Quality and Quantity in Robustness Checking Using Formal Techniques

Stefan Frehse

Quality and Quantity in Robustness Checking Using Formal Techniques

Stefan Frehse
Arbeitsgruppe Rechnerarchitektur
Fachbereich 3 - Mathematik und Informatik
Universität Bremen

DISSERTATION
Zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften
- Dr.-Ing. -

Kolloquium am 21. August 2013

Erstgutachter: Prof. Dr. Rolf Drechsler
Zweitgutachter: Prof. Dr. Alberto Garcia-Ortiz

Für Imke
und unsere echt krasse Herde

Danksagung

Mit der Dissertation beende ich meine Promotion im Fach Informatik. Für die Unterstützung während der Entstehung möchte ich mich bei einigen Menschen bedanken.

Für die intensive Betreuung meiner Promotion und die Begutachtung meiner Dissertation bedanke ich mich beim Leiter der *Arbeitsgruppe Rechnerarchitektur*: Rolf Drechsler. Er hat mich bereits im dritten Semester meines Informatikstudiums für aktuelle Forschungsfragen begeistert und eng in die Arbeitsgruppe einbezogen. Für die intensive Unterstützung während der Enstehung wissenschaftlicher Veröffentlichungen und das Herausarbeiten meines Dissertationsthemas möchte ich Görschwin Fey meinen Dank aussprechen.

Die gesamte Arbeitsgruppe Rechnerarchitektur bietet eine herausragend positive Arbeitsatmosphäre, dafür möchte ich mich sehr bedanken. Die spannenden und vielfältigen Diskussionen mit Finn Haedicke, Mathias Soeken und André Sülflow haben mich sehr inspiriert.

Ein herzliches Dankeschön geht an Jean Christoph Jung und Alexander Finder für die Durchsicht der Arbeit.

Ein großer Dank geht an meine Eltern, die mich in vielerlei Hinsicht intensiv unterstützen. Zu guter Letzt: Ich danke Imke Niemann für Ihre herausragende Motivation: *May the Force be with you!*

Bremen, Mai 2013, Stefan Frehse

Contents

Chapter 1

Introduction

In our daily life we are directly and indirectly using numerous computer systems. This number tends to grow in the future. Safety-critical computer systems assembled by several digital systems are integrated for example in cars, airplanes or are used in server systems. For example, during the last years *drive-by-wire* in cars came up to control the driver's command electronically rather than by a mechanical control systems. One advantage of such systems is that the input of the driver can be checked whether it keeps the car on track and correct the input if necessary. However, the dedicated computers behind those control systems are *special purpose processors* and are very complex while the application demands even for a very fast and correct processing.

Those system's complexity significantly increased over the last years since more and more features are implemented. A modern computer system contains several communicating *digital circuits* – or *chips*. These chips usually consist of millions of transistors. For example the IBM *Power7 Central Processing Unit* (CPU) introduced in 2011 consists of 1.2 billion transistors [SKS+11] while one of the first transistorized CPUs, the Intel 4004 released 1971, consists of only 2,300 transistors. This number, also referred to as *transistor count* doubles every 18 month according to Moore's Law. This leads to an exponential growth.

The continuously increasing integration density comes inherently with shrinking feature sizes since the size of the chips is kept approximately constant. Thus, the size of a single transistor is getting smaller and smaller. Besides the integration of more transistors the frequency is scaled up while the voltage is scaled down leading to a lower power consumption. That means, the chips run faster while consuming fewer energy.

However, due to these strong improvements one of the drawbacks are the circuits are less reliable [Bau05, Bor07, BBL+12, SKK+02]. More precisely,

the circuits are more sensitive to radiation. A *transient fault* occurs when enough energy affects the transistor's internal state. Thus, a transient fault may manipulate a logical value of an internal signal - the circuit is affected by a *soft error*. The logical value is flipped, i.e., inverted by 0 to 1 and vice versa which is also called as a *bit flip*. As one consequence the circuit may not operate as specified. Unlike physical manufacturing defects after fabrication a soft error is caused by external events that do not damage the chip permanently [Bau05].

The *Soft Error Rate* (SER) is measured in *Failure-In-Time* (FIT) defined by one failure in 10^9 hours caused by transient faults. A vast set of literature shows that the SER increases with continuously increasing integration density, e.g., [KK07]. That means, the higher the SER the higher the probability that a circuit causes a failure within that period of time.

In the past transient faults caused significant breakdowns [Bau02] since a misbehavior is life-threatening even in safety-critical systems. For example the *Therac-25* machine used for radiation therapy caused a critical overdose of six patient [LT93].

Handling transient faults appropriately has become one of the major challenges for future technology scaling [Bor07, BBL+12]. An important factor of improving the circuit's reliability is to tolerate transient faults by detecting and correcting the misbehavior. Reliability of a digital circuit is composed by several factors while fault tolerance is one of the most crucial parts. The reliability is usually measured in *Mean Time To Failure* (MTTF) and similar measures. MTTF specifies the average time that a system may fail. [KK07]

A digital circuit can be divided into *combinational logic* and *sequential logic*. In earlier technology generations, sequential logic was more sensitive for transient faults and was predominant analyzed in terms of transient faults rather than combinational logic. However, the probability that transient faults affect also combinational logic that may finally cause a failure increases with future technology generations. One reason is that scaling down the frequency decreases the time window that a transient fault can be stored in a memory element. This increases the probability that a faulty value is finally stored in the state elements since the state elements store more often in the same period of time. Consequently, a broader consideration of dealing with soft errors in combinational logic as well as sequential logic needs to be done which is addressed in this thesis.

In safety-critical systems the correct function must be ensured in every case. For example, an *airbag* must be fired in case of an accident and must not be fired during normal operation even under influences of external effects. Various standards have been published to formalize the requirements of

modern systems. An automotive related standard, ISO 26262 requires that a certain level of reliability needs to be assured to get finally certified.

In order to overcome these effects various *hardening techniques* are available to catch and handle these faults on hardware level. During the design process, the engineer usually adds redundancy to the hardware. Two important categories of those techniques taken at *design-level* are listed:

- *Error Correction Codes* (ECC): The classical Hamming code [Ham50] detects two errors and is able to correct one of them. A further ECC is the widely used *Reed-Solomon* (RS) [Ber68] code which is able to detect and correct errors as well. ECC is often implemented on top of storage elements. For example NVIDIA's recently manufactured *Fermi* GPU implements ECC on storage elements [NVI12]. Parts of the Power7 CPU are also ECC protected [SKS+11].

- *Redundancy*: Additional hardware is necessary in every case to tolerate faults [AK84]. A prominent example is the *Triple Modular Redundancy* (TMR) implementation [vN56]. The basic idea is to triplicate the functional unit and to add a majority voter to handle transient faults during operation.

  TMR is widely used in space applications since the SER increases with increasing altitude [GOSM08, Nic11]. The work of [Yeh96] describes a TMR-based implementation of the *Boeing 777* primary flight computer.

For various generally applicable techniques, tool support is already available. That means, those tools automatically implement these techniques. For example, TMRTool implements fully automatic TMR on an FPGA [Xil13].

Based on the widely applied techniques transient faults are caught to ensure that the circuit works as specified even in the presence of transient faults. However, the implementation itself might be faulty. Hence, the implementation needs to be verified. This process is called *robustness checking* and states the focus of this thesis. Robustness checking analyzes a circuit whether the implemented techniques tolerates transient faults appropriately.

