns show the number of remaining voters in the TMR circuit after the syntactic and semantic steps (without, with input, with input and output interfaces). In each case, we give the results obtained with the three logic domains.

The syntactic step eliminates all voters in circuits with a pipelined architecture such as adders, multipliers, or logarithmic units. With rolling pipelined architectures, a control part and a looped dataflow circuit may require voter protection (*e.g.,* none of the 28 voters of the shift/add multiplier are removed with only the syntactic analysis).

In general, control intensive circuits require a protection of their FSMs. Almost all memory cells of the serial flow comparator ($b01$) or the serial-to-serial converter ($b09$) have to be protected. Nevertheless, our analysis is capable of suppressing a significant amount of voters in many control intensive circuits. A circuit is usually composed of data and control flow parts and we can expect that most voters in the data flow part can be suppressed.

The logic domain D_2 is, most of the time, precise enough. However, correcting a bit-flip in D_2 (*e.g.,* $0 \rightarrow \overline{\mathrm{U}} \rightarrow \mathrm{U}$) looses information. In some circuits, like $b03$ and $b08$, substantial logical error masking is performed by an FSM and the analysis fails to detect it.

The precision of the domain D_3 allows us to achieve better optimizations than the domain D_2 in circuits $b03$ and $b08$ (see Table 4). With D_3, the corrupted FSM will recover to a precise state, while with D_2 its cells will recover to the correct unknown value U. This precise state plays a crucial role to show that the rest of the circuit, that depends on this FSM, will be cleaned up too.

The results for $SET(1, K)$ are shown in Table 5. The number of suppressed voters did not change with D_2. However, even the proposed approximations in Section 6.2 does not help to resolve the complexity problem for some circuits when analyzed with D_1 and D_3. The bottleneck results from the large number of corruption combinations if a single combinatorial cone includes many memory cells. For example, in the circuit $b03$, there is an FSM of 2 cells where each cell is connected through a combinatorial circuit to 26 memory cells (mainly controlling their enable signals). As a result, to approximate the impact of an SET in this FSM, we have to calculate all possible corruption combinations of 26 cells, which is 2^{26} configurations. The circuits that could not be analysed are marked by $*$ in Table 5.

The scalability of logic domains D_1, D_2, and D_3 has also been compared. Figure 4 presents the growth of the RSS S_i after i iterations (see Section 3) for the $b03$ and $b06$ circuits. The fixed point is reached with less iterations in D_2, and the number of states grows exponentially for D_1 versus linearly for D_2. The same behavior is observed in all considered circuits.

The logic domain D_3 reaches the fixed-point as fast as D_1 while keeping the same precision. This fact is demonstrated in Table 6 where we measured the number of cycles to calculate the RSS for each domain (the column "# iterations"). The column "seconds" gives the execution time spent to calculate the RSS, and the last column ,"# BDD nodes", gives the complexity of the RSS BDD representation in terms of allocated BDD nodes. On the one hand, the number of

Table 5: Voter Minimization, SET model, Boolean domains D_1 | D_2 | D_3.

	Circuit	FFs	Syn.	Semantic			Sem.Inp.			Sem.Out.		
				D_1	D_2	D_3	D_1	D_2	D_3	D_1	D_2	D_3
Data Flow Int.	FP Multiplier 8x8[1]	121	0	0	0	0	0	0	0	0	0	0
	Log.unit[1]	41	0	0	0	0	0	0	0	0	0	0
	Multiplier 8x8[36]	28	28	–	19	–	–	19	–	–	8	–
	ITC'99[21](subset)											
Control Flow Intensive	*b*01 Cmp. serial flows	5	3	3	3	3	3	3	3	3	3	3
	*b*02 BCD recognizer	4	3	2	3	3	2	3	3	2	3	3
	*b*03 Resource arbiter	30	29	–	29	–	–	29	–	–	29	–
	*b*06 Interrupt handler	9	3	3	3	3	3	3	3	3	3	3
	*b*08 Inclusion detector	21	21	–	21	21	–	21	0	–	21	0
	*b*09 Serial converter	28	21	–	20	–	–	20	–	–	20	–

