Formal Verification of Automatic Circuit Transformations for Fault-Tolerance

Dmitry Burlyaev

Univ. Grenoble Alpes; INRIA
dmitry.burlyaev@inria.fr

Pascal Fradet

INRIA; Univ. Grenoble Alpes
pascal.fradet@inria.fr

***Abstract*—We present a language-based approach to certify fault-tolerance techniques for digital circuits. Circuits are expressed in a gate-level Hardware Description Language (HDL), fault-tolerance techniques are described as automatic circuit transformations in that language, and fault-models are specified as particular semantics of the HDL. These elements are formalized in the Coq proof assistant and the properties, ensuring that for all circuits their transformed version masks all faults of the considered fault-model, can be expressed and proved. In this article, we consider Single-Event Transients (SETs) and fault-models of the form “at most 1 SET within k clock cycles”. The primary motivation of this work was to certify the Double-Time Redundant Transformation (DTR), a new technique proposed recently [1]. The DTR transformation combines double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers. That intricacy requested a formal proof to make sure that no single-point of failure existed. The correctness of DTR as well as two other transformations for fault-tolerance (TMR & TTR) have been proved in Coq.**

I. Introduction

Circuit tolerance towards soft (non-destructive, non-permanent) errors has become a design characteristic as important as performance and power consumption [2]. The increased risk of soft errors results from the continuous shrinking of transistor size that makes components more sensitive to radiation [3].

The most widely-used methods to make circuits fault-tolerant rely on hardware redundancy. Triple-Modular Redundancy (TMR) [4] remains the most popular technique along with Finite State Machine (FSM) encoding (one hot, hamming, *etc.*). Some more complex ones are based on time-redundancy (re-execution) [1], [5], [6]. All these techniques can be realized through automatic circuit transformations and some of them are already supported by CAD tools. Since fault-tolerance is typically used in critical domains (aerospace, nuclear power, etc.), the correctness of such transformations is essential. If there is little doubt about the correctness of simple transformations such as TMR, this is not the case for more intricate ones.

The overall correctness of an automatic circuit transformation for fault tolerance consists not only in its functional correctness when no soft errors occur but also in its proper behavior under error occurrences. Widely-used post-synthesis verification tools (*e.g.,* model checking) are simply inappropriate to prove that a transformation ensures some property for all possible circuits; only proof-based approaches are suitable.

We propose an approach using the Coq proof assistant [7] to formally verify the functional and fault-tolerance properties of circuit transformations. We define the syntax and semantics of a simple gate-level functional HDL to describe circuits. Fault models, that specify the kind and occurrences of faults to be masked, are formalized in the language semantics. In this paper, we focus on SETs and fault-models of the form “at most 1 SET every k cycles”. Fault-tolerance transformations are defined as recursive functions on the syntax of the language. Proofs rely mainly on relating the execution of the source circuit without faults to the execution of the transformed circuit *w.r.t.* the considered fault-model. They make use of several techniques (case analysis, induction on the type or the structure of circuits, co-induction on input streams).

While our approach is general, it has been originally developed to prove DTR, an involved transformation combining double-time redundancy, micro-checkpointing, rollback, several execution modes and input/output buffers [1]. If manual checks were quite useful to develop that transformation, they were error-prone and not convincing enough. This transformation served as an advanced case study. The correctness of DTR as well as two other transformations (among which TMR) have all been proved in Coq.

Section II introduces the syntax and semantics of our gate-level HDL. In section III, we present the specification of fault-models in the language formal semantics. Section IV explains the proof methodology adopted to show the correctness of circuit transformations. It is illustrated by examples taken from the simplest transformation: TMR. Section V introduces the DTR circuit transformation [1] and sketches the associated proofs. Section VI presents related work, summarizes our contributions and suggests a few extensions.

Throughout this article, we use standard mathematical and semantic notations. The corresponding Coq specifications and proofs are available online [8].

II. Circuit Description Language

We describe circuits at the gate level using a purely functional language inspired from Sheeran’s combinator-based languages such as μFP [9] or Ruby [10]. We equip our language with dependent types which, along with the language syntax, ensure that circuits are well-formed by construction (gates correctly plugged, no dangling wires, no combinational loops, . . .).

Contrary to μFP or Ruby, our primary goal is not to make the description of circuits easy but to keep the language as simple and minimal as possible to facilitate formal proofs. Our language contains only 3 logical gates, 5 plugs and 3 combining forms. It is best seen as a low-level core language used as the object code of a synthesis tool. We denote it as LDDL (Low-level Dependent Description Language).

A. Syntax of LDDL

A *bus* of signals is described by the following type

$$B := \omega \mid (B_1 * B_2)$$

A bus is either a single wire (ω) or a pair of buses. In other terms, buses are defined as nested pairs. The constructors of LDDL annotated with their types are gathered in Fig. 1. A circuit takes as parameters its input and output types and is either a logic gate, a plug, or composition of circuits.

Gates

$$\textsc{not} : \mathsf{Gate}\ \omega\ \omega \qquad \textsc{and}, \textsc{or} : \mathsf{Gate}\ (\omega * \omega)\ \omega$$

Plugs

$$\begin{array}{lll} \textsc{id} & : & \forall \alpha, \mathsf{Plug}\ \alpha\ \alpha \\ \textsc{fork} & : & \forall \alpha, \mathsf{Plug}\ \alpha\ (\alpha * \alpha) \\ \textsc{swap} & : & \forall \alpha\ \beta, \mathsf{Plug}\ (\alpha * \beta)\ (\beta * \alpha) \\ \textsc{lsh} & : & \forall \alpha\ \beta\ \gamma, \mathsf{Plug}\ ((\alpha * \beta) * \gamma)\ (\alpha * (\beta * \gamma)) \\ \textsc{rsh} & : & \forall \alpha\ \beta\ \gamma, \mathsf{Plug}\ (\alpha * (\beta * \gamma))\ ((\alpha * \beta) * \gamma) \end{array}$$

