Application of model checking to fault tolerance analysis*

Cinzia Bernardeschi$^{1[0000-0003-1604-4465],(\boxtimes)}$ and Andrea Domenici$^{1[0000-0003-0685-2864]}$

Dept. of Information Engineering, University of Pisa
{cinzia.bernardeschi,andrea.domenici}@ing.unipi.it

Abstract. A basic concept in modeling fault tolerant systems is that anticipated faults, being obviously outside of our control, may or may not occur. A fault tolerant system design can be proved to correctly behave under a given fault hypothesis, by proving the observational equivalence between the system design specification and the fault-free system specification. Additionally, model checking of a temporal logic formula which gives an abstract notion of correct behavior can be applied to verify the correctness of the design. The usage of model checking and temporal logic gives additional opportunities to better analyze the system behavior in presence of faults.

1 Introduction

Process algebras are a standard tool for the specification of concurrent systems. In order to specify a process and to prove its correctness, it is useful to decide which properties of the model are relevant and which ones can be ignored. Following [24], the semantics of processes is given in terms of labeled transition systems, which can describe their behavior in details, including their internal computations. It is common to define equivalences over labeled transition systems to verify if a process is a correct *implementation* of a *specification* process.

A widely used equivalence is *weak bisimulation*, or *observational* equivalence, first introduced by Milner [24], based on the idea that only the externally observable actions of a system are relevant in its interaction with the environment: Two systems are then observationally equivalent whenever no observation can distinguish them.

Model checking [12] is an alternative verification technique, in which the system is modeled using a process algebra or an automaton-based formalism and its correctness properties are expressed as temporal logic formulae [23]. Then these formulae are automatically checked on the specification of the system. Proofs are carried out by exhaustive search of the transition system of the model.

In fault-tolerance analysis, the main goal is verifying that a system works correctly in the presence of a given set of *anticipated faults*. In absence of implementation techniques to detect, confine and recover from erroneous states,

* Work partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

a system exhibits *failing behaviors* that deviate from the specified *normal*, or *correct*, behaviors. Different kinds of faults cause different kinds of failing behaviors, or *failure modes*, and the constraints on how faults are expected to occur in the system are expressed in the *fault hypothesis*. Given a set of anticipated faults, under a particular failure mode, a system is fault-tolerant with respect to the occurrence of faults as stated by the fault hypothesis if and only if the occurrence of such faults does not inhibit the system's ability to correctly satisfy its specification.

With the process-algebraic approach, checking for fault-tolerance is accomplished by defining a specification process that models the normal, or correct, behaviors, and an implementation process that models the possible behaviors in the presence of faults. The system is considered fault-tolerant with respect to the anticipated faults if the two processes are observationally equivalent.

This paper first discusses an issue related to the application of observational equivalence to fault-tolerance verification, then it reports results on the application of model-checking to the challenging problem of fault untestability in FPGA devices.

As reported in [20], one problem of using bisimulation equivalence for fault tolerance is that proving fault tolerance towards a given set of faults does not imply fault tolerance towards a subset of those faults. A typical example is that of compensating faults such as the loss and creation of messages in a communication channel. Fault actions are modeled as alternatives to the correct ones, but according to the standard process algebra semantics, when the correct actions are not enabled the system is forced to execute the fault actions. In these situations, faults are no longer random events independent of the system logic. Moreover, one fault may compensate the effects of another fault.

Testing Single Event Upsets (SEU) faults is a main concern in the development of aerospace applications based on FPGAs [17]. Such applications operate in an environment exposed to cosmic radiations that increase the likelihood of hardware faults. Radiation-hardened devices are expensive, so it is convenient to use on-line testing and on-the-fly reconfiguration to cope with radiation-induced faults. Given the large number of possible faults, it is important to optimize the test set in order to reduce execution time and energy consumption for on-line testing. Finding untestable faults is an important contribution to this purpose. The problem of fault untestability has been dealt with by modeling FPGA applications as state machines and using model checking to prove if the fault is untestable or not.

The paper is structured as follows: Section 2 reports related work, Section 3 discusses the use of process algebras and model checking in the analysis of the alternating bit protocol with multiple faults, Section 4 describes the application of model checking to the problem of fault untestability of FPGAs, and Section 5 concludes the paper.

2 Related work

A growing corpus of works on formal modeling and verification of fault tolerant systems has been produced, in particular concerning the application of process algebras and model checking. This section presents a small sample of the literature.

Partial model checking and μ-calculus are advocated by Gnesi et al. [16] to frame the problem of fault-tolerance verification within a general μ-calculus validation problem.

Francalanza and Hennessy [15] extend the Dπ language [18] to develop a behavioral theory of distributed programs in the presence of failures, using bisimulation equivalence to compare systems.