A '–' denotes an out of time termination of the analysis (>20 mins)

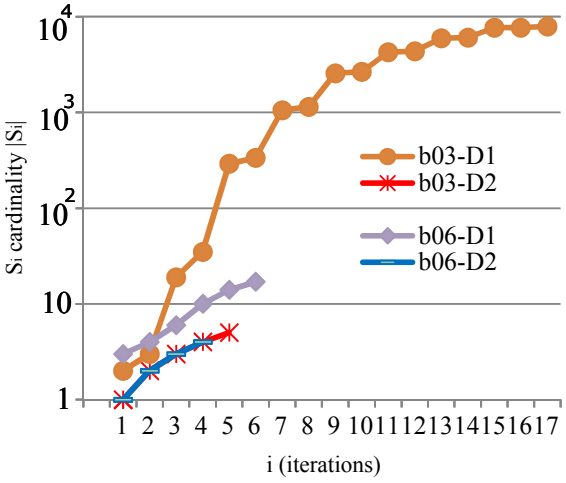


Figure 4: Logic Domain Comparison: Reachable State Space Size.

BDD nodes allocated to represent the RSS in larger circuits (*b*03, *b*08, *b*09) is much smaller with D_3 than with D_1. On the other hand, the BDD structures in D_3 require more variables and are more time consuming to manipulate. The domain D_3 overapproximates the RSS (see Section 3.3) which leads to less allocated nodes in the larger circuits. While it allows us to keep the necessary precision for optimizations comparable to the ones allowed by D_1, our existing implementation of D_3 would require further optimizations to be considered as an interesting compromise.

The bar graph of Figure 5 shows the ratio of the size of the RSS in D_1 to the corresponding size in D_2. The RSSs in D_1 are several orders larger than the corresponding ones in D_2. The most computation demanding step of the whole analysis is checking error propagation (see Section 5). A prohibiting growth of BDD structures representing the set of states E_i was observed with D_1

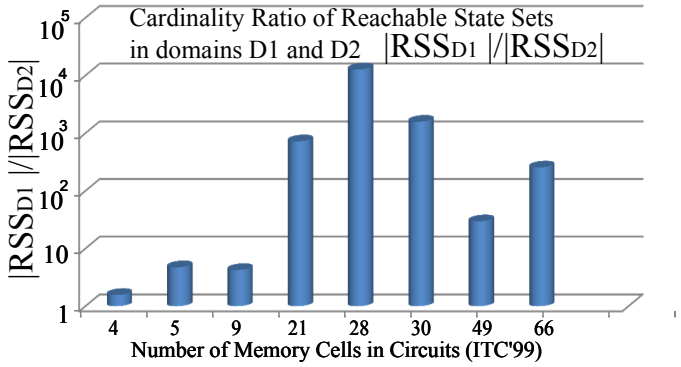


Figure 5: Logic Domain Comparison: Size Ratio of RSS.

Table 6: Time and memory resources to calculate the RSS.

		δ, sec	# iterations	seconds	# BDD nodes
b01	D_1	0.037	9	0.01	156
	D_2	0.037	6	0.01	78
	D_3	0.060	6	0.01	151
b02	D_1	0.020	9	0.005	81
	D_2	0.020	9	0.04	66
	D_3	0.024	9	0.01	127
b03	D_1	0.42	17	2.53	1506
	D_2	0.44	7	0.28	311
	D_3	875.670	7	235.13	668
b06	D_1	0.044	8	0.024	473
	D_2	0.052	6	0.018	130
	D_3	0.056	6	0.02	256
b08	D_1	0.364	40	3.14	27813
	D_2	0.356	5	0.02	324
	D_3	41.49	5	48.08	1222
b09	D_1	31.332	32	27.57	2919
	D_2	0.852	20	1.04	446
	D_3	>1000	-	-	-

for circuits of around 30 memory cells. The logic domain D_2 allows the analysis (with input and output interfaces) of much larger circuits, up to 100 cells.

