Patrick Klampfl, BSc

Soft-Error-Analysis
Evaluating the Quality of Protection Logic

Master's Thesis

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Roderick Bloem, PhD

Institute for Applied Information Processing and Communications

Advisors:
Ayrat Khalimov, MSc, Dr. Robert Könighofer

Graz, December 2016

Affidavit

Abstract

With shrinking component sizes and an increasing density, hardware did not just become more powerful. As a side effect, it suffers from an increasing susceptibility to faults caused by cosmic radiation and alpha particles.

Today's critical systems rely on error-detecting or error-correcting codes. Verifying them by hand is a time-consuming and error-prone task, which is why we propose new methods that can be applied automatically.

First, methods to detect bugs in a protection logic are presented. They can be used to either find unprotected components or to test if a protection logic reports errors too often. This is achievable by thorough formal methods, by a fast but inaccurate simulation-based technique, or by a semi-formal approach that provides more flexibility.

Afterwards, an approach to verify definitely protected components is shown. It formally proves components that are resilient in each situation by over-approximating the reachable state-space.

Our approaches do not assume any particular structure of the circuits and can be applied early in the design process. The experimental results show that our semi-formal approach already outperforms simulation if just a few input values are set open.

Acknowledgments

I am indebted to a lot of people for supporting me for this thesis. First, I am grateful to my advisor Robert Könighofer. I could learn a lot from him while working on my Bachelor's thesis and in later projects. He laid the foundations for this thesis, for which I am very thankful.

My sincere thanks also goes to my advisor Ayrat Khalimov, who gave lots of valuable advice throughout the creation of this thesis. I would like to thank him for countless hours of discussions, a lot of patience and extensive helpful feedback.

Furthermore, I am grateful to Roderick Bloem for providing ample food for thought and for reviewing this thesis. I would also like to thank Georg Hofferek for his great lecture *Logic and Computability* that aroused my interest in formal methods, and Shiri Moran for taking the time to explain how protection logic is implemented in practice.

I owe special thanks to my colleagues and friends at TU Graz for all the fruitful discussions and enjoyable hours during the last five years and to my family for making my education possible.

Moreover, I want to express my appreciation and love for Judith. Thank you for always being there for me.

Finally, I want to thank all the people who guided and accompanied me along the way.

Contents

List of Figures

1. Introduction

1.1. Overview

It was already in 1965 when Gordon Moore described the exponential growth of components per chip [1]. His observation that the number of transistors per integrated circuit doubles within a constant time span became a self-fulfilling prophecy. The term *Moore's law* has been coined, a so-called law that kept being valid over the last five decades.

The exponential growth of components did not only increase the performance of hardware, it also led to an increasing chip density. As a side effect, hardware became more and more susceptible to cosmic radiation and alpha-particles [2, 3, 4]. Such external influences can lead to bit flips (the change of internal value in a component) that in turn can temporarily cause a miscalculation [5]. These errors are known as *soft errors* because of a transient malfunction of the hardware, meaning that the hardware is not harmed permanently.

However, erroneous calculations can be troublesome in the best case, but they can also lead to financial loss or even life-threatening situations in the worst case. There exist counter-measures that aim at reliability to avoid such situations [6, 7, 8, 9]. One solution is to detect bit flips [10] via parity computations and fall back to a previously stored (correct) state. Another is to use *error correcting codes* (ECCs) to eliminate induced bit flips in the first place [11]. A very common example is *triple modular redundancy* (TMR), in which three copies of a module perform the same computation. The result is chosen by a majority-voter. Such systems work well but can be prohibitively expensive (e.g. more than 200% overhead of power consumption and area for TMR).

Obviously there is a need for both reliable and affordable solutions. Providing multiple redundancies of all parts fulfills the requirement of reliability, but it is costly. According to existing research, it is also unnecessary since big parts of most hardware are already protected against soft errors intrinsically [12]. Therefore, additional protection of these parts would be wasteful. Unfortunately it is not always obvious which parts are already protected.

Not every bit flip will lead to an error that is visible to the user [13], for example when the value of the flipped component is masked out (e.g., when the other input of an *and*-gate is set to false anyway) or when the internal state of the system is repaired again later by redundant components. Reporting such bit flips in order to reset the system to a previous state unnecessarily reduces the performance of the system.

On the other hand, an omission of protection logic in critical parts leaves the system vulnerable to soft errors that can compromise functionality. Finding such parts manually can be a cumbersome task as well.

1.2. Our Approaches

In this thesis, we present novel algorithms that address the previously mentioned challenge of developing optimal protection logic. In essence, we provide three different types of algorithms for analyzing sequential circuits consisting of *and*-gates and latches. Protection logic denotes a special part of these circuits dedicated to either correct errors or just detect errors and raise a special *alarm* output. The provided algorithms can be useful to find errors in the protection logic or to verify the robustness of a system.

We say that a component (latch or gate) is *protected* if there exists no execution of the circuit for which a (single) fault in that component can escape, meaning that no primary outputs of the circuit are corrupted without raising an alarm. Otherwise, if a fault in that component can escape and produce wrong output values, we call it *vulnerable*. A gratuitously raised alarm is called *false positive*.

The first provided set of algorithms is able to report components that are *vulnerable* to soft errors, meaning that the functionality of the system is compromised by flipping those. That is the case when the bit flip eventually manages to propagate to an output of the system. We said that this is not allowed unless a special *alarm* output is set to *true*. With these algorithms, it is possible to detect components that are not sufficiently protected. Note that not all of the provided algorithms are complete since it is not guaranteed that all vulnerable components are detected.

In addition to that, we provide algorithms that detect if bit flips that do not have any effect on the functionality of the system are reported unnecessarily. Such *false positives* occur when the alarm is raised even though the fault would never become visible to the user, for instance when it gets masked out. False positives can decrease the system's performance due to an unnecessary recovery to a previous state.

The third set of algorithms can be used to detect *definitely protected* components. With these, it is possible to verify the robustness of an added protection logic. This is done by proving that outputs of the system are always correct if no alarm is raised, no matter when a component is flipped. This is especially useful if it is combined with an incomplete algorithm for vulnerable components. If these algorithms detect a vulnerable component, then it is certainly not protected. But even if it is not reported by an incomplete algorithm, it could still be vulnerable. In this case, checking for protected components can provide more confidence.

The basic idea of all of our approaches in all of the three mentioned sets of algorithms is to compare a fault-free system with a modified copy in which we model faults. We are working under a single-fault assumption saying that at most one component is flipped at most once in an entire execution. Existing research shows that this is reasonably accurate [14]. For our implementations we only considered latches as components that could be hit by a fault, since a fault in a gate typically propagates to a latch, but our algorithms can be easily applied to gates as well. Categorization of components into *vulnerable* or *definitely protected* and the detection of *false positives* can be achieved with several methods: *simulation*, *formal* methods or *semi-formal* methods.

The *formal* methods are based on or similar to *bounded model checking*, meaning that the system's behavior is analyzed by considering all possible input traces up to a specified time-bound. This makes the formal methods *complete* approaches, since it es ensured that certain properties within specified bounds hold. These formal methods may start from an *initial state*, from *all states*, or from an *over-approximation* of the reachable state-space. Unfortunately, completeness often comes at the price of scalability.

Simulation, as opposed to formal methods, always starts with the initial state and uses input values from *test cases*. A test case provides concrete values for each primary input at each point in time up to a time bound, which is denoted as the length of the test case. This reduced input space can be processed a lot faster, but it is not complete anymore because it only covers a fraction of the possible executions. Hence, the number of detected components depends on the provided test cases. To make simulation complete it would be necessary to simulate all possible test cases, which is computationally infeasible for larger circuits, as our experiments show.

The *Semi-formal* methods lie in between of simulation and formal methods. They start from an initial state and symbolically encode a fault-model. They also work with *test cases*, but this time they do not necessarily have to contain only *concrete* input values. Instead, input values can also be left open, meaning that they represent both 0 and 1. A complete analysis is achieved by leaving all input values open, more open input values yield into more detected components, whereas an acceleration is achieved by using more concrete input values. This makes the semi-formal methods very flexible.

We use simulation to find vulnerable components by comparing a fault-free simulation (using concrete input values from a test case) with each possible faulty simulation. All combinations of the component to flip and the point in time to flip said component have to be simulated.

In contrast to that, test cases are not necessary for our formal model checking based approach. Here, we build a special circuit which contains the original circuit and a modified copy. Each component in the modified copy can be flipped by supplementary inputs. This special circuit has only one output, which is only true if the outputs in the modified copy can be set differently by flipping a latch without raising the alarm. A model checker can detect whether that single output of the circuit can be set to true within a specified time bound. In such a case, the circuit contains vulnerable components.

In the semi-formal algorithms, we encode the fault-free and the faulty system symbolically as a formula. It is only satisfiable if certain constraints hold. This can either be checked using SAT solvers, or with binary decision diagrams (BDDs). We present algorithms that use test cases to symbolically encode the operation of a fault-free and a faulty system in order to find *vulnerable* components or *false positives*. The input values in a test case can be concrete or left open to provide more flexibility.

For *formally* verifying that components are *definitely protected*, we also encode the operation of a system symbolically. No test case is necessary because all input values have to be open. The symbolically encoded operation does not start from the initial state alone. Instead, an over-approximation of the set of reachable states is used as a starting point. This can for example be achieved by only considering the initial state and successor states of all (potentially unreachable) states, or by unfolding for even more time steps. No matter in which of these states a component is flipped, it is required that no output can be corrupted without raising the alarm. We check if the internal state always recovers from a flip if no alarm is raised. In some cases, a simple over-approximation of reachable states might be too conservative because it still contains too many unreachable states, resulting in fewer detected definitely protected components. This can be circumvented by making the over-approximation more precise, at the cost of extra computational power.

The proposed algorithms can guide hardware developers by allowing them to verify their systems or by helping them to find bugs at an early stage of the development process, where the protection logic can still be repaired without much effort if necessary. Our reference implementation *OpenSEA* covers all of them and has been used to perform benchmarks. Both the source code and raw benchmark results are publicly available online[1].

[1]`https://github.com/p4p4/softerror-tools`

1.3. Thesis Outline

This thesis is structured as follows:

In Chapter 1, we give an overview explaining the problems related to soft errors and briefly outline how our approach can cope with them.

In Chapter 2, we describe the notation used throughout the thesis, basic terms and concepts in the field of soft errors and theoretical foundations that are necessary to understand the following chapters.

In Chapter 3, we propose techniques that can be used to find *vulnerable* components, which are not protected against soft errors and therefore can lead to an erroneous behavior of a system.

In Chapter 4, we discuss that not every bit flip may result in a faulty behavior of the system. We present algorithms that detect unnecessarily reported bit flips. Such *false positives* are undesirable because they reduce the system's performance.

In Chapter 5, we present a method to prove the resilience of components against soft errors.

In Chapter 6, we introduce extensions to the algorithms from the previous three chapters. Specifically, we show a way how it is possible to leave input values open instead of using test cases and how to use environment models to specify realistic use-cases. Additionally, we suggest how the three algorithm types can be combined.

In Chapter 7, we give insights to our reference implementation *OpenSEA*, dive into the different options and modes of the algorithms and discuss the used libraries and engines.

In Chapter 8, we discuss the performance of our algorithms by comparing benchmark results of our reference implementation.

In Chapter 9, we outline other approaches and compare them to our solution.

In Chapter 10, we sum up important results and conclude this thesis.

2. Preliminaries

In this chapter, we explain the fundamental backgrounds of our work. It talks about basic notation and data structures (and-inverter-graphs, BDDs) that are used throughout the following chapters. It also contains information about the terminology that we use over and over again in explanations. Here, we define the terms *vulnerable*, *false positive* and *definitely protected* in the context of soft error analysis. We also give examples of counter-measures against soft errors, which we call *protection logic*. Additionally, the concept of satisfiability and model checking are briefly explained. We also specify the structure of test cases, which are used as input for most of our algorithms. Finally, the concepts of concrete simulation and transition relations are explained, both of them are crucial to understand the algorithms presented in this thesis.

2.1. Basic Notation

A *literal* is either a propositional variable or its negation. It can be either true ($\top$, 1) or false ($\bot$, 0). A disjunction of literals is called *clause*. If a propositional formula only consists of a conjunction of clauses it is in *Conjunctive Normal Form* (*CNF*). A conjunction of literals is called *cube*. If a propositional formula only consists of a disjunction of cubes it is in *Disjunctive Normal Form* (*DNF*). Sets of variables are written with an overline, *e.g.* $\overline{x} = \{x_0, x_1, \ldots, x_n\}$. Vectors of concrete values (containing only 0 and 1) are written in bold,*e.g.* $\mathbf{x}$. *Sequential logic* denotes boolean circuits in which the output $\overline{o}$ depends not only on input values $\overline{i}$ but also on an internal state $\overline{x}$. In the initial state, they have pre-definded values. The next state $\overline{x}'$ is computed using $\overline{i}$ and the current state $\overline{x}$

2.2. Soft-Errors, Protection-Logic, Vulnerabilities and False-Positives

A *fault* is a flipped component (latch or gate) in a digital circuit, meaning that the truth value of the output of the affected component is inverted. Such faults are typically caused by external influences like radioactive decay and occur infrequently in just one or few components at the same time. If such a fault is visible to the user, *i.e.,* if primary outputs of the circuit change, it is called a *soft error*. *Soft*, because the transient error does not harm the hardware permanently [15].

Protection logic denotes a special part of a circuit with components dedicated to either detect or correct faults in order to prevent a soft error. We specify that this hardware-extension introduces a special output, the *alarm* output. Whenever a fault that leads to an erroneous behavior is detected, the alarm output is raised. The alarm should not be raised if the fault is masked out. The alarm signal can be used to only notify the user that the system might behave unexpectedly. It can also be used to reset the system, initiate some kind of software-recovery, or to trigger other (potentially expensive) actions.

There are several ways to add an error detection containing the special alarm-output to an existing circuit. A common solution is to compute parity sums of latches and introduce redundancy in the form of additional latches: Whenever the parity sums of the latch inputs from the previous time step and the parity sum from the latch outputs of the current time step are different (XOR), the alarm signal is raised.

Figure 2.1 shows an example of how parity-computation can be used to detect faults. An additional latch L_{e1} is added to protect the latches L_1 to L_n. If one latch (or an odd number of latches) is flipped, the alarm output is set to true. This example detects all possible single-faults, even those that may not be visible to the user.

An ideal protection-logic fulfills the following points. First, the *alarm* output should be raised when a fault happened that causes wrong output values. Second, the *alarm* output should never be raised gratuitously, meaning that the flipped component has no effect on the output values. And third, the

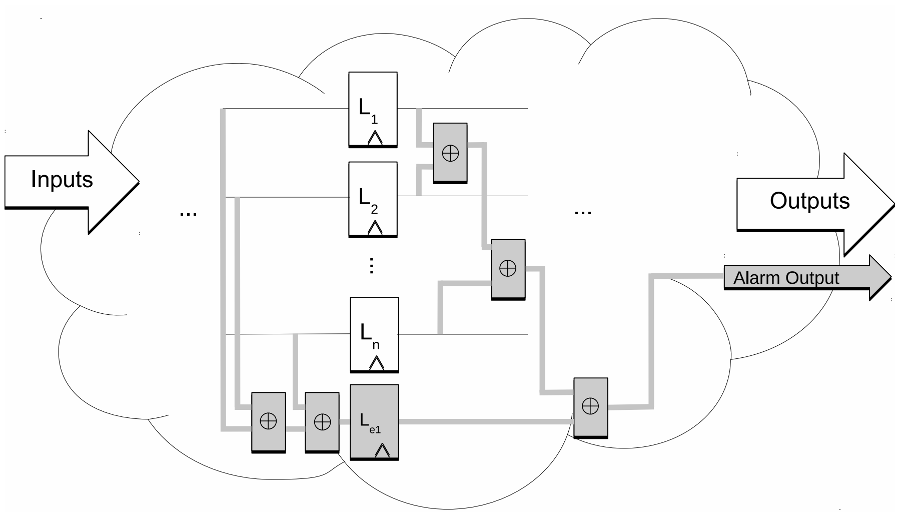


Figure 2.1.: The computation of parity sums is a very common method to detect soft errors.

protection logic should be as small as possible, since additional components cost power and area.

Since faults only happen rarely and in a small number of components, we are working under a single-fault assumption. At most one component is flipped at most once in an entire execution. Existing research shows that this is reasonably accurate [14].

We say that a component is *protected* if there exists no execution of the circuit in which any single fault in that component becomes visible to the user without raising an alarm. An alarm has to be raised at the latest when an output is corrupted. Otherwise, if a fault in that component can escape and produce wrong output values without an alarm, we call the component *vulnerable*.

A gratuitously raised alarm is called *false positive*. We do not want to raise the alarm too often because recovering to a previously stored correct state typically reduces the circuit's performance because of re-computations. An alarm is raised gratuitously if it is guaranteed that the current and all future output values are not affected by an induced fault.

2.3. Satisfiability

SAT solvers are a very powerful tool to check satisfiability for propositional formulas. A lot of the algorithms presented in this thesis utilize SAT solvers to analyze the quality of protection logic.

A SAT solver call is denoted by

$$\mathsf{sat} := \textsc{PropSat}(F(\overline{x})),$$

where $F(\overline{x})$ is a propositional formula in CNF. The variable sat is assigned to true if $F(\overline{x})$ is satisfiable, and false otherwise.