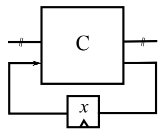
Circuits

$$\begin{array}{rlll} C ::= & \textit{Gates} & & \\ \mid & \textit{Plugs} & & \\ \mid & C_1 \multimap C_2 & : & \forall \alpha\ \beta\ \gamma, \mathsf{Circ}\ \alpha\ \beta \to \mathsf{Circ}\ \beta\ \gamma \\ & & & \quad \to \mathsf{Circ}\ \alpha\ \gamma \\ \mid & [\![C_1, C_2]\!] & : & \forall \alpha\ \beta\ \gamma\ \delta, \mathsf{Circ}\ \alpha\ \gamma \to \mathsf{Circ}\ \beta\ \delta \\ & & & \quad \to \mathsf{Circ}\ (\alpha * \beta)\ (\gamma * \delta) \\ \mid & \boxed{x}\!\!-\!C & : & \forall \alpha\ \beta, \mathsf{bool} \to \mathsf{Circ}\ (\alpha * \omega)\ (\beta * \omega) \\ & & & \quad \to \mathsf{Circ}\ \alpha\ \beta \end{array}$$

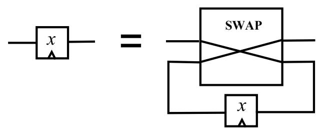
Fig. 1: LDDL Syntax

The sets of logical gates and plugs are minimal but expressive enough to specify any combinational circuit. The type of AND and OR, $\mathsf{Gate}\ (\omega{*}\omega)\ \omega$, indicates that they are gates taking a bus made of two wires and returning one wire. Likewise, NOT has type $\mathsf{Gate}\ \omega\ \omega$. Plugs, used to express (re)wiring, are polymorphic functions that duplicate or reorder buses: ID leaves its input bus unchanged, FORK duplicates its input bus, SWAP inverts the order of its two input buses, LSH and RSH reorder their three input buses.

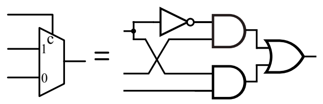
Circuits are either a gate, a plug, a sequential composition ($. \multimap .$), a parallel composition ($[\![., .]\!]$), or a composition with a cell (flip-flop) within a feedback loop ($\boxed{.}\!\!-\!.$). The typing of the sequential operator ensures that the output bus of the first circuit has the same type as the input bus of the second one. The typing of the parallel operator expresses the fact that the inputs (resp. outputs) of the resulting circuit is made of the inputs (res. outputs) of the two sub-circuits. The last operator (related to the μ operator of μFP) is the only way to introduce feedback loops in the circuit. $\boxed{x}\!\!-\!C$ is better seen graphically as the circuit



The circuit C can have any input/output bus but it also takes and returns a wire connected to a memory cell set to the Boolean value x (tt or ff). The main advantage of that operator is to ensure that any loop contains a cell. It prevents combinational loops by construction. Of course, it does not force all cells to be within loops. A simple cell without feedback is expressed as $\boxed{x}\!\!-\!\textsc{swap}$:



To illustrate the language, a multiplexer



can be expressed in LDDL as the expression

$$[\![\textsc{fork}, \textsc{id}]\!] \multimap \textsc{lsh} \multimap [\![\textsc{not}, \textsc{rsh} \multimap \textsc{swap}]\!] \multimap \textsc{rsh}$$
$$\multimap [\![\textsc{and}, \textsc{and}]\!] \multimap \textsc{or}$$

As common with low-level or assembly-like languages, LDDL is quite verbose. Recall that it is not meant to be used directly. It is best seen as a back-end language produced by synthesis tools. On the other hand, it is simple and expressive; its dependent types make inputs and outputs of each sub-circuit explicit and ensure that all circuits are well-formed.

B. Semantics of LDDL

From now on, to alleviate notations, we leave typing constraints implicit. All input and output types of circuits and corresponding buses always match.

The semantics of gates and plugs are given by functions denoted by $[\![.]\!]$. For instance, the semantics of ID, is the identity function ($[\![\textsc{id}]\!] x = x$) or the semantics of FORK is the function duplicating its bus argument ($[\![\textsc{fork}]\!] x = (x, x)$).

Taking into account errors (in particular, SETs) makes the semantics non deterministic. When a glitch produced by an SET reaches a flip-flop, it may be latched non-deterministically as tt or ff. Therefore, the standard semantics of circuits is not described as functions but as predicates. The second issue is the representation of a circuit state (*i.e.,* the current values of its cells). A solution could be to equip the semantics with an environment (*cell* $\to$ bool). We choose here to use the circuit itself to represent its state which is made explicit by the $\boxed{x}\!\!-\!C$ constructs.

The semantics of circuits is described by the inductive predicate step : $\mathsf{Circ}\ \alpha\ \beta \to \alpha \to \beta \to \mathsf{Circ}\ \alpha\ \beta$. The expression step $C\ a\ b\ C'$ can be read as "after one clock cycle, the circuit C applied to the inputs a may produce the outputs b and the new circuit (state) C' ". The rules are gathered in Fig. 2.

$$\text{Gates \& Plugs}\ \frac{[\![G]\!]a = b}{\mathsf{step}\ G\ a\ b\ G}$$

$$\text{Seq}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \qquad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{step}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\text{Par}\ \frac{\mathsf{step}\ C_1\ a\ c\ C_1' \qquad \mathsf{step}\ C_2\ b\ d\ C_2'}{\mathsf{step}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]}$$

$$\text{Loop}\ \frac{\mathsf{step}\ C\ (a, \mathsf{b2s}\ x)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{step}\ \boxed{x}\!\!-\!C\ a\ b\ \boxed{y}\!\!-\!C'}$$

Fig. 2: LDDL semantics for a clock cycle

Gates (or plugs) are stateless: they are always returned unchanged by step. The rules for sequential and parallel compositions are standard. The rule for $\boxed{x}\!\!-\!C$ makes use of the b2s function which converts the Boolean value of a cell into a signal and of the s2b predicate which relates a signal to a Boolean. The outputs and the new state (circuit) depend on the reduction of C applied to the inputs a and the signal corresponding to x. Non-determinism may come from the predicate s2b which relates a glitch to both tt and ff.