In order to evaluate the benefits of our analysis, TMR has been applied to the benchmarks with the minimized set of voters. The inserted voters are triplicated following the practice in the existing industrial tools to avoid a single-point of failure and to protect against SETs. The

Table 7: Frequency and area gain of optimized *vs* full TMR.

	TMR circuit	voters	MHz	gain	hw	gain
Data Flow I.	Pipel.FP.Mult.8x8	121	60.5		2338	
	Optimized	0	71.0	17.4%	1831	21.7%
	Pipel.log.un.	41	128.3		693	
	Optimized	0	184.1	43.5%	447	35.5%
	Shift/Add.Mult.8x8	28	106.0		537	
	Optimized	8	108.0	1.9%	408	24.0%
Control Flow Intensive	*b*01 Flows Compar.	5	162.6		126	
	Optimized	3	162.6	0%	114	9.5%
	*b*02 BCD recogn.	4	181.9		69	
	Optimized	2	206.6	13.6%	60	13.1%
	*b*03 Resourc.arbiter	30	81.6		594	
	Optimized	17	109.0	33.6%	576	3.0%
	*b*06 Interrupt Hand.	9	144.8		168	
	Optimized	3	144.8	0%	134	20.2%
	*b*08 Inclus.detect.	21	115.4		484	
	Optimized	0	142.4	23.4%	216	55.4%
	*b*09 Serial Convert.	28	89.4		584	
	Optimized	20	95.0	6.3%	565	3.3%

final circuits have been synthesized with *Synplify Pro* with no optimization applied (Resource Sharing, FSM Optimization, etc.). As a case study, we have chosen Flash-based ProASIC3 FPGA as a synthesis target. Its configuration memory is immune to soft-errors[5] and data memory is protected with voters. Table 7 compares the size and maximum frequency of the circuit with full TMR (*i.e.,* voters after each FF) versus TMR with the optimized number of voters. The gains are presented in terms of the required FPGA hardware Core Cells (*hw* column) and maximum synthesizable frequency (*MHz* column). The gain in the maximum frequency depends on the location of the removed voters (in the circuit critical path or not). The reduction in area directly depends on the number of suppressed voters (up to 55%).

8 Application to time-redundancy

In this article, we have only considered hardware redundancy (TMR) but our approach also applies to time redundancy. Time-redundant schemes mask errors by voting on re-computed data. Such schemes reuse the combinational part of the circuit and have a much lower hardware overhead. We present one of our time redundant techniques [14] and sketch how our minimization analysis can be used in that context.

The Triple-Time Redundant Transformation (TTR) [14] takes a sequential circuit and returns a triple time-redundant version fault-tolerant to SETs. The transformed circuit is obtained by substituting each original memory cell with a sub-circuit, called a *voting memory block* (Figure 6).

These blocks introduce five cells ($d, d', d''d, kA, kB$) and two voters (*votA*, *votB*) to record and vote on the recomputed bits. Two new control wires *fA* and *fB* are used to organize voting; they are set by a small centralized FSM (the *control block* not shown here). The former input and output of the replaced memory cell are denoted by *si* and *so*. The input stream must be upsampled (x3) (*e.g.,* from $i_1i_2...$ to $i_1i_1i_1i_2i_2i_2...$) and the transformed circuit produces an upsampled (x3) output stream (*e.g.,* from $o_1o_2...$ to $o_1o_1o_1o_2o_2o_2...$).

