UA^{2}TPG: An Untestability Analyzer and Test Pattern Generator for SEUs in the Configuration Memory of SRAM-based FPGAs*

Cinzia Bernardeschi Luca Cassano Andrea Domenici[†]
Luca Sterpone[‡]

March 29, 2019

Abstract

This paper presents UA^{2}TPG, a static analysis tool for the untestability proof and automatic test pattern generation for SEUs in the configuration memory of SRAM-based FPGA systems. The tool is based on the model-checking verification technique. An accurate fault model for both logic components and routing structures is adopted. Experimental results show that many circuits have a significant number of untestable faults, and their detection enables more efficient test pattern generation and on-line testing. The tool is mainly intended to support on-line testing of critical components in FPGA fault-tolerant systems.

Keywords Single Event Upset, SRAM-based FPGA, Untestability Analysis, Model Checking

1 Introduction and Related Work

Radiations in the atmosphere are responsible for introducing *Single Event Upsets (SEU)* in digital devices [2]. SEUs (or bit flips) have particularly adverse effects on Field programmable Logic Arrays (FPGAs) using SRAM technology, as they may permanently corrupt a bit in the configuration memory (correctable only with a reconfiguration of the device) and thus they may permanently modify the functionality of the system implemented in the FPGA device [20].

Testing techniques for digital systems may be divided into *off-line* [22, 32] and *on-line* [34, 9, 11, 24] ones. Off-line techniques are applied before running

*Postprint. Published in: Integration Volume 55, September 2016, Pages 85-97. DOI: https://doi.org/10.1016/j.vlsi.2016.03.004.

[†]C. Bernardeschi, L. Cassano, and A. Domenici are with the Department of Information Engineering, University of Pisa, Italy.

[‡]L. Sterpone is with the Dept. of Automatics and Informatics, Politecnico di Torino, Italy.

the system (such as scan chain-based tests performed by manufacturers before delivering the circuits). On-line techniques are employed by the end-user of the system and are applied while the system is running. On-line techniques may further be classified into *non-concurrent* and *concurrent* ones. Non-concurrent on-line techniques [34, 9] are applied while the system is running but during their execution the system does not perform its “nominal” operation; thus, non-concurrent techniques may or may not be transparent for the end-user depending on how long the test execution takes (test patterns-based techniques, like the one proposed in the paper, are non-concurrent on-line testing techniques). On the other hand, concurrent on-line techniques [11, 24] (such as memory readback) are applied while the system is running and are always transparent for the end user.

Also the occurrence of SEUs in the configuration memory of an FPGA-based system can be detected through memory readback as well as through on-line testing. Memory readback allows faults to be detected without interfering with the normal operation of the system, while on-line testing requires the operation of the system to be periodically halted in order to execute the test patterns. On the other hand, on-line testing has a wider applicability. While memory readback is supported only by recent devices, on-line testing can be applied to any device. This last point is particularly important in safety-related application fields where design standards, such as [16, 17], often recommend, or even require, designers to employ older but more robust and mature devices.

Automated test generation aims at finding input values (structured in test vectors, test patterns, or sets of test patterns) that can detect a large number of faults, while minimizing testing time. Many works addressing the problem of automatic test pattern generation (ATPG) for digital circuits have been published [1, 27], but very few of these works specifically address FPGAs. Test methods devised for ASIC circuits could be effective when used for testing structural defects in the FPGA chip, but they are not satisfactory when used for testing SEUs in the configuration memory of FPGAs [33]. In particular, it has been demonstrated [31] that test pattern generation methods based on the stuck-at fault model for ASIC circuits obtain too optimistic results when applied to FPGAs. The stuck-at fault model considers permanent faults at the input and output terminals of the logical components. To keep into account SEUs in the configuration bits of the FPGA chip, more accurate fault models should be considered.

In [34, 38], methods for testing of FPGA structural defects have been proposed. In [6], random testing is applied to analyze FPGA SEUs observability, and in [4] a genetic algorithm is used to generate test patterns for testing SEUs. The importance of detecting untestable faults has been recognized in works addressing various aspects of the analysis of fault untestability in digital systems, aimed at improving ATPG efficiency. In [30] and [28], a new subclass of untestable faults, called *register enable stuck-on*, is defined and a method for generating property specification language (PSL) assertions for proving the untestability of this class of faults is presented. In these papers, stuck-at faults on the clock-enable signals of registers at the register transfer level (RTL) are

addressed. The same authors proposed in [29] a hierarchical untestability identification method. The method addresses untestable faults in functional units, such as adders and multiplexers, at the RTL level. In [39], a preprocessing method for accelerating SAT-based ATPGs by eliminating untestable faults is presented. The method takes into account the stuck-at fault model and, as the authors declare, it addresses only *easy-to-classify* untestable faults. In [26], two algorithms (FILL and FUNI) for untestability demonstration of stuck-at faults are presented. FILL identifies large subsets of illegal states in synchronous sequential circuits, and FUNI finds untestable faults that require illegal states previously found by FILL to be detected.

In this paper we propose UA^{2}TPG, an untestability prover and automatic test pattern generator for SEUs in the configuration memory of SRAM-based FPGAs. The fault model adopted for SEUs in configuration bits controlling logic components and routing structures is more accurate than the classical ones usually considered [31, 40]. The proposed tool uses an *application-dependent* analysis, i.e., it statically determines which SEUs in the configuration bits actually used by a given system are not testable, without having to examine bits not required by the given application. Untestability is proved with a model-checking technique [12] for automatically verifying properties of finite state systems. Moreover, at the end of the untestability analysis, the tool generates a set of test patterns able to detect 100% of the testable SEUs by using the counter-example facility of model checking tools [19]. These test patterns could be used for on-line application-dependent testing of SEUs. The proposed untestability analysis approach can be applied to combinational circuits as well as to sequential ones.

Demonstrating the untestability of faults in a VLSI design offers the designer an evaluation of the degree of testability of the system, or, from the fault tolerance point of view, an estimation of the sensitivity to faults of the system. Similarly, test patterns generated for on-line testing could also be used as input patterns to stimulate the system during fault injection or radiation testing experiments.

Previous work by the authors analyzed only the problem of the unexcitability of SEUs in the configuration memory controlling logic resources [5] and routing resources [7], i.e., SEUs that can never be activated, ignoring failure propagation and masking. With respect to the state of the art, UA^{2}TPG represents the first tool able to analyze the untestability of SEUs in the configuration memory and to generate test patterns for such faults. The tool relies on the Symbolic Analysis Laboratory [3] framework, using on the SAL description language to model the structure of the netlist and temporal logic to specify untestability theorems. The SAL-SMC model checker is used to prove the untestability of faults. We used the BDD-based symbolic model-checking tool (SAL-SMC) instead of the SAT-based bounded model-checking one (SAL-BMC) because, as it has been experimentally demonstrated in [18], SAL-SMC is much more effective than SAL-BMC in finding counterexamples (and thus test patterns). Note that the use of model-checking (Boolean Satisfiability in particular) for automatic test patterns generation for stuck-at faults in digital circuits has already been

proposed in the literature, for example in [15], in [25] and in [36]. Nevertheless, the current paper is the first one to use a model-checking-based approach to analyze the testability of, and produce test patterns for, SEUs in the configuration memory of SRAM-based FPGA systems. The current version of UA^{2}TPG is available at: `http://www.ing.unipi.it/∼a009435/ua2tpg/`.

The remainder of this paper is organized as follows: in Section 2 the considered fault model is presented; in Section 3 some background about model checking and the SAL environment is introduced; Section 4 presents the UA^{2}TPG tool; Section 5 describes the analysis environment in which UA^{2}TPG works; Section 6 reports results from the application of the tool to some circuits from the ITC'99 benchmark; Section 7 concludes the paper.