A formal framework for the specification and verification of fault tolerant system designs was presented in [9]. The work was focused on the possibility of using automatic verification tools, exemplifying the use of tools working on a particular process algebra and automata-based semantics (CCS/MEIJE and networks of automata [11]) and the temporal logic ACTL [13].

The specification and verification of the GUARDS Inter-consistency mechanism is reported in[7, 8]. This fault-tolerance mechanism was developed within the European project GUARDS (Generic Upgradable Architecture for Real-time Dependable Systems [26]) as a component of an architecture for embedded safety-critical systems. The validation approach is based on model checking technique and exploits the verification methodology supported by the JACK environment [10].

A method for the verification of fault-tolerant distributed systems was presented by Jones and Pike [22], based on calendar automata [14] to model systems, and introducing the technique of *symbolic fault injection*. The SAL (*Symbolic Analysis Laboratory* model checker [25] is used for verification.

The Promela modeling language and the SPIN model checker [19] are used by John et al. [21] to present an approach to model *threshold-guarded* distributed algorithms.

3 Fault tolerance for systems with multiple faults

When observational equivalence is used to assess fault tolerance of a system with respect to a given set of anticipated faults, the issue of *fault monotonicity* must be considered. Simply stated, an equivalence criterion is fault monotonic if and only if fault tolerance of an implementation with respect to a set Φ of anticipated faults implies fault tolerance with respect to any subset Φ' of Φ.

Janowski [20] has shown that bisimulation is in general not fault-monotonic, using the alternating bit protocol as an example.

The purpose of the protocol is ensuring reliable communication over a medium which may loose messages, i.e., $\Phi_o = \{\text{omission}\}$ is the set of anticipated faults. A possible implementation of the protocol (Figure 1), similar to the one discussed in [20], consists of four processes: the Sender, the Receiver, and two communication channels: one for the delivery of the message, and another for the acknowledgment of message reception.

Sender and Receiver use the value of one bit to identify a message, so that the identifier bit of each message is the complement of the preceding message's bit; a new message is not sent until the sender receives acknowledgment of the current message. Since the channels can loose messages, both the Sender and the Receiver resend the same message or, respectively, acknowledgment repeatedly until the acknowledgment is received.

We first consider an implementation *Sys* of the protocol, represented in CCS as in the following, where S_0 and R_1 are the Sender and Receiver, respectively, A is the delivery channel, and B the acknowledgment one:

$$\begin{aligned}
S_0 &= in.S_0' \\
S_0' &= \overline{a_0}.S_0' + d_1.S_0' + d_0.S_1 \\
S_1 &= in.S_1' \\
S_1' &= \overline{a_1}.S_1' + d_0.S_1' + d_1.S_0
\end{aligned}$$

$$\begin{aligned}
A &= a_0.A_0' + a_1.A_1' \\
A_0' &= \overline{b_0}.A + \tau.A \\
A_1' &= \overline{b_1}.A + \tau.A
\end{aligned}$$

$$\begin{aligned}
B &= c_0.B_0' + c_1.B_1' \\
B_0' &= \overline{d_0}.B + \tau.B \\
B_1' &= \overline{d_1}.B + \tau.B
\end{aligned}$$

$$\begin{aligned}
R_1 &= b_0.R_0' + b_1.R_1 + \overline{c_1}.R_1 \\
R_0' &= \overline{out}.R_0 \\
R_0 &= b_1.R_1' + b_0.R_0 + \overline{c_0}.R_0 \\
R_1' &= \overline{out}.R_1
\end{aligned}$$

$$L = \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1\}$$

$$Sys = (S_0|A|B|R_1)\backslash L\ .$$

The above system is represented in Figure 1 as a network of communicating automata. Upon an *in* action at the system's external interface, the Sender sends the message to the Receiver through channel A by synchronizing on action a_0 or a_1 depending on the current value of the alternating bit (the first message is identified as 0). Upon receiving the message, the Receiver executes $\overline{out}$, meaning that the message is available at the interface. Next, the Receiver sends the acknowledgment by synchronizing with channel B on action c_0 or c_1 according to the value of the identifier bit of the received message.

Omission of messages or acknowledgments is represented by the τ actions in the processes for the channels, which can take a channel from state A'_0 or A'_1 (B'_0 or B'_1) to A (B) without executing the corresponding synchronization action.