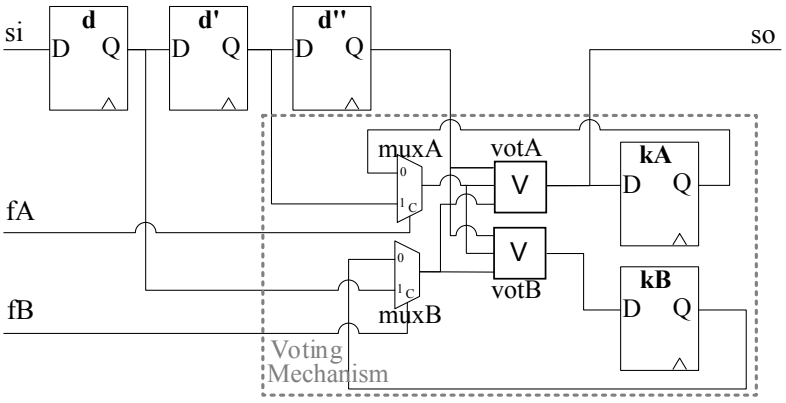


Figure 6: TTR voting memory block.

Consider a memory block reading three redundant copies of some bit b_1 re-computed by the combinational circuit. During the first cycle, b_1 is saved in d whereas, in the second cycle, it propagates to d' and the second copy of b_1 is saved in d. After three cycles, the cells d, d', d'' contain the three copies of b_1. During the fourth cycle, the following bit (*e.g.,* b_2) is computed and saved in d while the control block sets $fA = fB = 1$ to make the three b_1's propagate through the voters *votA* and *votB* for error masking and to the output *so*. To support triple-time redundancy, the vote on the three redundant b_1's must be performed two more times. Since the next redundant bit b_2 is filling d, d', d'', the kA and kB cells fetch the result of the first vote on b_1 (from d, d', d''). The vote in the next two cycles are performed on the values of cells d'', kA and kB by setting the control wires $fA = fB = 0$. When d, d', d'' are filled by the three copies of the next redundant bit b_2 the procedure repeats: they are voted (cycle 1: $fA = fB = 1$) and fetched by kA and kB to allow voting during the next two cycles (cycles 2-3: $fA = fB = 0$).

The advantage of TTR is to trade-off throughput for a low hardware overhead. For the benchmarks used in Section 7, TTR circuits are 1.6 to 2.1 times smaller than their TMR alternatives; they are also three times slower. The TTR scheme can be refined to allow dynamically changes of the level of time redundancy [17] (*e.g.,* to use redundancy only in critical situations). Further, combining dynamic redundancy with micro-checkpointing it is possible to mask faults with only double-time redundancy [16].

As in full TMR, voters are introduced for each original cell to prevent error accumulation and some of them may be useless. The same criteria apply in this context: if our voter minimization analysis suppresses the voters after memory cells $\{M\}$ in a TMR circuit and guarantees its tolerance *w.r.t.* $SET(1, K)$, the same analysis can also suppress the voting mechanisms in the TTR memory blocks that correspond to cells $\{M\}$ in the TTR circuit. The resulting optimized TTR circuit is guaranteed to be tolerant *w.r.t.* $SET(1, 3K + 1)$. Only the voting mechanism (the lower right part in Figure 6) is suppressed. The d,d',d'' chain remains to record the three recomputed bits and to propagate them to *so* but without error-masking.

Again, the simplest example is a pipelined architecture where all voters (except at the primary

outputs) are suppressed by our analysis. If a pipeline has n stages and each original memory cell is replaced by a memory block without voting, the number of stages becomes $3n$. With an upsampled input (x3) such circuit is tolerant *w.r.t.* $SET(1, 3n + 1)$ because any erroneous bit reaches the primary outputs within at most $3n + 1$ cycles where they are masked. For the 4-stage 32-bits floating-point multiplier [1], the tolerable fault-model is weakened from $SET(1, 4)$ to $SET(1, 13)$ but the overall TTR circuit size is reduced by 37%.

Our voter minimization approach might also be used to optimize the refined time-redundant schemes [16] and [17] but this application needs further investigation.