In contrast, *Model Checking* (MC) usually verifies a circuit with respect to a specification without external effects [CGP01]. While MC verifies the unaffected circuit, robustness checking verifies the circuit even in the case of external effects, i.e., in the presence of transient faults.

Figure 1.1 shows the enhanced design flow embedding robustness checking. The design is passed to a model checker verifying whether the circuit

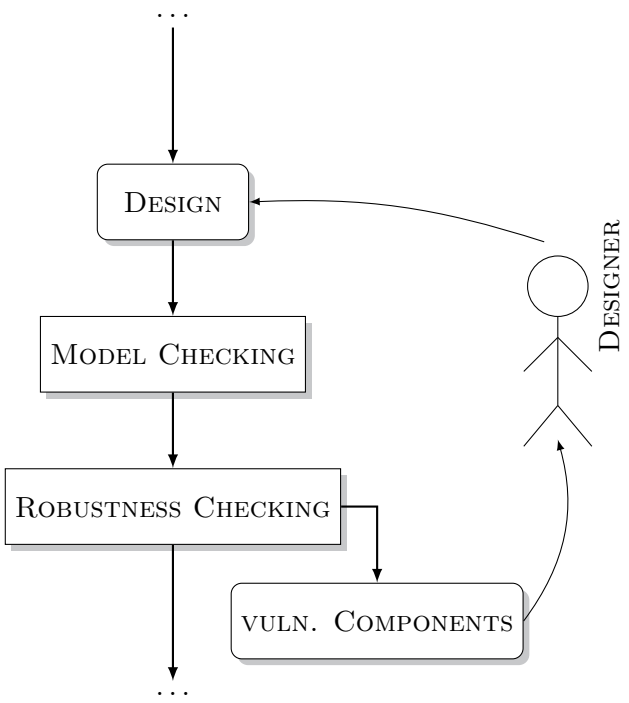


Figure 1.1: Robustness checking embedded into the design flow

implements the specification. Once the circuit is successfully verified, robustness checking is performed. The outcome of robustness checking is a set of components that are vulnerable against transient faults. A component is vulnerable if a transient fault at this component modifies the function of the circuit. Moreover, a measure of the quality of the circuit in the presence of transient faults is provided in term of a robustness measure.

If a desired level of robustness is assured the design is passed to further steps in the design flow. However, if a higher level of robustness is required the data about vulnerable components is provided to the designer who may implement additional techniques to improve the robustness. Afterwards the design is again passed to the model checker, i.e., the loop restarts.

This thesis focuses on analyzing a digital circuit with respect to transient faults, i.e., technique to perform robustness checking are proposed. The analysis is performed on logical level of a circuit while the circuit is given in a *Hardware Description Language* (HDL). Basically, two sets of components are delivered by robustness checking. The first set contains components that are not vulnerable against transient faults and the other set contains components that are vulnerable against transient faults.

In order to compute these sets the theoretical groundwork needs to be properly defined: 1) a fault model that describes the behavior and impact of transient faults, and 2) a computational model that calculates the impact of transient faults for each component are precisely introduced. Faults are logically injected, i.e., for each component that needs to be analyzed a dedicated *fault injection logic* is introduced into the circuit. Various algorithms are introduced which inject and analyze which behavior the respective faults cause.

The algorithms proposed in the thesis are divided into *formal-based methods* and *simulation*-based approaches. Formal methods completely analyze the entire search space and are therefore able to formally prove the absence of faulty behavior. In contrast, simulation covers only a limited portion of the search space and is usually not able to prove those properties since corner cases are missed. That means, simulation roughly approximates the solution but can handle larger circuits.

Bounded Model Checking (BMC) is a technique of functional verification that is able to disprove or prove a *temporal* property [BCCZ99] with respect to a circuit. A bounded time interval is analyzed by formulating a decision problem which is translated into a series of satisfiability problems solved by a *Boolean satisfiability solver*. BMC is proven to be PSPACE-complete [SC85] for the general class of temporal properties. Practically, BMC is very effective in finding bugs of a circuit by returning a trace that shows the particular misbehavior. Proving that a property holds on the circuit requires typically too much computational power using BMC. Further improvements have been developed by introducing, e.g., *Interpolation-based Model Checking* (IMC) [McM03] that makes BMC practically complete. IMC abstracts irrelevant facts while proving a property which was shown to be very effective even on industrial benchmarks.

Trivially, one can say that robustness checking can be easily translated into a model checking problem that can be solved by state-of-the-art model checkers. Of course, the modeling effort would be very low but this approach performs very poorly. During experiments it turned out that this translation into a model checking problem is outperformed by all formal approaches proposed in this thesis by a huge factor even for the smallest considered circuit. That means, exploiting problem domain knowledge leads to a much better performance and finally better quality of the result.

In contrast to BMC, *Automatic Test Pattern Generation* (ATPG) computes a set of test pattern that are simulated after *post-production* to filter out defective chips [DEFT09]. The generation of a test pattern is usually performed according to the *Stuck-At Fault Model* (SAFM). Basically, for each fault a test pattern is separately generated. In [Lar92] the ATPG

problem has been translated into a Boolean satisfiability problem which has been further improved in [DEFT09] called *SAT-based ATPG*. In industrial test flows SAT-based ATPG is very effective. ATPG is proven to be NP-complete [IS75] and theoretically nontrivial to solve. But typically the problem instances are solved very efficiently. ATPG is usually reduced to combinational circuits since *scan chains* are integrated in the circuits to arbitrarily justify values on the state elements. As a side-effect the complexity is reduced.

The algorithms of this thesis follow the methods of BMC, ATPG, and IMC to perform robustness checking. These original approaches are adapted for robustness checking. Moreover, in order to handle larger circuits a *divided-and-conquer* approach is introduced similar as in *Compositional Model Checking* [CLM89]. All these approaches are formal approaches providing exact results. However, in a powerful flow of robustness checking a random simulation is necessary to run fast pre-processing which is introduced in this thesis as well.

Overall, RobuCheck an integrated robustness checker is introduced that implements a highly-optimized flow. All approaches of this thesis are integrated and are freely configurable. These approaches can be called concurrently exploiting multi-core processor architectures or consecutively on single core processors. The huge search space of the underlying problem making robustness checking hard. Low-level optimization, pre-processing, and post-processing techniques are introduced to improve the overall performance.

In order to effectively assess the circuit's robustness three complexity issues are adequately addressed in this thesis: 1) all input scenarios, and 2) all transient faults need to be completely analyzed whether 3) all output sequences adhere the specification. These issues come inherently to compute a suitable set of *reachable states* to provide high quality results. This set of states directly influence the accuracy of the analysis which is deeply investigated and handled in RobuCheck.

All approaches introduced in this thesis are empirically evaluated on well-known academic circuits. Moreover, benchmarks coming from IBM show the effectiveness of RobuCheck on industrial circuits.

Related Work

To analyze the behavior of a circuit in the presence of transient faults various works have been published. Simulation and emulation [CMR+02, KPMH07, PCZ+08] based methods can handle large circuits but do not generally provide exact results.

A major difference of the approaches in the literature and the thesis's focused fault model is the level of modeling faults. The works [MZM06, MZM10, ZBD07] model radiation-induced transient faults on electrical level and all relevant masking effects. The size of those models is very large. Consequently, the approaches are only useful for small circuits. Moreover, multiple transient faults are considered in [MZM10] making the model even more complex.

[BBC+09] characterizes the state space after injecting faults based on certain properties. The works [BCG+10, BCT07] require more human interaction by providing properties manually. However, both approaches provide only a "yes"/"no" information of the respective system while [BCG+10] provides moreover a gradation of the considered properties as well.