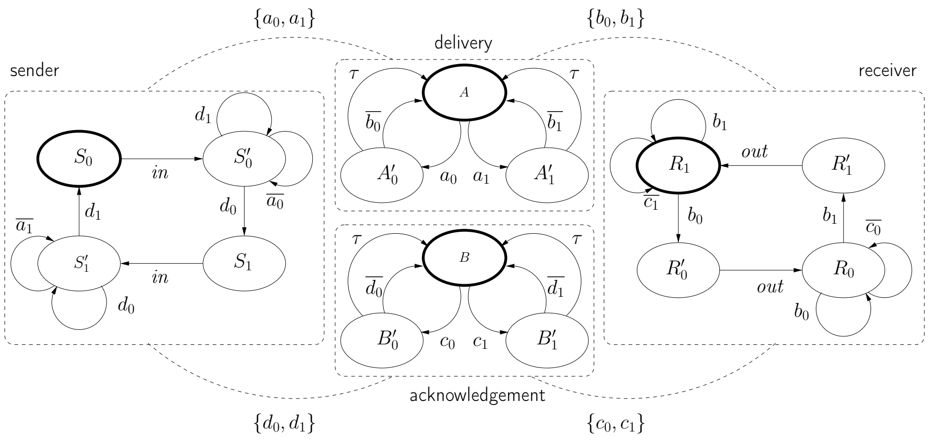

Fig. 1. Alternating bit protocol.

Let us now consider a system Sys_{oc} affected by the set $\Phi_{oc} = \{\text{omission}, \text{creation}\}$ of anticipated faults, i.e., we assume that the channels may drop messages or acknowledgments and also emit spurious ones. The CCS description of this system differs from the original one in the channel processes, as shown in Figure 2 and in the following code:

$$\begin{aligned}
A_{oc} &= a_0.A'_{oc0} + a_1.A'_{oc1} + \overline{b_0}.A_{oc} + \overline{b_1}.A_{oc} \\
A'_{oc0} &= \overline{b_0}.A_{oc} + \tau.A_{oc} \\
A'_{oc1} &= \overline{b_1}.A_{oc} + \tau.A_{oc}
\end{aligned}$$

$$\begin{aligned}
B_{oc} &= c_0.B'_{oc0} + c_1.B'_{oc1} + \overline{d_0}.B_{oc} + \overline{d_1}.B_{oc} \\
B'_{oc0} &= \overline{d_0}.B_{oc} + \tau.B_{oc} \\
B'_{oc1} &= \overline{d_1}.B_{oc} + \tau.B_{oc}
\end{aligned}$$

Creation is modeled by the additional transitions in state A (B), which execute a synchronization without changing state.

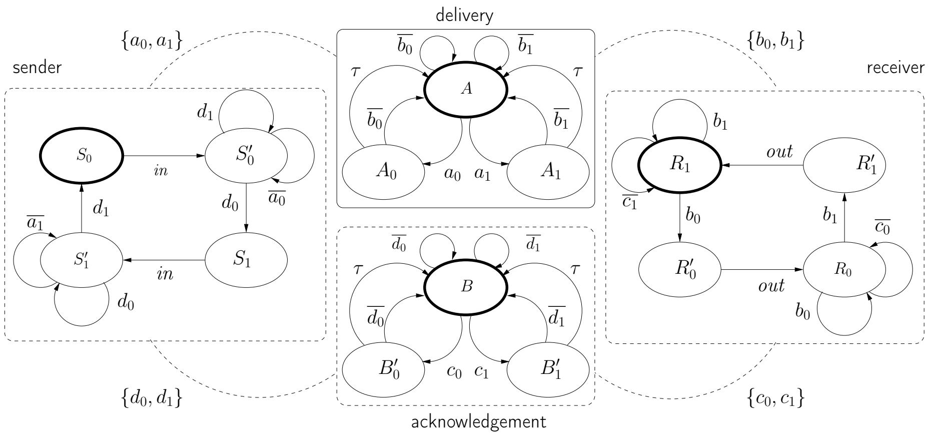

Fig. 2. Alternating bit protocol with omission or creation of messages or acknowledgments.

Fault tolerance of Sys_{oc} can be proved by checking that it is observationally equivalent to a process P specifying the intended behavior, namely, the alternation of *in* and *out* actions:

$$\begin{aligned} Sys_{oc} &= (S_0|A_{oc}|B_{oc}|R_1)\backslash L \\ P &= in.\overline{out}.P \\ P &\approx Sys_{oc} \end{aligned}$$

If instead we consider a system Sys_c with channels that can misbehave only by creating messages or acknowledgments and not by dropping them, we would expect the implementation to be still fault tolerant, given that in this case the set $\Phi_c = \{\text{creation}\}$ of anticipated faults is a proper subset of Φ_{oc}. In this case, we have:

$$\begin{aligned} A_c &= a_0.A'_{c0} + a_1.A'_1 + \overline{b_1}.A_c + \overline{b_0}.A_c \\ A'_{c0} &= \overline{b_0}.A_c \\ A'_{c1} &= \overline{b_1}.A_c \end{aligned}$$

$$\begin{aligned} B_c &= c_0.B'_{c0} + c_1.B'_{c1} + \overline{d_1}.B_c + \overline{d_0}.B_c \\ B'_{c0} &= \overline{d_0}.B_c \\ B'_{c1} &= \overline{d_1}.B_c \end{aligned}$$

$$Sys_c = (S_0|A_c|B_c|R_1)\backslash L \ .$$

It is immediate to show that Sys_c is not observationally equivalent to P, thus proving that bisimulation is in general not fault-monotonic. Therefore, proving that a system with more faults is observationally equivalent to the fault-free system does not guarantee that observational equivalence holds for any subset of faults.