If we are also interested in a satisfying assignment, then we write

$$(\mathsf{sat}, \mathbf{x}, \mathbf{y}, \ldots) := \textsc{PropSatModel}\big(F(\overline{x}, \overline{y}, \ldots)\big)$$

As in the previous call, sat is assigned to true if the formula F over the variables $\overline{x}, \overline{y}, \ldots$ is satisfiable. In such a case, $\mathbf{x}, \mathbf{y}, \ldots$ are satisfying assignments for the variables $\overline{x}, \overline{y}, \ldots$, respectively. If F is unsatisfiable, then sat is assigned to false and the variables $\mathbf{x}, \mathbf{y}, \ldots$ are meaningless. Let $\mathbf{x}$ be a cube (a conjunction of literals) over the variables $\overline{x}$. Given that $\mathbf{x} \wedge F(\overline{x}, \overline{y})$ is unsatisfiable, we will write

$$\mathbf{x}' := \textsc{PropUnsatCore}(\mathbf{x}, F(\overline{x}, \overline{y}))$$

to denote the extraction of an unsatisfiable core $\mathbf{x}' \subseteq \mathbf{x}$ such that $\mathbf{x}' \wedge F(\overline{x}, \overline{y})$ is still unsatisfiable.

Very often several similar SAT solver calls are necessary to solve the problems discussed in this thesis. In that case, incremental SAT-solving is very suitable: Instead of having multiple individual SAT solver calls it is possible to add additional constraints after a satisfiable SAT solver call for the next SAT solver call.

2.4. Bounded Model Checking

Model checking in general exhaustively checks a system against a provided specification [16]. An unbounded model checker is *complete*, *i.e.,* it considers the entire state space of the system: A certificate of correctness is issued if the system provably satisfies a specification at any time. It suffices to find just one counter-examle to know that a system does not satisfy its specification. In such a case, the model checking algorithm can already be stopped without checking the remaining states.

Bounded model checkers are a special *incomplete* variant that is typically used to find counter-examples [17, 18]. The type of bounded model checkers we are using to find vulnerable components (Section 3.1) use a special circuit as input. Such a circuit can have multiple inputs, but only one output, namely the *bad*-signal.

This model checker can prove whether it is possible to set the *bad*-signal to true or not within a specified *time bound*. This is faster than using unbounded model checkers, however, it can still be prohibitively expensive in terms of execution time.

Bounded model checkers do not prove that the system satisfies a model completely. Instead, they only check all input combinations up to a specified number of time steps (the upper bound) in order to set the *bad*-signal to true. This can be faster than an unbounded proof and might be sufficient for certain use-cases.

2.5. Tseitin Transformation

Most SAT solvers only work with formulas in conjunctive normal form (CNF). Converting a boolean formula to CNF using DeMorgan's law and distributive law may lead to an exponential growth of the transformed formula. Since we use CNFs mainly for satisfiability checks, equisatisfiable formulas are sufficient for these purposes.

Equisatisfiable means that the transformed formula is satisfiable if and only if the original formula is satisfiable. The Tseitin transformation uses auxiliary variables to generate an equisatisfiable formula with a linear increase of the formula size.

The idea is to introduce a new variable, in this case called r, for every sub-formula. [19] Rewrite rules for $\wedge$ and $\vee$ look as follows.
$r \leftrightarrow p \wedge q$ can be rewritten to $(\neg r \vee p) \wedge (\neg r \vee q) \wedge (\neg p \vee \neg q \vee r)$.
$r \leftrightarrow p \vee q$ can be rewritten to $(\neg p \vee r) \wedge (\neg q \vee r) \wedge (\neg r \vee p \vee q)$.

2.6. And Inverter Graphs (AIGs)

And Inverter Graphs (AIGs) [20] can be used as a compact representation of combinatorial circuits. AIGs are directed acyclic graphs with primary inputs, constants, nodes that have two inputs and inverters that can be applied to edges. These nodes represent *and*-gates that can be negated at inputs and outputs. With that it is possible to encode any boolean formula and thus any combinatorial circuit.

The AIGER[1] file-format specifies a way to encode an and-inverter-graph in an ASCII- or in a binary file-representation. A C-library that can be used to access such files is also available. The library also supports D-flip-flops, which are memory elements that store 1-bit. D-flip-flops (short for Delay-flip-flops) have a *data*-input, a *clock*-input and a *data*-output. The *clock*-signal alternates from high to low and form low to high within one time-cycle. After each (either positive or negative) edge on the *clock*-signal, the *data*-input value is copied to the *data*-output, or in other words: the output-signal equals the input value of the previous time step. AIGER is very well known in the hardware community (e.g. used for the hardware-model-checking-competition, synthesis competitions, etc). Tools like ABC can read and write AIGER files or convert other common formats like Verilog to AIGER.

Our algorithms assume that the circuits to check are represented as AIGs. Our reference implementation uses the AIGER library.

[1] `http://fmv.jku.at/aiger/FORMAT.aiger`

2.7. Binary Decision Diagrams (BDDs)

There are many different ways to represent boolean functions. SAT solvers typically use CNF-Formulas. Reduced ordered BDDs (ROBDDS or short BDDs) [21, 22] are a compact canonical representation of boolean formulas. *Canonical* means that two equivalent boolean formulas result in the same BDD if the same variable ordering is used. Figure 2.2 shows a BDD with the variable ordering $x_1 < x_2 < x_3$.

BDDs are *Directed Acyclic Graphs* (DAG) that consist of a root node, nodes for variables and terminal nodes (0 and 1). The root node defines a starting point that points to the first variable-node. Each variable-node has a *then* and an *else* edge, pointing either to another variable-node or to one of the terminal nodes, which represent the truth-constants.

In order to find the truth value of a valuation, one has to start at the root node and follow the edges of the variable nodes. If a variable is set to true, the *then* edge is followed, otherwise the *else* edge is followed. Eventually, a terminal node is reached, which determines the truth value of the function.

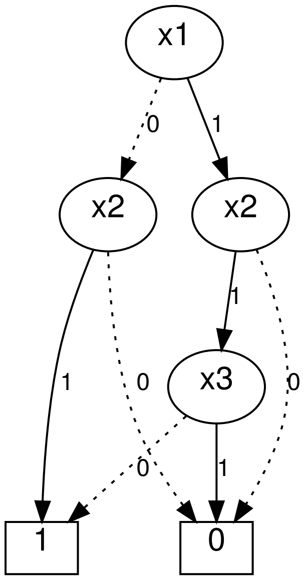


Figure 2.2.: Example of a simple BDD for the formula $\neg x_1 \wedge x_2 \wedge x_3 \vee x_2 \wedge \neg x_3$

The size of a BDD highly depends on the ordering of variables. Although the size can be exponential (regarding the number of variables), a good variable order can reduce the size significantly for a lot of functions. Most of the available libraries include several different algorithms for dynamic variable reordering. However, finding a good variable order can be a computationally hard task.

Satisfiability- and equivalence checks can be done in constant time, operations like conjunction or disjunction can be done in polynomial time.

2.8. Test Cases

Some of our presented algorithms use *test cases* which provide input values to analyze a circuit. Test cases are vectors of input vectors. An input vector defines the (usually concrete) input values of the circuit for one time step. Test cases can be represented as simple ASCII files: Each line represents one input vector. The number of input vectors, which we also call the *length* of a test case, can be arbitrary. The width of an input vectors has to match the number of inputs. The ordering of the inputs is from left to right.

This is an example test case for a two-inputs-circuit with a length of 3 time steps:

```
1 0
1 1
0 0
```

The previous test case consisted solely of concrete input values. Some algorithms also allow the verification engineer to leave parts of the test case undefined by writing a question mark instead of a concrete value. In such a way, the algorithms test all possible combinations of unspecified input-values. Here is an example:

```
1 0
? 1
? ?
```

The input values for the initial time step are fixed in that example. The next time step (*? 1*) can be both (*0 1*) or (*1 1*). Finally, the last time step can be set to all four possible 2-bit-combinations. As a result, the above test case with 3 open input values represents all possible combinations of concrete vaues, *i.e.,* it represents $2^3 = 8$ different concrete test cases instead of just one execution of the system.

2.9. Concrete Simulation

The functionality of logical circuits in AIG representation (Section 2.6) can be simulated in software. Concrete values (0 or 1) for inputs $\mathbf{i}$ (e.g. from a test case, 2.7) and the current state $\mathbf{x}$ result in concrete values of the outputs $\mathbf{o}$ (and an alarm output a) and next state $\mathbf{x}'$. The current state $\mathbf{x}$ is simply a vector that stores a concrete value for all flip-flops or latches in the circuit, the next state $\mathbf{x}'$ represents the input-values of these memory-elements. The following function call denotes a concrete simulate of a circuit for one time step:

$$(\mathbf{x}', \mathbf{o}, a) := \mathsf{sim}(\mathbf{x}, \mathbf{i})$$

To simulate a circuit using a concrete test case, it is necessary to start with an initial state (typically all latch values $\mathbf{x}$ set to 0).

With the concrete input-values (from a test case) and the concrete state-values (starting with the initial-state), the output- and next-state values can be computed by iteratively computing the results for each AND gate. The output-value of an AND gate is true if and only if both input-values are true.

Before computing outputs $\mathbf{o}$ and next state $\mathbf{x}'$ for another time step, one has to copy the values from the next-state $\mathbf{x}'$ to the current-state $\mathbf{x}$ and replace the values for the inputs $\mathbf{i}$ with the ones from the test case.

2.10. Transition Relation Unrolling

A circuit can be represented as a transition relation formula $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}')$ defining allowed state transitions. The outputs $\overline{o}$ and the next state $\overline{x}'$ are a function of the current state $\overline{x}$ and the (current) inputs $\overline{i}$. The relation-formula is only true for valid combinations of input-, output-, current-state- and next-state-values. The execution of a circuit over multiple time steps is encoded by *unrolling* the transition relation, *i.e.,* conjuncting additional instances of the transition relation that use the next state literals of the previous relation as state.

A transition relation can be generated in CNF using Tseitin transformation. Recall from 2.5 that Tseitin transformation adds auxiliary variables $\overline{t}$ for intermediate results like AND outputs. The transition relation can be computed iteratively by encoding each AND gate. Suppose the following AND gate: $lhs = rhs_0 \wedge rhs_1$. The output $lhs \in \overline{t}$ is defined by the two inputs rhs_0 and rhs_1. The two inputs can either be an input-literal of the circuit $i \in \overline{i}$, a current state literal $x \in \overline{x}$, or the output literal of another AND-gate $t \in \overline{t}$. The AND-gate can be represented by the three clauses $(\neg lhs \vee rhs_0) \wedge (\neg lhs \vee rhs_1) \wedge (\neg rhs_0 \vee \neg rhs_1 \vee lhs)$ Doing this for each AND-gate gives us an equisatisfiable transition relation $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}', \overline{t})$ which also contains auxiliary variables $\overline{t}$ introduced by intermediate AND-gates.

To unroll the transition relation for another time step, the current state $\overline{x}$ has to be replaced by the next state $\overline{x}'$. Additionally, fresh input variables have to be used. An unrolled transition relation for two time steps looks as follows: $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}', \overline{t}) \wedge T(\overline{x}', \overline{i_2}, \overline{o_2}, \overline{x}'', \overline{t_2})$. For n time steps: $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}', \overline{t}) \wedge T(\overline{x}', \overline{i_2}, \overline{o_2}, \overline{x}'', \overline{t_2}) \wedge ... \wedge T(\overline{x}^{(n)\prime}, \overline{i_n}, \overline{o_n}, \overline{x}^{(n+1)\prime}, \overline{t_n})$.

For reasons of readability we may omit Tseitin variables $\overline{t}$ from formulas whenever they are not a subject to discussion.

Most of our algorithms use these transition relations in CNF-representation together with a SAT solver, some of them use a similar idea based on BDDs instead. Since BDDs do not have to be in a CNF representation they do not contain any Tseitin variables $\overline{t}$. In the algorithms in Section 3.3 we also use modified transition relations $T_{err}(\overline{x}, \overline{i}, f, \overline{c}, \overline{o}, a, \overline{x}')$ or $T_{err}(\overline{x}, \overline{i}, f, \overline{c}, \overline{o}, a, \overline{x}')$. The first one contains an additional signal f that flips a specific predefined

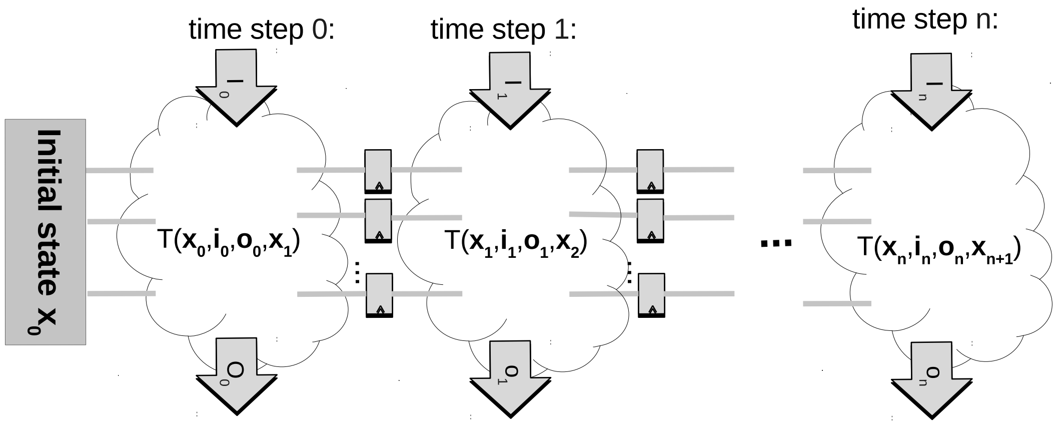


Figure 2.3.: A transition relation unrolled for *n* time steps

component (at the time step that is represented by the transition relation). The second one also has additional signals $\overline{c}$ that are used to choose *which* component should be flipped.

3. Detecting vulnerable Latches

Recall from Section 2.2 that a component is *vulnerable* if a fault in that component can compromise the functionality of the circuit without being detected.

More precisely, a latch is vulnerable if a bit flip in it can change outputs of the circuit without raising the special alarm-output. As opposed to this, a latch is *not* considered to be vulnerable if potential faults do not have an effect on primary outputs, for example when the internal state is repaired by error correcting codes. In addition to that, latches are also protected if the alarm is set to true before or at the latest when an output is corrupted.

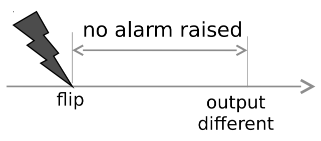


Figure 3.1.: A vulnerable latch over time

This chapter presents multiple algorithms that detect vulnerable latches.

First, we describe how model-checkers can be utilized to test if a circuit contains any vulnerable latches at all. Then, we present a way to find vulnerable latches by simulating a circuit with injected faults. After that, our symbolic algorithms based on SAT-solving or BDDs are explained. The first one (STA) symbolically encodes when to flip a certain latch in order to find a fault that becomes visible to the user. The second one (STLA) also encodes *which latch* has to be flipped when symbolically in order to produce soft-error escapes. Vulnerabilities are found if such a formula is satisfiable by setting the inputs to flip latches accordingly.

3.1. Bounded Model-Checking Approach

In this section, we explain how model-checkers (Section 2.4) can be used to test if a circuit contains vulnerable latches. This is achieved by converting a circuit with protection logic to a model-checking problem with a single *bad*-state. A model checker takes such a circuit with exactly one output (the *bad*-signal) and tries to find an input sequence that sets it to true.

For this purpose, we create two copies of the original circuit to build a sequential equivalence problem, as illustrated in Figure 3.2. Both of these copies use the same input values.

The first copy is untouched, whereas the second copy is modified in such a way that bit flips can be introduced. This is done by adding additional inputs which allow the model-checker to flip a certain latch at a certain point in time.

We ensure that a flip can be introduced only once to fulfill the single fault assumption. This is done by ignoring further flip-requests when there has already been a flip.

The output values of both circuit-copies are compared. To detect a vulnerable latch, at least one of the outputs must be different and that the alarm output of the modified copy has not been set to true before. A flipped latch is considered to be protected if the alarm output has been set to true before or when the outputs changed. The *bad*-signal can only be set to true if a latch can be flipped and outputs can be corrupted without raising the alarm.

Each input value for each time step (up to a specified time step bound) is a free input value, which can be set arbitrarily by the model-checker. Additionally, the model checker can choose when to flip which latch with the added special inputs. As a consequence, model checking is the most accurate, but also the computationally most expensive way to approach a soft error-analysis for vulnerable latches.

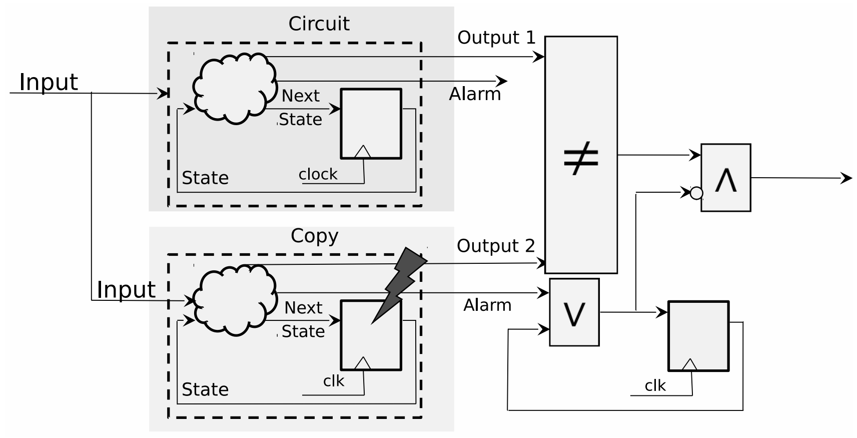


Figure 3.2.: Abstract schema of the model-checking approach.

3.2. Simulation Based Analysis

In the previous section we mentioned that model-checking is a very accurate method to search for vulnerable latches because all possible input combinations are examined. However, the huge input space may lead to a bad scalability, which is why simulation or emulation [23, 24] are widely used. In this section we present our simulation based algorithm, which analyzes the circuit using *test cases* (Section 2.8), which define the input values for the circuit. The underlying idea is that selecting a small set of test cases with a practical orientation may already suffice to find most vulnerable latches.