9 Related work

Existing industrial tools for applying TMR into FPGA protect against both kinds of soft error, SEUs and SETs. They include the Xilinx XTMR tool [11, 22], BYU/Los Alamos National Laboratory B-TMR [39], Synopsys's Synplify Premier [45], and Mentor Graphics Precision Hi-Rel [25]. In these tools, TMR is applied to circuit parts chosen by the user and, thus, the resulting circuits might not be fault-tolerant unless voters are inserted after *each* memory cell and primary circuit outputs. [25] proposes a protection technique against SEUs that requires only memory cells triplication with a majority voter insertion. But this approach relies on the assumption that only memory cells are influenced by radiation particles and that no signal perturbations in a combinatorial circuit occur. Thus, unlike our technique, the technique of [25] protects only against SEUs and not against SETs.

While our static analysis uses exclusively logical masking to tolerate transient errors, many other works rely on electrical and latching-window properties of hardware to estimate the chance that errors will not manifest in failures. This is the primary reason why a good part of research on voter insertion, Selective Triple-Modular Redundancy (STMR), and partial hardware redundancy mainly focus on probabilistic approaches [4, 34, 7, 42]. Contrary to our approach, they are not interested in formal guarantees that the final circuit tolerates a fault-model. [34] shows how selective voter insertion minimizes the negative timing impact of TMR. In [40], probabilities are used to apply TMR on selected portions of the circuit (STMR). In [42], STMR of combinational circuits specifies input interfaces using input signal probabilities. The main advantage of STMR over TMR is that the area of the STMR circuit is roughly two-thirds of the area of the TMR circuit. An original probabilistic-based idea is given in [33] that allows a certain level of degradation in output correctness in order to optimize TMR at a Data Flow Graph (DFG) abstraction level. While this technique is originally dedicated to heterogenous systems, it could be applied to Digital Signal Processing (DSP) hardware as well. Since the proposed methods are probabilistic, some errors may propagate to primary outputs. In our approach, the circuit is guaranteed to mask *all* possible errors of the considered fault model.

Other works use model-checking to guarantee user-defined fault-tolerance properties [43, 8]. [43] investigates which memory cells in SpaceWire node have to be protected so that even under an SEU occurrence the circuit keeps its functional properties, expressed as 39 assertions in linear temporal logic. If a cell is protected (fabricated with a special technology), an SEU cannot corrupt it. On the other hand, a protected cell consumes more power than a non-protected memory cell. As a result of verification-guided replacement of protected cells by their non-protected alternatives, a 4.45X reduction in power has been achieved. The work [8] formally proves that some system properties of ATM controller are kept if an SEU happens. The authors evaluate the probability to obtain the expected property under faults.

Another group of formal studies investigates sequential circuit robustness symbolically [9, 28] or by interpolation [18]. Since robustness is introduced probabilistically these work combine both

formal and probabilistic worlds.

While the aforementioned formal studies do not address voter minimization, their approaches to fault-tolerance and robustness are related to our work.

It is worth to notice that the introduced reachability analysis with multi-value encoding can be also interpreted within the well-known tainting dataflow-based analysis [24] and path sensitisation theories [19]. The former assigns a security-related mark to each information bit and tracks its propagation, just like we tag some bits as erroneous. The later approaches check if there is a path so that a signal change along that path alters the output. In our case, the signal change corresponds to an error injection, *e.g.,* a bit-flip, and we check whether this change can propagate to corrupt the output.

10 Conclusion

We proposed a logic-level verification-guided approach to minimize the number of voters in TMR circuits that guarantees a user-defined fault-model to be masked. Our approach is based on reachable state set computations and input/output interface specifications. In order to avoid analyzing the triplicated circuit, we introduced three logic domains, which allowed us to perform the analysis on a single copy of the circuit. Our analysis shows that some voters are useless and can be safely removed from the TMR application. We have used as case studies several arithmetic circuits as well as the benchmark suite *ITC'99*. They show that our technique allows not only a significant reduction in the amount of hardware resources (up to 35% for data flow intensive circuits and up to 55% for control flow intensive ones), but also a significant increase in the clock rate, compared to the full TMR method that inserts a voter after each memory cell.