However, we may observe that in this case creation faults are modeled as non-deterministic actions on the same footing as the correct behavior. It may be more natural to model them as internal actions as shown below for Φ_c:

$$\begin{aligned}
AA_c &= a_0.AA'_0 + a_1.AA'_1 + \tau.\overline{b_1}.AA_c + \tau.\overline{b_0}.AA_c \\
AA'_0 &= \overline{b_0}.AA_c \\
AA'_1 &= \overline{b_1}.AA_c
\end{aligned}$$

$$\begin{aligned}
BB_c &= c_0.BB'_{c0} + c_1.BB'_{c1} + \tau.\overline{d_1}.BB_c + \tau.\overline{d_0}.BB_c \\
BB'_{c0} &= \overline{d_0}.BB_c \\
BB'_{c1} &= \overline{d_1}.BB_c \ .
\end{aligned}$$

The case for Φ_{oc} is handled similarly.

In this case, observational equivalence is not satisfied in either case: creation faults only, and omission and creation faults.

In addition, observational equivalence between the behavior of the fault free system and that of the system affected by faults does not reveal some useful information in case of occurrence of faults. For example, infinite loops of τ actions could not be detected. It is then advisable to introduce model checking of temporal logic formulae to complement the techniques based on bi-simulation.

First, model checking a specification allows it to be validated with respect to properties expressed in temporal logic. For example, we can use μ-calculus to express the property that action $\overline{out}$ will eventually be executed (a minimal sanity check), with the formula

$$\alpha \triangleq \mu.Z(<\!-\!>\ tt \wedge [-out]Z) \ .$$

Model checking shows that α holds unsurprisingly for P, but it does not hold for any of the implementations considered above. This is caused by the fact that channels may drop messages or acknowledgments indefinitely by executing τ actions.

The failure to prove property α shows that there exists a path in which *out* is not executed. This is caused by cycles in which messages can be created and lost. And what we can see is that such property is false also on the original version of the protocol, with omission only. This because the specification of the protocol allows the Sender (Receiver) to re-send or loose the same message an unbounded number of times.

The example proves tolerance to the creation and omission of messages of the protocol, in a context in which faults can freely occur. Other properties of the fault tolerant system can be proved by model checking if actions modeling of faults are made explicit. In this case we can prove for example, that the system satisfies property α in case of one omission fault. What can be done is to state an assumption on fault occurrences, and prove tolerance in that specific case.

In real system modeling, this approach reduces the state space explosion problem. From the specification point of view, explicit actions modeling faults and a new process, the fault hypothesis process, which synchronizes with the system and states the possible occurrences of faults, are introduced in the specification.

4 Untestability of faults: SEUs in SRAM-based FPGAs

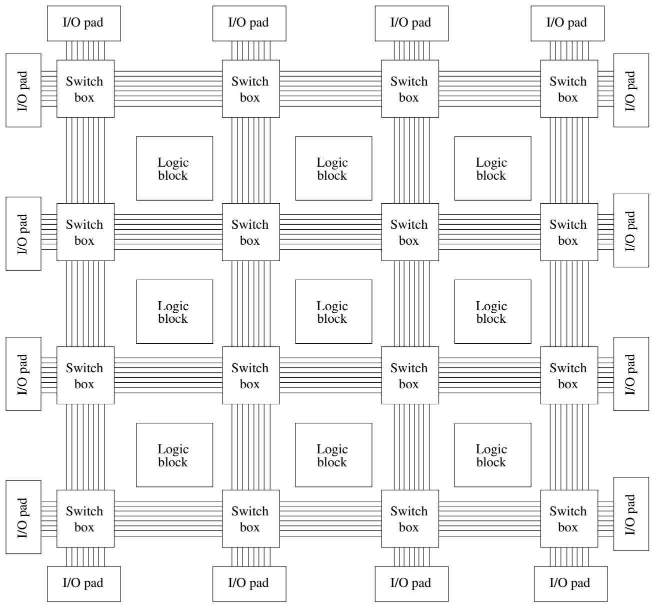

Fig. 3. Structure of an FPGA.

SRAM-based FPGAs are programmable devices made of logic blocks interconnected by switch elements called *switch boxes*, in a structure shown in a very simplified and scaled-down way by Figure 3. A logic block contains a small number of memory elements and combinatorial logic. The latter is implemented with configurable *look-up tables* (LUT), which behave logically as associative memories mapping each combination of inputs to the corresponding value of a given Boolean function. Figure 4 (a) is a logical representation, as a Karnaugh map, of a LUT configured to implement the disjunction of its four inputs.