Simulating a boolean circuit with input-values from a test case is an easy task (Section 2.9). Our simulation-based algorithm to detect vulnerable latches compares output values from a correct simulation with output values from a fault simulation, in which a bit flip has been injected. The algorithm contains lots of nested loops because each latch of the circuit can be flipped at each time step of the test case. This results in numerous fault simulations.

A fault simulation starts at the point in time where the bit flip is injected. It starts with the same state as the fault-free simulation, except of the one latch that is flipped, and it uses the same input values as the fault-free simulation. The fault simulation has three stopping criteria. If the alarm is raised, the faulty-simulation can be stopped since a latch can only be vulnerable if no alarm is raised. The simulation can also be aborted if the output values of the faulty and the fault-free simulation are different. In this case a vulnerable latch is detected (because the first stopping criterion, a raised alarm signal, did not hold). Finally, a fault simulation can also be stopped when the state of both simulation instances is the same again. That happens when the internal error produced by the bit flip vanishes (and did not corrupt any outputs before, e.g. due to error correcting codes). Both instances have the exact same internal state and thus the same output-values from that point on until the end of the test case. Omitting these useless comparisons is just a performance optimization of the algorithm.

In the beginning (Lines 2-5) of Algorithm 1, the circuit is simulated once starting from the initial state $\mathbf{x}_1$ using the input values from the provided test case t. Here, no latch is flipped at all. The resulting state- and output-values are stored for later comparisons with fault simulations.

Then, all possible fault simulations are executed: The algorithm flips latches and searches for situations in which outputs are set differently to the correct simulation without an alarm being raised. Each latch is flipped at each point in time. It is vulnerable if the output is different in any of the future time steps without raising an alarm before (Lines 13 - 22). That is the case if the first stopping criterion (alarm raised) did not hold, but the second one (outputs different) did.

As already stated, the third stopping criterion (Line 23) is an optimization to speed up the algorithm, since it does not make any sense to compare outputs in future states if both simulations have the exact same internal state.

Algorithm 1 SIM

```
1: procedure AnalyzeSim(test case $t$)
2:     $\mathbf{x}_1$ := init                                          ▷ run correct simulation once:
3:     for $i = 1$ to $len(t)$ do
4:         $(\mathbf{x}_{i+1}, \mathbf{o}_i, a_i) := \mathsf{sim}(\mathbf{x}_i, t[i])$
5:     end for
6:     for  each latch $C$ do                                          ▷ simulate each single flip:
7:         $is\_vulnerable := false$
8:         for $i = 1$ to $len(t)$ do                                  ▷ time to flip
9:             if $is\_vulnerable$ then
10:                break
11:            end if
12:            $\mathbf{x_{i,e}} := \{\mathbf{x_{i1}}, \ldots, \neg\mathbf{x_{iC}}, \ldots, \mathbf{x_{in}}\}$              ▷ flip latch
13:            for all $j \geq i$ do                                   ▷ time steps after flip
14:                $(\mathbf{x_{j+1,e}}, \mathbf{o_e}, a_e) := \mathsf{sim}(\mathbf{x_{j,e}}, t[j])$
15:                if $a_e$ then                                       ▷ ok, detected
16:                    break
17:                end if
18:                if $\mathbf{o} \neq \mathbf{o}_e$ then                          ▷ error escaped
19:                    $vulnerable\_latches$.add($C$)
20:                    $is\_vulnerable := true$
21:                    break
22:                end if
23:                if $\mathbf{x_{j+1}} = \mathbf{x_{j+1,e}}$ then                ▷ ok, corrected
24:                    break
25:                end if
26:            end for
27:        end for
28:    end for
29: end procedure
```

3.3. Semi-Formal Approach

The two following algorithms developed by us show a way to find vulnerable latches using a SAT solver. (The algorithms can also easily be implemented using BDDs instead). This can be achieved by a technique called sequential equivalence checking, in which the transition relation is unrolled (Section 2.10) for the length of the provided test case, similar to bounded model checking. Searching for vulnerable latches can be formulated as a satisfiability problem by conjuncting additional vulnerability constraints to the unrolled transition relation. Recall that the vulnerability constraints require that outputs are different due to a bit flip without raising an alarm before.

We present two variants of the algorithm. The first one has to be executed for each latch independently, only the point in time when the single bit flip is introduced is encoded symbolically. In the second variant, the point in time *and* the location (choice of component) to flip is encoded symbolically. In Section 6.1, we will show how both algorithms can be adapted to also support free input values.

3.3.1. Point in Time Symbolic

As in all the algorithms presented so far, outputs of a faulty circuit (with a bit flip) somehow have to be compared with a fault-free circuit model. In contrast to the simulation-based algorithm, the point in time to flip a latch is encoded symbolically in this algorithm. That means it is not necessary to introduce a fault at each time step and check afterwards if the flipped latch is vulnerable. Instead, the SAT solver chooses *when* to flip a certain latch in order to produce wrong outputs without an alarm being raised. A vulnerable latch is found if the constructed formula is satisfiable. This algorithm has to be executed for each latch individually.

Algorithm 2 STA (Symbolic Time Analysis)

```
1: procedure AnalyzeSTA(test case $t$, component $C$)
2:     $\mathbf{x} := \mathsf{init}$
3:     $\mathsf{syb\_state} := \mathsf{init}$
4:     $\overline{f} := \emptyset$
5:     $F := \mathsf{true}$
6:     $T_{err}(\overline{x}, \overline{i}, f, \overline{o}, a, \overline{x}')$ is a modified transition relation where the additional input variable $f$ defines if the output of component $C$ is flipped.

7:     for $i = 1$ to $len(t)$ do
8:         $(\mathbf{x}', \mathbf{o}, a) := \mathsf{sim}(\mathbf{x}, t[i])$
9:         $(\mathbf{x}'_e, \mathbf{o}_e, a_e) := \mathsf{sim\_err}(C, \mathbf{x}, t[i])$
10:        if $\mathbf{o} \neq \mathbf{o}_e$ and $a_e = \mathsf{false}$ then
11:            return $(i, i)$
12:        end if
13:        $(\overline{x}', \overline{o}', f_i) := \mathsf{createFreshVars}(|\overline{x}|, |\overline{o}|, 1)$
14:        if ($\mathbf{o} = \mathbf{o}_e$ and $\mathbf{x}' = \mathbf{x}'_e$) or $a_e = \mathsf{true}$ then
15:            $F := F \wedge T(\mathsf{syb\_state}, t[i], \overline{o}', \mathsf{false}, \overline{x}')$
16:        else
17:            $F := F \wedge T_{err}(\mathsf{syb\_state}, t[i], f_i, \overline{o}', \mathsf{false}, \overline{x}')$
18:            $F := F \wedge \bigwedge_{f_a \in \overline{f}} (f \rightarrow \neg f_a)$
19:        end if
20:        $(\mathsf{sat}, \mathbf{f}) := \textsc{PropSatModel}\big(F \wedge (\overline{o}' \neq \mathbf{o})\big)$
21:        if sat then
22:            return $\big((j$ such that $f_j$ is true in $\mathbf{f}), i\big)$
23:        end if
24:        $\mathbf{x} := \mathbf{x}'$,    $\mathsf{syb\_state} := \overline{x}'$,    $\overline{f} := \overline{f} \cup f_i$
25:    end for
26:    return None
27: end procedure
```

The input for STA are a test case t and a component C to check in a circuit.

The algorithm consists of a concrete part, which is used to generate correct output- and state-values, and a symbolic part. The symbolic part consists of an unrolled modified transition relation, in which it is possible to flip the latch to test in one of the time steps in order to produce output values that are different to the correct simulation.

In the beginning, the concrete state **x** is set to the initial state (each latch 0). Correspondingly, syb_state, which stores the symbolic state, is set to *false* for each latch.

F stores the formula which will be used for SAT solver calls. F is satisfiable if a soft error can escape if C is flipped at some point in time. In the beginning it is set to true. Throughout the execution of the algorithm, more and more constraints get appended to it.

$T_{err}(\overline{x}, \overline{i}, f, \overline{o}, a, \overline{x}')$ is a modified transition relation with an additional input f, which is used to flip the output of the component C. At each time step, an instance of this transition relation is appended.

The loop of the algorithm (Lines 7 - 25) is executed for at most the number of time steps of the test case t.

First, the correct next state- and output-values are computed by a concrete simulation using the current state and the current input vector from the test case (Line 8). Similarly, a concrete simulation with the same input vector is performed, with the only difference that the output of the component C is flipped. In addition to the faulty next state and outputs, the alarm signal a_e of the fault simulation is stored.

If flipping the latch already immediately produces different outputs without raising the alarm (Line 10), then the C is vulnerable. In such a case the execution can be terminated without having to calling the SAT solver. Otherwise the execution continues with the symbolic part of the algorithm.

In Line 13, fresh variables for the next state $\overline{x}'$ and output $\overline{o}'$ are generated. Additionally, a variable f_i, which can be used to flip the output of C, is created. These new variables are used for the next instance of the transition relation for the current time step.

If both concrete simulations resulted in the same output- and next-state values, then a flip in that state has no effect to the system at all. Similarly, the component C is also protected in that time step if the flip immediately led to a raised alarm signal in the same state. In both of these two cases (Line 14) the transition relation for the current time step $T(\text{syb\_state}, t[i], \overline{o}', \text{false}, \overline{x}')$

does not contain an f signal. Otherwise, if the system does not recover until the next state or if no alarm is raised in the same state, a modified transition relation $F \wedge T_{err}(\mathsf{syb\_state}, t[i], f_i, \overline{o}', \mathsf{false}, \overline{x}')$ containing f_i is appended to F. The variable f_i leads to a bit flip in C if set to true. Due to our single fault assumption, a constraint defining that at most one f_i signal can be set to true (Line 18) (*i.e.,* the latch can be flipped at most once) is added.

Note that the alarm a is required to be false in the transition relations, since only such situations are of interest when testing if the current latch is vulnerable.

The SAT solver call (Line 20) contains the unrolled transition relation, which uses inputs from the test case and requires the alarm signal to be false all the time, and constraints that the outputs have to be different at the current point in time. This can only be satisfiable by setting one of the variables $f \in \mathbf{f}$ to true.

The latch C is vulnerable if the formula is satisfiable. It is possible to produce wrong output values at the current point in time without a raised alarm with the provided test case if C is flipped at time step j, where f_j is the only true variable in the assignment.

If the sat solver could not yet find a satisfying assignment (a point in time to flip C which makes the latch vulnerable now), then the algorithm tries to find one in a future time step in another iteration of the loop. But first, a switch to the next state is performed (Line 24)

3.3.2. Point in Time And Location Symbolic

In the previous algorithm, the point in time *when* to flip a latch was already symbolic, meaning that the SAT solver chooses when a latch has to be flipped in order to find a vulnerability.

The algorithm presented in this section builds up on the previous algorithm, since it also symbolically encodes the point in time to flip. In contrast to the previous algorithm, this one does not have to be executed for each latch individually. Instead, the SAT solver also choose *which* latch should be flipped, not just *when* a fixed latch has to be flipped.

Instead of running STA for each latch individually, one execution of STLA suffices to check all latches.

Algorithm 3 STLA (Symbolic Time Location Analysis)

```
1: procedure AnalyzeSTLA(test case $t$)
2:     $\mathbf{x} := \mathsf{init}$
3:     $\mathsf{syb\_state} := \mathsf{init}$
4:     $\overline{f} := \emptyset, \mathsf{vulnerable} := \emptyset$
5:     $F := \big((\sum c_k) = 1\big)$
6:     $T_{err}(\overline{x}, \overline{i}, f, \overline{c}, \overline{o}, a, \overline{x}')$ is a modified transition relation where the output
7:         of latch $C_k$ is flipped if the variables $c_k$ and $f_j$ are both true.
8:     for $i = 1$ to $len(t)$ do
9:         $(\mathbf{x}', \mathbf{o}, a) := \mathsf{sim}(\mathbf{x}, t[i])$
10:        $(\overline{x}', \overline{o}', f_i) := \mathsf{createFreshVars}(|\overline{x}|, |\overline{o}|, 1)$
11:        $F := F \wedge T_{err}(\mathsf{syb\_state}, t[i], f_i, \overline{o}', \overline{c}, \mathsf{false}, \overline{x}')$
12:        $F := F \wedge \bigwedge_{f_a \in \overline{f}} (f \rightarrow \neg f_a)$
13:        $(\mathsf{sat}, \mathbf{f}, \mathbf{c}) := \textsc{PropSatModel}\big(F \wedge (\overline{o}' \neq \mathbf{o})\big)$
14:        while sat do
15:            $\mathsf{vulnerable} := \mathsf{vulnerable} \cup \big((j$ such that $f_j$ is true in $\mathbf{f}),$
16:                                    $(C_k$ such that $c_k$ is true in $\mathbf{c}), i\big)$
17:            $F := F \wedge \neg c_k$
18:            $(\mathsf{sat}, \mathbf{f}, \mathbf{c}) := \textsc{PropSatModel}\big(F \wedge (\overline{o}' \neq \mathbf{o})\big)$
19:        end while
20:        $\mathbf{x} := \mathbf{x}'$,   $\mathsf{syb\_state} := \overline{x}'$,   $\overline{f} := \overline{f} \cup f_i$
21:    end for
22:    return vulnerable
23: end procedure
```

The previous algorithm from Section 3.3.1 has a lot of similarities to this one. That is why the following explanation mainly focuses on the differences.

This algorithm uses a slightly different version of a modified transition $T_{err}(\overline{x}, \overline{i}, f, \overline{c}, \overline{o}, a, \overline{x}')$. In addition to the f signal, which means that a flip should be done *now*, the relation also talks about $\overline{c}$ signals. A signal $c_k \in \overline{c}$ set to true indicates that the component C_k should be flipped (if the f signal is set to true as well).

Due to our single fault assumption, the first constraint requires that at most one latch can be flipped (Line 5).

As in the previous algorithm, the results of the concrete simulation using input values from the test case are stored. Likewise, it is required that there is at most one point in time when a flip can be introduced.

In contrast to STA, no fault simulation is executed.

Again, a modified transition relation for the current time step is appended, in which the input values from the test case are used and the alarm signal is required to be false. Each latch C_i can only be flipped at the current time step j by setting both c_i and f_j to true.

With help from the correct output values from the concrete simulation (Line 9) we create a clause saying at least one of the output values must be different in the current time step and call the SAT solver (Line 13/18).

A vulnerable latch is found if the query is satisfiable. The information which latch has to to be flipped when in order to produce wrong outputs can be found in the satisfying assignment. The one and only c_k signal set to true corresponds to the latch C_k. f_j set to true in the assignment $\mathbf{f}$ indicates that the latch hast to be flipped at time step j. The error has an effect on the output at the current time step i because the current query is satisfiable.

Since multiple latches might produce wrong output values in the current step when being flipped in this or in a previous step, the SAT call is performed in a loop until all latches that can corrupt the current output are found. For each detected vulnerable latch C_i, the negated control signal $\neg c_i$ is added as a constraint in order to find other vulnerable latches. The algorithm proceeds to the next time step as soon as the formula to find vulnerable latches in the current time step turns unsatisfiable.

4. Detecting False Positives

The algorithms presented in the preceding chapter searched for *vulnerable* latches, i.e., undetected faults that affect the circuit's output values. Vulnerabilities are *false negatives*, because the alarm should have been raised, but it was not. As opposed to this, gratuitously raised alarms, which we call *false positives*, are also undesirable.

They decrease a circuit's performance due to an unnecessary reset to a previously stored preceding state. That is why we developed two semi-formal algorithms to find false positives.

Flipping a latch leads to a false positive if the alarm is set to true unnecessarily, *i.e.,* if it is guaranteed that the current and all future output values are correct. Looking at all potentially infinitely many output values does not work for verification, which is why we propose an approximate approach that can find some, but not necessarily all definite false positives. Instead of checking the infinitely many future output values, we search for situations where a flip does not affect any output values until the internal state recovers within a finite duration again (*e.g.,* via ECCs), as can be seen in Figure 4.1.

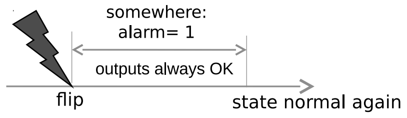


Figure 4.1.: A false positive over time

This definition of false positives can in principle also be checked via simulation, bounded model checking and with our semi-formal approach, similar to searching for vulnerable latches (Chapter 3). In this chapter, we only present a semi-formal approach by describing two algorithms which are closely related to the algorithms for vulnerable latches. In the first one in Section 4.1, the point in time to flip a latch is symbolic, while for the second one in Section 4.2 both the component to flip *and* the point in time are symbolic.

Both of the algorithms per default perform a *quantitative*-analysis, meaning that multiple situations where flipping a certain-latch results in a false positive are reported, not just one. The reason is that false positives *only* reduce the performance of a system and are therefore tolerable if they happen only rarely, whereas soft errors caused by *vulnerable* latches are system-critical and therefore should never happen.

4.1. False Positives - Point in Time Symbolic

This algorithm is a version of STA (Section 3.3.1) which has been adapted to find false positives instead of vulnerabilities. As in STA, The point in time when a latch can be flipped is encoded symbolically.

The adapted algorithm FP 1 can be found on page 33.

FP 1 computes a set Superfluous that stores 5-element quintuples. The quintuple $(C, t, j, k, i) \in$ Superfluous means that the latch C can be flipped at time step j while executing test case t. This will result in an alarm at time step k. At time step i, the error is gone again, meaning that the internal state is identical as it would have been without a flip. The outputs were correct at all times.

The algorithm consists of a simulation-based part, and a symbolic part. $\overline{a}$ is a vector that stores the variables a_i representing the value of the alarm output of the transition relation at the time step i.

In the simulation part, a fault-free and a fault simulation is performed. First, the fault-free simulation stores values for the next state, the outputs and the special alarm output ($\mathbf{x}'$, $\mathbf{o}$ and a respectively). The whole procedure is aborted if the alarm signal a in the fault-free simulation is set to true (Line 7

- 10). In such a case, the parity computation is wrong and should be fixed first before re-running the algorithm.