We demonstrated that the choice of the logic domain influences the scalability of the analysis and its precision. We considered both SEU and SET fault-models and explained the modeling methodogy. As the experimental results show, the same level of optimization can be reached for both fault-models, but the SET model implies a potentially large number of corruption combinations to be examined, which can cause an analysis bottleneck.

Further research directions include the application of our approach to other fault-tolerance techniques, taking into account other optimization criteria like frequency and making the approach modular. We review these three topics in turn.

Frequency maximization

Voters ordering, discussed in Section 3.2, could also take into account other optimization criteria than voter minimization. For instance, we may increase the maximum synthesizable frequency by removing first the voters on the critical path. However, removing a voter from the critical path may make another path critical. Thus, the choice of the next voter to remove depends not only on the existing ordering but also on the current critical path. However, the critical path strategy may not result in the minimal number of voters. In this sense, the two criteria "number of voters" and "synthesizable frequency" are orthogonal, and bi-criteria optimization must be studied.

Modularity

Applying our analysis in a modular manner can increase its scalability and, consequently, the applicability of the proposed technique to larger circuits. The hierarchical compositional design of today's circuits makes it natural to decompose a circuit to the IPs of its block-by-block structure. Such structural partitioning requires the deep design understanding and has already been used

in the model checking of Intel CPUs [6]. In our case, the presented analysis can be applied to circuit sub-components after the decomposition. After the minimization of internal voters in each sub-circuit, the components should be interconnected again to rebuild the whole design. However, the interconnection wires should include voters to guaranty the fault-tolerance property of the final optimized circuit. Such an approach is not optimal even if the local input/output specifications are precise, because some of the interconnection voters may be redundant. Only a global analysis on the blocks containing such wire with a voter (as an input or output wire) can safely remove interconnection voters.

If a decomposition in sub-circuits is not known, the circuit netlist has to be automatically divided and the input-output specifications of its parts have to be found. These steps by themselves present complex tasks and require deep investigation. Here, we sketch some preliminary ideas about how these problems can be solved. First, a circuit netlist can be separated according to some syntactic criteria, *e.g.,* the circuit cuts should be performed at wires that are included in the biggest number of sequential loops. Such an approach eliminates as many sequential loops as possible by reducing the number of sequential loops in each sub-component. It limits the number of potential points where the voters have to be inserted.

After the circuit decomposition, our semantic analysis can be applied to each of its sub-parts. The main difficulty lies in the identification of input/output specification of each sub-circuit to perform the local semantic analyses. Figure 7 presents three cases of the circuit separation: *a)* sequential, *b)* parallel, and *c)* feedback decomposition.

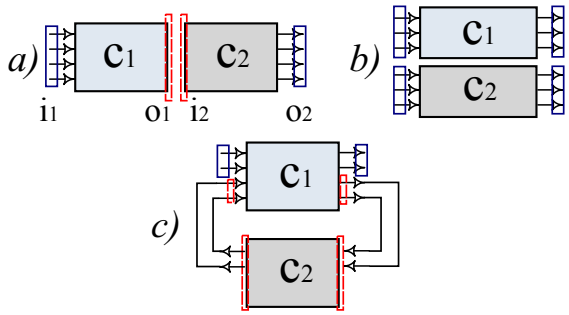


Figure 7: *a)* sequential, *b)* parallel, and *c)* feedback circuit decomposition.

While the input/output specification for c_1 and c_2 sub-circuits can be extracted from the global specification in the parallel decomposition (case b, Figure 7), the sequential and feedback decompositions (cases a and c) create unknown internal specifications (marked in red). They have to be found for each sub-part. Consider, for instance, the unknown input specification i_2 for the sequential decomposition (case a). The signals in i_2 are the outputs o_1. Since the netlist c_1 and its input specification i_1 fully describe the behavior of c_1, o_1 and i_2 can be described by the same NBA. In the worst case, such NBA could be as big as c_1 multiplied by the size of i_1, which can be prohibitive for the following semantic analysis of c_2 sub-circuit. Consequently, the extracted NBA should be over-approximated to lower the complexity. Naturally, the over-approximation may influence the precision of the further semantic voter minimization in c_2. The feedback decomposition is even more complex because of the mutual dependency between sub-components c_1 and c_2.