The most similar works compared to the thesis are [HH08, HHC+09, KPJ+06] which analyze all components of a circuit where the work [HPB07, SLM07] focusing on state elements. All approaches analyze the impact of the sequential behavior of transient faults. Either the approaches perform symbolically a fixed-point characterization which is restricted to relatively small circuits or the approaches itself are restricted to a certain class of circuits [HH08, HHC+09].

The approaches in this thesis analyze any kind of digital circuit at logic level by considering the complete space of transient faults for each component. Transient faults are adequately modeled for a single but arbitrary time frame where the impact of transient faults are sequentially analyzed. The approaches in this thesis provide a detailed analysis for each component for any sequential circuit. The thesis focuses on formal techniques to analyze transient faults.

Research Work

The thesis is based on the prior work of [FD08, FSD09] that formulate the basis for formal robustness checking. However, several extensions and refinements in terms of performance, accuracy, and completeness have been published by the author of this thesis. The entire work would not be possible without the strong support of the coauthors of the respective papers. Thanks to all my coauthors.

In the following an overview of my research is briefly described.

This thesis focuses on the author's research work about formal robustness checking. Several scientific works were published at national and international conferences [SFFD09, FFD10, FFSD10, FF10, FHD+11, FRF12, FFA+12, RFF12] and journals [FSFD11, FSSFa10] based on peer-review.

The work of [FFD10] entitled *A better-than-worst-case robustness measure* received a *Best Paper Award* in the category testing. Moreover, the authors were invited to give a talk about their work at the *International Test Conference* (ITC).

Parts of the thesis were also presented at the PhD-Forum of the *Asia and South Pacific Design Automation Conference* (ASP-DAC) conference.

The work [FFA+12] was a joint work with the *IBM Haifa Research Lab* where the author of this thesis had an internship.

A preliminary version of the verification tool RobuCheck has been published and demonstrated at the University Booth of the *Design, Automation, and Test in Europe* (DATE) conference.

To provide an overview which paper is included in the respective chapters the following list shows the relations and the further structure of the thesis:

- Chapter 2: In this chapter the fundamentals are presented to keep the thesis self-contained.
- Chapter 3: The next chapter introduces the basic fault model. Transient faults are modeled and the impact of these faults on the circuit's behavior is introduced. This chapter contains parts of the works [FFD10, FSFD11, FFA+12].
- Chapter 4: After introducing the different kinds of behavior two robustness measures are introduced originally published in [FFD10, FSFD11].
- Chapter 5: Having the basic groundwork introduced a basic algorithm to compute the circuit's robustness is proposed. Moreover, the influence of approximate reachability information is analyzed. Parts of this chapter were published in [FSFD11, FHD+11].
- Chapter 6: Concrete engines are introduced in this chapter that follow the idea of the basic algorithm from the previous chapter. Mainly, formal-methods based approaches are introduced but also a simulation-based approach. Moreover, a simple model checker covering a new idea of approximating reachable states is proposed as well. At the end RobuCheck is presented that integrates all engines to determine the circuit's robustness. Parts of this chapter were published in [FFSD10, FSFD11, FHD+11, FFA+12].
- Chapter 7: All introduced algorithms are evaluated on several benchmarks. Beside mainly the evaluation of robustness checking results of approaches to efficiently compute interpolants and a comparison of the

introduced model checker against a state-of-the-art model checking are presented.

- Chapter 8: Finally, the thesis ends with a conclusion and a direction of the future work.

Moreover, numerous works are published in related areas that are listed below:

- The works of [SFWD12, WGF+11, ZFWD11, JFWD10, FWD10, WGF+09] are about *reversible logic* including *automated debugging* and *testing*. Moreover, an integrated development environment called RevKit for developing algorithms around reversible logic has been published as an open-source framework.
- The work of [HFF+11] integrates various satisfiability solvers into a domain specific language via C++ called metaSMT. Numerous *front-ends*, *middle-ends*, and *back-ends* are available to configure an optimal solver for a dedicated problem. metaSMT is published as an open-source framework that is already integrated in various tools, e.g, [HLGD12, RF12].

Chapter 2

Preliminaries

In this chapter the theoretical background is introduced to keep the thesis self-contained. For further details on the respective topic a reference is provided.

2.1 Directed Acyclic Graph

A common data structure in computer science is a graph consisting of nodes and edges formally introducted as follows:

Definition 2.1. *A* Directed Acyclic Graph *(DAG) is a tuple* $G = (V, E)$ *with a finite set of* nodes V *and a finite set of* directed edges $E \subseteq V \times V$. *If* $e = (v, v') \in E$, v *is called* source node *and* v' *is called* target node.

The function $\mathsf{in}(v) = \{e_{i_1}, \ldots, e_{i_o}\}$ *denotes the set of* incoming *edges, i.e., where* v *is the target node. The function* $\mathsf{out}(v) = \{e_{i_1}, \ldots, e_{i_p}\}$ *denotes the set of* outgoing *edges, i.e., where* v *is the source node.*

2.2 Boolean Logic

2.2.1 Boolean Functions

The set of Boolean values is given by $\mathbb{B} = \{0, 1\}$, where 0 is also denoted by FALSE or $\bot$, and 1 which is also denoted by TRUE or $\top$ [Weg87].

Definition 2.2. *An assignment* ϕ *is a mapping from a set of variables* X *to Boolean values, i.e.,* $\phi(x) \in \mathbb{B}$ *for all* $x \in X$.

Definition 2.3. *A Boolean function* f *is a mapping of the form* $f : \mathbb{B}^n \to \mathbb{B}$. *Further,* f *is often defined over a set of Boolean variables* $X = \{x_1, \ldots, x_n\}$ *and is also written as* $f(x_1, \ldots, x_n)$.

Boolean functions can be differently represented by, e.g., *truth tables*, *Boolean expressions*, *Binary Decision Diagrams*, and, *Conjunctive Normal Form*. A simple representation form is based on Boolean operations (connectivities) that forms a *Boolean expression*. Such operations are for example *negation* $(\neg, \bar{\cdot})$, *conjunction* $(\wedge, \cdot)$, *disjunction* $(\vee, +)$, *exclusive-or* $(\oplus, \hat{})$, and further. A more detailed explanation is given in [Weg87].

Example 2.1. *Given a set of Boolean variables* $X = \{x_1, x_2, x_3\}$ *with* $x_i \in \mathbb{B}$*, a Boolean function* f *is given by:* $f(x_1, x_2, x_3) = x_1 \cdot x_2 \vee \neg x_1 \cdot x_3$*. The function* f *evaluates to TRUE under the assignment* $x_1 = 1$*,* $x_2 = 1$*,* $x_3 = 0$*.*

2.2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are graph-based data structures to represent Boolean functions [Bry86]. For several years they exclusively represented the back-end for many tasks during the design process like verification or test generation since manipulation and comparison is very efficient for those systems once the BDD is built-up. However, they are still effectively used in many applications as also in this thesis. Recently, [XWMB12] shows that BDDs are still used in industrial-strength verification flows that successfully verifies systems while more modern approaches fail.

A BDD consists of edges and nodes forming a *Directed Acyclic Graph* (DAG). Each internal node of the BDD is commonly computed by the *Shannon Decomposition*. A *Reduced Ordered Binary Decision Diagram* (ROBDD) is a canonical representation of a Boolean function. Construction and manipulation of BDDs can be done very efficiently for a wide range of Boolean functions. The book of [DB98] provides an overview of the techniques and applications of BDDs.