The complete semantics is given by a co-inductive predicate eval : Circ $\alpha\ \beta \rightarrow$ Stream $\alpha \rightarrow$ Stream β which describes the circuit behavior for any infinite stream of inputs.

$$\text{Eval}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{eval}\ C'\ is\ os}{\mathsf{eval}\ C\ (i : is)\ (o : os)}$$

If C applied to the inputs i returns after a clock cycle the outputs o and the circuit C' and if C' applied to the infinite stream of inputs is returns the output stream os then the evaluation of C with the input stream $(i : is)$ returns the stream $(o : os)$.

The variable-less nature of LDDL spares the semantics to deal with bindings and environments. It avoids many administrative matters (reads, updates, well-formedness of environments) and facilitates formalization and proofs.

III. Specification of Fault Models

There are two main types of soft errors caused by particle strikes: Single-Event Upsets (SEUs) (*i.e.,* bit-flips in flip-flops) and SETs (*i.e.,* glitches propagating in the combinational circuit). An SEU can be modeled by changing the value of an arbitrary memory cell between two clock cycles. In this article, we focus on SETs and fault-models allowing at most 1 SET within k clock cycles, written $SET(1,k)$. An SET in a combinational circuit can lead to the non-deterministic corruption of any memory cell connected (by a purely combinational path) to the place where the SET occurred. Since an SET may potentially lead to several bit-flips, the $SET(1,k)$ model subsumes $SEU(1,k)$. In order to model SETs, glitches and their propagation must be represented in the semantics. We use signals that can take 3 values: 0, 1, or a glitch written $\lightning$. We often abuse the notation and denote a wire by its signal value.

Glitches propagate through plugs and gates (*e.g.,* AND$(1,\lightning) = \lightning$) but can be also logically masked (*e.g.,* OR$(1,\lightning) = 1$ or AND$(0,\lightning) = 0$). If a corrupted signal is not masked, it is latched as tt or ff (both (s2b $\lightning$ tt) and (s2b $\lightning$ ff) hold).

The semantics of circuits for a cycle with an SET is represented as the inductive predicate stepg $C\ a\ b\ C'$ that can be read as "after one cycle with an SET occurrence, the circuit C applied to the inputs a may produce the outputs b and the new circuit/state C'". The main rules for stepg are gathered in Fig. 3.

$$\text{Gates}\ \frac{}{\mathsf{stepg}\ G\ a\ \lightning\ G}$$

$$\text{SeqL}\ \frac{\mathsf{stepg}\ C_1\ a\ b\ C_1' \qquad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\text{SeqR}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \qquad \mathsf{stepg}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\dots$$

$$\text{LoopC}\ \frac{\mathsf{stepg}\ C\ (a, b2s\ x)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\!\!-\!C\ a\ b\ \boxed{y}\!\!-\!C'}$$

$$\text{LoopM}\ \frac{\mathsf{step}\ C\ (a, \lightning)\ (b,s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\!\!-\!C\ a\ b\ \boxed{y}\!\!-\!C'}$$

Fig. 3: LDDL semantics with SET (main rules)

The rule (Gates) asserts that stepg introduces a glitch after a logical gate. The two rules for sequential composition represents two mutually exclusive cases where the SET occurs in left sub-circuit (SegL) or in the right one (SegR). The rule for the parallel operator is similar. The rule (LoopC) represents the case where an SET occurs inside C. The rule (LoopM) represents the case where an SET occurs at the output of the memory cell x which is taken as an input by C. To summarize, stepg introduces non-deterministically a single glitch after a cell or a logical gate. Hence, if a circuit has n gates and m cells, it specifies $n+m$ possible executions.

The fault-model $SET(1,k)$ is expressed by the predicate setk_eval : Nat $\rightarrow$ Circ $\alpha\ \beta \rightarrow$ Stream $\alpha \rightarrow$ Stream β:

$$\text{SetN}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (n-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ n\ C\ (i : is)\ (o : os)}$$

$$\text{SetG}\ \frac{\mathsf{stepg}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (k-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ 0\ C\ (i : is)\ (o : os)}$$

The first argument of setk_eval plays the role of a clock counter. A glitch can be introduced (by stepg) only if the counter is 0 (SetG). When a glitch is introduced, the counter is reset to enforce at least $k-1$ normal execution steps (SetN).

IV. Overview of Correctness Proofs

Describing the proofs in details is out of scope of this paper (and would be tiresome). Instead, we outline the common proof structure of the transformations we have studied. We illustrate the main steps using examples taken from the correctness proof of the simplest one: TMR.

Transformation

Each fault-tolerance technique is specified by a program transformation on the syntax of LDDL. They are all defined by induction of the syntax and replacement of each memory cell by a memory block (a small circuit). The TMR transformation takes a circuit of type Circ α β and returns a circuit of type Circ $((\alpha * \alpha) * \alpha)$ $((\beta * \beta) * \beta)$. Inputs/outputs are triplicated to play the role of the inputs/outputs of each copy.

$$\begin{array}{lcl} \text{TMR}(X) & = & [\![[\![X, X]\!], X]\!] \quad \textit{with } X \textit{ a gate/plug} \\ \text{TMR}(C_1 \multimap C_2) & = & \text{TMR}(C_1) \multimap \text{TMR}(C_2) \\ \text{TMR}([\![C_1, C_2]\!]) & = & \text{S}_1 \multimap [\![\text{TMR}(C_1), \text{TMR}(C_2)]\!] \multimap \text{S}_2 \\ \text{TMR}(\boxed{x}\!\!-\!C) & = & \boxed{x}\!\!-\!\boxed{x}\!\!-\!\boxed{x}\!\!-\!(\mathsf{vot} \multimap \text{TMR}(C) \multimap \text{S}_3) \end{array}$$

where S_1, S_2, S_3 are reshuffling plugs (*e.g.,* S_1 has type Plug $(((\alpha*\beta)*(\alpha*\beta))*(\alpha*\beta))$ $(((\alpha*\alpha)*\alpha)*((\beta*\beta)*\beta))$ and reshuffles the input bus accordingly). Each cell is replaced by three cells followed by a triplicated voter (vot) made of a majority voter for each copy.