2 Effects of SEUs in the Configuration Memory of SRAM-based FPGAs

An FPGA [23] is an array of programmable logic blocks, interconnected through a programmable routing architecture and communicating with the output through programmable I/O pads (Figure 1).

Programming an FPGA device consists in downloading a programming code, called a *bitstream*, in its configuration memory. The bitstream determines the hardware structure, and thus the functionality, of the system to be implemented in the FPGA. SEUs occurring in the configuration memory of an FPGA device may alter the functionality performed by logic blocks or their interconnections. In this paper, we define as *untestable* a fault in the configuration memory that does not change the values of the external outputs with respect to the values produced by the fault-free system, for the same inputs.

A fault may be untestable if it affects a configuration bit that (i) controls an unused resource, or (ii) is a don't-care bit in the configuration of a used resource, or (iii) controls a used resource whose output is masked at further stages of the signal propagation path, or (iv) affects a potentially used bit that is never activated by the possible resource inputs (unexcitable fault).

2.1 SEUs in Logic Components

The functional fault model for SEUs in the configuration memory controlling the logic resource of an FPGA proposed in [31] is assumed in this work.

The effects of SEUs in the configuration memory cannot be accurately modeled as stuck-at faults. In the stuck-at fault model, an SEU in the configuration memory of a component causes the output of the component to be stuck at a given value. In the fault model considered in this work an SEU in the configuration memory of a LUT causes the faulty LUT to produce an incorrect output only when the configuration of its input values is the one associated with the faulty configuration bit. On the other hand, the faulty LUT will behave correctly for every other configuration of its input values. Figure 2(a) shows an SEU causing a bit flip in the configuration bit associated with input (0000). In

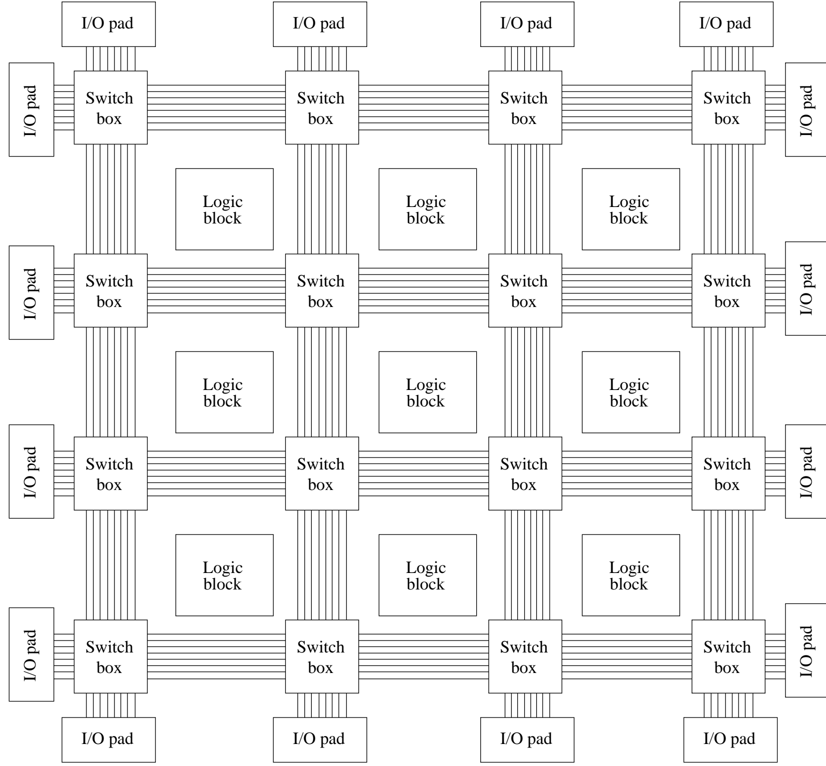


Figure 1: General architecture of an FPGA device.

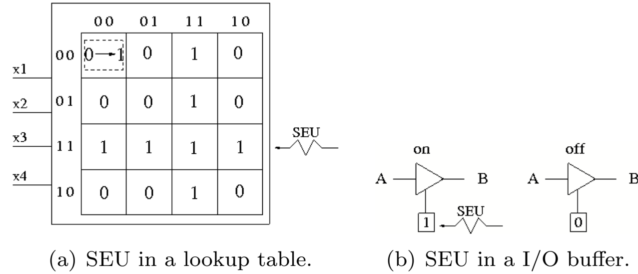
this case the logic function implemented by the LUT changes from the correct function $y = x_1 \cdot x_2 + x_3 \cdot x_4$ to the faulty function $y_f = x_1 \cdot x_2 + x_3 \cdot x_4 + \bar{x_1} \cdot \bar{x_2} \cdot \bar{x_3} \cdot \bar{x_4}$.

It may be observed that in the example the behavior of the fault-free LUT and the faulty one is different only when the values of the input signals are (0000).

An SEU in the configuration bit of an I/O buffer causes an undesired connection or disconnection between two wires. Figure 2(b) shows a 1 to 0 bit flip causing a disconnection between points A and B.

As an example of an untestable fault, assume, with reference to Figure 3, that an SEU affects the configuration bit of `LUT_2` (a logical OR) associated with input (10). This input configuration can never appear at the input of `LUT_2`, since the first bit is the logical AND of `i0` and `i1` and the second bit is the logical OR of the same signals, so that this fault is untestable because unexcitable.

An example of masked fault is a flip in the configuration bit of `LUT_0` (a logic AND) corresponding to input (11). In this case, the output of `LUT_0` is zero and the output of `LUT_1` is one, so that the output of `LUT_2` takes the correct value, thus masking the fault in `LUT_0`.



(a) SEU in a lookup table. (b) SEU in a I/O buffer.

Figure 2: Effects of an SEU in lookup tables and buffers.

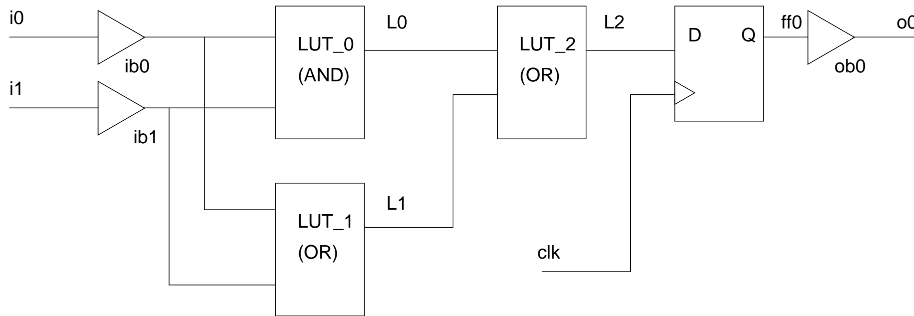


Figure 3: An example netlist.

2.2 SEUs in Routing Components

The basic elements in the FPGA interconnect are wires and *programmable interconnect points* (PIP). A PIP is a programmable switch that may connect two or more wires, controlled by one or more configuration bits. Figure 4 shows two PIPs, P_1 and P_2, which can connect one wire (a and b) to one or more wires of a set; in this example, a is connected to x and b to y. Since a configuration bit may also control more than one PIP, an SEU in a configuration bit may affect several nodes [40].

More precisely, the PIP model considered is based on the one found in the Xilinx Virtex-II devices. This type of PIPs is controlled by two to four configuration memory bits, and when one of these bits is affected by a bit-flip it can generate one of the topological modifications considered by the proposed method. Routing and multiplexers are not controlled by single configuration memory bits but by a group of them. It is possible that one configuration bit controls more PIPs but not multiple multiplexers.