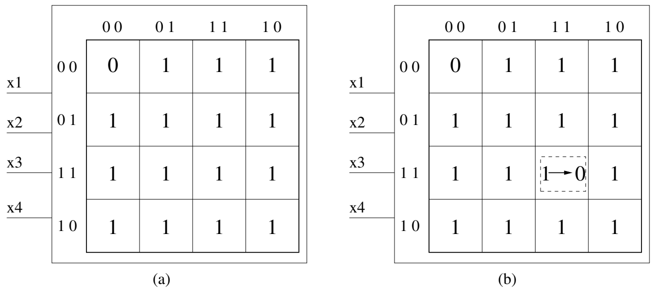

Fig. 4. A LUT and the effect of a SEU.

A switch box is a matrix of switches called *Programmable Interconnect Points* (PIPs) [27], which route signals by connecting pairs of wires. An example of switch-box is shown in Figure 5 (a), where P_i and P_j are PIPs; P_i is programmed to connect the input wire A to the output wire B. Similarly, P_j is programmed to connect the input wire C to the output wire D. The connection between two wires, such as A to B, is called a *routing segment*.

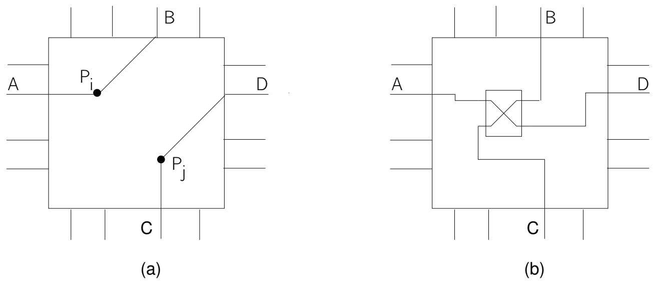

Fig. 5. Bridge fault.

Programming an SRAM-FPGA device consists in downloading a configuration code, called a bitstream, into its configuration memory. The bitstream determines the functionality of each LUT and the configuration of the PIPs. The bitstream is generated by a tool from a high-level hardware design language (e.g., Verilog or VHDL). As an intermediate step, the Verilog/VHDL description is synthesized into a *logic netlist* showing the logical interconnections of FPGA components, such as LUTs and memory elements. Figure 6 shows a simplified logic netlist for a system composed of three 2-input LUTs, one D flip-flop, a clock generator, two input buffers (i.e., signal amplifiers), and one output buffer. The LUTs implement two AND and one OR gate.

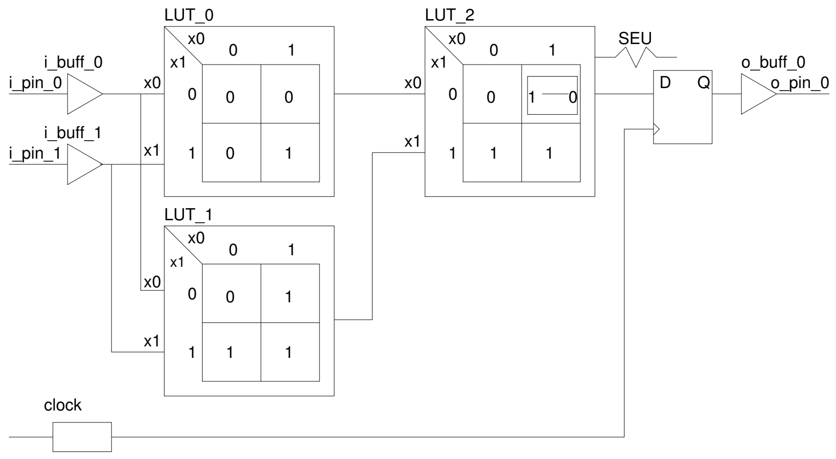

Fig. 6. SEU unexcitability.

In a further step, the logic netlist is transformed into a *place-and-routed netlist* including information on the physical placement of the components and on the configuration of the switch boxes.

Both LUTs and switch boxes are affected by various types of faults, including *Single Event Upsets* (SEU), caused by radiations, especially in aerospace applications [2]. SEUs affecting a LUT change its logic function, whereas those affecting a switch box modify the interconnection topology among logic blocks.

An example of a faulty LUT is shown in Figure 4 (b), where the dashed box represents a SEU that flips a bit from 1 to 0, so that the LUT maps the input vector (1111) to the output 0. However, the output of the faulty LUT differs from the one of the fault-free LUT only for that specific input. That fault is activated only when the input associated with the faulty configuration bit is applied to the LUT.