2.2.3 Conjunctive Normal Form

Definition 2.4. *Let* $X = \{x_1, \cdots, x_n\}$ *be a set of Boolean variables then* $\mathsf{Lit}(X) = \{x, \overline{x} \mid x \in X\}$ *is called the set of* literals *of* X*.*

Definition 2.5. *A* clause *is a disjunction of literals, i.e.,* $F = l_1 \vee l_2 \vee \cdots \vee l_n$ *is a clause with* $l_i \in \mathsf{Lit}(X)$ *where* X *is a set of Boolean variables.*

A clause is also modeled as a set of literals since the structure of the formula is implicitly defined, i.e., the clause $F = l_1 \vee l_2 \vee \ldots \vee l_n$ is also written as $F = \{l_1, l_2, \ldots, l_n\}$. The size of a clause is given by the number of containing literals and is written as $|F|$. A clause that contains only one literal is called *unit clause*. The *empty clause* contains no literals, is logically equivalent to FALSE, and is denoted by $\square$.

Definition 2.6. *A* Conjunctive Normal Form *(CNF) is a conjunction of clauses. That means the CNF $M = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ is a Boolean formula consisting of clauses F_i with $1 \leq i \leq m$.*

A CNF is also modeled as a set of clauses, i.e., a CNF is also written as $M = \{F_1, F_2, \ldots, F_m\}$. The size of a CNF is given by the number of containing clauses written as $|M|$. The function $\mathsf{Var}(M)$ returns all variables occurring in the formula M.

Definition 2.7. *Given a CNF $M = \{F_1, \ldots, F_m\}$ defined over the variables $\mathsf{Var}(M) = \{x_1, \ldots, x_n\}$ and an* assignment *ϕ to the variables of M with $\phi(x_i) \in \mathbb{B}$. The assignment ϕ satisfies F_i, written $\phi \models F_i$, if at least one literal evaluates to TRUE. The CNF M is satisified by ϕ, written $\phi \models M$, if every clause F_i is satisfied by ϕ, i.e., $\forall F_i \in M.\phi \models F_i$. If there exists an assigmments satisfying M, then M is called* satisfiable*, otherwise M is called* unsatisfiable*.*

Definition 2.8. *Let $F = \{l_1, \ldots, l_n\}$ be a clause and M be a Boolean function, then $F_{|M}$ is a* stripped clause *by the variables of M, i.e, $F_{|M} = F \cap \mathsf{Lit}(\mathsf{Var}(M))$.*

Example 2.2. *Given a clause $F = \{x_1, \bar{x}_2, x_3\}$ and a Boolean function $M(x_1, x_2) = x_1 \cdot \bar{x_2}$, then the stripped clause by M is given by $F_{|M} = \{x_1, \bar{x}_2\}$. That means, the stripped clauses does not contain the variable x_3, because M is not defined over x_3.*

Definition 2.9. *Given an unsatisfiable CNF $M = \{F_1, \ldots, F_n\}$, then M' is called* unsatisfiable subformula *with $M' \subseteq M$, if M' is unsatisfiable. Further, M' is called* minimal *unsatisfiable subformula, if for all $F \in M$, the formula $M' \setminus \{F\}$ is satisfiable. The term* (minimal) unsatisfiable subformula *is also known as* (minimal) unsat-core *of a CNF formula.*

Definition 2.10. *A* minterm *is a conjunction of literals, i.e., $H = l_1 \wedge l_2 \wedge \ldots \wedge l_n$ is a minterm with $l_i \in \mathsf{Lit}(X)$ where X is a set of Boolean variables.*

Definition 2.11. *A* Disjunctive Normal Form *(DNF) is a disjunction of minterms. That means, $G = H_1 \vee H_2 \ldots \vee H_n$ is a DNF.*

2.2.4 Boolean Satisfiablity – The SAT Problem

Definition 2.12. *The* Boolean Satisfiablity Problem *(SAT Problem) is a decision problem that takes as input a Boolean expression $f : \mathbb{B}^n \mapsto \mathbb{B}$ and computes whether f is satisfiable computed by the function $\mathsf{SAT}?(f) \in \{$TRUE, FALSE$\}$. $\mathsf{SAT}?(f)$ returns* TRUE *if there is an assignment ϕ such that $\phi \models f$, i.e., the function f evaluates to TRUE under the assignment α, otherwise* FALSE *is returned.*

Boolean Satisfiability (also known as Satisfiability Problem or SAT problem) states the question whether a Boolean function is satisfiable. This problem is of great interest for a various of theoretical and practical reasons. The SAT problem was the first known NP-complete problem proved by Stephen A. Cook in [Coo71]. Therefore, from a theoretical point of view we do not expect a general solution to compute satisfiability efficiently under assumption that $\mathsf{P} \neq \mathsf{NP}$ [AB09]. However, not every probem instance requires exponential run time and due to the great advances in the area of SAT-solving, hard problem instances became solved very efficiently.

Various real-world problems from planning in *Artificial Intelligence* (AI) [KS92, Rin12] over *Automatic Test Pattern Generation* (ATPG) [DEFT09] and Model Checking (MC) [CGP01] of Boolean circuits to automated debugging of software [SVAV05] are translated into a SAT-problem and are effectively solved. Due to the great success, these SAT solvers are applied in various fields in computer science to solve difficult problems.

Before solving a suitable problem using a SAT solver, a decision problem needs to be formulated and then translated into a CNF. That means, the problem instance from the problem domain needs to be translated into a Boolean function and is then checked for satisfiability by a SAT solver. After deciding satisfiability, the result is translated back into the problem domain and in case that the formula is satisfiable the satisfying assignments are translated back if necessary.

Despite the high complexity of solving the SAT problem various algorithms came up to solve the problem efficiently. These algorithms forms the basis for today's modern SAT solvers whose insights are briefly introduced in the following.

SAT solver

A *SAT solver* usually gets a CNF formula as input and returns whether the CNF is satisfiable or unsatisfiable, i.e., a SAT solver implements the function SAT?. Additionally, almost all SAT solvers compute satisfying assignments if one exists. Due to the great achievements to boost the performance of the SAT solver in recent years, CNFs with several hundreds of thousand clauses and variables are routinely solved. Empirical studies about the craft behind SAT solvers are presented, e.g., in [KSMS11]. The first algorithm was the *Davis-Putnam* (DP) algorithm [DP60] which performs *resolution*, a simple inference rule to deduce the empty clause in case of an unsatisfiable CNF. The two years later proposed *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm improves the DP algorithm in terms of memory

consumption [DLL62] by replacing resolution with search and backtracking. The DPLL algorithm forms the basis for the most modern SAT solvers.

At the beginning of the 21^{st} century, SAT-solving became very attractive, because more elaborated techniques like *conflict analysis*, *2-watched literal scheme*, and, *non-chronological backtracking* have been proposed to solve practical problems, e.g., proposed with the SAT solver GRASP [MSS99], MiniSAT [ES03], and Chaff [MMZ+01b].

Today's annual SAT competitions[1] since 2002 show the increasing efficiency of new techniques. Further, it has been shown that fine-tuning the internal heuristics and engineering internal data structures of those solvers has a strong impact on the performance.