After that, if the algorithm was not aborted, a fault simulation with C flipped is computed. A false positive is detected using simulation only if the alarm a_e of the fault simulation is true but the output values do not differ and the error vanishes in the next state (Line 14 - 19). In that case it is not necessary to call a SAT solver in that time step. Instead, we only unroll the transition relation for the current time step (Line 16). It is required that the output values have to be the same as in the fault-free simulation (the symbolic outputs are set to $\mathbf{o}$ from the correct simulation). With that, future SAT solver calls can find false positives caused by flips in previous time steps. However, adding an f-variable in the transition relation to flip the latch in the current time step would not make any sense since a flip in that state always recovers before the next state.

If no false positive caused by a flip in the current time step is found, the algorithm continues with the symbolic part in which a false positive in the current step caused by a flip in one of the previous steps is searched (Line 20 - 37).

It is impossible to find a false positive in the current or any following step by flipping C now if it immediately produces wrong output values. In that case, adding an f-signal in the transition relation for the current step does not make any sense either (Line 20 - 21). Flipping in this particular state can impossibly result in a false positive in any future state since outputs are already wrong at that time.

In all other situations, the unrolled transition relation is extended by an f signal to flip C in the current time step (Line 24 - 27).

Finally, the SAT solver is called to search false positives at the current point in time. F contains the unrolled transition relation where input values from the test case are used. The transition relation for one particular point in time can be flipped by setting the corresponding f variable to true. Recall that we required that the output-values are correct in each step. For the current SAT call, we also require that the error is already gone in the next state $(\overline{x}' = \mathbf{x}')$ and that the alarm was raised before at least once.

The query is satisfiable if flipping C in one of the previous time steps leads to a false positive. By parsing the satisfying assignment $(\mathbf{f}, \mathbf{a})$ it is possible to find out when C has to be flipped (via $\mathbf{f}$) in order to raise an alarm at a specific point in time (via $\mathbf{a}$) such that no output is changed and the internal state recovers from the flip in the next step $(i + 1)$.

The algorithm can find multiple points in time that produce a false positive by flipping latch C. For that, the blocking clause $\neg f_j$ is added to prevent reporting the same point in time again. If one is only interested if the latch is susceptible to false positives at all (qualitative analysis), the algorithm could also already be stopped at the first time a query is satisfiable.

Algorithm 4 FP(S) 1

```
1: procedure AnalyzeFP1(test case $t$, component $C$)
2:     Superfluous $= \emptyset$,   $\overline{f} := \emptyset$,   $\overline{a} := \emptyset$
3:     $\mathbf{x} :=$ init,   syb_state $:=$ init,   $F :=$ true
4:     $T_{err}(\overline{x}, \overline{i}, f, \overline{o}, a, \overline{x}')$ is a modified transition relation where the output
               of latch $C$ is flipped if the variable $f$ is set to true.
5:     for $i = 1$ to $len(t)$ do
6:         $(\mathbf{x}', \mathbf{o}, a) :=$ sim$(\mathbf{x}, t[i])$
7:         if $a =$ true then
8:             print("Alarm raised without error.")
9:             return
10:        end if
11:        $(\mathbf{x}'_e, \mathbf{o}_e, a_e) :=$ sim_err$(C, \mathbf{x}, t[i])$
12:        $(\overline{x}', \overline{o}', a_i) :=$ createFreshVars$(|\overline{x}|, |\overline{o}|, 1)$
13:        $\overline{a} := \overline{a} \cup a_i$
14:        if $\mathbf{o} = \mathbf{o}_e$ and $\mathbf{x}' = \mathbf{x}'_e$ and $a_e =$ true then
15:            Superfluous $=$ Superfluous $\cup (C, t, i, i, i+1)$
16:            $F := F \wedge T($syb_state$, t[i], \mathbf{o}, a_i, \overline{x}')$
17:            $\mathbf{x} := \mathbf{x}'$,   syb_state $:= \overline{x}'$
18:            continue
19:        end if
20:        if $(\mathbf{o} \neq \mathbf{o}_e)$ then
21:            $F := F \wedge T($syb_state$, t[i], \mathbf{o}, a_i, \overline{x}')$
22:        else
23:            $(f_i) :=$ createFreshVars$(1)$
24:            $F := F \wedge T_{err}($syb_state$, t[i], f_i, \mathbf{o}, a_i, \overline{x}')$
25:            $F := F \wedge \bigwedge_{f_a \in \overline{f}} (f \rightarrow \neg f_a)$
26:            $\overline{f} := \overline{f} \cup f_i$
27:        end if
28:        $($sat$, \mathbf{f}, \mathbf{a}) :=$ PropSatModel$\big(F \wedge (\overline{x}' = \mathbf{x}') \wedge \bigvee_{k=0}^{i} a_k\big)$
29:        while sat do
30:            $j =$ an index such that $f_j =$ true in $\mathbf{f}$
31:            $k =$ the lowest index such that $a_k =$ true in $\mathbf{a}$
32:            Superfluous $=$ Superfluous $\cup (C, t, j, k, i+1)$
33:            $F := F \wedge \neg f_j$
34:            $($sat$, \mathbf{f}, \mathbf{a}) :=$ PropSatModel$\big(F \wedge (\overline{x}' = \mathbf{x}') \wedge \bigvee_{k=0}^{i} a_k\big)$
35:        end while
36:        $\mathbf{x} := \mathbf{x}'$,   syb_state $:= \overline{x}'$
37:    end for
38: end procedure
```

4.2. False Positives - Point in Time And Location Symbolic

This algorithm is a version of STLA (Section 3.3.2) which has been adapted to find false positives instead of vulnerabilities. Both the point in time when a latch is flipped and the latch itself are encoded symbolically.

FP 2 is not just based on STA, it obviously has a lot of similarities with FP 1 from the previous section. It can be found on page 35.

The main difference is that the algorithm shown in this section searches for vulnerabilities in *all* latches simultaneously, whereas the previous had to be executed for each one individually. To achieve that, the transition relation which is generated at each time step of the test case has additional inputs $\overline{c}$. A latch C_j is only flipped if both $c_j \in \overline{c}$ and $f_i \in \overline{f}$ are set to true.

This algorithm is more concise compared to the previous one. Since there is no fault simulation in this version of the algorithm, a lot of the optimizations from FP 1 are not applicable and therefore omitted. In exchange, it is not necessary to loop over all latches.

The satisfying assignment is used to find out which latch (via **c**) has to be flipped when (via **f**) in order to raise an alarm at a specific point in time (via **a**) so that the outputs have always been correct and the circuit will have recovered from the flip in the next step $(i + 1)$.

The algorithm can find multiple traces to produce a false positive by flipping a latch C_i. For that, only the constraint $\neg(f_j \wedge c_i)$ has to be added to prevent the SAT solver from reporting the same trace again. For a qualitative analysis, it is only necessary to add the constraint $\neg c_i$ instead to prevent latch Ci from being reported again.

After adding a Superfluous trace, the SAT solver is called again to find additional traces in the current time step (inner loop, Line 30 - 37), or in a future time step (outer loop).

Algorithm 5 FP(S) 2

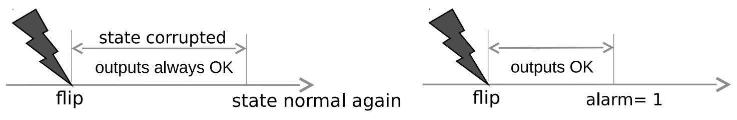
```
1: procedure AnalyzeFP2(test case $t$)
2:     Superfluous = $\emptyset$,    $\overline{f} := \emptyset$,    $\overline{a} := \emptyset$
3:     $\mathbf{x} :=$ init,    syb_state := init,    $F :=$ true
4:     $T_{err}(\overline{x}, \overline{i}, f, \overline{c}, \overline{o}, a, \overline{x}')$ is a modified transition relation where the output
           of latch $C_i$ is flipped if the variables $c_i$ and $f$ are both true.
5:     for $i = 1$ to $len(t)$ do
6:         $(\mathbf{x}', \mathbf{o}, a) := \mathsf{sim}(\mathbf{x}, t[i])$
7:         if $a =$ true then
8:             print("Alarm raised without error.")
9:             return
10:        end if
11:        $(\overline{x}', \overline{o}', a_i, f_i) := \mathsf{createFreshVars}(|\overline{x}|, |\overline{o}|, 2)$
12:        $\overline{a} := \overline{a} \cup a_i$
13:        $F := F \wedge T_{err}(\mathsf{syb\_state}, t[i], f_i, \mathbf{o}, a_i, \overline{x}')$
14:        $F := F \wedge \bigwedge_{f_a \in \overline{f}} (f \rightarrow \neg f_a)$
15:        $(\mathsf{sat}, \mathbf{f}, \mathbf{c}, \mathbf{a}) := \textsc{PropSatModel}\big(F \wedge (\overline{x}' = \mathbf{x}) \wedge \bigvee_{k=0}^{i} a_k\big)$
16:        while sat do
17:            return $\big((j$ such that $f_j$ is true in $\mathbf{f}), (C_k$ such that $c_k$ is true in $\mathbf{c}), i\big)$
18:            $F := F \wedge \neg(f_j \wedge c_i)$
19:            $(\mathsf{sat}, \mathbf{f}, \mathbf{c}, \mathbf{a}) := \textsc{PropSatModel}\big(F \wedge (\overline{x}' = \mathbf{x}) \wedge \bigvee_{k=0}^{i} a_k\big)$
20:        end while
21:        $\mathbf{x} := \mathbf{x}'$,    syb_state $:= \overline{x}'$,    $\overline{f} := \overline{f} \cup f_i$
22:        $F := F \wedge \Big((\overline{x}' = \mathbf{x}') \rightarrow \bigwedge_{f_a \in \overline{f}} \neg f_a\Big)$
23:    end for
24:    return None
25: end procedure
```

5. Detecting Definitely Protected Latches

Searching for errors like vulnerabilities in a protection logic is one way to analyze the quality of protection logic. Another approach is to search for definitely protected latches, meaning that it is ensured that an alarm is raised when a soft error happens. Recall that a component is protected if a flip in it does never corrupt primary output values without being detected.

Since it is impossible to compare the potentially infinitely many output values of a flip, we are instead searching for situations where the internal state recovers from a bit flip. A component is definitely protected if it can be proven that each single bit flip is either detected or if the state is repaired, *i.e.,* it is the same as without a flip, within finite time, as depicted in Figure 5.1.

Contrary, if the state is different, it is not possible to classify the component. It might be that the value of a primary output is wrong in some future time step, or they might as well be correct all the time.



(a) Error vanishes (b) Error gets detected

Figure 5.1.: A definitely protected latch must always fulfill one of these two properties

A latch $l_i \in \bar{l}$ is 1-step protected if

$$T(\overline{x},\overline{i},\overline{o},a,\overline{x}') \land T\big((x_1,\ldots,\neg x_i,\ldots,x_n),\overline{i},\overline{o}_e,a_e,\overline{x}'_e\big) \land \neg a_e \land (\overline{o} \neq \overline{o}_e \lor \overline{x}' \neq \overline{x}'_e) \tag{5.1}$$

is unsatisfiable.

Such a check means: no matter in which state the circuit is and no matter which inputs are used, if latch l_i flips its value, the error is either already gone in the next time step and the output values are correct, or the alarm is raised immediately when the fault happens.

However, some error-detection or error-correction methods might take more than just one clock cycle to raise an alarm or to repair the internal state. Hence, searching for 1-step protected components might not be sufficient to prove that a component is protected. For this purpose we also present methods to check for k-step protection.

Such checks may be too strict because it is possible to start from any state, possibly states that are unreachable from the initial state where parity bits are wrong. This issue can be resolved by doing a more precise over-approximation of the state spaces or by computing the state space with unbounded methods like PDR [25].

In the following sections, we propose one algorithm that checks each latch individually (5.1) and one that checks all at the same time (5.3) for 1-step protection. Additionally, we present algorithms to chek for k-step protection, again one variant to verify latches individually (5.2) and one to verify them simultaneously (5.4).

5.1. Testing latches individually for 1-step protection

The following algorithm checks one latch after the other if it is 1-step protected. A latch is one-step protected if a flip in any state will either raise the alarm or affect neither the next output nor the next state. In other words: if the alarm is not raised, the flip must be gone after one time step.

Algorithm 6 DP 1

```
1: ProtectedLatches = $\emptyset$
2: procedure DefinitelyProtected1
3:     $T := T(\overline{x}^*, \overline{i}^*, \overline{o}^*, a^*, \overline{x}) \wedge \neg a^*) \vee \overline{x} = \mathsf{init}$
4:     $T := T \wedge T(\overline{x}, \overline{i}, \overline{o}, a, \overline{x}') \wedge \neg a$
5:     for $l \in$ Latches do
6:         $T_l := T \wedge T_{err}((x_1, ..., \neg x_l, ..., x_n), \overline{i}, \overline{o_e}, a_e, \overline{x_e}') \wedge \neg a_e$
7:         $T_l := T_l \wedge (\overline{o} \neq \overline{o_e} \vee \overline{x}' \neq \overline{x_e}')$
8:         sat := PropSat$(T_l)$
9:         if sat = false then
10:            ProtectedLatches := ProtectedLatches $\cup \{ l \}$
11:        end if
12:    end for
13: end procedure
```

As stated earlier, checking equation 5.1 might be too conservative. Because of that, an additional step is added in Line 3. With that we assert that the state we start from is either the initial state or a successor form some previous state $\overline{x}^*$. It is also required that there has not been raised an alarm in that predecessor-state. This is still a conservative over-approximation of the state-space, however, trivially illegal states that could violate the single-fault assumption (e.g. wrong values in parity latches) can already be excluded.

For some benchmarks, the over-approximation in Line 3 might still not be accurate enough. A more precise (but computationally more expensive) over-approximation would look as follows:

$$\left(T(\overline{x}_0, \overline{i}_0, \overline{o}_0, \mathsf{false}, \overline{x}_1) \wedge \bigwedge_{i=1}^{j} T(\overline{x}_i, \overline{i}_i, \overline{o}_i, \mathsf{false}, \overline{x}_{i+1}) \right) \vee \overline{x}_{j+1} = reach(\mathsf{init}, j) \tag{5.2}$$

where $\overline{x}_{j+1} = reach(\mathsf{init}, j)$ denotes all states reachable from the initial state *within* j time steps and the term in parentheses denotes all states reachable from *any* state after *exactly* j steps. The approximation can be made arbitrarily precise by setting j big enough.

Line 4 adds a fault-free transition relation $T(\overline{x}, \overline{i}, \overline{o}, a, \overline{x}')$ and restricts it to situations where the alarm is set to false to rule out situations where an alarm is raised without a flip. If the j-step over-approximation (5.2) is used it is necessary to work with that state-space instead.

Each latch is tested individually if flipping it might lead to an error of the system (the for-loop).

Similarly to the fault-free relation in Line 4, a faulty transition relation is conjuncted in line 6. Here, the latch under verification is negated to simulate a bit flip. All other latches except the flipped one and all inputs are the same one as in the relation without a flip. The modified transition relation might result in different output- or next state values compared to the fault-free relation. Again, we are only interested in situations where no alarm is raised because only these situations can be susceptible.

The clauses in Line 7 limit the search space to situations where the output or next state is different because of the introduced flip.

The tested latch is definitely protected if the query is unsatisfiable. Without raising an alarm it is not possible to alter an output or the next state.

5.2. Testing multiple latches simultaneously for 1-step protection

The previous algorithm tested one latch after the other for 1-step protection. This was achieved by computing a transition relation in which that latch has been flipped and by calling the SAT solver afterwards.

The idea of this algorithm is to only create one modified faulty transition relation T_{err} in which the SAT solver can choose which of all latches should be flipped (Line 5). With that, latch l_i can be flipped by setting one of the additional inputs $c_i \in \overline{c}$ to true.

A SAT-query that is immediately unsatisfiable indicates that all latches are definitely protected. If however the query is satisfiable, a counter example is found. The signal c_i set to true in the satisfying assignment **c** means that flipping latch l_i might modify the next-state or an output without raising an alarm. Latch l_i is removed from the set of definitely protected latches and the control signal c_i is set to false. This is done until the formula turns unsatisfiable. All remaining latches are definitely protected.

Algorithm 7 DP 2

```
1: ProtectedLatches = Latches
2: procedure DefinitelyProtected2
3:     $T := T(\overline{x}^*, \overline{i}^*, \overline{o}^*, a^*, \overline{x}) \wedge \neg a^*) \vee \overline{x} = \text{init}$

4:     $T := T \wedge T(\overline{x}, \overline{i}, \overline{o}, a, \overline{x}') \wedge \neg a$
5:     $T := T \wedge T_{err}(\overline{x}, \overline{i}, \overline{c}, \overline{o_e}, a_e, \overline{x_e}') \wedge \neg a_e$

6:     $T := T \wedge (\overline{o} \neq \overline{o_e} \vee \overline{x}' \neq \overline{x_e}')$
7:     while $(\text{sat}, \mathbf{c}) := \text{PropSatModel}(T)$ do
8:         $T := T \wedge \neg c_i$, where $c_i$ is the only true variable in $\mathbf{c}$
9:         ProtectedLatches := ProtectedLatches $\setminus \{ l_i \}$
10:    end while
11: end procedure
```

5.3. Testing latches individually for k-step protection

Not every system recovers from a bit flip within one time step. The two algorithms presented so far do not detect a latch as definitely protected if the system needs more than one time step to recover, even if the output is correct all the time.

The algorithms in this and the following section can test if the system either detects a soft error or if it recovers within at most k time steps without affecting any of the outputs.