Relations between source and transformed circuits

The correctness property relates the execution of the source circuit without fault to the execution of the transformed circuit under a fault-model. Most of the lemmas also relate the states and executions of the source and transformed circuits. These relations are expressed as inductive predicates.

For TMR, a key property is that an SET can corrupt only a single redundant copy and that such corruption stays confined in that copy. To express corruption, we use a predicate relating source and transformed programs expressed on the syntax of LDDL. The corruption of the first copy of a transformed circuit C^T *w.r.t.* to its source circuit C is expressed by the predicate $\overset{c}{\sim}_1$. The main rule is

$$\text{CLoop}\ \frac{C \ \overset{c}{\sim}_1\ C^T}{(\boxed{x}\!\!-\!C) \ \overset{c}{\sim}_1\ (\boxed{z}\!\!-\!\boxed{x}\!\!-\!\boxed{x}\!\!-\!(\mathsf{vot} \multimap C^T \multimap \text{S}_3))}$$

which states that if C is in relation with C^T and the second and third memory cells of the transformed circuit are the same as the cell of the source circuit, then $\boxed{x}\!\!-\!C$ and its transformed version are in relation. The other rules just check recursively this source/transformed circuit relationship. For instance, the rule for the parallel construct is

$$\text{CPar}\ \frac{C_1 \ \overset{c}{\sim}_1\ C_1^T \qquad C_2 \ \overset{c}{\sim}_1\ C_2^T}{([\![C_1, C_2]\!]) \ \overset{c}{\sim}_1\ (\text{S}_1 \multimap [\![C_1^T, C_2^T]\!] \multimap \text{S}_2)}$$

The same relations exist for other options of redundant copy corruption ($\overset{c}{\sim}_2$ and $\overset{c}{\sim}_3$) and for each possible corruption of the triplicated bus ($\overset{b}{\sim}_1$, $\overset{b}{\sim}_2$, $\overset{b}{\sim}_3$). In the following, we write $\overset{c}{\sim}$ for the relation $\overset{c}{\sim}_1 \vee \overset{c}{\sim}_2 \vee \overset{c}{\sim}_3$.

Key properties and proofs

Properties and their associated proofs can be classified as:

- properties "for all circuits" relating their source and transformed versions for a one cycle reduction. They are usually proved by a simple structural induction on the structure of LDDL expressions;
- similar properties but for known sub-circuits introduced by the transformations (*e.g.,* voters). They are proved by examining all possible cases of corruption or SET occurrences.
- properties about the complete (infinite) execution of source and transformed circuits. They are proved by co-induction on the stream of inputs.

The main lemmas state how the transformed circuit evolves when it is in a correct state and one SET occurs (stepg), or when it is in a corrupted state and it executes normally (by step). For TMR we have for instance:

$$\begin{array}{l} \mathsf{step}\ C_1\ a\ b\ C_2 \ \wedge\ \mathsf{stepg}\ \text{TMR}(C_1)\ (a,a,a)\ b3\ C_2^T \\ \Rightarrow\ \ C_2\ \overset{c}{\sim}\ C_2^T \end{array}$$

It can be read as: if C_1 reduces by step in C_2, and its transformed version $\text{TMR}(C_1)$ reduces by stepg in a circuit C_2^T, then C_2^T is the transformed version of C_2 with at most one corrupted redundant copy ($C_2\ \overset{c}{\sim}\ C_2^T$). In other terms, a glitch can corrupt only one of copies of the TMR circuit.

The following lemma

$$\begin{array}{l} C_1\ \overset{c}{\sim}\ C_1^T \ \wedge\ \mathsf{step}\ C_1\ a\ b\ C_2 \\ \Rightarrow\ \ \mathsf{step}\ C_1^T\ (a,a,a)\ (b,b,b)\ \text{TMR}(C_2) \end{array}$$

ensures that a corrupted transformed circuit comes back to a valid state after one normal reduction step.

The main correctness theorems state that for related inputs the normal execution of the source circuit and the execution (under the considered fault-model) of the transformed circuit give related outputs. A complete execution is modeled using infinite streams of inputs/outputs and the proof should proceed by co-induction.

The correctness of the TMR transformation is expressed as

$$\begin{array}{l} \mathsf{eval}\ C\ i\ o \ \wedge\ \mathsf{setk\_eval}\ 2\ \text{TMR}(C)\ n\ (\mathsf{tripl}\ i)\ o3 \\ \Rightarrow\ \ o\ \overset{s}{\sim}\ o3 \end{array}$$

TMR masks all faults of the fault-model $SET(1,2)$, so it tolerates an SET every other cycle. The stream of primary inputs for the transformed circuit is the input stream i where each element (bus) is triplicated (tripl i). The stream of primary outputs of the transformed circuit ($o3$: Stream $((\beta*\beta)*\beta)$) is a triplicated version of the output stream (o : Stream β) with at most one corrupted element in each triplet ($\overset{s}{\sim}$ relation). Indeed, the fault-model allows an SET to occur after the final voters. These SETs cannot be corrected internally but, since the outputs are triplicated, masking is still possible by voting in the surrounding circuit.

Practical issues

Taylor-made tactics had to be written for LDDL syntax and semantics. They helped to shorten and to automatize parts of the proofs.