The topological modifications caused by a faulty configuration bit can be described as follows [40]: (i) *Open*, where two wires are disconnected; (ii) *antenna*, where a new connection is established between an unused wire and a used one; (iii) *conflict*, where a new connection is established between two used wires; and (iv) *bridge*, where an existing connection is broken and a new one is

Table 1: Logical effects of routing faults.

Logical effect	Description
Stuck-at-0 on P_1 (P_2)	$x = 0$ ($y = 0$)
Stuck-at-1 on P_1 (P_2)	$x = 1$ ($y = 1$)
Bridge P_1 — P_2	$x = b$, $y = a$
Wired-AND P_1 — P_2	$x = y = a \wedge b$
Wired-OR P_1 — P_2	$x = y = a \vee b$
Wired-MIX P_1 — P_2 (P_1)	$\begin{cases} x = 1, y = 0 & \text{if } a \neq b \\ x = a, y = b & \text{if } a = b \end{cases}$
Wired-MIX P_1 — P_2 (P_2)	$\begin{cases} x = 0, y = 1 & \text{if } a \neq b \\ x = a, y = b & \text{if } a = b \end{cases}$

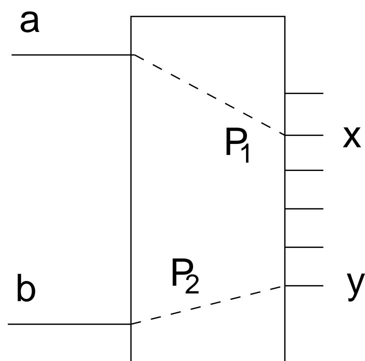


Figure 4: Routing example.

created between a used wire and one wire of the broken connection.

The effects of these topological modifications on the circuit's operation depend on several factors, and can be modeled at the logical level [40] as shown in Table 1, taking Figure 4 as an example. Note that two nodes take complementary values when affected by a wired-mix, therefore the two lines for this effect specify which node takes the 1 value.

It may be observed that a given SEU in the configuration bit associated with a PIP can propagate to different routing segments, and that the same SEU can have different effects on the routing segments through which it propagates, as discussed in [40].

An example of untestable routing fault is shown in Fig. 5. If the fault-free network on the left is affected by a Wired-OR between P_1 and P_2, it becomes logically equivalent to the one on the right, which in turn is equivalent to the fault-free one.

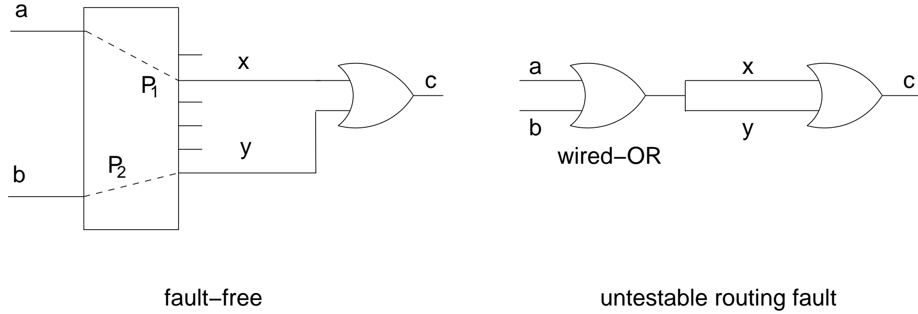


Figure 5: Example of untestable routing fault.

3 Model checking

Model checking [12] is an automated verification technique used for the specification and verification of concurrent systems. In particular, model checking relies on generating a finite state model of the system. A state space is a graph consisting of nodes representing states of the system, and edges representing state transitions. The state transition graph contains all possible behaviors.

Properties of the system are then expressed in a *temporal logic* language with respect to this model; for example, one may specify that a given relationship among system variables (such as signal values) holds for certain subsets of the state space, e.g., for all states, or for some states following a state where another condition holds, or other possible time orderings. Properties expressed in temporal logic are automatically verified by model-checking algorithms.

Let `p` and `q` be temporal logic formulas. Typical temporal operators are:

- `G(p)` means that `p` is always true;
- `F(p)` means that `p` will eventually be true;
- `U(p,q)` means that `p` is true until a state is reached where `q` is true; and
- `X(p)` means that `p` is true in the next state.

Typical properties expressed with temporal logic formulas are *safety*, in the form $\mathtt{G}(\neg\chi)$, stating that the undesired condition χ is never satisfied, and *liveness*, in the form $\mathtt{G}(\mathtt{F}(\psi))$ or $\mathtt{G}(\gamma \rightarrow \mathtt{F}(\psi))$, stating that the desired condition ψ will be eventually satisfied or that the desired condition ψ will be eventually satisfied if condition γ is satisfied.

Given a state transition graph M and a formula ϕ, the assertion $M \vdash \phi$ means that property ϕ holds for the system specified by M, i.e., M is a *model* of the formula ϕ.

The verification procedure is an exhaustive search of the state space. Moreover, if a property is not satisfied, model checking will produce a counterexample that can be used to locate the problem in the system.

3.1 The SAL Environment

The Symbolic Analysis Laboratory (SAL) is a framework for the formal modeling and analysis of systems [3]. Systems are modeled in the SAL language and their properties can be specified in temporal logic, and verified by the SAL model checker.

The SAL language is a strongly-typed description language. Supported types are Booleans, scalars, integers and integer subranges, records, arrays and abstract data types. Expressions consist of constants, variables, applications of Boolean, arithmetic, bit-vector operations and array and record selection and update.

A SAL model is made of one or more `Modules`. A SAL module is a self-contained specification of a transition system. A module consists of a state set definition, an `Initialization` of the state and a list of `Transitions`.

The state set is defined by four disjoint sets of `Input`, `Output`, `Global` and `Local` variables. The input and global variables are *observed*, in the sense that their value can only be read. The output and local variables are *controlled*, in the sense that their value can be both read and written. Each SAL variable has two values, the *current* value (denoted, e.g., by `x`) and the *next* value (denoted, e.g., by `x'`) valid in the current and the next state (respectively) of the module.

The transitions of a module can be specified *variable-wise*, by means of `Definitions`, or *transition-wise* by means of `Transitions`.

A definition is a system invariant represented as a simple assignment between a controlled variable and the result of an expression, as follows:

$$\textit{variable} = \textit{expression;}$$

Transitions are assignments between next-state variables and the result of expressions. A transition is of the form:

$$\textit{variable'} = \textit{expression;}$$

The initialization is used to specify an initial value for all or some of the controlled state variables of the module.

A list of transitions can be specified as a `Guarded Command`. A guarded command is composed of a guard, i.e., a Boolean condition defined on state variables, and one or more transitions. The guard must be satisfied in order to perform the transitions. A guarded command is of the form:

$$\begin{aligned} \textit{guard} \rightarrow\ & \textit{variable}_1\textit{'} = \textit{expression}_1 \\ & \dots \\ & \textit{variable}_n\textit{'} = \textit{expression}_n \end{aligned}$$

The SAL language provides the `IN` construct that denotes choice among a set of values.

The language allows the composition of different modules. Several modules can be collected in a SAL `Context`. Contexts may also include constants, types declarations and theorems.

The transition graph built by SAL explores all possible reachable states starting from the initial state and for every possible combination of the input variables.

The SAL-SMC (Symbolic Model checker) allows properties to be specified in Linear Time temporal Logic (LTL) and in Computation Tree Logic (CTL). In particular, we use LTL as an assertion language.

4 The UA^{2}TPG tool

UA^{2}TPG works on a representation of the FPGA-based system at the netlist level obtained before the place-and-route phase of the FPGA development process. This representation is sufficient to analyze logic faults. In order to analyze routing faults, the E^2STAR tool is used after place-and-route to generate the list of possible faults with their effects.