Algorithm 8 DP 3

```
1: ProtectedLatches = $\emptyset$
2: procedure DefinitelyProtected3
3:     $T := T(\overline{x}^*, \overline{i}^*, \overline{o}^*, a^*, \overline{x}_1) \wedge \neg a^*) \vee \overline{x}_1 = \text{init}$
4:     $T := T \wedge T(\overline{x}_1, \overline{i}_1, \overline{o}_1, a_1, \overline{x}'_1) \wedge ... \wedge T(\overline{x}'_{k-1}, \overline{i}_k, \overline{o}_k, a_k, \overline{x}'_k)$
5:     $T := T \wedge \neg a_1 \wedge \neg a_2 \wedge ... \wedge \neg a_k$
6:     for $l \in$ Latches do
7:         $T_l := T \wedge T_{err}((x_{1_1}, ..., \neg x_{1_l}, ..., x_{1_n}), \overline{i}_1, \overline{o}_{e1}, a_{e1}, \overline{x}'_{e1})$
8:         $T_l := T_l \wedge T(\overline{x}'_{e1}, \overline{i}_2, \overline{o}_{e2}, a_{e2}, \overline{x}'_{e2}) \wedge ... \wedge T(\overline{x}'_{k-1_e}, \overline{i}_k, \overline{o}_{ek}, a_{ek}, \overline{x}'_{ek})$
9:         $T_l := T_l \wedge \Big((\neg a_{e1} \wedge \overline{o}_1 \neq \overline{o}_{e1})$
               $\vee(\neg a_e 1 \wedge \neg a_{e2} \wedge \overline{o}_2 \neq \overline{o}_{e2})$
               $\vee \ldots$
               $\vee\big(\neg a_e \wedge \neg a_{e2} \wedge \ldots \wedge \neg a_{ek} \wedge (\overline{o}_k \neq \overline{o}_{ek} \vee \overline{x}'_k \neq \overline{x}'_{ek})\big)\Big)$
10:        sat := PropSat$(T_l)$
11:        if sat = false then
12:            ProtectedLatches := ProtectedLatches $\cup \{ l \}$
13:        end if
14:    end for
15: end procedure
```

An over-approximation of the reachable state space is performed the same way as it is done in the two previous algorithms.

In Line 4, the transition relation is unrolled for k time steps, starting with the states from the over-approximation. Only state transitions where the alarm signal is not raised are of interest (Line 5).

The same has to be done with the latch under verification flipped (Line 7). The flipped transition relation in turn also gets unrolled for k time steps (Line 8).

The condition to find counter-examples for definitely protected latches in Line 9 is a bit more complicated. It reads as follows: A latch is not definitely protected if it produces a different output value after a flip without raising the alarm before. It might also be vulnerable if the next state after k steps is not the same again, even if the outputs are always correct. In such a case one can not be sure if a latch is definitely protected because it might still result in wrong output values without raising the alarm after more than k time steps.

If that query however is unsatisfiable, the latch is definitely protected; The system always repairs its state within at most k time steps. If the flip leads to a wrong output value, it is ensured that the alarm is raised in that or in an earlier time step.

If the algorithm finds a lot of counter examples where latches might not be definitely protected because the state does not recover within k time steps, then increasing k might help finding more definitely protected latches.

5.4. Testing multiple latches simultaneously for k-step protection

The algorithm presented in this section adapts the previous algorithm in such a way that multiple latches can be tested simultaneously for k-step protection. This is achieved by the same way as the 1-step protected algorithm DefinitelyProtected2 has been adapted from DefinitelyProtected1 to check multiple latches simultaneously.

Again, the underlying idea is to use only one modified transition relation T_{err} where each of the latches $l_i \in L$ can be flipped by a corresponding signal $c_i \in \overline{c}$.

The difference for k-step protection is that the fault-free and the faulty transition relation have to be unrolled for k instead of just 1 time step. No output is allowed to change without raising an alarm and after k steps without any alarm. Additionally, the state has to be the same as without any flip after k steps.

This algorithm starts by assuming that all latches are k-step protected and refines the set of definitely protected latches by using counter-examples from satisfying assignment until the formula eventually turns unsatisfiable.

Algorithm 9 DP 4

```
1: ProtectedLatches = Latches
2: procedure DefinitelyProtected4
3:     $T := T(\overline{x}^*, \overline{i}^*, \overline{o}^*, a^*, \overline{x}_1) \wedge \neg a^*) \vee \overline{x}_1 = \text{init}$

4:     $T := T \wedge T(\overline{x}_1, \overline{i}_1, \overline{o}_1, a_1, \overline{x}'_1) \wedge ... \wedge T(\overline{x}'_{k-1}, \overline{i}_k, \overline{o}_k, a_k, \overline{x}'_k)$
5:     $T := T \wedge \neg a_1 \wedge \neg a_2 \wedge ... \wedge \neg a_k$

6:     $T := T \wedge T_{err}(\overline{x}_1, \overline{i}_1, \overline{c}, \overline{o}_{e1}, a_{e1}, \overline{x}'_{e1})$
7:     $T := T \wedge T(\overline{x}'_{e1}, \overline{i}_2, \overline{o}_{e2}, a_{e2}, \overline{x}'_{e2}) \wedge ... \wedge T(\overline{x}'_{k-1_e}, \overline{i}_k, \overline{o}_{ek}, a_{ek}, \overline{x}'_{ek})$

8:     $T := T \wedge \Big((\neg a_{e1} \wedge \overline{o}_1 \neq \overline{o}_{e1})$
               $\vee(\neg a_{e1} \wedge \neg a_{e2} \wedge \overline{o}_2 \neq \overline{o}_{e2})$
               $\vee \ldots$
               $\vee(\neg a_e \wedge \neg a_{e2} \wedge \ldots \wedge \neg a_{ek} \wedge (\overline{o}_k \neq \overline{o}_{ek} \vee \overline{x}'_k \neq \overline{x}'_{ek}))\Big)$
9:     while $(\text{sat}, \mathbf{c}) := \text{PropSatModel}(T)$ do
10:        $T := T \wedge \neg c_i$, where $c_i$ is the only true variable in $\mathbf{c}$
11:        ProtectedLatches := ProtectedLatches \ { $l_i$ }
12:    end while
13: end procedure
```

6. Algorithm Extensions

This chapter presents extensions to the algorithms presented in previous chapters. First, we show how some or all input values can be left open in test case based algorithms. Afterwards, we explain how to use environment models to define when outputs are relevant and to restrict the number of allowed input combinations for open inputs. Finally, we show how the findings of an algorithm for definitely protected latches can be used to speed up the search for vulnerable latches (and vice versa).

6.1. Free Inputs Modes

The algorithms for vulnerable latches (Chapter 3) and false positives (Chapter 4) used test cases with concrete inputs only. Each of them can be extended in such a way that they also support free input values. It is possible to leave some inputs open at defined points in time, whereas the other inputs can still be concrete. With that, all presented algorithms can be used as a bounded model-checker by simply defining each input value for each time step (up to a specified time step bound) as a free input value. Using free input values in test cases allows to be more flexible, whereas concrete input values can be processed faster. In some cases however, free inputs might reduce the length or number of necessary test cases.

6.1.1. Free Inputs Modes for Simulation-based Algorithm

Internally, simulation (Section 3.2) can only deal with concrete input values. To simulate one open input value, it is necessary to simulate the circuit twice: once with the concrete input value 0, and once with the concrete input value 1.

If a test case (Section 2.8) contains n open input values, it is necessary to simulate the whole circuit 2^n times, *i.e.* the number of simulations grows exponentially to the number of open input values.

Algorithm 10 SIM 1

```
1: procedure ANALYZESIMFREEINPUTS(test case $t$)
2:     $t_{concrete}[]$ := generateConcreteTestCases($t$)
3:     for $i := 1 \dots 2^n$ do
4:         ANALYZESIM($t_{concrete}[i]$)
5:     end for
6: end procedure
```

6.1.2. Free Inputs Modes for semi-formal Algorithms

The symbolic nature of our SAT solver based algorithms presented in Section 3.3 for vulnerable latches and 4.1 and 4.2 for false positives makes them perfectly suitable for free input values.

All of them contained a fault-free concrete simulation $(\mathbf{x}', \mathbf{o}, a) := \mathsf{sim}(\mathbf{x}, t[i])$ The results were used for a comparison with a transition relation T_{err}, which contained a bit flip. Some also contained a concrete faulty simulation as a performance optimization.

However, as stated in the previous section, the number of simulations grows exponentially with the number of open input values. Besides that, the huge number of simulations barely harmonize with the symbolic part of the semi-formal algorithms. Because of that we decided to encode a transition relation (Section 2.10) instead of a concrete simulation for the fault-free part when free input values are involved. The faulty concrete simulations had to be dropped as well.

The new fault-free *symbolic*-simulation produces an unrolled transition relation $T(\overline{x}, \overline{i}, \overline{o}, \overline{x}') \wedge T(\overline{x}', \overline{i_2}, \overline{o_2}, \overline{x}'') \wedge ...$ which grows in size if a lot or only free input values are used and which is smaller if lots of concrete input values are used. That is because concrete values consisting of truth values only do not result in a new literal whereas open input values have to be represented by a new boolean literal.

In the end, it is necessary to compare the faulty transition relations T_{err} with the fault-free transition relations $T(\overline{x}_i, \overline{i}_i, \overline{o}_i, a_i, \overline{x}'_i)$ instead of comparing with results from a concrete simulation $(\mathbf{x}'_i, \mathbf{o}_i, a_i) := \mathsf{sim}(\mathbf{x}_i, t[i])$.

Changing semi-formal algorithms to BDD based algorithms can often be achieved without much effort. Because of that, the concept of free input values for the BDD based algorithms is the same as for SAT-based algorithms. Again, the concrete fault-free simulation has to be exchanged by a symbolic formula. In the case of BDDs, it has to be exchanged by a BDD representation of the fault-free transition relation. For free input values fresh BDD variables are generated, whereas concrete input values keep the BDD small.

6.2. Environment Models

In certain situations some output values might be irrelevant, for example when an output indicating that data on a bus is ready to be read is set to false. If that is the case, then it does not matter whether the output values on the data-bus are flipped or not. Such situations do not have to be reported as vulnerabilities. On the contrary, a false positive should still be reported if only irrelevant outputs are corrupted.

The optional environment models can be used to specify under which circumstances an output is relevant. It is encoded as a circuit as well.

Environment circuits can use the same inputs as the original circuit and all outputs signals of the original circuit. For the sake of implementational simplicity, the number of inputs has to match: The input signals of the environment circuit are a concatenation of the inputs and outputs of the original circuits. However, the environment model does not necessarily have to make use of all provided input signals. Each output of the environment model corresponds to one of the original circuit. An environment output set to true indicates that the corresponding output in the original circuit is relevant at this point in time.

Some algorithms also support test cases that contain open input-values, which can be set arbitrarily. However, some input combinations may never occur in a practical mode of operation. An optional output can be added to restrict the number of allowed input combinations. This additional output is intended to be used as a formula talking about the input values, which is true whenever the input combination is allowed. With that, a SAT solver is required to set the unspecified input values in such a way that that this output stays *true*. If at some point it is not possible to set the open inputs accordingly, the algorithms won't find any further vulnerabilities/false positives from this point forward.

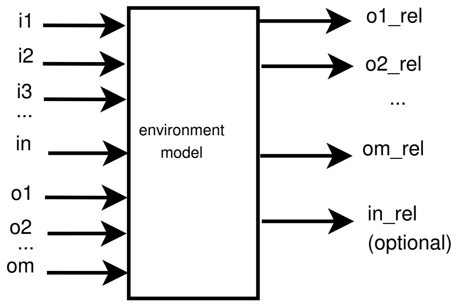


Figure 6.1.: An environment model defining relevance of outputs and optionally also allowed input combinations

6.2.1. Environment Models for Simulation-based Algorithm

Internally, the simulation-based Algorithm (Section 3.2) works with concrete input values only, even when a mode with free-inputs test cases is used (Section 6.1.1). Recall that we model the environment as a standard circuit with inputs and outputs. Therefore we can also perform a concrete simulation of the environment model.

```
1: $\bar{i}_{env} := t[i] \cup outputs[i]$
2: $(\bar{o}_{rel}, input\_ok) := simulate1step(env, \bar{i}_{env})$
```

First, the concrete input values from the test case $t[i]$, which were also used for the original circuit, are concatenated with the output values of the concrete simulation. This new vector $\bar{i}_{env}$ is used as input for the simulation of the environment model.

The outputs of the environment simulation $\bar{o}_{rel}$ define which of the outputs of the original circuit is relevant at the moment and which is not.

The optional output *input_ok* defines if the input combination is allowed, which is only useful for the free-input mode. If *input_ok* is set to false, the algorithm can be aborted for the current test case t because an illegal input combination has been used.

When comparing the outputs of the correct and faulty simulation, only the outputs that are relevant according to $\bar{o}_{rel}$ have to be checked. A modified irrelevant output does not indicate a vulnerable latch, only a modified relevant output does.

6.2.2. Environment Models for SAT- and BDD-based Algorithms

This section describes how symbolic (SAT- or BDD-based) algorithms for vulnerable latches (Section 3.3) and for false positives (Chapter 4) can be modified to support an environment model.

Symbolic algorithms that use test cases with concrete input values only can make use of an environment model similarly as the simulation based algorithm: Relevant outputs are computed via concrete simulations as well. The concrete input values are taken from the test case and the concrete output values are taken from the concrete fault-free simulation. With the help of these values, relevant outputs can be computed.

When searching for vulnerable latches, the clauses saying that an output is different is omitted when an output is irrelevant. Similarly, clauses requiring that an output is equal can be omitted for irrelevant latches in algorithms for false positives.

When free input values are involved, concrete simulation is not applicable anymore for symbolic algorithms. For these modes it is necessary to encode the environment model symbolically as well. This is achieved by computing a transition relation of the environment.

The transition relation $T_{env}(\overline{x}_{env}, \overline{i}_i, \overline{o}_i, \overline{o}_{rel}, input\_ok, \overline{x}'_{env})$ uses the same input literals $\overline{i}_i$ as the original circuit. Additionally the output literals of the original circuit $\overline{o}_i$ can be used as input for the environment model. The outputs $\overline{o}_{rel}$ are a vector of literals that define which of the original outputs $\overline{o}_i$ are relevant.

Since the relevance of outputs now depends on open inputs, it is not possible to use simulation to compute which output is relevant and then leave out the constraint for that output. Instead, it is necessary to define that the formula to compare outputs depends on the literals that define the relevance of the corresponding outputs.

In order to restrict the choice of input combinations to allowed input combinations only, the literal $input\_ok$ is added as a unit clause.

6.3. Combining the different Algorithm types

We are classifying components into four categories: *vulnerable*, *definitely protected*, *false positive* or unknown. All of our algorithms can be used to exclude some components, meaning that only a subset of all components is checked.

Obviously, a component can not be both vulnerable and definitely protected at the same time. It makes sense to exclude the findings of one algorithm before running the other.

Similarly, one might want to first run a mode/a simpler test case that can be executed faster but does not find all components. Afterwards, a slower mode/more sophisticated test cases can be used to focus on the remaining unclassified components.

7. Implementation

As a proof of concept, we implemented each of the algorithms presented in Chapter 3 (Vulnerable Latches), 4 (False Positives) and 5 (Definitely Protected Latches) in C++. The bounded model-checking algorithm from Section 3.1 is realized as a small standalone tool AlarmToMC that converts a circuit into a model-checking problem. All other discussed algorithms are implemented in our comprehensive soft error tool named *OpenSEA*.

This chapter explains decisions made, format specifications and conventions created for our reference implementation of the said algorithms.

7.1. Format specifications

OpenSEA can analyze circuits in AIGER[1] file-format (version 20071012) with included error-detection logic. The last output has to be the special *alarm* output, as can be seen in Figure 7.1.

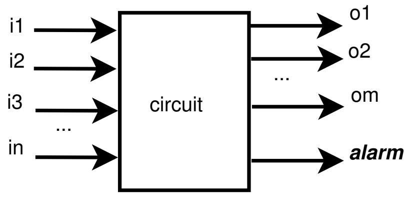


Figure 7.1.: The input circuit including the special *alarm* output

Some algorithms require test cases (Section 2.8) which provide input values. Test-cases can either be generated randomly or be provided as ASCII text

[1]http://fmv.jku.at/aiger/FORMAT.aiger

files. One line contains an input vector for one time step, open input values are declared by a question mark instead of 1 or 0.

Most modes support an optional environment model (Section 6.2) which can be used to specify under which circumstances an output is relevant. Like the original input circuit, they are encoded in the AIGER format as well. The input- and output- specification of environment models can be seen in Figure 6.1 on page 47.

After analyzing the quality of the protection logic, the tool results in a list of errors or a list of protected latches (depending on the selected algorithm). Example executions of *OpenSEA* are listed in the Appendix (page 74ff).

When searching for vulnerable latches the algorithm outputs a list of the detected components. Optionally, a detailed error-trace can be generated for each vulnerable latch consisting of the concrete input values for each time step, the point in time, at which the latch has to be flipped, and the point in time, at which the error has an effect on the output values.

When searching for false positives, the tool generates a list of the superfluous alarms, each consisting of a flipped component C, the point in time j where it has been flipped, a point in time k where the alarm was raised gratuitously, and a point in time $i + 1$ where the state is the same as it would have been without flipping in step j. Recall that to be a false positive, the output values between j and $i + 1$ must not change. If the test case contained free input variables, an input trace with the concrete input values to reproduce the error is generated.

When searching for definitely protected latches, a list of latches that are definitely protected is generated.

7.2. Algorithms and Modes

The following tables list algorithms that have been implemented in *OpenSEA* and references them to the corresponding sections of this thesis.