All the transformations use known sub-circuits (*e.g.,* voters) and many basic properties must be shown on them. Such properties are often of the form

$$\text{Pstepg}\ \frac{P\ a \qquad \mathsf{stepg}\ C\ a\ b\ C'}{Q(a,b,C')}$$

with P and Q representing pre- and post-conditions, respectively. These properties on stepg entail to consider all possible SET occurrences. For TMR, which introduces triplicated voters, this can be done using standard proofs. The transformation DTR introduces much bigger sub-circuits, which would lead to very large proofs since dozens of different cases of SET need to be considered. Fortunately, Coq permits proofs by reflection which, in some cases, permits to replace manual proofs by automatic computations. We use largely this feature for known circuits. It amounts to

- define fstepg a functional version of stepg, which, for a given circuit and particular input, computes the set of the possible outputs and circuits in relation by stepg;
- prove that if $(b, C') \in (\mathsf{fstepg}\ C\ a)$ then stepg $C\ a\ b\ C'$;
- define (or generate) equivalent functional (Boolean) versions P_b and Q_b of the predicates P and Q.

Then, a proof by reflection of the property (Pstepg) proceeds by generating all possible inputs, then it filters them by P_b, executes fstepg on all elements of that set and, finally, checks that Q_b returns true on all results. In this way, reflection automatizes the exploration of all fault occurrences and most of the proof boils down to computations.

V. Correctness of Double-Time Redundancy

The initial motivation of this work was to certify DTR, an involved circuit transformation that we recently proposed [1]. Hereafter, we outline DTR and the main parts of its proof.

A. DTR Transformation Overview

The main assets of DTR are its much lower hardware overhead than TMR and its ability to mask SETs using double-time redundancy instead of a triple overhead (in time or in space). DTR uses double-time redundancy to detect errors and a micro-checkpointing and a rollback mechanisms to re-execute the faulty cycle for recovery. Since, according to the fault-model, no error can occur immediately after the last error, time-redundancy can be switched-off during the recovery phase to "accelerate" the circuit twice. Along with input and output buffers to record inputs and to produce delayed outputs, it makes errors and recovery absolutely transparent to the surrounding circuit. Error detection followed by the recovery to a correct state may take up to 9 clock cycles; therefore DTR masks errors from the $SET(1, 10)$ fault-model.

The DTR transformation consists of four parts (see Fig. 4):

1) substitution of each original memory cell with a *memory block* and threading of control wires within the circuit;

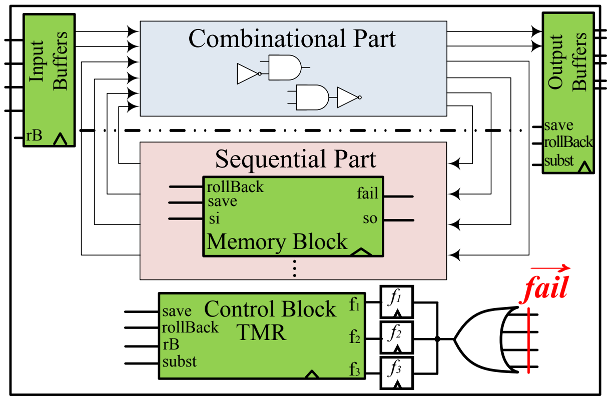


Fig. 4: Transformed circuit for DTR.

2) addition of a *control block*;
3) addition of *input buffers* to all circuit primary inputs;
4) addition of *output buffers* to all circuit primary outputs.

Further, the input stream should be upsampled twice in order to introduce enough redundancy in the transformed circuit to detect errors caused by SETs.

1) Memory blocks: Each original memory cell is substituted with a memory block (see Fig. 5) that stores the results of signal propagation through the combinatorial circuit along with recovery bits (or checkpoint bits). It consists of:

- two cells d and d' (the data bits) to save redundant information for comparison (with EQ) to detect errors; since the input stream is upsampled twice, d and d' normally contain the same value each every other cycle;
- two cells r and r' (the recovery bits) with enable-input to keep the value of the input during four clock cycles. If an error is detected in any memory block, a synchronous rollback occurs in all blocks. It retrieves correct values from r' cells and the circuit recovers from the erroneous state using a third recomputation.

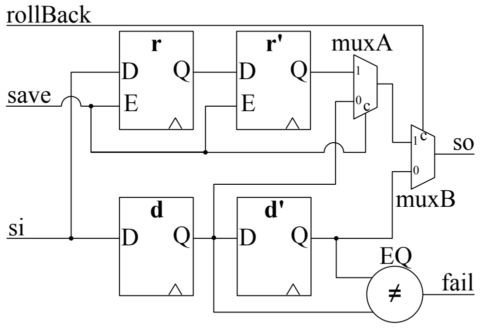


Fig. 5: DTR memory block.

2) Control block: When a memory block detects an error, the *fail* signal, going to the control block, is raised and latched in three error signaling cells (f_i in Fig. 4). Then, the control block emits a series of control signals to memory blocks (*e.g.,* $save$ and $rollBack$) to schedule rollback and recovery. Its functionality can be described as the FSM of Fig. 6. The states

0 and 1 compose the *normal* mode which raises alternatively the *save* signal used as an enable signal to organize a 4-cycle delay in the r-r' memory block cells. When an error is detected (*i.e.,* $f_i = 1$), the FSM enters the recovery mode for 4 cycles (states $2, 3, 4, 5$) and raises appropriate signals.

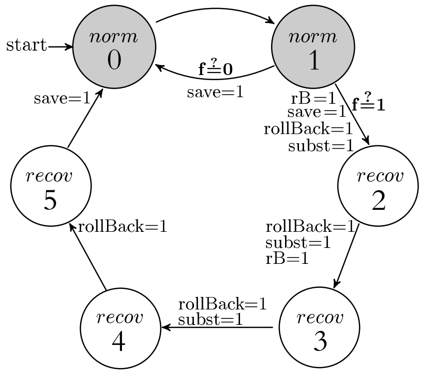


Fig. 6: FSM of the DTR control block: "$\overset{?}{=}$" denotes a guard, "$=$" an assignment and, by default, signals are set to 0.