In the following, we first show how to use the SAL language to model the netlist and SEUs occurring in the configuration memory of the device, then we show how to write an untestability theorem using LTL and finally we illustrate the execution flow of the tool.

4.1 Modeling SRAM-based FPGA Netlists

A netlist is described by a SAL `MODULE`. Inputs and outputs of the system are modeled by `INPUT` and `OUTPUT` variables. Each component in the netlist is modeled as a `LOCAL` variable that represents the output of the component itself.

The semantics of each component except flip-flops is described by `Definitions`. The semantics of an input buffer can be simply described as an assignment between a local variable, modeling the buffer, and an input variable, modeling the associated input pin. Similarly the semantics of an output buffer can be described as an assignment between two local variables: the variable modeling the input to the buffer is assigned to the variable modeling the buffer. Examples of SAL specifications input/output buffers, which refers to the circuit of Figure 3 are:

```
ib0 = i0; ob0 = ff0;
```

The semantics of LUTs is described by the corresponding logic functions. The 2-input lookup table `LUT_2` implementing the OR function is modeled as:

```
L2 = L0 OR L1;
```

where `L2`, `L0`, and `L1`, are the logical variables representing the outputs of the LUTs.

Multiplexers can be described by an `IF THEN ELSE` clause:

```
y IN IF(s=FALSE) THEN x1 ELSE x2 ENDIF;
```

where s represents the select signal for the multiplexer.

Flip-flops are described by `Transitions`. D-flip-flops can be described as a simple assignment between the next value of the flip-flop and the current value of its input:

```
ff0' = L2;
```

where L2 represents the input signal and ff0 the content of flip-flop. Other types of flip-flops can be described by an IF THEN ELSE clause. For example, FDCE flip-flops are described as follows, where q represent the state of the flip-flop, e represents the clock enable signal, c the clear signal and d the input signal.

```
q' IN IF(c=TRUE)
        THEN {FALSE}
      ELSIF (e=FALSE)
        THEN {q}
      ELSE {d} ENDIF;
```

4.2 Modeling SEUs in the Configuration Memory

We model the effects of SEUs affecting the configuration bits associated with logic components by modifying the functionality performed by the faulty component.

We model SEUs affecting configuration bits controlling routing resources, according to the caused logical effect. For example, let a and x be two wires connected through a PIP P_1, with a carrying the output of a component C. An SEU causing a stuck-at 0 on P_1 is modeled by changing the function performed by C to $FALSE$.

As described in Section 2, a given SEU in a configuration bit controlling a PIP can be propagated through a number of routing segments with different logical effects. Thus, an SEU having multiple propagation points is modeled by modifying the functions performed by all the affected components, according to the logical effects associated with the SEU.

In practice, modeling an SEU results in changing the SAL representation of the netlist, which provides a complete and detailed description of the system structure. A detailed knowledge of the underlying FPGA architecture, in particular the physical location of the configuration bits controlling each resource, is not needed. Each fault generated by the tool is a possible configuration of a given resource, which differs from the correct one as specified by the netlist, independent of which particular bit causes a fault.

We observe that most of the untestable SEUs in LUTs may be related to don't-care configuration bits. As an example, let us consider a 3-input logic function mapped on a 4-input LUT: In this case 8 out of the 16 configuration bits controlling the LUT will be don't-care and, if faulty, will not affect the correct behavior of the circuit. Nevertheless, when generating test patterns for fault detection, is very difficult or even impossible for the design/test engineer to have a detailed knowledge of the don't-care configuration bits. Thus, without an automatic untestability analyzer, test pattern generation tools would be asked to try to test every possible SEU. Moreover, when considering SEUs in the routing resources, the untestability of an SEU is caused by much more complex logic masking effects that are very hard to statically determine at design time; thus again, an automatic untestability analysis tool such as UA^{2}TPG would be beneficial to the generation of test patterns.

4.3 System specification

In order to analyze the untestability of a given SEU, we build the SAL model of the composition of the correct circuit and of the faulty one, we connect them to the same inputs and we check whether the outputs of the two systems are always the same or not.

Figure 6 reports the specification of the simple netlist shown in Figure 3, where `LUT_0` implements an AND function, `LUT_1` and `LUT_2` implement the OR function.

More precisely, the SAL code describes both the fault-free circuit and the circuit resulting from the SEU in the configuration bit of `LUT_2` associated with input (10). The tool produces the function computed by `LUT_2` after the fault, in a sum-of-products form ($(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2)$, where x_1 and x_2 are the LUT inputs). In the specification of the faulty circuit, variables are renamed by adding the suffix `_F`.

The two circuits share the same inputs, including the clock, and they have the same definitions and transitions, except for the definition of the faulty component. The SAL model checker proves that the SEU in this example is untestable.

The model of the system is a transition system describing the system behavior. Whenever the clock (`clk`) is true, the corresponding guarded command is executed, thus computing the new state of both circuits. If n is the number of input variables, there are 2^n possible inputs at each clock cycle. The transition system built by SAL explores all possible reachable states, starting from the initial state and applying every possible combination of the inputs at every clock cycle.

We observed that, even for a small circuit, the number of all faults is very high. To mitigate this problem, we analyze the circuit in a modular way.

For each combinational component C being tested, we prune the whole circuit of the components (including device input and output terminals) that do not affect (directly or indirectly) the inputs of the component (i.e., not in the input cone), or are not affected by its output (not in the output cone), or do not affect the components in the output cone. We call the resulting subcircuit the *region* of C. Figure 7 illustrates this idea.

A similar method is used for faults in PIPs, where we extract the regions of the components affected by each fault.

4.4 Identifying Untestable SEUs

An SEU is untestable if the output of the correct circuit always equals the output of the faulty circuit for the same inputs. We define an untestability theorem as an LTL safety formula, in the form $G(\neg(O_0 \neq O_0^F \ \vee \ \ldots \ \vee O_n \neq O_n^F)))$ where O_i is the i-th output of the correct circuit and O_i^F is the i-th output of the faulty circuit. Such formula states that it is always false that the output of the correct system is different from the output of the faulty circuit. Thus, if for a given SEU the theorem is proved, the SEU is demonstrated to be untestable.

In Figure 6, we show the theorem `untestability_th` for the analysis of the untestability of the SEU of the circuit of Figure 3.

The tool produces a list of untestable faults, reporting the affected component and the associated input configuration. A sample output for the considered example follows:

```
5;      %number of untestable faults
LUT_2 10;
LUT_1 11;
LUT_0 11;
LUT_0 01;
LUT_0 10;
```

4.5 Generating Test Patterns for Testable SEUs

If the theorem is not proved, a counter-example is automatically produced by the model checker. In general, a counter-example is one of the possible sequences of assignments of the inputs of the model that caused the theorem not to be proved. In particular, the counter-example provided by SAL-SMC after trying to prove an untestability theorem is a sequence of input vectors applied to the inputs of the system that caused the output of the faulty system to be different from the output of the correct one. In other words, the sequence of input vectors produced by the model-checker in the counter-example is a test pattern able to test the SEU injected in the system. In Figure 8 we show the counter-example provided by the SAL model checker when trying to demonstrate the untestability theorem related to the SEU in the configuration bit of `LUT_2` (see Figure 3) associated with input configuration (00). The counter-example shows that the untestability assumption is violated at least when the input sequence $(00), (01)$ is applied. The first test vector places (00) at the inputs of `LUT_2`, and consequently the D input of the flip-flop receives a 0 in the fault-free circuit and a 1 in the faulty one. This difference appears at the circuit output when the second test vector is applied. In this case, any value of the second test vector would do, and $(0, 1)$ is the value selected by the tool.