7.2.1. Modes for Vulnerabilities

Algorithm	Mode	Description
`SIM`	0	only for concrete input values
(3.2)	1	capable of free inputs (currently at most 64)
`STA`	0	copy whole transition relation when unrolling
(3.3.1)	1	compute transition relation on the fly
	2	capable of free inputs
`STLA`	0	standard mode
(3.3.2)	1	capable of free inputs
`BDD`	0	1-hot encoding for $\overline{c}$
(3.3.2)	1	cardinality constraints for $\overline{c}$
	2	binary encoding for $\overline{c}$
	3	binary encoding for $\overline{c}$ and $\overline{f}$
	4	binary encoding for $\overline{c}$ and $\overline{f}$, free inputs

The BDD based algorithms is a variant of the SAT-based STLA algorithm. It currently does not support environment models.

7.2.2. Modes for False Positives

FP denotes a quantitative analysis, meaning that multiple false positives per component are reported, whereas FPS only reports at most one false positive per component.

Algorithm	Mode	Description
FP(S)	0	`STA` based (4.1)
(4)	1	`STLA` based (4.2)
	2	`STA` based + capable of free inputs
	3	`STLA` based + capable of free inputs

7.2.3. Modes for Definitely Protected Latches

Algorithm	Mode	Description
DP	1	testing latches individually for 1-step protection (5.1)
(5)	2	testing latches simultaneously for 1-step protection (5.2)
	3	testing latches individually for k-step protection (5.3)
	4	testing latches simultaneously for k-step protection (5.4)

Algorithms for definitely protected algorithms do not yet support environment models.

7.2.4. Available Engines

Our SAT-based algorithms (STA, STLA, FP(S) and DP) rely on external SAT solvers. *OpenSEA* provides accesses to SAT solvers via an abstract interface, therefore SAT solver libraries are interchangeable. At the moment, the following SAT solvers are available and can be selected via a command-line parameter:

- `MiniSat 2.2.0` [26]
- `Lingeling ayv-86bf266-140429` [27]
- `PicoSAT 960` [28]

Our BDD-based algorithms use `cudd 3.0.0` by Fabio Somenzi [29]. Integrating Sylvian [30] as an alternative engine would be interesting since it can run parallel on multiple CPU-cores.

The concrete simulation of AIGER circuits was implemented by us.

7.3. AlarmToMC - detecting vulnerable latches via bounded model-checking

AlarmToMC is a simple standalone tool that converts an AIGER circuit (Figure 7.1) into a model checking problem to detect whether that circuit contains any vulnerable latches or not.

The resulting circuit (as described in Section 3.1, Figure 3.2) is again an AIGER circuit that can be tested with model checkers like BLIMC or IC3.

The original circuit contains vulnerable latches if the model checker is able to find a *bad* trace for the converted circuit.

7.4. AddParityTool - adding a simple parity-net to circuits

AddParityTool adds a basic parity net to an existing AIGER circuit and returns a protected AIGER circuit, as can be seen in Figure 7.2. The resulting circuit contains an *alarm* output, which is raised as soon as a latch is flipped.

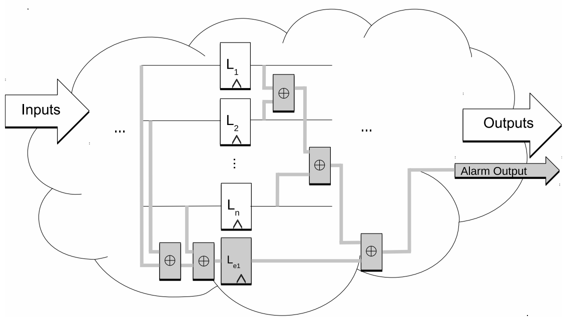


Figure 7.2.: Resulting circuit of AddParityTool

The tool allows the user to specify a percentage of latches to protect (0-100%), which is useful for benchmarks. Additionally, the number of latches that are protected by one additional parity-latch can be specified.

8. Experimental Results

In this chapter, we evaluate the performance of algorithms implemented in *OpenSEA*.

Note that all following charts in this chapter are cactus-plots with a logarithmic y-axis showing the execution time. The x-axis shows the number of benchmarks that can be solved within that time limit. Throughout this chapter, we are referring to the algorithms and their modes using the abbreviations and mode numbers defined in Section 7.2.

8.1. Benchmarks and Benchmarking Environment

Benchmarking Environment

All tests have been carried out on machines with two Intel Xeon E5430 CPUs, each with four cores running at 2.66GHz, operated from a 64bit Linux. Since our tool at the moment uses only one core, up to 8 benchmarks can be executed simultaneously on one machine. An execution timeout was set to 10000 seconds. The memory limit was set to 6 GiB RAM.

The SAT-based algorithms used `MiniSat 2.2.0`, BDD based algorithms used `cudd 3.0.0`.

IWLS Benchmarks

We used most of the publicly available IWLS 2002[1] and IWLS 2005[2] benchmarks for measuring the performance of our implementation. In total, we chose 558 benchmarks with up to roughly 1000 inputs, 1400 outputs, 10000 latches and 80000 and-gates.

For this, we first converted the circuits into the AIGER format using ABC [31]. Then, we added a generic protection logic to them which compares the parity sum of the previous latch inputs and the current latch outputs, as explained in Section 2.2. Our simple AddParityTool (Section 7.4) can be used to protect a specified percentage of all latches. This is useful for generating circuits that are partly vulnerable and partly protected. We chose to protect 90% of the latches for our experiments. Pairs of two latches are protected by one additional latch. The added error detection logic consequently increased the circuit size by an average of 5 AND-gates and half a latch per latch to protect.

The error detection logic *immediately* raises an alarm when the output of a protected latch is flipped.

ADD Benchmarks

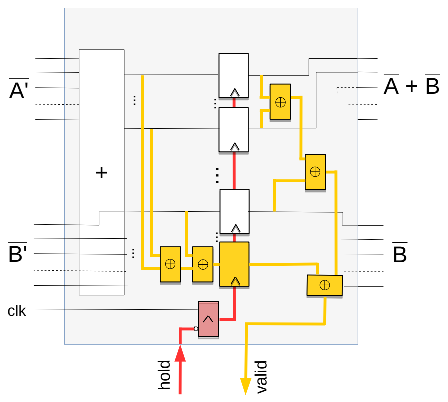
The IWLS circuits were reasonable for first experiments since it was possible to start with a huge amount of benchmarks without too much effort. Unfortunately, it is too cumbersome to completely understand the structure of all of them or to build in a cleverer protection logic. Therefore, we decided to create a more realistic benchmark which can also be parametrized in size.

The motivation for creating the ADD benchmarks (Figure 8.1) was to mimic a pipelined processor. The circuit performs a multiplication of the input by $(n+1)$. The multiplication is carried out by n additions over n time steps.

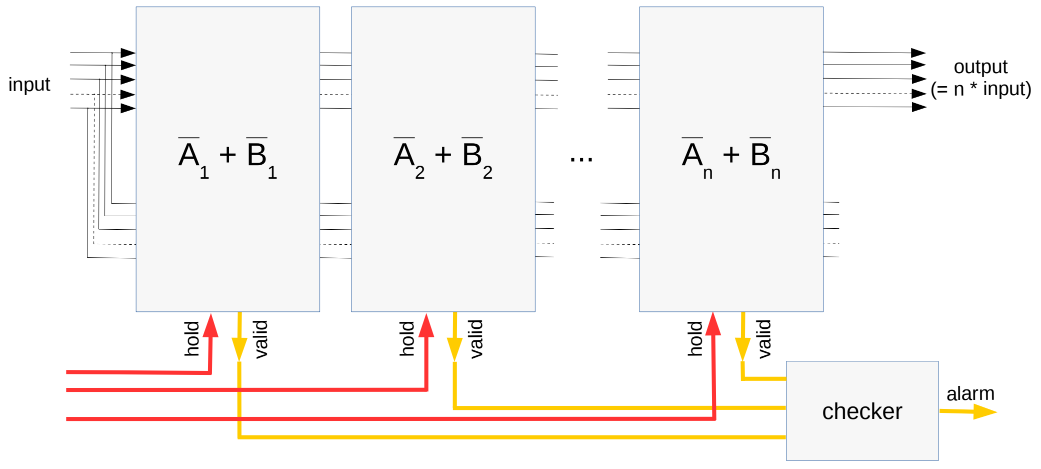
The circuit consists of n layers. Each layer has two input vectors $\overline{A}'$ and $\overline{B}'$ and two output vectors. The first output vector is the sum of both input

[1] `http://www.eecs.berkeley.edu/~alanmi/benchmarks/iwls2002/`

[2] `http://www.eecs.berkeley.edu/~alanmi/benchmarks/`



(a) single module


input
$\overline{A}_1 + \overline{B}_1$
$\overline{A}_2 + \overline{B}_2$
...
$\overline{A}_n + \overline{B}_n$
output
(= n * input)
hold
valid
hold
valid
hold
valid
checker
alarm

(b) complete pipelined circuit

Figure 8.1.: ADD benchmarks. The number of inputs and the number of layers is parametrized.

vectors from the previous state, the second output vector equals the second input vector from the previous state. A special *hold* input can be used to prevent values in the output buffer from getting overwritten by the current input-values of that layer. The *valid* signal is a simple parity computation of the latches from the output buffer of the layer, similar to the protection for the IWLS benchmarks.

The *checker* circuit raises the alarm output if the *valid*-signal is set to *false* in one layer and the *hold*-signal of the succeeding layer is set to *false* as well. That is the case when a bit flip in one layer would corrupt the internal state of the succeeding layer.

The benchmarks were written in Verilog and converted to AIGER using `vl2mv`[3] and ABC. Both the number of layers (up to 10) and the number of inputs/outputs (up to 128) are parametrized in order to provide a set of scaling benchmarks. A list of the benchmarks can be found in Appendix D.

ADD TMR Benchmarks

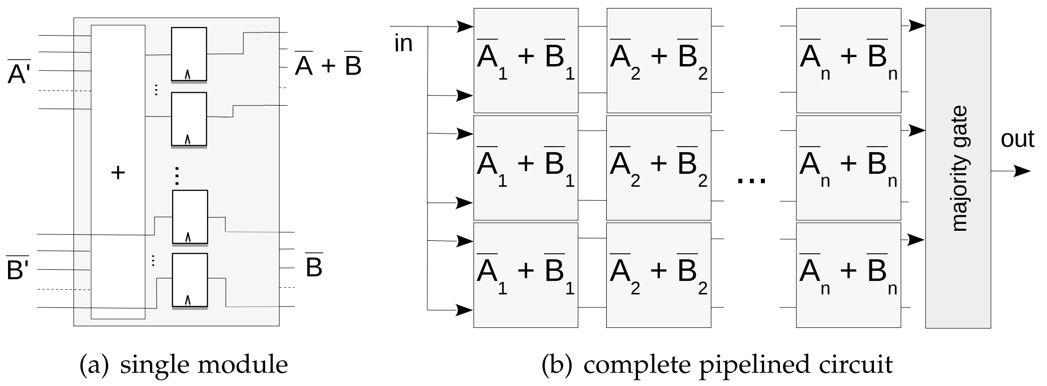
These circuits have the same functionality as the ADD benchmarks. They differ in implementing error *correction* instead of error *detection*.

Error correction is implemented via triple modular redundancy (TMR), as can be seen in Figure 8.2. The basic idea is to build a system that is resistant if only few faults happen. This is achieved by adding two additional copies of the circuit. All three copies produce the same output values if no fault happens. The majority gate after the final layer sets the final outputs by performing a majority vote: An output signal is set to *true* if at least two copies computed the value *true* for that output.

These benchmarks do not contain an *alarm* signal. Instead of reporting an error, it is corrected automatically. After a fault happens, it takes up to the number of layers time steps until the internal state recovers again. But the outputs remain correct as long as not more than of the three copies is corrupted at the same time.

The circuits were also designed in Verilog and then translated to AIGER. The list of our 25 TMR benchmarks starts with a circuit with 2 layers and 8 inputs and ends with one consisting of 10 layers and 128 inputs. It can be found in Appendix D. Note that TMR protection leads to a circuit overhead of over 200%, since it is necessary to add two identical copies of the circuit and a majority gate.

[3] `http://vlsi.colorado.edu/~vis`



$\overline{A'}$
$\overline{B'}$
+
$\overline{A} + \overline{B}$
$\overline{B}$
in
$\overline{A}_1 + \overline{B}_1$
$\overline{A}_2 + \overline{B}_2$
...
$\overline{A}_n + \overline{B}_n$
majority gate
out
(a) single module
(b) complete pipelined circuit

Figure 8.2.: ADD TMR benchmarks. The number of inputs and the number of layers is parametrized.

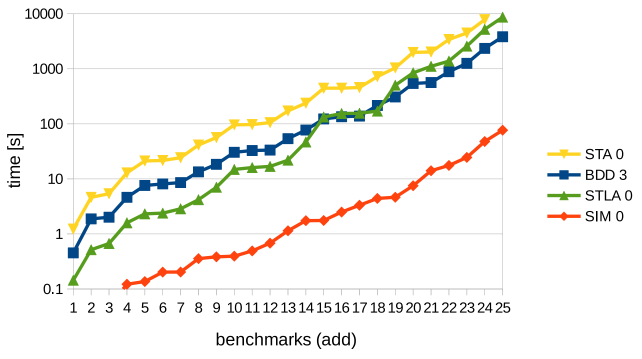
8.2. Performance Evaluation for the Detection of Vulnerable Latches

8.2.1. Results with concrete Inputs only

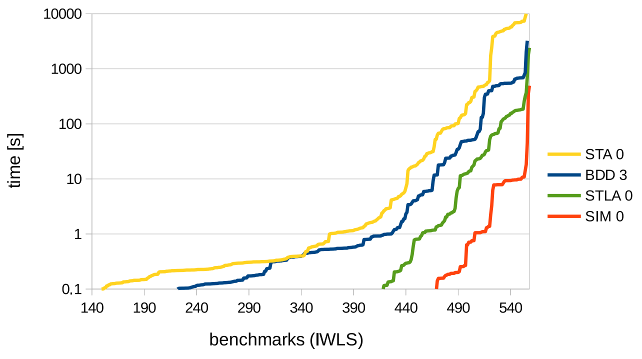
OpenSEA implements multiple algorithms that are capable of finding *vulnerable* components (Chapter 3). Figure 8.3 shows their execution times for the ADD and IWLS benchmarks with concrete test cases. As an input, three random concrete test cases with a length of 15 time steps are used for each benchmark. As can be seen, the simulation based algorithm (SIM) is the fastest for completely concrete test cases.

Making not just the point in time but also the latch to flip symbolic clearly leads to a significant speed-up (STA vs STLA).

The BDD-based algorithm, which is an adaption of the SAT-based STLA algorithm, lies between STA and STLA for the IWLS benchmarks. Interestingly, BDD outperforms STLA on more complicated ADD benchmarks. To our surprise, we found out that the BDD-based algorithm works better with disabled variable reordering.



(a) ADD benchmarks



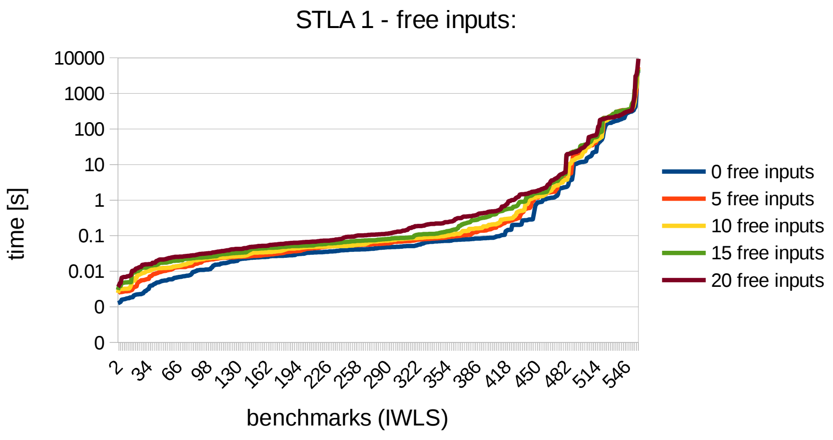
(b) IWLS benchmarks

Figure 8.3.: Execution times to detect vulnerable components. 3 concrete test cases, 15 time steps.

8.2.2. Amount of Unspecified Input Values

Using free input values in test cases allows to be more flexible, whereas concrete input values can be processed faster. In some cases however, free inputs might reduce the necessary length or number of test cases.

In this section, we compare the run-time of the SAT-based `STLA` algorithm using free input values with the naive simulation-based approach (`SIM`). Recall that each free input value in the worst case doubles the run-time when using a naive exhaustive approach.



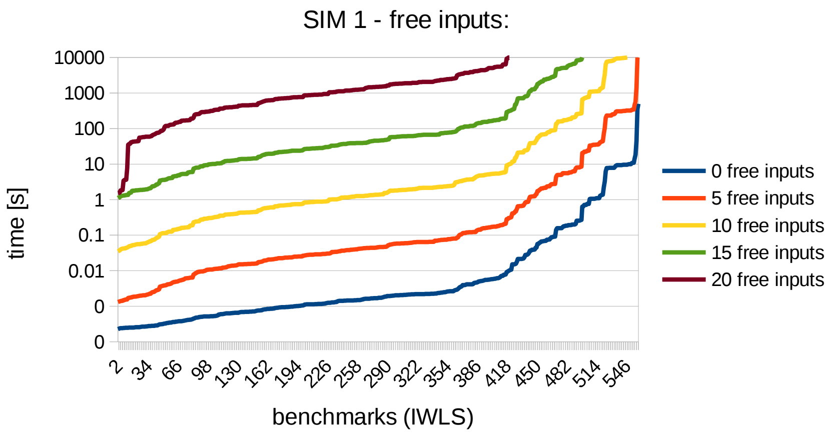


Figure 8.4.: Using free input values scales significantly better for the `STLA 1` algorithm. Benchmarks are 90% protected, 3 test cases, each of a length of 15 time steps. The free input values are located within the first time step(s).