During the recovery process, the control block switches-off double time redundancy speeding-up the circuit which, in a few cycles, catches up the state it should have had if no error had occurred. According to the fault-model, no error may occur immediately after the last error which allows us to perform such "acceleration". The control block itself is a small circuit protected against SETs using TMR.

3) Input/Output Buffers: To prevent disrupted input/output behavior during recovery and to guarantee transparency for the surrounding circuit, additional input and output buffers are necessary. They are inserted at each primary input and output of the original circuit. Input buffers keep the two last inputs which are only used for re-computation during recovery. Output buffers delay outputs (and introduce a two-cycles latency) in order to emit the previously recorded correct outputs during the recovery process. They are designed to be also fault-tolerant to any SET occurring inside or even at their outputs. To achieve this property, the primary outputs are triplicated in space. Buffers are controlled by the rB, $rollBack$, $subst$, $save$ signals. We refer the reader to [1] for a detailed presentation of their internal structure and behavior.

We illustrate the basic functionalities of DTR using a simple scenario where the upsampled stream $a\ a\ b\ b\ c\ c\ d\ d\ e\ e\ f\ f \ldots$ is sent to the circuit but a SET in the combinational part corrupts the first occurrence of b (written $\tilde{b}$). We use quadruplets like $(c, s, [d, d', r, r'], s)$ to denote the cycle number (c), the state of the control block (s) at the beginning of the cycle (see Fig. 6) and the state of the memory blocks (the values to be used as output are in bold). We start when the error is about to be detected: the four first values $a, a, \tilde{b}, b$ have been stored in memory blocks. The execution proceeds as follows:

$$\begin{array}{lll} (1, 0, [b, \tilde{\mathbf{b}}, b, a]); & (2, 1, [\tilde{c}, b, b, \mathbf{a}]); & (3, 2, [\mathbf{b}, \tilde{c}, b, b]); \\ (4, 3, [\mathbf{c}, b, b, b]); & (5, 4, [\mathbf{d}, c, b, b]); & (6, 5, [e, \mathbf{d}, b, b]); \\ (7, 0, [e, \mathbf{e}, e, b]); & (8, 1, [f, \mathbf{e}, e, b]); & (9, 0, [f, \mathbf{f}, f, e]) \end{array}$$

In cycle 1, the error is detected and the f_i cells (see Fig. 4) are set to 1. The value $\tilde{c}$ is read; it is potentially corrupted since it may be the result of the propagation of $\tilde{b}$ (which is corrupted) through the combinational circuit. In cycle 2, the control block goes in the recovery mode and performs the rollback (transition $1 \mapsto 2$, Fig. 6). It emits control signals that ensure that the values stored in the recovery bits r' (as well as those stored the input buffers) are used instead of the usual inputs for the third recomputation. The circuit is now in a speedup mode and double time redundancy is suspended. The cycles 2 to 4 are computed using the values a, b, c. The control signals entail that the read value, which is stored in d, is used as output in the next cycle. Then, double redundancy resumes and the recovery line r-r' retakes correct values in the next cycles. At the 9^{th} cycle, the state is exactly the state that would have been reached at the same 9^{th} cycle without error. During the recovery, input buffers also re-inject previous values synchronously with the memory blocks, and output buffers produce delayed correct outputs (see [1] for the complete description).

B. Formal Specification and Proofs

While TMR is a well-established transformation and its properties are doubtless, DTR is a novel and much more complex technique. Our goal was to ensure that no single point of failure existed: in particular, any SET in memory blocks, combinational logic, input or output buffers, control block, and control wires should be masked. The number of possible error scenarios is very large (about 10 cases each for memory block and output buffers). Moreover, the normal execution mode has a two-cycle period which doubles the number of corruption cases. Full confidence in DTR correctness for all possible circuits and errors can only be achieved with a formal proof-based approach.

Transformation

The core DTR transformation is defined very much like TMR as presented in Section IV. It takes an original circuit of type Circ α β and substitutes each memory cell with a memory block returning a circuit of type Circ $(\alpha * ((\omega * \omega) * \omega))$ $(\beta * ((\omega * \omega) * \omega))$. The three wires $((\omega * \omega) * \omega)$ correspond to the control signals $((save * rollBack) * fail)$ that propagate through all memory blocks. Input (resp. output) buffers are plugged to each primary input (resp. output) by recursion on the input type α (resp. output type β). Plugging input/output buffers and the control block to the transformed circuit returns a circuit of type Circ α $((\beta * \beta) * \beta)$. The triplicated output interface of type $((\beta * \beta) * \beta)$ represents the triplicated original output bus.

In the following, we write $\textsc{mb}(d, d', r, r', C)$ to denote a memory block with values d, d', r, r' (see Fig. 5) plugged to a circuit C.

Relations between source and transformed circuits

Most of the inductive predicates relating states and executions of the source and transformed circuits have several versions depending on the state of the control block $(0, 1, \ldots)$.

For instance, the predicate dtr0 expresses the relation between a transformed circuit and its source version(s) when the control block is in state 0. The state of a memory block is of the form $[y, y, y, x]$ where the values x and y are the two values taken successively by the corresponding cells of the source version. Therefore, the state of the transformed circuit is in relation with two successive source circuits. dtr0 is defined inductively in a similar way as $\overset{c}{\sim}$ predicates in Sec. IV. The main rule relates the memory block to the states of the two source circuits:

$$\frac{\mathsf{dtr0}\ C_0\ C_1\ C^T}{\mathsf{dtr0}\ (\boxed{x}\!-\!C_0)\ (\boxed{y}\!-\!C_1)\ \textsc{mb}(y, y, y, x, C^T)}$$

The memory block should be of the form ($d = d' = r = y$; $r' = x$) where x and y are values of the corresponding cells of the circuits $\boxed{x}\!-\!C_0$ and $\boxed{y}\!-\!C_1$, respectively. Those two circuits represent two successive states of the source circuit.