The UA^{2}TPG tool extracts a test pattern for each testable fault from the model checker output, as shown below:

```
LUT_0 00;  2; (0,0) (1,0);
LUT_1 00;  2; (0,0) (0,1);
LUT_1 10;  2; (1,0) (0,1);
LUT_1 01;  2; (0,1) (0,1);
LUT_2 00;  2; (0,0) (0,1);
LUT_2 01;  2; (0,1) (1,0);
LUT_2 11;  2; (1,1) (1,0);
```

Each row reports the affected component, the input configuration associated with the fault, the length of the test pattern and the sequence of test vectors. For example, the test pattern `(1,1)(1,0)` can be applied to test the SEU in the configuration memory of LUT_2 corresponding to the LUT input `(1,1)`.

5 The Analysis Environment

The proposed tool works in conjunction with an EDIF parser and the E^2STAR tool [10]. The parser is a tool able to translate the EDIF description of the netlist, produced by the HDL synthesis tool, into an intermediate description of the topology of the netlist in terms of connections among logic components and functionalities performed by components. Moreover, the parser is also able to produce a list of the effects of SEUs occurring in configuration bits associated with the logic resources used by the system under analysis. In particular, the parser produces the list of the faulty functions associated to each LUT in the system and to each SEU in the configuration memory controlling the LUT. E^2STAR is a static analyzer of the configuration memory of SRAM-based FPGA devices. Given an FPGA device and a placed-and-routed design, E^2STAR is able to determine the configuration bits actually used by the design and which are the logical effects of SEUs occurring in the configuration bits controlling the routing resources, according to the fault model previously presented. E^2STAR analyzes the configuration bits affecting all routing resources, including the ones that dispatch signals within CLBs.

Faults are injected over all configuration bits except those controlling unused resources, i.e., faults are not injected over bits that do not affect either logic functions or the architecture topology, therefore the configuration bits analyzed include unused or not programmed bits that could generate shorts.

The overall structure of the analysis environment is shown in Figure 9. After the HDL specification of the system has been synthesized, the netlist description file and the list of the effects of SEUs in the configuration bits controlling logic resources are generated by the parser from the EDIF representation of the netlist.

E^2STAR produces a file containing the list of the SEUs in the configuration bits associated with the routing elements of the FPGA. For each SEU the file produced by E^2STAR contains the list of all the components to which the SEU propagates, and the logical effect that each SEU has on each involved component.

An excerpt of the output of E^2STAR is the following:

```
15 2;
0 bridge 22 0 21 0;
1 stuck-at-1 16 0;
```

This entry means that routing fault number 15 has two effects: a bridge between input pin 0 of component 22 and input pin 0 of component 21, and a stuck-at-1 on input pin 0 of component 16.

The overall execution flow of UA^{2}TPG is shown in Figure 10. The Logic Fault list contains the list of SEUs in the LUTs of the implemented system, and the faulty LUT function associated with each SEU. The Routing Fault list contains the list of the possible SEU in the configuration memory controlling routing resources, and for each SEU, the list of its effects. The tool performs the following steps:

1. Build the model of the fault-free system (as described in Section 4.3) starting from the Netlist Description file.
2. Build the untestability theorem (as described in Section 4.4).
3. For each SEU α:
   (a) Build the model of the faulty system from the logical effects induced by the SEU.
   (b) Invoke SAL-SMC on the untestability theorem.
      - If the untestability theorem is proved (thus α is untestable) then save α in the list of the untestable SEUs.
      - If the untestability theorem is not proved (thus α is testable) then extract the test pattern able to detect α from the counter-example provided by SAL-SMC (as described in Section 4.5) and save it in the list of test patterns.

At the end of the untestability analysis the list of the untestable SEUs and the list of test patterns for all the testable SEUs are generated. The list of test patterns contains a test for each testable SEU. This list can then be compressed by eliminating all the duplicated test patterns and all those test patterns that are prefix of longer ones.

Note that the UA^{2}TPG could also be applied earlier during the design process, before the place-and-route phase. In this case the analysis carried out by E^2STAR is not needed and the untestability analysis is performed only on the configuration bits controlling the logic resources. Moreover, it is worth noting that, while E^2STAR and the EDIF parser work only on Xilinx devices, UA^{2}TPG, which is actually the core of the proposed analysis flow, is completely independent of the FPGA vendor and model.

6 Experimental results

In the following, results of the application of the proposed tool to some circuits from the ITC'99 benchmark [14] are reported. This is a set of benchmark circuits that are meant to be used for experimentation on Design for Testability (DFT) and Automatic Test Pattern Generation (ATPG). The tested circuits (Table 2) provide a diversified set of test cases composed of sequential circuits with a single clock signal, no tristate buses or internal memories, modeled at the RTL level, ranging from 4 to 106 LUTs and from 4 to 59 FFs.

We synthesized the VHDL code of the circuits using the Xilinx ISE CAD tool. As a target device we adopted the Xilinx Virtex-II XC2VP30 device. The characteristics of the designs used in the experiments are shown in Table 3, which reports for each circuit the number of SEUs affecting logic and routing resources (columns LSEU and RSEU, respectively), the number of look-up tables (LUT), flip-fLops (FF), multiplexers (Mux), input and output buffers (IB and OB).

Table 2: Selected benchmark circuits.

Circuit	Function
SFC	Compare serial flows
BCD	Recognize binary coded decimal numbers
ARB	Resource arbiter
HDR	Interrupt handler
SEQ	Find inclusions in sequences of numbers
CVT	Serial-to-serial converter
VOT	Voting system
MET	Interface to meteo sensors

Table 3: Characteristics of the benchmarks.

Circuit	LSEU	RSEU	LUT	FF	Mux	IB	OB
SFC	124	547	9	5	0	3	2
BCD	52	304	4	4	0	2	1
ARB	954	5,910	76	37	0	5	4
HDR	104	566	9	8	0	3	6
SEQ	504	2,689	40	21	0	10	4
CVT	692	3,872	53	28	0	2	1
VOT	660	3,942	52	24	0	12	6
MET	1,216	7,203	106	59	11	11	10

The computer used for the experiments was equipped with an Intel Core i5 (QuadCore) 2.67 GHz, 256 KB L1 Cache, 1 MB L2 Cache, 8MB L3 Cache, 4 GB RAM.

In Table 4 we show the results of the analysis performed by E^2STAR to the considered circuits. The table shows the number of critical configuration memory bits (RF) identified by the tool, and the number of affected nodes classified by logical effect: Stuck-at-0 (Sa0), Stuck-at-1 (Sa1), Wired-And (Wand), Wired-Mix (Wmix), and Bridge (Br). In the examined circuits, Wired-Or effects were not observed. It may be observed that, as we previously discussed, the number of propagation points per SEU in the configuration bits controlling the routing structure is much higher than the actual number of SEUs itself.

Results obtained from the application of the proposed tool are shown in Table 5. The table shows for each circuit the total number of faults in configuration bits controlling both logic and routing resources (SEU column), the number of untestable SEUs affecting logic resources (UL), the number of untestable SEUs affecting routing resources (UR), the total number of untestable SEUs (UT), and the execution time (Time), including check for untestability and

Table 4: Effects of SEUs in the routing elements.

Circuit	RF	Sa0	Sa1	Wand	Wmix	Br
SFC	547	708	2,944	5	7	0
BCD	304	118	339	5	7	102
ARB	5,910	8,105	21,661	1,423	1,431	2,320
HDR	566	372	790	0	18	305
SEQ	2,689	3,074	8,061	464	496	1,217
CVT	3,872	6,569	15,948	567	512	1,908
VOT	3,942	4,603	10,727	482	692	1,498
MET	7,203	10,390	27,720	1,143	1,387	3,602

Table 5: Results from the application of UA^{2}TPG.