The chart in Figure 8.4 shows the effect of free input values on the execution time for both the `STLA` and the `SIM` algorithm. `SIM` is the fastest algorithm when using only concrete inputs, as already mentioned in previous chapters. Just for concrete input values, converting a circuit to a CNF transition relation, unrolling it and then performing SAT solver calls does not pay off compared to a fast simulation. The SAT-query is already a complicated formula because of the $\overline{f}$ and $\overline{c}$ literals, even with concrete inputs only. However, that changes when free input values come into play. Using more free input values with STLA does not add a lot of extra runtime cost (top graph). The performance of SIM is however drastically reduced when more input values are set as open.

The STLA algorithm has the crucial ability to utilize the flexibility of SAT solvers. Because of that, the symbolic algorithm outperforms the simulation approach when it comes to scalability regarding free-input values. Both algorithms roughly have the same performance when using about 5 free input values. For more free input values, STLA is the best choice, for fewer to none, SIM is faster.

It is arguable that optimizations in SIM could lead to a speed-up. For example the fact that several input combinations might lead to the same next state could be used to prevent the algorithm from doing the same simulation more than once. That certainly is a valid objection, it is however questionable if such an optimiation can be implemented easily. SAT solvers on the other hand do such optimizations intrinsically due to the smbolically encoded queries.

8.2.3. Comparison with Model Checking

In the previous section we discussed the influence of free input values. Model checking carries the idea of free inputs to an extreme: each input value for each time step (up to a specified time step bound) is a free input value. As a consequence, model checking is the most accurate, but also the computationally most expensive approach to a soft error-analysis.

OpenSEA is capable of analyzing a circuit in a full model-checking mode, meaning that no concrete input values in a test case are used. Each mode that supports free inputs can be used for that.

Since most model checkers are highly optimized, this section shows results of a model checker (namely BLIMC) as well. For this purpose, we wrote a small tool (AlarmToMC) which converts a circuit with protection logic to a model-checking problem, which again is a special type of circuit. Model checkers typically take a circuit with exactly one output and try to find an input sequence which sets this error-output to true.

Figure 8.5 illustrates the enormous differences in execution time between a model-checking approach (using BLIMC and our `STLA 1`) and concrete input values only (`STLA 0`). Our `STLA` algorithm in full MC mode is competitive with `BLIMC`.

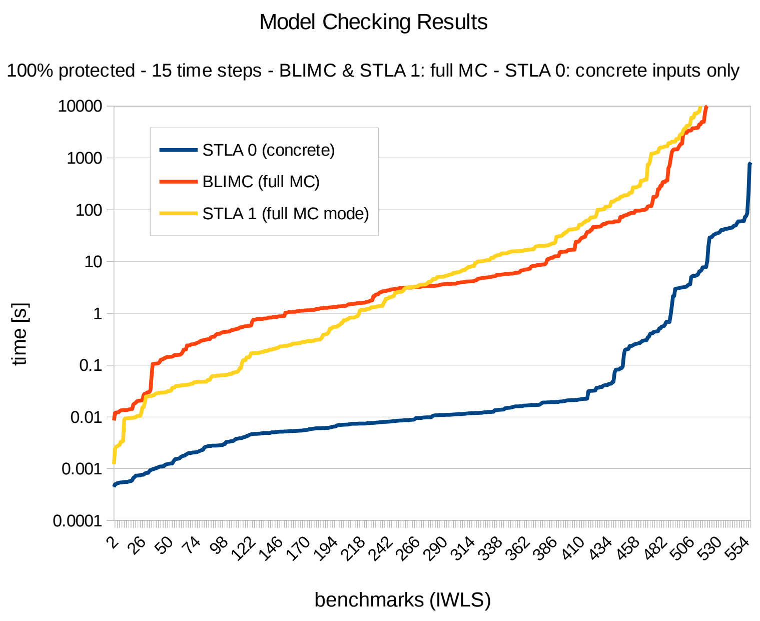


Figure 8.5.: Both `BLIMC` model-checking for 15 time steps and `STLA 1` with 15 time steps of only free inputs try out all possible input combinations, whereas `STLA 0` uses only one concrete test case with a length of 15 time steps.

8.2.4. Length of the Test Cases

The length of test cases (number of time steps) is one parameter that has an effect on the execution time. Longer test cases obviously result in a longer execution time. Figure 8.6 illustrates the influence of test case lengths: The `STLA` algorithm was executed with test case lengths of 15, 30, 60 and 100 time steps. No free input values were used, the randomly generated test cases consisted solely of concrete input values (0 or 1). The chart indicates an increase of execution time of roughly a half order of magnitude when doubling the test case length. An increasing test case length in `SIM` only leads to a linearly growing run-time.

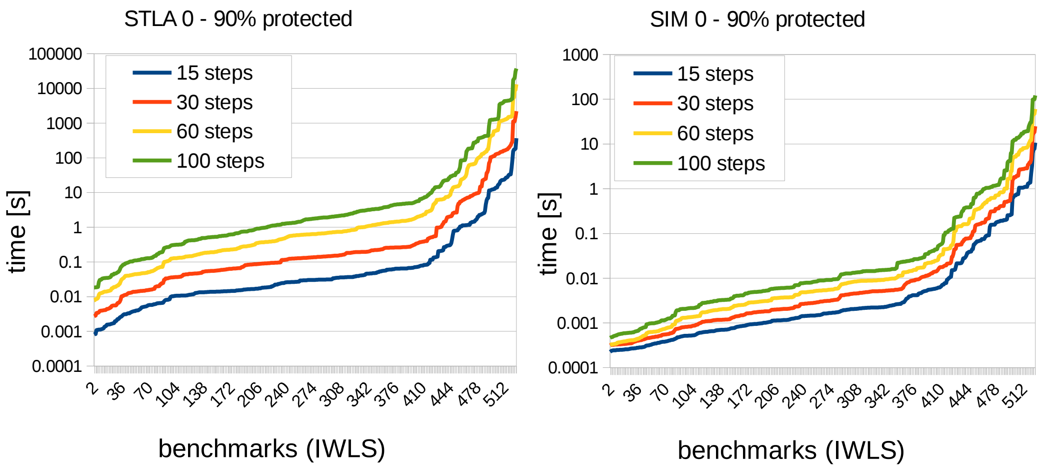


Figure 8.6.: `STLA 0` and `SIM 0` execution times for different test case lengths

8.3. Performance Evaluation for the Detection of False Positives

Figure 8.7 shows the execution time for detecting false positives. Recall that `FP` performs a *quantitative* and `FPS` a *qualitative* analysis. Mode 0 checks each latch individually, as in `STA`, and mode 1 checks all of them individually, as in `STLA`.

It can be seen that the qualitative analysis outperforms the quantitative analysis and that testing all latches at once is faster than checking them individually for most of the bigger benchmarks.

Recall from Chapter 4 that a false positive is detected when a bit flip does not change any outputs and the internal state of the circuit eventually

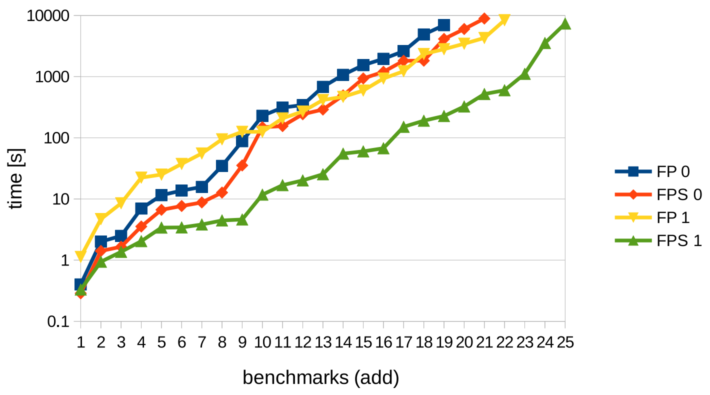


Figure 8.7.: Execution times to detect false positives. ADD benchmarks, 3 concrete test cases, 15 time steps

recovers again. The ADD circuits are perfectly suitable for benchmarking these algorithms since it takes up to the number of layer time steps until the fault is repaired again or until the fault becomes visible to the user by propagating to the outputs.

8.4. Performance Evaluation for the Detection of Definitely Protected Latches

Recall (Chapter 5) that a definitely protected component can be flipped in any valid state without any undetected errors. Thus, either an alarm is raised or the internal state recovers again from the flip without affecting any output values.

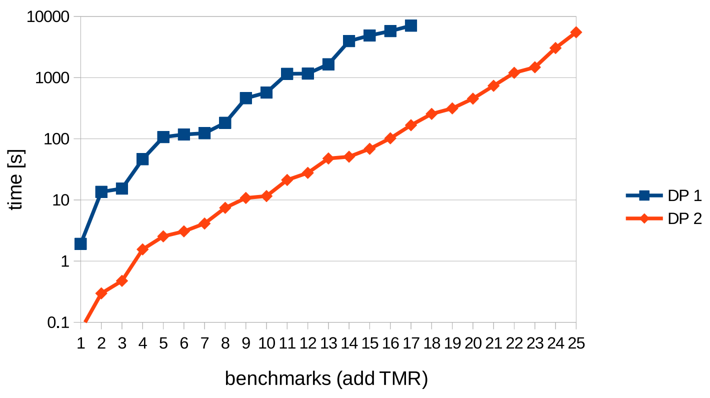
We chose the ADD TMR benchmarks for testing the performance of our algorithms because they are known to be hard to verify. These circuits do not have an *alarm* output. It takes up to the number of layers time steps until the internal state is the same again as it would have been without a bit flip.

Figure 8.8 shows the performance of our algorithms for definitely protected latches. We ran our ADD TMR benchmarks by starting from any state that is reachable from any state after 10 time steps. This way, we made sure that the check does not contain checks that already start from illegal states that violate our single-fault assumption. Modes `DP 1` and `DP 3` check each latch individually, `DP 2` and `DP 4` check all latches at once.

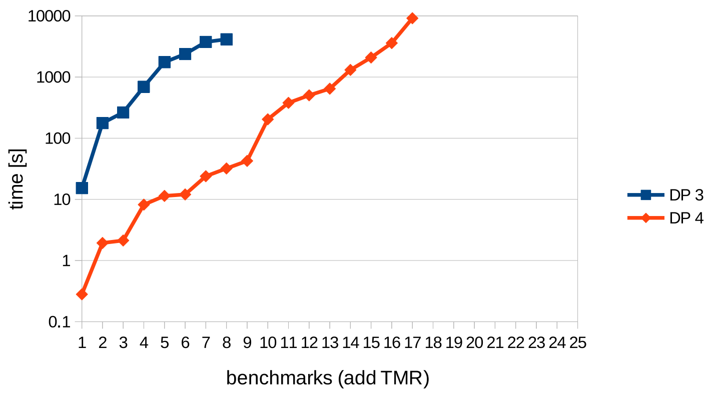
The algorithms that check for 1-step protection could only classify the latches from the last layer of our ADD TMR circuits as definitely protected. Latches in others than the last layer are not identified as protected because it takes more than one time step until the currupted value is masked out in the majority-voting layer. With our algorithms for k-step protection, we were able to detect all latches that are definitely protected. The parameter `k` has also been set to `10`, since the benchmarks can have up to 10 layers.

It turned out that checking for k-step protection took approximately two magnitudes of order longer than checking for 1-step protection. Testing all latches at once is about one order of magnitude faster than checking each one individually, in both algorithm types.

Furthermore, we found out that modes that check multiple latches at once (modes `2` and `4`) perform better if more latches were definitely protected. This makes sense since these algorithms work with satisfying assignments that provide counter-examples with latches that are not definitely protected. They run until the queries eventually become unsatisfiable. In contrast, modes `1` and `3` (that test latches individually) were faster if fewer latches were definitely protected. It is obviously easier to find just one satisfying assignment (if the latch is not definitely protected) than to prove that a formula is unsatisfiable (i.e. the latch is definitely protected). Proving something unsatisfiable means to rule out *all* possible combinations of truth values. In contrast, the SAT solver's work is already done if it manages to find just *one* satisfying assignment.



(a) testing 1-step protection



(b) testing k-step protection, k set to 10

Figure 8.8.: Execution times to detect definitely protected components.

9. Related Work

In 2006, Krautz et al. proposed a way to evaluate the coverage of error detection logic using BDDs [32]. They used a fault injection model similar to our BMC based approach (Section 3.1) in which they compare a fault-free device with a faulty one. The output of their fault injection model is defined by a property checker that defines multiple properties by comparing primary outputs of both circuit copies. A sequential equivalence check of this circuit is performed up to a defined number of time steps by creating a BDD representation. The number of input combinations that make properties true are counted. Among other properties, they count a property similar to our definition of *vulnerabilities* and the number of injected faults. A coverage is computed using these numbers.

Holcomb et al. showed another way to compute the *failure in time* rate by performing a system-level analysis [33]. In contrast to our algorithms, their methods rely on formal specifications. Our approaches only need to know which output is the *alarm* output.

Fey, Frehse and Drechsler presented a bounded model-checking based technique that computes a lower and an upper bound of the fault tolerance [34]. This is done by an approximate reachability analysis, since a full reachability analysis is considered to be computationally infeasible. They compute both an over- and an under-approximation of the reachable states and use them as initial states in a time-bounded check. With that, it is possible to provide a lower and an upper bound for classifying vulnerable components. The over-approximation contains more than the actually reachable states. Because of that, some components might be mistakenly classified as vulnerable because they could modify primary outputs in an unreachable state. Similarly, an under-approximation detects fewer vulnerable components due to a smaller state-space. Both numbers together define a lower and an upper bound. Our approach to detect definitely protected latches (Chapter 5) only makes use of an over-approximation because we wanted define a very strong check.

A year later, the same three authors together with Arbel and Yorav proposed additional approaches to classify components. This time, they use interpolation for an over-approximation of the reachable state space and compute fixed-points [35]. This way, only states that are relevant to the property to prove are considered.

In 2014, Arbel, Koyfman, Kudva and Moran published a structural approach to detect parity-protected memory elements [36]. For this purpose, they first search for potential error detection circuits and for latches that are potentially protected by those using functional analysis. Afterwards, they use SAT-calls to prove that these latches are indeed protected by the parity net. The unique characteristic of their method is that they analyze a circuit locally rather than looking at the behavior of the entire system. Their method can be applied whenever error checker latches are present, which is not the case for our ADD TMR benchmarks presented in Section 8.1.

Methods that systematically construct robust systems[37] might make our work and all methods to verify error detection and error correction obsolete one day. However, as long as these intelligent compilers are not yet perfect (if they will ever be) verification of circuits will stay important.

10. Conclusions

In this thesis we presented multiple approaches that can be used to analyze how hardware reacts to faults that are caused by cosmic radiation. For this purpose, we defined the terms *vulnerable*, *definitely protected* and *false positive*. Our solutions try to automatically detect whether a component is part of such a category.

In Chapter 3, we described methods to detect *vulnerable* components. A component is vulnerable if a fault in it can eventually become visible to the user on the outputs of a circuit. We started by showing a model-checking based approach that can only detect whether the circuit under test contains any vulnerable latches or not. This is achieved by creating a new circuit that contains two copies of the original circuit. One copy is unchanged, whereas the other is modified in such a way that latches can be flipped via additional inputs. We defined that the one and only output of the new circuit is set to 1 whenever the outputs of both copies are set differently without raising the special alarm output before. The (bounded) model-checker tries to find an input sequence for which this output can be set to true.

Afterwards, we described a simulation-based algorithm. Vulnerable components are detected by comparing a fault-free simulation with a faulty ones. A *test case* provides concrete input-values for the simulations. The output values of the correct simulation are compared with the ones from the faulty simulations, in which we encoded a bit flip.

In addition to that, we proposed two semi-formal SAT-based algorithms, in which we symbolically encoded the circuit. In one of them only the *point in time* when a component should be flipped was symbolic but the component to flip was fixed. In the other, both the component and the point in time to flip it were selected by the SAT solver. Again, the input values were provided by a test case. A satisfying assignment determines when a vulnerable component has to be flipped (and also which one, in the second algorithm). Finally, we developed a BDD-based version of the mentioned approach.

In Chapter 4, we presented versions of the SAT-based algorithms that detect *false positives* with input values from test cases. We said that components lead to a false positive if a bit flip raises the alarm, even though the outputs would not have changed. Similar to the symbolic algorithm for vulnerable components, we described a variant in which only the point in time is symbolic and another where the component to flip is also encoded symbolically.

In Chapter 5, we did not search for errors in the protection logic. Instead, we presented a method to prove that a component is *definitely protected*. No matter when a component is flipped, we require that either the alarm is raised or the internal state of the circuit recovers again without affecting any outputs. For this purpose, we are using an over-approximation of the reachable states. In addition to that, all input values are open. There are variants of the algorithm to check for 1-step protection and for k-step protection. Both have one version that checks each component individually and one that checks all components simultaneously.

Afterwards, we showed how each of the test case based algorithms can be adapted in order to support free input values. With that, it is possible to leave some or all input values open for an extensive analysis. Free input values provide more flexibility and might help to find more vulnerable components or false positives, whereas concrete input values can be processed faster. In addition to that, we explained how environment models can be used together with our existing algorithms. They can be used to define when outputs are relevant or to restrict the way open input values can be set.

The experimental results of our reference implementation *OpenSEA* showed that semi-formal algorithms like `STLA` scale better when it comes to free input values, whereas an analysis using concrete test cases only is carried out best with a simple simulation based approach (`SIM`). In any case, concrete input values can be processed faster than free input values.

The results indicate that our `STLA` algorithm with free input values seems to be a good compromise between a simulation based- and a full model-checking approach. Testing all components at once for definite protection is superior compared to testing them individually.

In essence, model-checking should be used if there is no or little knowledge of the circuit under test and/or if completeness is important. Simulation should be used for fast results if accuracy is not the most important factor. Knowledge of the operation of the circuit might be beneficial compared to random simulations. Semi-formal methods are the best solution for everything in between, *i.e.*, if there is at least some knowledge how the circuit works. Concrete input values can be used to increase the algorithm's performance and open input values can help to classify more components.