The corresponding predicate when the control block is in state 1 relates a transformed circuit to three successive source circuits. Indeed, in that state, the memory block is of the form $[z, y, y, x]$ where x, y and z are three successive values taken by the source circuit.

Several versions of these predicates are needed to represent the corruption cases. For instance, the predicate dtr1d expresses the relation between a transformed circuit whose d cells are potentially corrupted and its source version when the control block is in state 1. The main rule is:

$$\frac{\mathsf{dtr1d}\ C_0\ C_1\ C_2\ C^T}{\mathsf{dtr1d}\ (\boxed{x}\!-\!C_0)\ (\boxed{y}\!-\!C_1)\ (\boxed{z}\!-\!C_2)\ \textsc{mb}(w, y, y, x, C^T)}$$

that is, r', (resp. d' and r) should hold the same values are the first (resp. second) source circuit; d has no constraint (*i.e.,* can be corrupted). Other predicates are also needed to relate the source and transformed versions when the control block is in the recovery mode.

Key lemmas

Using the aforementioned predicates, we can define lemmas that show how the transformed circuit evolves with and without SETs. First, it can be shown that, initially, the transformed circuit is in relation with the source circuit *i.e.,*

$$\mathsf{dtr0}\ C\ C\ \textsc{dtr}(C)$$

Then, all cases of state evolution are covered. For instance, the following property for a reduction with no SET

$$\begin{aligned} \mathsf{dtr0}\ C_0\ C_1\ C^T\ &\Rightarrow\ \mathsf{step}\ C_1\ a\ b\ C_2\\ &\Rightarrow\ \mathsf{step}\ C^T\ \{a, \{0, 0, 0\}\}\ b'\ C'^T\\ &\Rightarrow\ b' = \{b, \{0, 0, 0\}\}\\ &\ \ \wedge\ \mathsf{dtr1}\ C_0\ C_1\ C_2\ C'^T \end{aligned}$$

states that, if the original circuit evolves from C_1 to C_2 with input a, then the corresponding transformed circuit C^T with input a and signals *save* $= 0$, *rollBack* $= 0$, and *fail* $= 0$ returns the same output b and the same global signals. Further, if C^T is related to (C_0, C_1) with dtr0, the returning state C'^T is related to (C_0, C_1, C_2) with dtr1.

Similarly, if the *rollBack* signal is corrupted (has a glitch), then memory blocks may output an incorrect value (wrongly selected by muxB, Fig.5). Since this value goes through the combinational circuit and may be fetched by memory blocks, their d values may be corrupted. This is formalized as

$$\begin{aligned} \mathsf{dtr0}\ C_0\ C_1\ C^T\ &\Rightarrow \mathsf{step}\ C_1\ a\ b\ C_2\\ &\Rightarrow \mathsf{step}\ C^T\ \{a, \{0, ↯, 0\}\}\ b'\ C'^T\\ &\Rightarrow \exists x, b' = \{x, \{0, ↯, 0\}\}\\ &\ \ \wedge \mathsf{dtr1d}\ C_0\ C_1\ C_2\ C'^T \end{aligned}$$

These properties are shown by simple structural induction. Similar properties on input and output buffers are proved using reflection. The proofs for the collection of input and output buffers plugged to the primary input and output buses are proved by induction of the input and output types. Proofs for the triplicated control block make a critical use of the main properties proved for the TMR transformation.

The property corresponding to a reduction by stepg of the whole transformed circuit proceeds by inspection of all cases of fault occurrences and application of the properties mentioned above. It can be shown that in all cases the transformed circuit returns to a correct state after less than 10 reduction steps after an SET.

Main theorem

The main correctness theorem is expressed as

$$\begin{aligned} &\mathsf{step}\ C_0\ a\ b\ C_1\\ \wedge\quad &\mathsf{step}\ \textsc{dtr}(C_0)\ a\ b_1\ C^T\ \wedge\ \mathsf{step}\ C^T\ a\ b_2\ C_1^T\\ \wedge\quad &\mathsf{eval}\ C_1\ i\ o\ \wedge\ \mathsf{setk\_eval}\ 10\ C_1^T\ n\ (\mathsf{upsampl}\ i)\ oo\\ \Rightarrow\quad &\mathsf{outDTR}\ (b, o)\ oo \end{aligned}$$

It assumes that no error occurs during the first two cycles (second line of the theorem). This is due to the arbitrary initialization of memory cells (buffers, memory blocks) performed by the transformation. Since the recovery bits are not properly set, a rollback and the following recovery would be incorrect. The stream of primary inputs of the transformed circuit (upsampl i) is the input stream i where each bit is repeated twice. The fault-model $SET(1, 10)$ is expressed by the predicate (setk_eval 10) that may use stepg at most once every 10 cycles (and uses step otherwise). The predicate outDTR relates the output stream (of type Stream β) produced by the source circuit to the output stream (*oo* of type Stream $(\beta * (\beta * \beta))$) of the transformed circuit. The two first values of the transformed stream are not meaningful since output buffers introduce a latency of two cycles. The predicate outDTR states that if the first stream has value a at position i, then the second stream will have a triplet with at least two a's at position $2 * i + 1$. We can guarantee the correctness of only two values because we allow an SET to occur even at the primary outputs.

VI. Conclusions

Many efforts have been devoted to the formal functional verification of circuits [11]. It is usually performed for specific circuits using model-checking or SAT solving techniques.

However, such an approach is inappropriate to prove the correctness of a synthesis or transformation tool for all possible circuits; theorem proving must be used instead.

Still, proof-assistants have been mostly used for functional circuit verification. Let us cite, among many others, the application of ACL2 to prove the out-of-order microprocessor architecture FM9801 [12], HOL for the Uinta pipelined microprocessor [13], and Coq for an ATM Switch Fabric [14]. The language proposed by Braibant [15] is close to our LDDL language and has been used to prove the correctness of parametric combinational circuits (*e.g.,* n bits adders).