Due to the increasing number of multi-core systems, recently SAT-solving on different CPU cores became an active research area. A *portfolio solver* encapsulates different solvers and starts the solving process by starting all integrated solvers in separate threads or processes. The result of the fastest solver is returned and all solvers are terminated. Various SAT solvers behave differently on diverse CNFs due to their distinguished internal heuristics and parameters. Adjusting one parameter can help to solve one instance much faster while it slows down solving other instances. An advanced selection whether a specific solver is suitable to solve a CNF faster than another solver is very difficult and often comes inherently to solve the SAT problem itself. However, a balanced combination of various solvers may overcome this issues which is also recently addressed in these SAT competitions by introducing new tracks of parallel solving.

Resolution Proofs

Today's modern SAT solvers are *Conflict-Driven Clause-Learning* (CDCL) solvers. These solvers are inspired by modern DPLL-based algorithms including backtracking, and the generation of conflict clauses. However, they can be understood as proof systems performing resolution [PD11] introduced in this section.

Given a CNF formula M, the procedure *resolution* tries to produce a proof that M is a contraction, i.e., unsatisfiable. The proof is produced by the application of the following *RESolution inference rule* (RES) with $F_i \cup \{l\}$ and $F_j \cup \{\neg l\}$ are clauses with $F_i \cup \{l\}, F_j \cup \{\neg l\} \in M$:

$$\frac{F_i \cup \{l\} \quad F_j \cup \{\neg l\}}{F_i \cup F_j}\ \text{RES} \tag{2.1}$$

[1] Organized on `http://www.satcompetition.org`

where l is called *pivot variable*, $F_i \cup \{l\}$ and $F_j \cup \{\neg l\}$ are called *antecedents*, and $F_i \cup F_j$ is called *resolvent*. Let $\mathsf{Res}(F_i, F_j, l)$ be the function that computes the resolvent. Basically, the RES rule states that $F_i \cup \{l\} \wedge F_j \cup \{\neg l\}$ implies $F_i \cup F_j$. If a sequence of resolution rules has been applied which finally derives the empty clause, the CNF is unsatisfiable since a conflict occurs. Recall the empty clause is false.

Remark. In [Rob65], it has been shown that the resolution proof system with the single rule RES is sound and complete.

Example 2.3. *Given an unsatisfiable CNF M with*

$$M = \{\{x_1, x_2\}, \{x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{\neg x_1, \neg x_2\}\}.$$

A sequence of applying RES *to derive the empty clause is given by the following figure:*

$$\dfrac{\dfrac{\{x_1, x_2\} \quad \{x_1, \neg x_2\}}{\{x_1\}} \qquad \dfrac{\dfrac{\{x_1, x_2\} \quad \{\neg x_1, x_2\}}{\{x_2\}} \qquad \{\neg x_1, \neg x_2\}}{\{\neg x_1\}}}{\square}$$

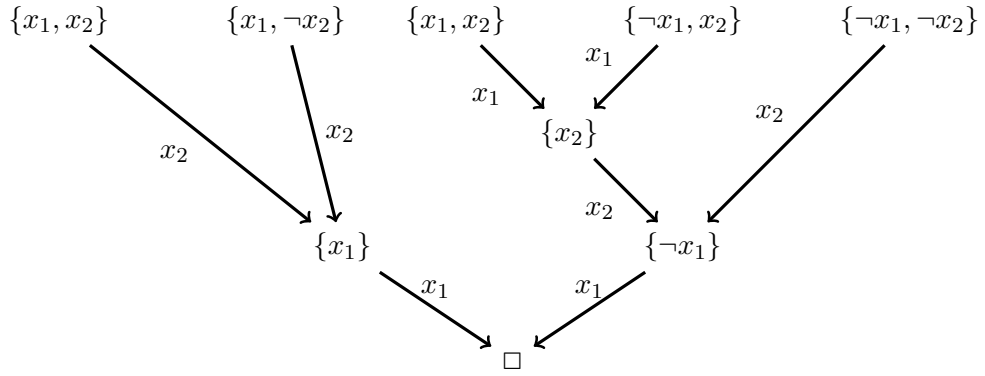
△

The sequence of applying RES to derive the empty clause is called *resolution proof*. In order to handle resolution proofs a data structure is introduced.

Definition 2.13 (Resolution Proof)**.** *A* resolution proof *is a* Directed Acyclic Graph *(DAG) $R = (V, E, \mathsf{piv}, \mathsf{cl}, c)$, where V is a finite set of nodes, E is a finite set of edges, piv specifies the pivot variable which is labeled at the edges, cl is the clause function computing a resolvent and $c \in V$ is called* sink *representing the empty clause and has no outgoing edges. Nodes with no incoming edges are* original *nodes from the input formula, all remaining nodes are* derived *nodes. For all derived nodes it holds: let $\alpha, \beta, \gamma \in V$ with $(\alpha, \gamma) \in E$ and $(\beta, \gamma) \in E$ and $\mathsf{cl}(\gamma) = \mathsf{Res}(\alpha, \beta, \mathsf{piv}(\gamma))$. The sink node represents the empty clause, i.e., $\mathsf{cl}(c) = \square$.*

Moreover, given a resolution proof an unsatisfiable core can be easily extracted. The original clauses reached backwards from the sink node form an unsatisfiable core since they are necessary to derive a conflict.

Example 2.4. *Reconsider Example 2.3, the following figure represents the resolution proof.*



Arrows mark edges between nodes (clauses) and variables on edges denote the pivot variables. △

In the following the number of nodes in the resolution proof is considered as the size of the proof necessary to derive the empty clause. Unfortunately, the size of a resolution proof can be very large with respect to the size of the input formula. For example, the proof size of *Pigeonhole Problems* in inherent, i.e., it requires always an exponentially sized resolution proof [AB09].

For other instances, the size significantly depends on the choice of the antecedents and pivot variables. Thus, the heuristics of the SAT solver influences at least structurally the proof and often also semantically. However, there are various techniques to shrink resolution proofs, e.g., [BIFH$^+$11].

A modern SAT solver can be instrumented to produce a resolution proof. For instance, the SAT solvers `PicoSAT` [Bie08] and `MiniSAT` [ES03] in its proof tracing versions produce resolution proofs. However, tracing the proof slows down the solve process by a considerable factor. Thus, proof tracing is only activated if necessary.

Moreover, there are several tools beside the SAT solvers to process resolution proofs. For example, `tracecheck`[2] verifies whether a resolution proof correctly deduces the empty clause based on the resolution inference rule. Those tools are useful when it has to be verified that a SAT solver correctly concludes that a CNF is unsatisfiable [ZM03].

Craig Interpolants

Craig interpolation has been introduced in [Cra57] and was later named by the author William Craig. An interpolant describes intuitively the relation

[2] Available at `http://fmv.jku.at/booleforce/`

between logical formulas and is formally defined for Boolean formulas as follows:

Definition 2.14. *Given a pair of Boolean formulas* (A, B) *such that* $A \wedge B$ *is unsatisfiable.* A Craig interpolant $\hat{I}$ *is a Boolean formula with the following properties: 1)* $A \implies \hat{I}$*, 2)* $\hat{I} \wedge B$ *is unsatisfiable, and 3)* $\mathsf{Var}(\hat{I}) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$*.*

Intuitively, $\hat{I}$ abstracts formula A, but contains sufficient constraints to contradict formula B.

Theorem 2.1. *Given a propositional formula pair* (A, B) *where* $A \wedge B$ *is unsatisfiable, then there exists always an interpolant [Cra57].*

An *interpolation system* is a procedure to obtain interpolants. This system is called *proof-based* when it is based on a resolution proof and it is called *model-based* when it is based on model enumeration.