A SEU in the configuration memory of PIPs may cause several types of topological modifications [28] that manifest themselves as logical faults on the output wires of the affected switch box [1], namely: *stuck-at-0* (*stuck-at-1*), when a wire is stuck at the 0 (1) logic value; *bridge*, when the values of two wires are exchanged; *wired-AND* (*wired-OR*), when the value of an output wire is the AND (OR) of the values of two input wires; and *wired-MIX*, when the values of two output wires B and D are mixed so that they keep their correct values if the values of the respective input wires are equal, otherwise they take the complementary values. Figure 5 (b) shows the effects of a logical bridge fault.

When FPGAs are used in safety-critical applications, on-line testing is one of the methods that can be applied at run-time to detect SEUs, and possibly reconfigure the FPGA. Testing relies on a pre-computed set of test patterns, and it is important to optimize this set with respect to the contrasting goals of maximum fault coverage and minimum execution time and resource usage. One way of optimizing the test set is finding offline the faults that can be excluded from the set because they are undetectable, i.e., they are either unexcitable or masked. A fault is *unexcitable* if the combination of input values that could activate it will never be fed to the affected component. A fault is *masked* if it cannot propagate wrong values to the external pins of the device.

In [6], model checking has been applied to prove unexcitability of SEUs and the counter-example facility of the model checker has been used to generate test patterns. The logic netlist was used to model faults in the LUTs and the route-and-placed netlist to model faults in the switch boxes. For faults in a LUT, the logic function of the faulty LUT was generated; for faults in the interconnect, their logical effects were modeled. The routing faults and their effects were computed using an external tool, E^2STAR [6], that operates on the place-and-routed netlist.

```
netlist : CONTEXT =
BEGIN
    circuit: MODULE =
    BEGIN
        INPUT i_pin_0: BOOLEAN;           INPUT i_pin_1: BOOLEAN;
        OUTPUT o_pin_0: BOOLEAN;
        LOCAL i_buff_0: BOOLEAN;          LOCAL i_buff_1: BOOLEAN;
        LOCAL LUT_0: BOOLEAN;             LOCAL LUT_1: BOOLEAN;
        LOCAL LUT_2: BOOLEAN;
        LOCAL d_ff_0: BOOLEAN;            LOCAL O_buff_0: BOOLEAN;
    DEFINITION
        i_buff_0 = i_pin_0;               i_buff_1 = i_pin_0;
        LUT_0 = (i_buff_0 AND i_buff_1);  LUT_1 = (i_buff_0 OR i_buff_1);
        LUT_2 = (i_LUT_0 OR LUT_1);
        o_buff_0 = d_ff_0;                o_pin_0 = o_buff_0;
    INITIALIZATION
        d_ff_0 = FALSE;
    TRANSITION
        d_ff_0' = LUT_2;
    END;
END
```

Fig. 7. SAL specification.

The behavioral model of the FPGA application is built in the Symbolic Analysis Laboratory (SAL) framework [25], and the unexcitability property is expressed as an LTL logic formula that checks whether the configuration activating the fault can be generated, starting from any possible input sequence of the FPGA.

The SAL input language describes a system (*context*) as the parallel composition of *modules*, each representing a state machine defined by its input, local, and output variables, by definitions equating variable values to functions of other variables, and by transitions equating the *next* values of variables (denoted by primes) to functions of the current state.

The SAL code for the netlist shown in Figure 6 is reported in Figure 7. In this case, only one module is sufficient. The behavior of LUTs and buffers is described by definitions:

– the behavior of LUTs is described by the corresponding logic functions;
– the behavior of an input buffer is described as an assignment between a local variable, modeling the buffer, and an input variable, modeling the associated input pin;
– similarly, two local variables are used for output buffers, one modeling the buffer and the other modeling the output pin.

Flip-flops are described by transitions, that are executed at each clock cycle.

If a fault f in an n-input LUT affects a location corresponding to the input vector $i_f = (v_0, v_1, \dots v_{n-1})$ (e.g., $i_f = (10)$ in *LUT_2* of Figure 6), fault f is unexcitable if the configuration i_f can never occur in the context C modeling the whole system [4], therefore the property of unexcitability in LTL has the general form

$$C \vdash \mathbf{G}(\neg(x_0 = v_0 \wedge x_1 = v_1 \wedge \cdots \wedge x_n = v_{n-1})) \ .$$

The unexcitability property for the SEU in the configuration bit of *LUT_2* associated with input (10) is then

```
unex_LUT_2_10: THEOREM
    circuit |- G(NOT(LUT_0 = TRUE AND LUT_1 = FALSE));
```

The theorem holds, since this fault can never be excited, because it is not possible that the output of *LUT_1* is `0` while the output of *LUT_0* is `1` (Figure 6), because *LUT_1* implements the OR function, and *LUT_0* implements the AND of the same input signals. Similarly, the formula for the SEU in the configuration bit of *LUT_0* associated with input `11` is the following:

```
unex_LUT_0_11: THEOREM
    circuit |- G(NOT(i_buff_0 = TRUE AND i_buff_1 = TRUE));
```