Proof-assistants have also been used to certify tools used in circuit synthesis. An old survey of formal circuit synthesis is given in [16]. More recently, S. Ray et al. proved circuit transformations used in high-level synthesis with ACL2 [17]. Braibant and Chlipala certified in Coq a compiler from a simplified version of BlueSpec to RTL [18].

To the best of our knowledge, our work is the first to certify automatic circuit transformations for fault-tolerance. Contrary to most of the works which specify circuits within the logic of the prover, we use a gate-level HDL. This approach permits to reason on circuits (gates and wires) and to model SETs as glitches occurring at specific places. Automatic fault-tolerance techniques are easily specified by program transformations on the syntax of LDDL. Furthermore, its variable-less nature allowed a simple semantics (without environments) that facilitated formalization and proofs.

Our approach is general and applicable to many fault-tolerant transformations. We used it to prove the correctness of TMR and DTR but also of Triple-Time Redundant Transformation (TTR), a simpler and more straightforward time redundancy technique where each computation cycle is triplicated and followed by votings. However, our initial motivation was the proof of DTR whose correctness was far from obvious. While we relied on many manual checks to design the transformation, only Coq allowed us to get complete assurance. The formalization of DTR did not reveal real errors but a few imprecisions. For instance, we stated in [1] that the control block was protected using TMR without making it clear how it was connected to the rest of the circuit. We had to introduce three cells to record the value of the *fail* signal and to slightly change the definition of its internal FSM.

The approach makes an essential use of two features of Coq: dependent types and reflection. Dependent types provided an elegant solution to ensure that all circuits were well-formed. Such types are often presented as tricky to use but, in our case, that complexity remained confined to the writing of libraries for the equality and decomposition of buses and circuits. Reflection was very useful to prove properties of known sub-circuits; it would had been much harder without it.

The size of specifications and proofs for the common part (LDDL syntax and semantics, libraries) is 5000 lines of Coq (excluding comments and blank lines), 700 for TMR, 3500 for TTR and 7000 for DTR. Checking all the proofs takes around 45 min on an average laptop. The overall effort for the complete development is hard to estimate. Completing the proof of DTR alone took roughly 5 man-months. The Coq files for these proofs are available online [8].

We believe that additional user-defined tactics could make the proofs of LDDL transformations much smaller and automatic. Indeed, the key parts are to define the predicates relating the source and transformed circuits and to state the lemmas. The proofs themselves are, for the most part, straightforward inductions. The proposed framework could also be used to prove other fault-tolerance mechanisms (*e.g.,* the transformations for adaptive fault-tolerance we present in [19]) or well-known techniques used in circuit synthesis (*e.g.,* FSM-encoding). More generally, proof-assistants are now sufficiently mature to consider the formal certification of the whole circuit synthesis chain including optimizations.

Acknowledgments

Thanks are due to Frédéric Besson and Damien Pous for answering several questions about Coq. We got inspiration from James R. Wilcox's blog to make our proofs axiom free.

References

[1] D. Burlyaev, P. Fradet, and A. Girault, "Automatic time-redundancy transformation for fault-tolerant circuits," in *FPGA*, 2015, pp. 218–227.

[2] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer*, vol. 38, no. 2, pp. 43–52, Feb. 2005.

[3] 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, pp. 389–398.

[4] J. von Neumann, "Probabilistic logic and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[5] C. Chan, D. Schwartz-Narbonne, D. Sethi, and S. Malik, "Specification and synthesis of hardware checkpointing and rollback mechanisms," in *Design Automation Conference*, 2012, pp. 1222–1228.

[6] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: Concepts, overhead analysis, and implementation," in *FPGA*, 2007, pp. 188–196.

[7] Coq development team. The coq proof assistant, software and documentation available at http://coq.inria.fr/, 1989-2014.

[8] "Coq proofs of circuit transformations for fault-tolerance," available at https://team.inria.fr/spades/fthwproofs/, 2014-2015.

[9] M. Sheeran, "muFP, A language for VLSI design," in *LISP and Functional Programming*, 1984, pp. 104–112.

[10] G. Jones and M. Sheeran, "Designing arithmetic circuits by refinement in Ruby," *Sci. Comput. Program.*, vol. 22, no. 1-2, pp. 107–135, 1994.

[11] A. Gupta, "Formal hardware verification methods: A survey," *Form. Methods in System Design*, vol. 1, no. 2-3, pp. 151–238, Oct. 1992.

[12] J. Sawada and W. A. Hunt Jr., "Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability," *Formal Methods in System Design*, pp. 187–222, 2002.

[13] P. J. Windley and M. L. Coe, "A correctness model for pipelined multiprocessors," in *Theor. Provers in Circuit Design*, 1994, pp. 33–51.

[14] S. Coupet-Grimal and L. Jakubiec, "Certifying circuits in type theory," *Formal Asp. Comput.*, vol. 16, no. 4, pp. 352–373, 2004.

[15] T. Braibant, "Coquet: A coq library for verifying hardware," in *Proc. of Certified Programs and Proofs - CPP*, 2011, pp. 330–345.

[16] R. Kumar, C. Blumenrhr, D. Eisenbiegler, and D. Schmid, "Formal synthesis in circuit design. A classification and survey," in *FMCAD*, 1996, pp. 294–309.

[17] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang, "Formal verification for high-assurance behavioral synthesis," in *Int. Symposium on Automated Technology for Verification and Analysis*, 2009, pp. 337–351.

[18] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*, 2013, vol. 8044, pp. 213–228.

[19] D. Burlyaev, P. Fradet, and A. Girault, "Time-redundancy transformations for adaptive fault-tolerant circuits," in *NASA/ESA Conf. in Adaptive Hardware and Systems, AHS*, jun 2015.