Proof-based System Various proof-based interpolation systems have been proposed based on resolution proofs. But basically, they can be divided into two groups: a *symmetric system* independently developed in [Hua95, Kra97, Pud97], and *McMillan's interpolation system* (MIS) [McM03]. Further, [DKPW10, Wei12] compare the strengths of both systems and provide a parametric interpolation systems to generate different interpolants from the same proof. An essay about the influences of the performance of various interpolation systems is until now publicly not available to the best of the author's knowledge.

Interpolants are practically exploited in *SAT-based Model Checking* [McM03] which significantly improved the field of formal hardware verification in terms of handling industrial-sized circuits. In McMillan's work on interpolation-based model checking he introduced MIS which is briefly revisited in the following. MIS can be understand as a method which annotates each node of a resolution proof based on incoming edges that finally produces an interpolant.

Definition 2.15 (McMillan's Interpolation System – MIS)**.** *Given a resolution proof* $R = (V, E, \mathsf{piv}, \mathsf{cl}, c)$ *for the formula pair* (A, B) *with* $A \wedge B$ *is unsatisfiable. Each node* $v \in V$ *is annotated with the* annotate*-function as follows:*

- *if* $\mathsf{in}(v) = 0$*:*
  - *if* $\mathsf{cl}(v) \in A$ *then* $\mathsf{annotate}(v) = \mathsf{cl}(v)_{|B}$

  - *if* $\mathsf{cl}(v) \in B$, *then* $\mathsf{annotate}(v) = \top$

- *else: let* $(v_1, v) \in E$ *and* $(v_2, v) \in E$
  - *if* $\mathsf{piv}(v) \in \mathsf{Var}(B)$, *then* $\mathsf{annotate}(v) = \mathsf{annotate}(v_1) \wedge \mathsf{annotate}(v_2)$
  - *if* $\mathsf{piv}(v) \notin \mathsf{Var}(B)$, *then* $\mathsf{annotate}(v) = \mathsf{annotate}(v_1) \vee \mathsf{annotate}(v_2)$

Finally, $\mathsf{annotate}(c)$ *represents the interpolant where* c *is the sink node.*

The computation of an interpolant can be done in linear time and space with respect to the size of the resolution proof. However, the proof itself can be exponentially larger than the size of the input formula $A \wedge B$ according to the resolution proof's size.

Model-based System The work of [CIM12] propose an approach to compute interpolants based on model enumeration. Given the formula pair (A, B) the approach basically 1) enumerates all assignments satisfiying A, 2) minimizes these assignments that still contradict B, and, 3) creates a DNF using these assignments. Finally, the DNF represents the interpolant.

Algorithm 1:
Model-based interpolation systems

- **Input:** The input of the approach is the formula pair (A, B) given in CNF.
- **Output:** The output is either the interpolant $\hat{I}$ of (A, B) in DNF or a satisfying assignment in case that $A \wedge B$ is satisfiable.
- **Description:** The code is listed in Pseudocode 1. Basically, the approach enumerates models of A, i.e., $\phi \models A$ as long as no more assignments exists. At the beginning interpolant $\hat{I}$ is FALSE (Line 2) and the approach starts all possible models, i.e., $A' = A$ (Line 3). The `while`-loop iterates as long as no more assignments exist, i.e., A' is unsatisfiable (Line 5) - the determined interpolant $\hat{I}$ is returned - or there is an assignment that satisfies B, i.e., $A \wedge B$ is satisfiablity - TRUE is returned in Line 11.

  Suppose there is an assignment $\phi \models A'$ (Line 7) represented as minterm. The first steps is to remove all variables of ϕ that occur only in A, i.e., ϕ' is defined over common variables of A and B. In

```
1  begin
2      Î = FALSE;
3      A' = A;
4      while TRUE do
5          if !SAT?(A') then
6              return Î;
7          φ = model(A');
8          φ' = projection(φ, Var(A) ∩ Var(B));
9          φ'' = cubeEnlargement(φ', A');
10         if SAT?(B ∧ φ'') then
11             return SAT + φ'';
12         φ''' = cubeEnlargement(φ'', B);
13         Î = Î ∨ φ''';
14         A' = A' ∧ ¬φ''';
15     end
16 end
```

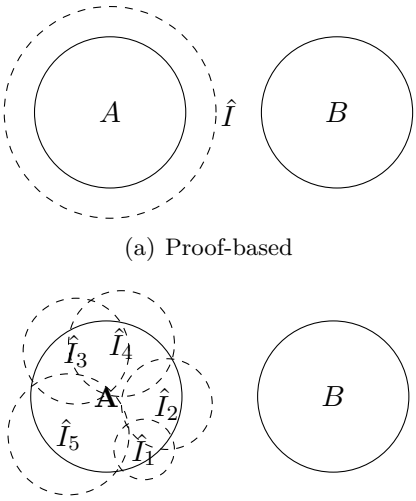
Pseudocode 1: Model-based interpolation systems.

Line 9 the minterm ϕ is minimized by the function `cubeEnlargement`. The implementation of that function is transparent to the algorithm. Basically, the function tries to minimize an assignment that still holds for an interpolant. Concrete realizations of this function are provided in Section 6.7.2.

Finally, a new minimized minterm ϕ'' is returned. If this minterm satisfies B then $A \wedge B$ is satisfiable.

The minterm p''' is added to the interpolant $\hat{I}$ where the negation of the minterm is added to A' to avoid to recomputed the same assignment.

Brief Comparison The proof-based systems are based on resolution proofs whereas the model-based systems are based on model enumeration. However, since the resolution proof might be very large the memory consumption may require gigabytes of memory. In contrast, the model-based system does not require the resolution proof but enumerates a huge number of assignments. In Figure 2.1 the main difference is illustrated. While the



(a) Proof-based

(b) Model-based

Figure 2.1: Types of Interpolation Systems

proof-based approach computes the interpolant at once along the resolution proof, the model-based approach iteratively construct interpolants by computing minterms.

Both systems are implemented and run concurrently to get the best of both. Details about the implementation are presented later in this thesis in Section 6.7.2.

In the following, the function $\mathsf{itp}(A, B)$ denotes the interpolant of the formula pair (A, B) independent on the respective realizations.

2.3 Circuits and Automata

Digital circuits can be represented in different ways. In this thesis circuits are assumed to be given as a graph-based representation introduced more formally in the next section. In formal verification an automata-based presentation is commonly used which is introduced in the following. Each graph-based representation of a circuit can be easily translated into a automata-based representation. Thus, if necessary a automata-based presentation is used.

2.3.1 Digital Circuits

Definition 2.16 (Circuit)**.** *A* sequential circuit $\mathcal{C} = (V, E)$ *is a* Directed Acyclic Graph *(DAG) consisting of a finite set of nodes* $V = \{v_1, \ldots, v_n\}$,

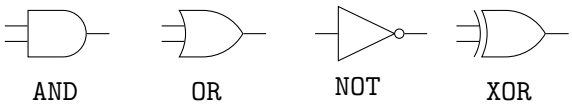


Figure 2.2: Typical gates

and a finite set of edges $E = \{e_1, \ldots, e_m\}$. *Nodes correspond to components and edges correspond to signal connections between the compnents. The terms node and component are used interchangeably. The size of a circuit is given by the number of components, i.e.,* $|V|$ *also written as* $|\mathcal{C}|$.