This theorem does not hold, and a trivial counter-example is shown:

```
Counter_example: (i_pin_0 = true, i_pin_1 =true)
```

The formulae for routing faults in the switch box of Figure 6, referring to the logical netlist, are shown in Table 1, where $\hat{A}$ and $\hat{C}$ are the values of A and C (true if equals to 1, false otherwise) [5]. A Stuck-at 0 (1) on P_i is unexcitable if the signal on A is always 0 (1). A Bridge between P_i and P_j is unexcitable if the value of A always equals the value of C. A Wired-AND between P_i and P_j is unexcitable if the value of B always equals $A \wedge C$ and the value of C always equals $A \wedge C$. A Wired-OR between P_i and P_j is unexcitable if the value of A always

Table 1. Unexcitability formulae for routing faults.

s-a-0 on P_i	$C \vdash \mathbf{G}(\neg(\hat{A}))$
s-a-1 on P_i	$C \vdash \mathbf{G}(\neg(\neg\hat{A}))$
bridge between P_i and P_j	$C \vdash \mathbf{G}(\neg(\hat{A} \neq \hat{C}))$
Wired-AND between P_i and P_j	$C \vdash \mathbf{G}(\neg((\hat{A} \neq (\hat{A} \wedge \hat{C})) \vee (\hat{C} \neq (\hat{A} \wedge \hat{C}))))$
Wired-OR between P_i and P_j	$C \vdash \mathbf{G}(\neg((\hat{A} \neq (\hat{A} \vee \hat{C})) \vee (\hat{C} \neq (\hat{A} \vee \hat{C}))))$
Wired-MIX between P_i and P_j	$C \vdash \mathbf{G}(\neg((\hat{A} \neq \hat{C}) \wedge (\neg\hat{A} \vee \hat{C})))$

equals $A \vee C$ and the value of C always equals $A \vee C$. A Wired-MIX between P_i and P_j is unexcitable if (i) the value of A always equals the output of C or (ii) the value of A is 1 and the value of C is 0, or viceversa. The unexcitability theorem associated with the Bridge fault in Figure 5 is:

```
unex: THEOREM circuit |- G(NOT(B=D));
```

Experimental results [6] show that a substantial number of SEU faults are not excitable. Knowing which faults are unexcitable reduces significantly the time needed for test pattern generation and testing. In the same framework, untestability of faults, that includes both unexcitability and fault masking, is analyzed. Masked faults are found by comparing the values at the output pins of the fault-free system to the values at the output pins of the faulty system at each clock cycle, considering the full end-to-end paths from input to output. The counter-example gives information on the test vector that must be applied at every input to test the fault. The ability of model checkers to produce counter-examples has been used in our framework to generate test patterns for testable faults, optimized with a genetic algorithm [3].

5 Conclusions

This work reports on applications of model checking to different issues related to fault tolerance. In particular, the problem of assessing fault tolerance in systems with multiple faults modeled with process algebras has been discussed, and a method to analyze untestability of hardware faults has been presented. Model checking has been shown as useful complement to method based on weak-bisimulation equivalence. Model checking on state-machine based models has been shown experimentally to be an effective tool to improve the performance of on-line testing for systems affected by radiation faults.

References

1. Battezzati, N., Sterpone, L., Violante, M.: Reconfigurable Field Programmable Gate Arrays for Mission-Critical Application. Springer Science & Business Media. (2011)