These modularity issues are complex but important and valuable since many other static analyses of circuits could benefit from them.

References

[1] *Open Source Hardware IPs: OpenCores project*, Michael Dunn- Logarithm Unit; Launchbird Design Systems, Inc.-Floating Point multiplier.

[2] CUDD: CU Decision Diagram Package, release 2.5.0. `http://vlsi.colorado.edu/~fabio/CUDD/`. Accessed: 2014-09-01.

[3] MLCUDDIDL: An OCaml interface for the CUDD BDD library. `http://pop-art.inrialpes.fr/~bjeannet/mlxxxidl-forge/mlcuddidl/index.html`. Accessed: 2014-09-01.

[4] *Proceedings 2003 International Test Conference (ITC 2003), Breaking Test Interface Bottlenecks, 28 September - 3 October 2003, Charlotte, NC, USA*. IEEE Computer Society, 2003.

[5] Neutron-induced single event upset SEU. *Microsemi Corporation*, (55800021-0/8.11), August 2011.

[6] M. Aagaard, R. Jones, and C.-J. Seger. Formal verification using parametric representations of boolean constraints. In *Design Automation Conference (DAC)*, pages 402–407, 1999.

[7] B. B. Alagoz. Fault masking by probabilistic voting. *OncuBilim Algorithm And Systems Labs*, 9(1), 2009.

[8] S. Baarir, C. Braunstein, et al. Complementary formal approaches for dependability analysis. In *IEEE Int.Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 331–339, 2009.

[9] S. Baarir et al. Feasibility analysis for MEU robustness quantification by symbolic model checking. In *Proceedings in Formal Methods of Software Design*, 2011.

[10] A. Bogorad et al. On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance. *IEEE Trans. on Nuclear Science*, pages 2804–2806, 2011.

[11] B. Bridgford, C. Carmichael, and C. W. Tseng. Single-event upset mitigation selection guide. *Xilinx Application Note XAPP987*, 1, 2008.

[12] P. Brinkley, P. Avnet, and C. Carmichael. SEU mitigation design techniques for the XQR4000XL. 2000.

[13] S. P. Buchner and M. P. Baze. Single-event transients in fast electronic circuits. *IEEE NSREC Short Course*, pages 1–105, 2001.

[14] D. Burlyaev. Design, optimization, and formal verification of circuit fault-tolerance techniques. *PhD thesis Joseph Fourier University/INRIA*, November 2015.

[15] D. Burlyaev, P. Fradet, and A. Girault. Verification-guided voter minimization in triple-modular redundant circuits. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014.

[16] D. Burlyaev, P. Fradet, and A. Girault. Automatic time-redundancy transformation for fault-tolerant circuits. *International Symposium on Field-Programmable Gate Arrays*, pages 218–227, February 2015.

[17] D. Burlyaev, P. Fradet, and A. Girault. Time-redundancy transformations for adaptive fault-tolerant circuits. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8, 2015.

[18] G. Cabodi and S. Singh, editors. *Complete and Effective Robustness Checking by Means of Interpolation*. Formal Methods in Computer-Aided Design (FMCAD), 2012.

[19] A. C. L. Chiang, I. S. Reed, and A. V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Trans. Computers*, 21(2):189–195, 1972.

[20] E. M. Clarke, J. R. Burch, O. Grumberg, D. E. Long, and K. L. McMillan. Mechanized reasoning and hardware design. chapter Automatic Verification of Sequential Circuit Designs, pages 105–120. 1992.

[21] F. Corno, M. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *Design Test of Computers*, pages 44–53, 2000.

[22] X. Corporation. Xilinx TMRTool. Product brief. 2006.