A component can be:

- *a primary input* PI*, a primary output* PO*, or a state element* FF*,*
- *a* primitive *commonly used gate exemplary shown in Figure 2.2,*
- *a more complex module for example a multiplier* MULT*, or an* Arithmetic Logic Unit *(ALU). A complex module might also relate to a statement of a* Hardware Description Language *such as VHDL [Ash01] or Verilog [TM96].*

All available types of components are summarized in the set LIB.

Additional information is associated to nodes and edges expressed by mappings of the following form:

- $\mathsf{type} : V \rightarrow \mathtt{LIB}$ *maps each node to a type,*
- $\mathsf{ord} : E \rightarrow \mathbb{N}$ *maps each edge to a natural number*
- $\mathsf{bv} : V \cup E \rightarrow \mathbb{N}$ *maps a bit-width in terms of a natural number to an edge or node, also written as* $|g|$ *for* $g \in V$*. Usually, the outgoing node of an primitive gate has a bit-width of* 1*. More complex modules may have a higher bit-width.*

The set $X \subseteq V$ *denotes the set of primary inputs* PI*, i.e., components with no incoming edges. The set* $Y \subseteq V$ *the set of primary outputs* PO*, i.e., components with no outgoing edges. The set* $S \subseteq V$ *contains state elements* FF*. A sequential circuit consists of state elements where the state elements may have different values in different times controlled by clock denoted by* time frame.

In order to address the values at a certain time frame the following notation is used; An additional argument of the sets denotes a certain time frame, .e.g, $X(t)$ *adresses the primary inputs at time frame* t*. Ananlogously this notation is used for primary outputs, and state elements.*

A bit-level *circuit exclusively consists of primitive gates with single-bit outputs.*

Definition 2.17. *A* combinational *circuit is a digital circuit without state element i.e.,* $S = \emptyset$.

2.3.2 And-Inverter-Graphs – AIGs

And-Inverter-Graphs (AIGs) [Hel63, DJBT81] are a subset of sequential circuits which consider a certain set of gates. An AIG consists only of `AND`-gates, state elements (`FF`), and, special annotations to signals whether the value is inverted or not. AIGs are very simple to handle and can be compactly represented using dedicated data structures. Each circuit can be efficiently translated into an AIG, i.e., converting a circuit can be done in linear time and space with respect to the number of components of the original circuit [Tse68].

AIGs are commonly used in formal hardware verification tools. Various optimization techniques are available to reduce the size of AIGs, e.g., [ZKKSV06, EMS07]. Even the translation of an AIG into a CNF can be done very efficiently which is typically required to reason about a circuit. Before translation, the AIG is unrolled for a desired number of time frames and afterwards converted into CNF.

Various software packages are available to represent and manipulate AIGs: AIGER [Bie12] and the ABC verification tool [Gro12].

2.3.3 Finite State Machine

Besides the DAG-based representation of sequential circuits the functional behavior can be represented by a *Finite State Machine* (FSM) as well. This kind of representation is often used in formal verification since problems related to reachability analysis are performed which are commonly performed on automatas. Those FSMs can be derived from sequential circuits considering fan-in and fan-out cone of state elements that correspond to transition functions. Intuitively, an FSM reflects the behavior over the time frames in terms of state-to-state relations based on the stimuli of the inputs and produces output values for each transition.

Essentially, reachability information can be computed on this data structure in order to conclude that a circuit may reach unwanted behavior in terms of a *bad state*. However, in the following an FSM is formally introduced.

Definition 2.18. *A* Finite State Machine *(FSM) of a sequential circuit is a tuple* $M = (I, S, T)$. *The set* $I \subseteq S$ *describes the set of* initial states.

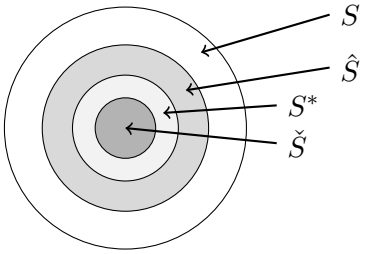


Figure 2.3: Relation of sets of states

The state space S is given by $S = \mathbb{B}^n$ where n is the number of state bits of the circuit $\mathcal{C}$. The transition relation $T(s, s')$ is TRUE if there is transition from state s to state s'. [CGP01]

Operators to compute reachability information are defined as well.

Definition 2.19. *The set* $\mathsf{img}(Q) = \{s' \in S \mid \exists s \in Q \wedge T(s, s')\}$ *computes all successor states reachable in one step from states of set $Q \subseteq S$ based on the transition relation T. A* path *is a sequence of states $s_0 \to ... \to s_n$ such that for $0 \leq i < n$ the transition function $T(s_i, s_{i+1})$ is TRUE. The* length *of a path is the number of transitions from the start state to the end state. Let $\mathsf{img}^0(Q) = Q$ and $\mathsf{img}^{i+1}(Q) = \mathsf{img}(\mathsf{img}^i(Q))$. All states reachable from the initial state I in any number of steps are given by $S^* = \bigcup_{i \geq 0} \mathsf{img}^i(I)$. States in the set S^* are called* reachable states *and* img *is named as* exact image operator. *Let* $\widehat{\mathsf{img}}$ *be an over-approximate operator with the following properties:* $\mathsf{img}(Q) \subseteq \widehat{\mathsf{img}}(Q)$ *and therefore it holds $S^* \subseteq \hat{S}$ with $\hat{S} = \bigcup_{i \geq 0} \widehat{\mathsf{img}}^i(I)$.*

The set $\hat{S}$ contains at least all reachable states, i.e., $\forall s \in S^* \implies s \in \hat{S}$ and might contain also states that are not reachable from the initial state, i.e., $\exists s \in \hat{S} \implies s \notin S^*$. The sets of states are also often described by a Boolean formula whose models corresponds to states. Those formulas can be often represented very compactly. In the following sets of states and Boolean formulas of states are used interchangeably. A detailed relation of the sets of states is visually characterized in Figure 2.3.

An approximation can be described manually by a designer or can be generated by various available tools, e.g.,based on BDDs[3]. A trivial over-approximation is the complete state space, i.e., S, and a trivial under-approximation is given by $\check{S}(l) = \mathsf{img}^l(I)$, i.e., all states that are reachable in l steps from the initial state.

[3] Available under `http://vlsi.colorado.edu/~fabio/CUDD/`

Definition 2.20. *The* diameter *of an FSM M denoted by* $\mathsf{dia}(M)$ *is the length of the longest path in the set of shortest paths between pairs of states in S^*. The* reachability diameter *of M denoted by* $\mathsf{rd}(M)$ *is the length of the longest path that starts from the initial state.*

Definition 2.21. *A* scenario *is a tuple $\tau = ((X_0, \ldots, X_n), S_0)$ where X_i $(0 \leq i \leq n)$ is an assignment to the primary inputs at time frames i and S_0 is an initial assignment to the state elements for the first time frame.*

2.4 Functional Verification

Functional verification is an important step during the design process that checks whether the *Circuit Under Verification* (CUV) fulfills the desired specification.

Model Checking (MC) [CGP01] is a major part of functional verification that formally verifies whether a circuit completely fulfills the specification. The specification is given as a set of temporal properties such as *Computation Tree Logic* (CTL), or *Linear Temporal Logic* (LTL) [Eme95, Pnu77]. The circuits are usually provided as a FSM. Various algorithms came up to perform model checking on FSMs, e.g., *Symbolic Model Checking* [CMCHG96] that performs a fixed-point computation that checks whether the circuit's states intersect states not uncovered by the specification.

In contrast to MC, *Equivalence Check* is a further technique to ensure correctness during the design process. In EC, a golden model (specification) is checked against an implementation which is performed for each level of the design process.