Circuit	SEU	UL	UR	UT	Time (min)
SFC	671	0	1	1	1.29
BCD	356	6	2	8	0.66
ARB	6,864	694	214	908	62.80
HDR	670	8	6	14	1.24
SEQ	3,903	76	66	142	16.34
CVT	4,564	244	73	317	35.48
VOT	4,602	288	136	424	31.39
MET	8,419	640	671	1,311	263.10

test pattern generation. Figure 11 shows, for each circuit, the percentage of untestable SEUs in configuration bits controlling logic (UL/LSEU) and routing (UR/RSEU) resources, respectively, and the total untestability percentage (UT/SEU).

The experiments show that all the considered circuits have a number of faults that cannot be tested. The average untestability is 6.6%. The highest untestability is 15.5% for the meteo sensors interface, while the lowest is 0.15% for the serial flow comparator. SEUs in logic resources seem to be much harder to test than SEUs in routing resources. This may be explained, if we take two points into account: (i) the excitation of an SEU in a configuration bit controlling a LUT depends on the values of all the inputs of the LUT while, as we previously discussed, the excitation of an SEU in a configuration bit controlling a PIP depends on the value of one or two signals; and (ii) as we previously discussed, each SEU in the routing structure has a very large number of propagation points, thus they are more likely to be propagated. If we consider only SEUs in logic resources we find an average untestability of 29.8%, with a

Table 6: Results for previous tools (presented in [5, 7]).

Circuit	Unex-L	Unex-R	Unex-SEU
SFC	0	0	0
BCD	6	0	6
ARB	462	170	632
HDR	8	0	8
SEQ	21	0	21
CVT	190	0	190
VOT	231	0	231
MET	483	321	804

peak of about 72.7% for the resource arbiter. Considering only SEUs in routing resources we find an average untestability of 2.8%, with a peak of about 9.3% for the Meteo sensors interface.

Table 6 shows results only about the unexcitability of SEUs in logic and routing components of the considered circuits, calculated with the tools presented in [5, 7]. Comparing this table with Table 5, we note that a considerable number of faults are always masked.

6.1 Test pattern generation

Table 7 reports results from the test pattern generation performed by UA^{2}TPG. The table shows the number of faults of each circuit (SEUs) and the length of the test patterns generated by the proposed tool before (TPLen) and after (Compacted) compaction. TPLen has been computed as the sum of the length of the tests including the reset cycle between different tests. The table shows that TP compression is very effective in reducing the length of TPs. This efficacy can be explained by the fact that the model checker methodically explores the input space in a fixed order, thus producing counter-examples having long prefixes in common. Longer TPs tend to contain shorter TPs, which are eliminated by compaction. Further improvements are not currently supported by the tool, but additional techniques, such as genetic algorithms [4], can be applied to minimize the test pattern set.

In order to assess the effectiveness of the test pattern generation process performed by UA^{2}TPG, we performed a random test experiment using the SEU simulator presented in [8]. Each circuit was simulated by applying 10000 randomly generated test patterns and simulating the occurrence of each SEU in the configuration memory of the circuit, one at a time. Table 8 reports the fault coverage values obtained by this experiment. It can be noticed that, although the number of random test patterns was always (apart from the meteo sensors interface) much larger than the number of test patterns generated by UA^{2}TPG, the fault coverage obtained with random testing is always lower than 100%, and, for 4 out of 8 circuits, even lower than 50%.

Table 7: Automatic Test Pattern Generation Results.

Circuit	SEUs	TPLen	Compacted
SFC	671	3,424	460
BCD	356	2,292	224
ARB	6,864	58,890	1,062
HDR	670	2,930	258
SEQ	3,903	99,218	6,307
CVT	4,564	86,374	5,456
VOT	4,602	43,032	5,198
MET	8,419	219,564	14,827

Table 8: Random testing coverage (10.000 test vectors).

Circuit	Random_Cov
SFC	100.0%
BCD	98.0%
ARB	47.8%
HDR	98.7%
SEQ	4.9%
CVT	45.5%
VOT	76.8%
MET	29.8%

6.2 Validation of Results

The same simulator mentioned above was used to assess the correctness of the results produced by UA^{2}TPG. First, the SEUs identified as testable by UA^{2}TPG were exhaustively injected and the simulated circuit was fed with the test patterns generated by the proposed tool: In this way we verified that the test patterns generated by UA^{2}TPG were actually able to detect all the testable SEUs. Then, in order to verify that the faults identified as untestable by UA^{2}TPG were actually untestable, we exhaustively injected them and we fed the simulated circuit with 100,000 randomly generated test patterns: this additional validation experiment showed that none of the faults identified by UA^{2}TPG as untestable had been detected.

6.3 Performance

Table 9 compares computational and time complexity for the different circuits, related to the respective number LR of logical resources (LUTs, flip-flops, and

Table 9: Complexity of the SAL-generated models.

Circuit	LR	IN	NTR	Time (min)
SFC	14	3	1053	1.29
BCD	8	2	110	0.66
ARB	113	5	10009	62.80
HDR	17	3	115	1.24
SEQ	61	10	1002	16.34
CVT	81	2	1434	35.48
VOT	78	12	1266	31.39
MET	176	11	10010	263.10

multiplexers) and number IN of inputs. The number NTR of nodes in the transition relationship for each SAL-generated model has been adopted as a measure of computational complexity. Since this value varies with each different fault, the maximum value is reported. The time required for the analysis performed by UA^{2}TPG ranges from some seconds up to some minutes for very small and medium size circuits. For larger circuits the required time is a few hours. We believe that these times are reasonable taking into account the inherent complexity of sequential ATPG. In fact, the aim of this work is achieving full coverage of testable faults, thus guaranteeing correct functionality. This is achieved at a high cost in time complexity for the *off-line* test pattern generation, but it provides a very effective and efficient set of patterns to be employed at run-time. Effectiveness results from the application of a formal method, and efficiency results from (i) analyzing only the resources actually used by the application, and (ii) excluding tests for untestable faults, which would included in test sets generated with random testing techniques.

Further, we observe that UA^{2}TPG is mainly intended to support the analysis of critical components in fault-tolerant designs. Ensuring the absence of SEUs in the configuration memory of these components, e.g., a voter in a TMR architecture, guarantees safe operation of the system, under the single failure assumption. Techniques for fault tolerant design include fault detection and masking and system recovery and rely on dedicated components, such as comparators, voters, and cyclic redundancy code checkers [37]. These critical components are relatively small compared to full systems, and a thorough testing of their circuits is affordable with the proposed tool, as indicated by the above results for the voting system and for the serial flow comparator. For the voter, the length of the test pattern is 5198 clock cycles to test 4602 SEUs, and for the comparator the length is 470 clock cycles to test 671 SEUs.

In general, untestability checking and test pattern generation can be applied to any modular part of a larger application. For example, they can be used in reconfigurable systems, where the FPGA is partitioned in regions which host different subsystems at different times [35].

7 Conclusions and Future Work

We have proposed a static analysis tool for the untestability proof and automatic test pattern generation for SEUs in the configuration memory of SRAM-based FPGA systems. The tool is based on an accurate fault model for both logic components and routing structures. The test patterns generated by the tool can detect 100% of the testable SEUs and may be used for in-service application-dependent testing of the system. The application of the tool to some circuits from ITC'99 benchmark shows that many circuits have a significant number of untestable faults.

Untestability results reduce the effort required by automated test patterns generators. From the point of view of fault tolerance, UA^{2}TPG can perform a worst case assessment of the sensitivity to SEUs of FPGA-based systems and the generated test patterns can be used to drive the circuit in fault injection or radiation testing experiments.