[23] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[24] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[25] R. D. Do. New tool for FPGA designers mitigates soft errors within synthesis. December 2011.

[26] P. Dodd, M. Shaneyfelt, J. Schwank, and G. Hash. Neutron-induced soft errors, latchup, and comparison of SER test methods for SRAM technologies. *International Electron Devices Meeting*, pages 333–336, 2002.

[27] G. Even, J. S. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In *Proc. 4th Int. Conf. on Int. Prog. and Combinatorial Opt.*, pages 14–28, 1995.

[28] G. Fey, A. Sülflow, and R. Drechsler. Computing bounds for fault tolerance using formal techniques. In *Proceedings of the 46th Design Automation Conference, DAC*, pages 190–195, 2009.

[29] S. Habinc. Functional triple modular redundancy FTMR. *European Space Agency Contract Report*, (FPGA-003-01), December 2002.

[30] K. Hass and J. Ambles. Single event transients in deep submicron CMOS. In *42nd Midwest Symposium on Circuits and Systems*, pages 122–125 vol. 1, 1999.

[31] J. P. Hayes, I. Polian, and B. Becker. An analysis framework for transient-error tolerance. In *25th IEEE VLSI Test Symposium (VTS 2007), 6-10 May 2007, Berkeley, California, USA*, pages 249–255, 2007.

[32] T. Heijmen. Soft-error vulnerability of sub-100-nm flip-flops. *14th IEEE Int.On-Line Testing Symposium*, pages 247–252, 2008.

[33] T. Imagawa, H. Tsutsui, H. Ochi, and T. Sato. A cost-effective selective tmr for heterogeneous coarse-grained reconfigurable architectures based on dfg-level vulnerability analysis. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 701–706, March 2013.

[34] J. M. Johnson and M. J. Wirthlin. Voter insertion algorithms for FPGA designs using triple modular redundancy. In *FPGA*, pages 249–258, 2010.

[35] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 43:85–103, 1972.

[36] S. Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

[37] H. T. Nguyen and Y. Yagil. A systematic approach to SER estimation and solutions. *Proc. Int. Reliability Physics Symp.*, pages 60–70, April 2003.

[38] I. Polian, B. Becker, M. Nakasato, S. Ohtake, and H. Fujiwara. Low-cost hardening of image processing applications against soft errors. In *21th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2006), 4-6 October 2006, Arlington, Virginia, USA*, pages 274–279, 2006.

[39] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving FPGA design robustness with partial TMR. *IEEE International Reliability Physics Symposium*, pages 226–232, 2006.

[40] O. Ruano, P. Reviriego, and J. Maestro. Automatic insertion of selective TMR for SEU mitigation. *European Conference on Radiation and its Effects on Components and Systems*, pages 284 – 287, 2008.

[41] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of CAD-93*, pages 42–47, 1993.

[42] P. Samudrala et al. Selective triple modular redundancy based single-event upset tolerant synthesis for FPGAs. *IEEE Transactions on Nuclear Science*, pages 284 – 287, October 2004.

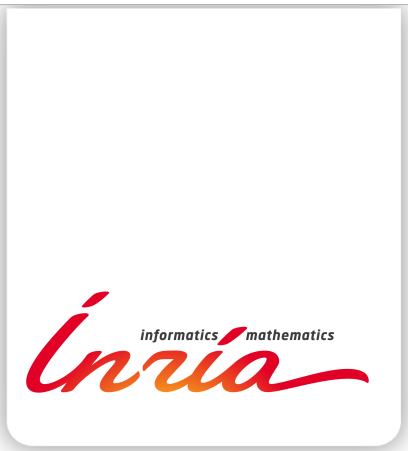
[43] S. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *DATE '07*, pages 1–6, 2007.

[44] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[45] A. Sutton. Creating highly reliable FPGA designs. *Military&Aerospace Technical Bullentin*, Issue 1:5–7, 2013.

[46] J. von Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies, Princeton Univ. Press*, pages 43–98, 1956.

[47] J. Ziegler et al. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399