2. Baumann, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability **5**(3), 305–316 (2005). https://doi.org/10.1109/TDMR.2005.853449
3. Bernardeschi, C., Cassano, L., Cimino, M.G., Domenici, A.: GABES: a Genetic Algorithm Based Environment for SEU Testing in SRAM-FPGAs. Journal of Systems Architecture **59**(10, Part D), 1383–1254 (2013). https://doi.org/http://dx.doi.org/10.1016/j.sysarc.2013.10.006, http://www.sciencedirect.com/science/article/pii/S1383762113001975
4. Bernardeschi, C., Cassano, L., Domenici, A.: SEU-X: A SEU un-excitability prover for SRAM-FPGAs. In: 18th IEEE International On-Line Testing Symposium, IOLTS 2012. pp. 25–30 (2012)
5. Bernardeschi, C., Cassano, L., Domenici, A., Sterpone, L.: Unexcitability analysis of SEus affecting the routing structure of SRAM-based FPGAs. In: Great Lakes Symposium on VLSI 2013 (part of ECRC), GLSVLSI'13, Paris, France, May 2-4, 2013. pp. 7–12 (2013)
6. Bernardeschi, C., Cassano, L., Domenici, A., Sterpone, L.: UA^{2}TPG: An untestability analyzer and test pattern generator for SEUs in the configuration memory of SRAM-based FPGAs. Integration **55**, 85–97 (2016). https://doi.org/10.1016/j.vlsi.2016.03.004
7. Bernardeschi, C., Fantechi, A., Gnesi, S.: Formal Validation of the GUARDS Inter-consistency Mechanism. In: Felici, M., Kanoun, K. (eds.) Computer Safety, Reliability and Security. SAFECOMP 1999. Lecture Notes in Computer Science, vol. 1698, pp. 420–430. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48249-0_36
8. Bernardeschi, C., Fantechi, A., Gnesi, S.: Formal validation of fault-tolerance mechanisms inside GUARDS. Rel. Eng. & Sys. Safety **71**(3), 261–270 (2001). https://doi.org/10.1016/S0951-8320(00)00078-8
9. Bernardeschi, C., Fantechi, A., Simoncini, L.: Formally verifying fault tolerant system designs. Comput. J. **43**(3), 191–205 (2000). https://doi.org/10.1093/comjnl/43.3.191
10. Bouali, A., Gnesi, S., Larosa, S.: The integration project for the JACK environment. Tech. Rep. CS-R9443, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands (1994)
11. Boudol, G.: Notes on algebraic calculi of processes. In: Apt, K. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol. 13, pp. 261–303. Springer, Berlin, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_9
12. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (Apr 1986). https://doi.org/10.1145/5397.5399, http://doi.acm.org/10.1145/5397.5399
13. De Nicola, R., Fantechi, A., Gnesi, S., Ristori, G.: An action based framework for verifying logical and behavioural properties of concurrent systems. In: Larsen, K.G., Skou, A. (eds.) Computer Aided Verification. pp. 37–47. Springer, Berlin, Heidelberg (1992). https://doi.org/10.1007/3-540-55179-4_5
14. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 3253, pp. 199–214. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_15

15. Francalanza, A., Hennessy, M.: A theory of system behaviour in the presence of node and link failures. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005 – Concurrency Theory. pp. 368–382. Springer, Berlin, Heidelberg (2005)
16. Gnesi, S., Lenzini, G., Martinelli, F.: Logical specification and analysis of fault tolerant systems through partial model checking. Electronic Notes in Theoretical Computer Science **118**, 57–70 (feb 2005). https://doi.org/10.1016/j.entcs.2004.09.032
17. Graham, P., Caffrey, M., Zimmerman, J., Sundararajan, P., Johnson, E.: Consequences and categories of SRAM FPGA configuration SEUs. In: In Proceedings of the International Conference on Military and Aerospace Programmable Logic Devices (MAPLD 2003 (2003)
18. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. Information and Computation **173**(1), 82–120 (2002). https://doi.org/10.1006/inco.2001.3089
19. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (May 1997). https://doi.org/10.1109/32.588521
20. Janowski, T.: On bisimulation, fault-monotonicity and provable fault-tolerance. In: Johnson, M. (ed.) Algebraic Methodology and Software Technology. AMAST 1997. Lecture Notes in Computer Science, vol. 1349, pp. 292–306. Springer, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0000478
21. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Towards modeling and model checking fault-tolerant distributed algorithms. In: Bartocci, E., C.R., R. (eds.) Model Checking Software. SPIN 2013. Lecture Notes in Computer Science, vol. 7976, pp. 209–226. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39176-7_14
22. Jones, B., Pike, L.: Modular model-checking of a byzantine fault-tolerant protocol. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. NFM 2017. Lecture Notes in Computer Science, vol. 10227, pp. 163–177. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_12
23. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems Specification. Springer-Verlag, Berlin, Heidelberg (1992)
24. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)
25. de Moura, L., Owre, S., Rueß, H., Rushby, J.M., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, 16th International Conference, (CAV 2004). Lecture Notes in Computer Science, vol. 3114, pp. 496–500. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_45
26. Powell, D., Arlat, J., Beus-Dukic, L., Bondavalli, A., Coppola, P., Fantechi, A., Jenn, E., Rabejac, C., Wellings, A.: GUARDS: a generic upgradable architecture for real-time dependable systems. IEEE Transactions on Parallel and Distributed Systems **10**(6), 580–599 (June 1999). https://doi.org/10.1109/71.774908
27. Rodriguez-Andina, J., Moure, M., Valdes, M.: Features, Design Tools, and Application Domains of FPGAs. Industrial Electronics, IEEE Transactions on **54**(4), 1810–1823 (Aug 2007). https://doi.org/10.1109/TIE.2007.898279
28. Sterpone, L., Violante, M., Sorensen, R., Merodio, D., Sturesson, F., Weigand, R., Mattsson, S.: Experimental Validation of a Tool for Predicting the Effects of Soft Errors in SRAM-Based FPGAs. Nuclear Science, IEEE Transactions on **54**(6), 2576–2583 (dec 2007). https://doi.org/10.1109/TNS.2007.910122