As further work we intend to explore advanced techniques as *on-the-fly* model-checking [21] (that avoids an explicit construction of the complete state space) and *state-space abstraction* [13] (that works on conservative over-approximations of the system states) to tackle the state-space explosion problem typical of model checking approaches.

Acknowledgments

The authors wish to thank the anonymous referees for their valuable comments and suggestions.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. John Wiley & Sons., 1990.

[2] R.C. Baumann. Radiation-induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305 – 316, September 2005.

[3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An Overview of SAL. In *Proceedings of the Fifth NASA Langley Formal Methods Workshop (LFM 2000)*, pages 187–196, 2000.

[4] C. Bernardeschi, L. Cassano, M.G.C.A. Cimino, and A. Domenici. Application of a genetic algorithm for testing SEUs in SRAM-FPGA Systems. In *Proceedings of the 6th HiPEAC Workshop on Reconfigurable Computing (WRC2012)*, 2012.

[5] C. Bernardeschi, L. Cassano, and A. Domenici. SEU-X: a SEu UnuXecitbility prover for SRAM-FPGAs. In *Proceedings of the 18th IEEE International On-Line Testing Symposium (IOLTS2012)*, June 2012.

[6] C. Bernardeschi, L. Cassano, A. Domenici, G. Gennaro, and M. Pasquariello. Simulated Injection of Radiation-Induced Logic Faults in FPGAs. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011.

[7] C. Bernardeschi, L. Cassano, A. Domenici, and L. Sterpone. Unexcitability Analysis of SEUs Affecting the Routing Structure of SRAM-based FPGAs. In *Proceedings of the Great Locations Symposium on Very Large Scale of Integration (GLSVLSI2013)*, May 2013.

[8] C. Bernardeschi, L. Cassano, A Domenici, and L. Sterpone. ASSESS: A Simulator of Soft Errors in the Configuration Memory of SRAM-Based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(9):1342–1355, Sept 2014.

[9] Cinzia Bernardeschi, Luca Cassano, Mario G.C.A. Cimino, and Andrea Domenici. GABES: A genetic algorithm based environment for SEU testing in SRAM-FPGAs. *J. of Systems Architecture*, 59(10, Part D):1383–1254, 2013.

[10] Cinzia Bernardeschi, Luca Cassano, Andrea Domenici, and Luca Sterpone. Accurate Simulation of SEUs in the Configuration Memory of SRAM-based FPGAs. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2012)*, 2012.

[11] Carl Carmichael, Earl Fuller, Phil Blain, and Michael Caffrey. SEU mitigation techniques for Virtex FPGAs in space applications. In *Proceeding of the Military and Aerospace Programmable Logic Devices International Conference (MAPLD)*, 1999.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[13] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[14] F. Corno, M. Sonza Reorda, and G. Squillero. RT-Level ITC'99 Benchmarks and First ATPG Results. *IEEE Des. Test*, 17:44–53, July 2000.

[15] S. Eggersgluss and R. Drechsler. A highly fault-efficient SAT-based ATPG flow. *Design Test of Computers, IEEE*, 29(4):63–70, Aug 2012.

[16] European Committee for Electrotechnical Standardization (CENELEC). EN 50129: Railway applications - Communications, signaling and processing systems - Safety related electronic systems for signaling, February 2003.

[17] International Organization for Standardization (ISO). 26262-5: Road vehicles - Functional safety - Part 5. Product development: hardware level, December 2009. Draft.

[18] G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *Proceedings of the International Conference on Software Testing Verification and Validation*, pages 41–50, April 2009.

[19] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, September 2009.

[20] P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson. Consequences and Categories of SRAM FPGA Configuration SEUs. In *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03)*, September 2003.

[21] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 191–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] N.G. Herron, E.J. Thorne, Q. Wang, A. Correale, and T.A. Dick. Testing a programmable logic device with embedded fixed logic using a scan chain, jul 18 2006. (US Patent 7,080,300).

[23] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[24] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. Exploiting Self-Reconfiguration Capability to Improve SRAM-based FPGA Robustness in Space and Avionics Applications. *ACM Transactions on Reconfigurable Technology and Systems*, 4:8:1–8:22, December 2010.

[25] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, Jan 1992.

[26] D.E. Long, M.A. Iyer, and M. Abramovici. FILL and FUNI: Algorithms to Identify Illegal States and Sequential Untestable Faults. *ACM Transaction on Design Automation of Electronic Systems*, 5(3):632–657, 2000.

[27] Alexander Miczo. *Digital Logic Testing and Simulation*. John Wiley & Sons., 2003.

[28] J. Raik, H. Fujiwara, R. Ubar, and A. Krivenko. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In *Proceedings of the 17th Asian Test Symposium (ATS'08)*, pages 21–26, 2008.

[29] J. Raik, A. Rannaste, M. Jenihhin, T. Viilukas, R. Ubar, and H.; Fujiwara. Constraint-Based Hierarchical Untestability Identification for Synchronous Sequential Circuits. In *Proceedings of the 16th European Test Symposium (ETS'11)*, pages 147–152, 2011.

[30] J. Raik, R. Ubar, A. Krivenko, and M. Kruus. Hierarchical Identification of Untestable Faults in Sequential Circuits. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'07)*, 2007.

[31] M. Rebaudengo, M. Sonza Reorda, and M. Violante. A new functional fault model for FPGA application-oriented testing. In *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*, pages 372 – 380, 2002.

[32] M. Renovell, P. Faure, J.M. Portal, J. Figueras, and Y. Zorian. IS-FPGA : a new symmetric FPGA architecture with implicit scan. In *Proceedings of the International Test Conference*, pages 924–931, 2001.

[33] M. Renovell, J.M. Portal, P. Faure, J. Figueras, and Y. Zorian. Analyzing the Test Generation Problem for an Application-Oriented Test of FPGAs. In *Proceedings of the IEEE European Test Workshop*, pages 75 –80, 2000.

[34] M. Rozkovec, J. Jenicek, and O. Novak. Application Dependent FPGA Testing Method. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD10)*, pages 525 –530, sept. 2010.

[35] D. Sorrenti, D. Cozzi, S. Korf, L. Cassano, J. Hagemeyer, M. Porrmann, and C. Bernardeschi. Exploiting dynamic partial reconfiguration for on-line on-demand testing of permanent faults in reconfigurable systems. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 203–208, Oct 2014.

[36] P. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, Sep 1996.

[37] L. Sterpone, M. Sonza Reorda, M. Violante, F. Lima Kastensmidt, and L. Carro. Evaluating different solutions to design fault tolerant systems with SRAM-based FPGAs. *J. Electron. Test.*, 23(1):47–54, February 2007.

[38] M. Tahoori. Application-Dependent Testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):1024 –1033, 2006.

[39] D. Tille and R. Drechsler. A Fast Untestability Proof for SAT-based ATPG. In *Proceedings of the 12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'09)*, pages 38–43, 2009.