Appendix

A. OpenSEA vulnerable latches example

The following Listing shows how to use *OpenSEA* to find vulnerable latches

```
./immortal-bin -i shiftreg.aig -b stla -tcr 1 5 -d
[LOG] Input-File: shiftreg
[LOG] Inputs: 1, Latches: 3, Outputs: 1, ANDs: 6
[LOG] Back-End: stla, mode = 0
[LOG] #Errors found: 3

=================================================
Latch: 4 flipped at i=0
Error happened at timestep i=0
[SIM] i=?: state | inputs | outputs | next state
-------------------------------------------------
[ OK] i=0: 000 0 00 000
[ERR] i=0: 100 0 10 000 <<< flipped in this state! <<<
   ↪ wrong output in this state!

=================================================
Latch: 6 flipped at i=0
Error happened at timestep i=1
[SIM] i=?: state | inputs | outputs | next state
-------------------------------------------------
[ OK] i=0: 000 0 00 000
[ERR] i=0: 010 0 00 100 <<< flipped in this state!
[ OK] i=1: 000 0 00 000
[ERR] i=1: 100 0 10 000 <<< wrong output in this state!

  [...]

[LOG] Overall execution time: 0.000582 sec CPU time, 0
   ↪ sec real time.
```

Flipping literal 4 (= first latch) immediately results in a wrong output value (first output), whereas flipping literal 6 (= second latch) in time step 0 does not affect the output before time step 1. In both cases, the alarm output (second output) should have been raised.

B. OpenSEA false positives example

The following Listing shows how to use *OpenSEA* to find false positives

```
./immortal-bin -i fp.delay1.aag -b fp -tcr 1 3 -d
[LOG] Input-File: fp.delay1
[LOG] Inputs: 1, Latches: 4, Error Latches: 1, Outputs:
   ↪ 1, ANDs: 15
[LOG] Back-End: fp, mode = 0
[LOG] #False positive traces found found: 7

  component=4, flip_ts=0, alarm_ts=1, error_gone_ts=3
[SIM] i=?: state | inputs | outputs | next state
----------------------------------------------------
[ OK] i=0: 00000 0 00 00000
[ERR] i=0: 10000 0 00 01011 <<< flipped in this state!
[ OK] i=1: 00000 1 10 10010
[ERR] i=1: 01011 1 11 10100
[ OK] i=2: 10010 1 10 11000
[ERR] i=2: 10100 1 10 11000 <<< error gone in next state!

  component=6, flip_ts=0, alarm_ts=1, error_gone_ts=2
[SIM] i=?: state | inputs | outputs | next state
----------------------------------------------------
[ OK] i=0: 00000 0 00 00000
[ERR] i=0: 01000 0 00 00111 <<< flipped in this state!
[ OK] i=1: 00000 1 10 10010
[ERR] i=1: 00111 1 11 10010 <<< error gone in next state!

  [...]

[LOG] Overall execution time: 0.00148 sec CPU time, 0 sec
   ↪  real time.
```

Flipping literal 4 (= first latch) in the initial time step does not affect any output value under the provided (random) test case. Two time steps later, the next-state is the same again as without any flip. Therefore the alarm has been raised gratuitously in time step 1

C. OpenSEA definitely protected latches example

The following Listing shows how to use *OpenSEA* to find definitely protected latches

```
 ./immortal-bin -i toggle.perfect.aag -b dp -m 2 -d
[LOG] Tool has been started
[LOG] Input-File: toggle.perfect
[LOG] Inputs: 3, Latches: 3, Error Latches: 1, Outputs:
    ↪ 3, ANDs: 18
[LOG] Back-End: dp, mode = 2
[LOG] #Definitely protected latches found: 3 (100 %)

Definitely protected latches:
8       10      12
[LOG] Overall execution time: 0.000507 sec CPU time, 0
    ↪ sec real time.
```

The DP back-end searches for definitely protected latches. Here, mode 2 is used, where multiple latches are tested simultaneously if they are definitely 1-step protected (Algorithm in Section 5.2). As an output, the tool prints the detected definitely protected latches.

The reported latches do not have to be checked if they are vulnerable because it is proven that the system always either recover from a flip in that latch without affecting an alarm or that an alarm is raised before or when an output is wrong.

It might however still be the case that flipping a definitely protected latch can lead to false positives, which is why one might also want to run one of the algorithms dedicated to that problem.

D. Benchmark Properties and raw Results

Table D1 contains information about the circuit size of the ADD benchmarks described in Section 8.1 and the raw benchmark results of our algorithms presented in Chapter 3 and 4. The circuits follow the naming convention `add_i[in]_l[l]`, whereas **[l]** denotes the number of layers and **[in]** denotes the bit-width of the number to multiply and the number of outputs. The number of inputs of a circuit equals the bit-width plus the number of layers, since each layer adds an additional *hold* input-signal.

circuit	IN	FF	OUT	AND	STA 0	BDD 3	STLA 0	SIM 0	FP 0	FPS 0	FP 1	FPS 1
add_i8_l2	10	40	8	577	1.2	0.5	0.1	0.0	0.4	0.3	1.1	0.3
add_i8_l4	12	80	8	1219	5.4	2.0	0.7	0.0	2.5	1.7	8.5	1.4
add_i8_l6	14	120	8	1861	12.9	4.6	1.6	0.1	7.0	3.5	22.3	2.0
add_i8_l8	16	160	8	2503	24.3	8.6	2.3	0.2	15.8	6.6	55.5	3.8
add_i8_l10	18	200	8	3145	41.2	13.4	4.2	0.4	34.8	12.7	94.6	3.4
add_i16_l2	18	72	16	1153	4.6	1.9	0.5	0.0	2.0	1.4	4.7	0.9
add_i16_l4	20	144	16	2435	21.6	8.1	2.9	0.2	13.7	7.7	37.5	3.4
add_i16_l6	22	216	16	3717	56.4	18.4	7.0	0.4	88.0	35.4	125.3	4.5
add_i16_l8	24	288	16	4999	105.2	33.3	16.1	0.7	343.3	149.0	269.2	16.8
add_i16_l10	26	360	16	6281	171.7	53.8	21.8	1.1	680.9	290.1	467.4	20.1
add_i32_l2	34	136	32	2305	21.2	7.6	2.4	0.1	11.5	8.8	25.1	4.6
add_i32_l4	36	272	32	4867	95.9	32.8	16.9	0.5	312.2	155.6	208.1	11.8
add_i32_l6	38	408	32	7429	237.4	77.2	46.9	1.8	1070.2	493.1	592.0	60.1
add_i32_l8	40	544	32	9991	444.2	138.3	153.9	3.3	2601.6	933.1	1225.7	55.2
add_i32_l10	42	680	32	12553	720.9	215.2	170.1	4.4	4884.5	1822.3	2345.5	67.4
add_i64_l2	66	264	64	4609	97.1	30.4	14.9	0.4	229.3	242.8	125.4	25.4
add_i64_l4	68	528	64	9731	444.8	134.7	133.0	2.5	1957.6	1814.0	932.1	190.4
add_i64_l6	70	792	64	14853	1040.9	307.7	503.0	4.6	6937.1	4129.7	2811.8	226.9
add_i64_l8	72	1056	64	19975	2018.5	562.8	844.1	14.1	timeout	8902.3	4287.1	519.8
add_i64_l10	74	1320	64	25097	3407.2	884.4	1380.2	17.5	timeout	timeout	8364.6	599.1
add_i128_l2	130	520	128	9217	456.6	122.7	155.7	1.7	1541.8	1196.8	416.1	151.4
add_i128_l4	132	1040	128	19459	1985.1	541.1	1107.2	7.5	timeout	6002.3	3450.4	324.6
add_i128_l6	134	1560	128	29701	4466.6	1256.1	2574.7	24.6	timeout	timeout	timeout	1111.7
add_i128_l8	136	2080	128	39943	7806.1	2343.4	5198.8	47.7	timeout	timeout	timeout	3543.1
add_i128_l10	138	2600	128	50185	timeout	3819.7	8642.0	76.6	timeout	timeout	timeout	7363.0

Table D1.: ADD circuit properties and execution times for finding *vulnerable* latches and *false positives*

Table D2 shows circuit properties of the ADD TMR benchmarks described in Section 8.1 and results for the algorithms presented in Chapter 5. They follow the same naming convention as the ADD benchmarks: The first number indicates the bit-width of the number to multiply and in this case also the number of inputs and outputs. The second number denotes the number of layers, i.e. the number of additions in the pipelined architecture.

circuit	IN	FF	OUT	AND	DP 1	DP 2	DP 3	DP 4
add_tmr_i8_l2	8	108	8	550	1.9	0.1	15.3	0.3
add_tmr_i8_l4	8	216	8	1468	13.6	0.5	176.6	2.1
add_tmr_i8_l6	8	324	8	2386	46.4	1.6	693.1	8.2
add_tmr_i8_l8	8	432	8	3304	106.6	4.1	1747.0	12.0
add_tmr_i8_l10	8	540	8	4222	182.5	7.4	4138.5	23.9
add_tmr_i16_l2	16	204	16	1150	15.4	0.3	263.4	1.9
add_tmr_i16_l4	16	408	16	3076	117.6	3.1	3742.5	32.0
add_tmr_i16_l6	16	612	16	5002	459.7	10.8	timeout	203.7
add_tmr_i16_l8	16	816	16	6928	1163.6	27.7	timeout	502.4
add_tmr_i16_l10	16	1020	16	8854	1651.7	47.7	timeout	1305.9
add_tmr_i32_l2	32	396	32	2350	123.1	2.5	2376.4	11.4
add_tmr_i32_l4	32	792	32	6292	1153.1	21.2	timeout	378.3
add_tmr_i32_l6	32	1188	32	10234	3953.5	68.5	timeout	3589.2
add_tmr_i32_l8	32	1584	32	14176	5752.3	166.6	timeout	9143.4
add_tmr_i32_l10	32	1980	32	18118	7048.6	255.2	timeout	timeout
add_tmr_i64_l2	64	780	64	4750	569.9	11.6	timeout	42.6
add_tmr_i64_l4	64	1560	64	12724	4881.3	102.0	timeout	2081.6
add_tmr_i64_l6	64	2340	64	20698	timeout	314.0	timeout	timeout
add_tmr_i64_l8	64	3120	64	28672	timeout	734.4	timeout	timeout
add_tmr_i64_l10	64	3900	64	36646	timeout	1197.4	timeout	timeout
add_tmr_i128_l2	128	1548	128	9550	timeout	50.9	timeout	644.3
add_tmr_i128_l4	128	3096	128	25588	timeout	452.8	timeout	timeout
add_tmr_i128_l6	128	4644	128	41626	timeout	1477.1	timeout	timeout
add_tmr_i128_l8	128	6192	128	57664	timeout	3041.9	timeout	timeout
add_tmr_i128_l10	128	7740	128	73702	timeout	5511.2	timeout	timeout

Table D2.: ADD TMR circuit properties and execution times for finding *definitely protected* latches

All of the listed results and many more (including the ones of the 558 IWLS benchmarks) from all of our implemented modes together with the source code are available online in our repository[1].

[1] `https://github.com/p4p4/softerror-tools`

Bibliography

[1] G. E. Moore. "Cramming More Components onto Integrated Circuits." In: *Electronics* 38.8 (Apr. 1965), pp. 114–117 (cit. on p. 1).

[2] T. J. O'Gorman. "The effect of cosmic rays on the soft error rate of a DRAM at ground level." In: *IEEE Transactions on Electron Devices* 41.4 (Apr. 1994), pp. 553–557. ISSN: 0018-9383. DOI: 10.1109/16.278509 (cit. on p. 1).

[3] J. C. Pickel. "Effect of CMOS Miniaturization on Cosmic-Ray-Induced Error Rate." In: *IEEE Transactions on Nuclear Science* 29.6 (Dec. 1982), pp. 2049–2054 (cit. on p. 1).

[4] J. F. Ziegler. "Terrestrial cosmic ray intensities." In: *IBM Journal of Research and Development* 42.1 (Jan. 1998), pp. 117–140 (cit. on p. 1).

[5] R. Baumann. "Soft Errors in Advanced Computer Systems." In: *IEEE Des. Test* 22.3 (May 2005), pp. 258–266 (cit. on p. 1).

[6] N. D. P. Avirneni and A. Somani. "Low Overhead Soft Error Mitigation Techniques for High-Performance and Aggressive Designs." In: *IEEE Transactions on Computers* 61.4 (Apr. 2012), pp. 488–501 (cit. on p. 1).

[7] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems (3rd Ed.): Design and Evaluation*. Natick, MA, USA: A. K. Peters, Ltd., 1998 (cit. on p. 1).

[8] T. M. Austin. "DIVA: A reliable substrate for deep submicron microarchitecture design." In: *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. IEEE. 1999, pp. 196–207 (cit. on p. 1).

[9] F. A. Bower, D. J. Sorin, and S. Ozev. "A mechanism for online diagnosis of hard faults in microprocessors." In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2005, pp. 197–208 (cit. on p. 1).

[10] T. M. Austin and V. Bertacco. "Deployment of better than worst-case design: solutions and needs." In: *2005 International Conference on Computer Design*. Oct. 2005, pp. 550–555 (cit. on p. 1).

[11] R. Hamming. "Error Detecting and Error Correcting Codes." In: *Bell System Technical Journal* 26.2 (1950), pp. 147–160 (cit. on p. 1).

[12] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. M. Austin. "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor." In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. 2003, pp. 29– (cit. on p. 2).

[13] S. A. Seshia, W. Li, and S. Mitra. "Verification-guided Soft Error Resilience." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '07. 2007, pp. 1442–1447 (cit. on p. 2).

[14] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. "On latching probability of particle induced transients in combinational networks." In: *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. June 1994, pp. 340–349 (cit. on pp. 3, 9).

[15] S. Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325, 9780123695291 (cit. on p. 8).

[16] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008 (cit. on p. 10).

[17] J. R. Burch and D. E. Long. "Efficient Boolean Function Matching." In: *Proceedings of the 1992 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '92. Santa Clara, California, USA, 1992, pp. 408–411 (cit. on p. 11).

[18] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs." In: *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference*. 1999, pp. 193–207 (cit. on p. 11).

[19] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus." In: *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*. 1983, pp. 466–483 (cit. on p. 12).

[20] A. Kuehlmann, S. Member, V. Paruthi, F. Krohm, and M. K. Ganai. "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification." In: *IEEE Trans. CAD* 21 (2002), pp. 1377–1394 (cit. on p. 12).

[21] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: (1986), pp. 677–691. ISSN: 0018-9340 (cit. on p. 13).

[22] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. 2013 (cit. on p. 13).

[23] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. M. Austin. "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework." In: *ICCD*. 2008, pp. 363–370 (cit. on p. 20).

[24] K. K. Goswami, R. K. Iyer, and L. Young. "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis." In: *IEEE Transactions on Computers* 46 (1997), pp. 60–74 (cit. on p. 20).

[25] N. Een, A. Mishchenko, and R. Brayton. "Efficient Implementation of Property Directed Reachability." In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD '11. Austin, Texas: FMCAD Inc, 2011 (cit. on p. 37).

[26] N. Eén and N. Sörensson. "An Extensible SAT-solver." In: *Theory and Applications of Satisfiability Testing, 6th International Conference*. 2003, pp. 502–518 (cit. on p. 53).

[27] A. Biere. "Lingeling and Friends Entering the SAT Challenge 2012." In: *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*. Vol. B-2012-2. Department of Computer Science Series of Publications B, University of Helsinki. 2012, pp. 33–34 (cit. on p. 53).

[28] A. Biere. "Picosat essentials." In: *Journal on Satisfiability, Boolean Modeling and Computation (JSAT* (), p. 2008 (cit. on p. 53).

[29] F. Somenzi. *CUDD: BDD package, University of Colorado, Boulder*. `http://vlsi.colorado.edu/~fabio/CUDD/` (cit. on p. 53).

[30] T. van Dijk and J. van de Pol. "Sylvan: multi-core decision diagrams." In: *Tools and algorithms for the construction and analysis of systems*. Vol. 9035. Lecture notes in computer science. Apr. 2015, pp. 677–691 (cit. on p. 53).

[31] R. Brayton and A. Mishchenko. "ABC: An Academic Industrial-strength Verification Tool." In: *Proceedings of the 22Nd International Conference on Computer Aided Verification*. Conference on Computer Aided Verification '10. 2010, pp. 24–40 (cit. on p. 56).

[32] U. Krautz, M. Pflanz, C. Jacobi, H. W. Tast, K. Weber, and H. T. Vierhaus. "Evaluating Coverage of Error Detection Logic for Soft Errors using Formal Methods." In: *Proceedings of the Design Automation Test in Europe Conference*. Vol. 1. Mar. 2006, pp. 1–6 (cit. on p. 68).

[33] D. Holcomb, W. Li, and S. A. Seshia. "Design as you see FIT: System-level soft error analysis of sequential circuits." In: *2009 Design, Automation Test in Europe Conference Exhibition*. Apr. 2009, pp. 785–790 (cit. on p. 68).

[34] G. Fey, A. Sulflow, S. Frehse, and R. Drechsler. "Effective Robustness Analysis Using Bounded Model Checking Techniques." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011), pp. 1239–1252 (cit. on p. 68).

[35] S. Frehse, G. Fey, E. Arbel, K. Yorav, and R. Drechsler. "Complete and effective robustness checking by means of interpolation." In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. 2012, pp. 82–90 (cit. on p. 69).

[36] E. Arbel, S. Koyfman, P. Kudva, and S. Moran. "Automated Detection and Verification of Parity-protected Memory Elements." In: *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '14. San Jose, California, 2014, pp. 1–8 (cit. on p. 69).

[37] C. Zhao and S. Dey. "Improving Transient Error Tolerance of Digital VLSI Circuits Using RObustness COmpiler (ROCO)." In: *Proceedings of the 7th International Symposium on Quality Electronic Design*. ISQED '06. 2006, pp. 133–140 (cit. on p. 69).