[40] M. Violante, N. Battezzati, and L. Sterpone. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer Science & Business Media, 2011.

```
untestability : CONTEXT =
  BEGIN
   untest_circuit : MODULE =
   BEGIN
   % Input pins and clock
    input clk, i0, i1 : boolean
   % Fault-free circuit
    local ib0, ib1 : boolean
    local L0, L1, L2 : boolean
    local ff0, ob0 : boolean
    output o0 : boolean
   % Faulty circuit
    local ib0_F, ib1_F : boolean
    local L0_F, L1_F, L2_F : boolean
    local ff0_F, ob0_F : boolean
    output o0_F : boolean
   DEFINITION
   % Fault-free circuit
    ib0 = i0; ib1 = i1;
    L0 = ib0 AND ib1; L1 = ib0 OR ib1;
    L2 = L0 OR L1;
    ob0 = ff0; o0 = ob0;
   % Faulty circuit
    ib0_F = i0; ib1_F = i1;
    L0_F = ib0_F AND ib1_F;
    L1_F = ib0_F OR ib1_F;
    L2_F = NOT L0_F AND L1_F OR L0_F AND L1_F;
    ob0_F = ff0_F; o0_F = ob0_F;
   INITIALIZATION
    ff0 = FALSE; ff0_F = FALSE;
   TRANSITION
    [ clk = false --> % do nothing
    [] clk = true -->
           ff0' = L2; ff0_F' = L2_F; ]
  END;
  untestability_th : THEOREM
    untest_circuit |- G(NOT(o0 /= o0_F));
END
```

Figure 6: Example of SAL specification

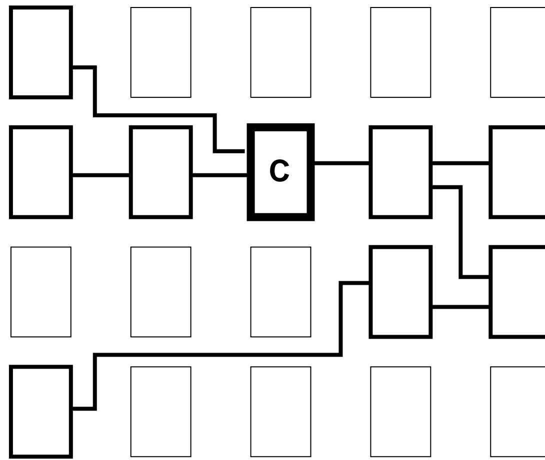


Figure 7: The region of component C (in thick lines).

```
Step 0:
--- Input Variables (assignments) ---
i0 = false
i1 = false

Step 1:
--- Input Variables (assignments) ---
i0 = false
i1 = true
```

Figure 8: A SAL-SMC counter-example

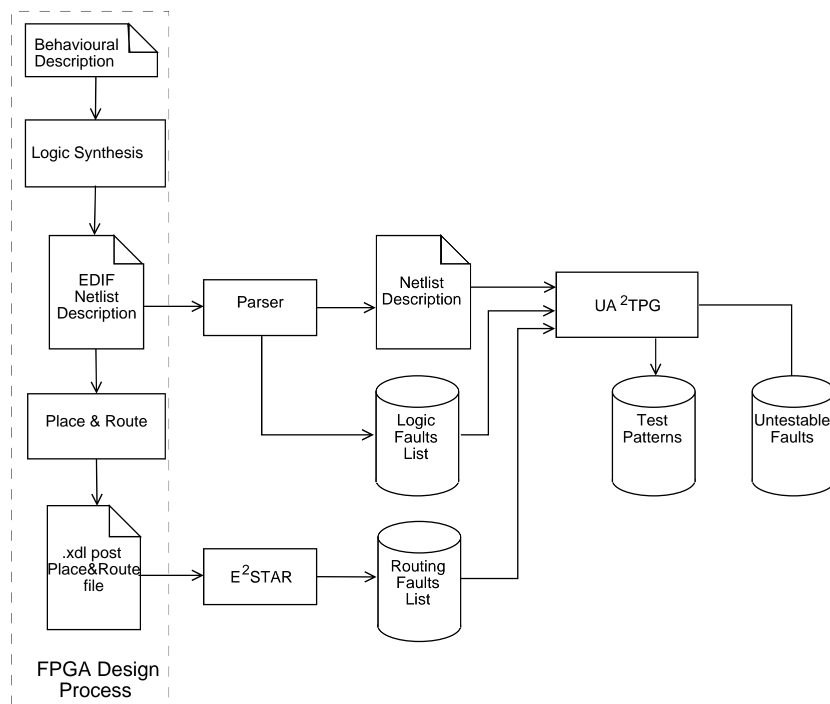


Figure 9: Flow Diagram of the Untestability Analysis and Automatic Test Pattern Generation Environment.

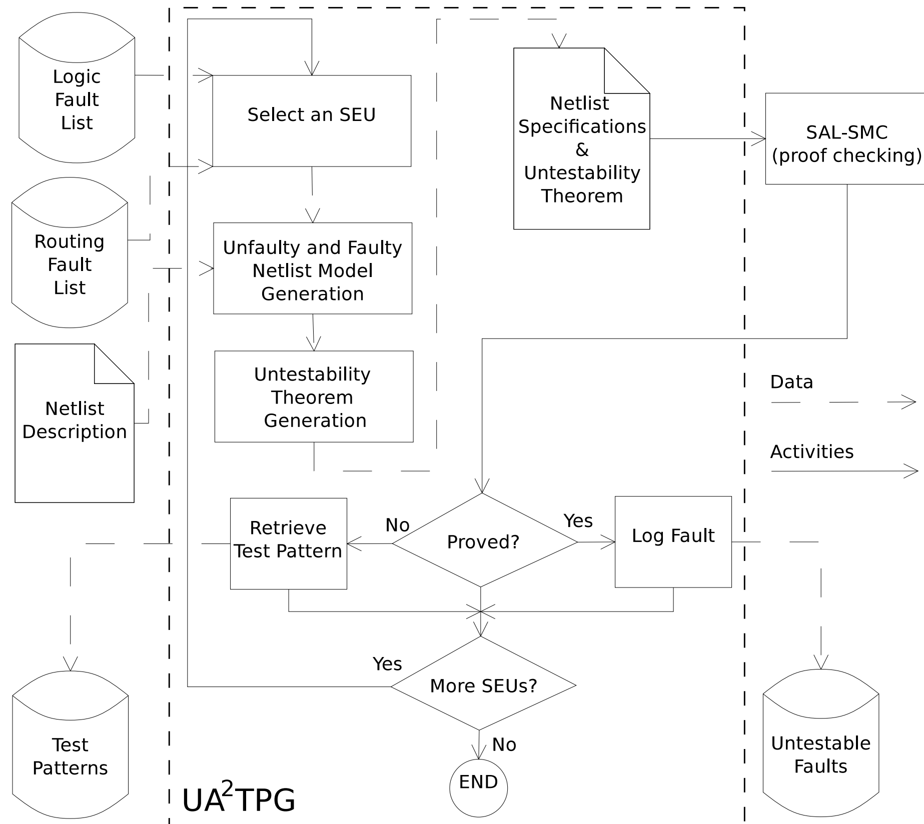


Figure 10: The execution flow of UA^{2}TPG.

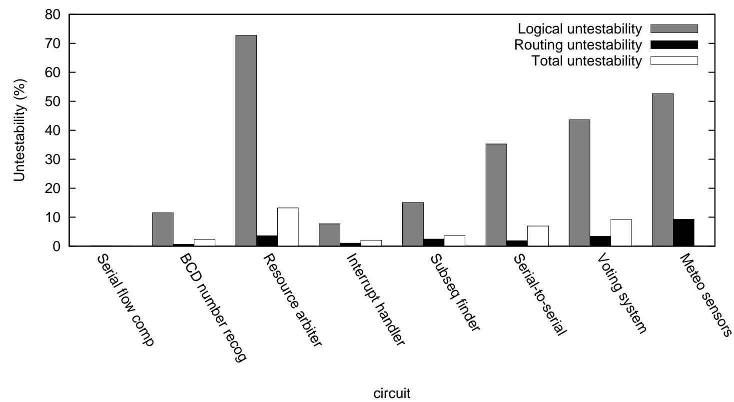


Figure 11: Fault untestability for the considered circuits.