In order to verify that a CUV completely fulfills a desired specification all possible computations scenarios need to be checked against the specification. Formally verifying a design comes inherently with considering all possible scenarios to provide a complete analysis.

Techniques behind the formal verification mainly rely on powerful reasoning engines as for example BDDs, or SAT solvers. Since those engines fully analyze the entire search space a complete answer is provided that empowers the design process. If a CUV violates the specification, i.e., the CUV does not behave as the desired, techniques based on formal verification will uncover this misbehavior.

The field of formal hardware verification is an intensive research area. Various verification techniques are successfully applied in complex hardware verification in industry [Kai11]. The work of [KGN+09] describes how a complex processor design consisting of several million components has been formally verified.

In contrast to formal verification, simulation-based functional verification only partially covers all possible scenarios and therefore misses corner cases. Simulation or *constrained-random* simulation is still commonly applied in functional verification to close the manufacturing and verification gap. Since several years the trend is that the ability to manufacture a design is significantly higher than the ability to formally verify a design.

2.4.1 Bounded Model Checking (BMC)

Bounded Model Checking (BMC) [BCCZ99] is a model checking technique based on Boolean satisfiability to show or to refute that a design fulfills certain properties. Due to its great success large processor designs have been formally verified [VLP+05]. Therefore formal verification became more and more accepted within the design groups despite its high computational complexity [CKOS04].

BMC is able to prove or disprove a temporal property provided in LTL with respect to a design. In particular, if a bug has been found, a trace or stimuli called a *counterexample* is provided in order to understand the misbehavior of the design by simulating the trace. Basically, the technique behind BMC is to iteratively check whether a property holds during a certain number of time frames. By increasing this number of time frames eventually 1) a counterexample is found which violates the property, or 2) sufficiently many time frames are considered and therefore the design fulfills the property. Consequently, if there is behavior of the circuit that does not adhere the property BMC will find this bug. Otherwise, i.e., the circuit does completely fulfills the property BMC provides a proof. Since BMC relies on formal reasoning engines such as SAT solvers a complete analysis is guaranteed.

A more detailed explanation of BMC is revisited in the following since parts of the thesis are related to BMC. More insights are presented in the original work [BCCZ99].

Problem Formulation

BMC comes inherently with translating a certain number of time frames and a negated property into a CNF. The CNF is then checked for satisfiablity. This step is described below:

Given an FSM $M = (I, T, S)$ of a sequential circuit and a predicate P describing the property (specification). The following formula forms the basis for all BMC instances that are checked by a SAT solver:

$$\mathsf{BMC}(l) = I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1}) \wedge P(s_l) \tag{2.2}$$

If $\mathsf{BMC}(l)$ is satisfiable for any l then there exists a path $s_0 \to s_1 \to \ldots \to s_l$ with $s_0 \in I(s_0)$, $(s_i, s_{i+1}) \in T$ for $0 \leq i < l$ and $s_l \in P(s_l)$. That means, there is a path in the system that matches undesired behavior. Consequently, the design does not fulfill the property. In contrast, if the formula is unsatisfiable the design fulfills the property for a path of length l.

The BMC instances are iteratively checked starting from $l = 0$ up to the *completeness threshold* (CT) denoted by l_{cmpl}. Once a BMC instance becomes satisfiable for any $l \in [0, l_{\mathsf{cmpl}}]$ the property is violated and a counterexample is extracted from the satisfying assignments. Otherwise, the BMC instance is unsatisfiable for all those values of l the property holds on the circuit.

Large upper bounds of the completeness thresholds can be determined which is itself a nontrivial task [CKOS04]. Computing exact values in reasonable run times for general LTL properties constitutes still an open problem. Linear bounds have been proven for a subset of LTL properties [BCCZ99]. Various methods have been proposed in order to provide a complete verification without computing the exact completeness threshold. Due to these strong improvements [McM03, GS05, CLM89] LTL model checking is very effective despite its high complexity since it is shown to be PSPACE-complete [SC85].

2.4.2 Interpolation-Based Model Checking

BMC is sound and complete [BCCZ99] once the completeness threshold is known. In industrial verification flows it turned out that BMC is mostly used as a bug finder by computing a counterexample. Proving that a circuit fulfills a property BMC needs to reach the *completeness threshold* which is in general unknown making BMC impractically as discussed above. Moreover, the completeness threshold cannot be practically reached for moderate-sized circuits since the complexity increases exponentially which results in huge search spaces.

Craig interpolants [Cra57] are exploited in the work of McMillan [McM03] to provide a completeness argument without knowing the completeness threshold. Interpolants are used in order to over-approximate the image of the transition relation. Those interpolants abstract facts that are irrelevant to prove the property making BMC with interpolation very effective. Once a fixed-point of the over-approximate image has been found the circuit is

proven to be correct with respect to the property since all necessary states have been discovered. This approach is called *Interpolation-based Model Checking* (IMC) in the following.

Problem Formulation

Reconsider the BMC Formula 2.2 partitioned into two parts (A, B):

$$A := I(s_0) \wedge T(s_0, s_1) \quad \text{and} \quad B := \bigwedge_{1 \leq i < l} T(s_i, s_{i+1}) \wedge P(s_l)$$

Formula A contains the initial predicate and the first transition relation while the formula B contains the remaining transitions and the predicate of the property. The common variables of A and B are the state variables corresponding to s_1.

Suppose the formula $A \wedge B$ is unsatisfiable for a certain l. That means, there is no path of length l that violates the property. An interpolant $\hat{I} = \mathsf{itp}(A, B)$ can be computed. The interpolant $\hat{I}$ over-approximates the image of the first transition relation since $A \implies \hat{I}$ and $\mathsf{Var}(\hat{I}) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$. Since $B \wedge \hat{I}$ is unsatisfiable based on the construction of $\hat{I}$ that means no path of length $l-1$ from an over-approximate state of $\hat{I}$ can reach a state of the property. This construction is used to compute an over-approximation of the exact set of reachable states by iteratively computing interpolants until a fixed-point is reached.

The interpolant $\hat{I}$ is added to the set A by replacing the variable s_1 through s_0 with:

$$A := (I(s_0) \vee \hat{I}(s_0)) \wedge T(s_0, s_1)$$

The formula $A \wedge B$ is again checked for satisfiablity and two cases may occur:

- If the formula is unsatisfiable a new interpolant is computed and added to A. Eventually, the disjunction of all previously computed interpolants implies the new interpolant - a fixed-point is reached and it is proven that the circuit fulfills the property since no state of the over-approximate state space reaches a state of the property.
- In contrast, if the formula is satisfiable a probably spurious path has been found since the first state may start from an non-reachable state over-approximated by the interpolant. However, in that case l is increased, all interpolants are discarded and the entire procedure restarts. The consequence of increasing l is that the new interpolants

```
1  begin
2      if I(s_0) ∧ P(s_0) then return TRUE;
3      l = 1;
4      while TRUE do
5          φ = I(s_0);
6          while TRUE do
7              A := φ ∧ T(s_0, s_1);
8              B := ⋀_{1≤i<l} T(s_i, s_{i+1}) ∧ P(s_l);
9              if SAT?(A ∧ B) then
10                 if φ = I(s_0) then
11                     return TRUE
12                 l = l + 1;
13                 break;
14             end
15             Î = itp(A, B);
16             if Î ⟹ φ then return FALSE;
17             φ = φ ∨ Î(s_0);
18         end
19     end
20 end
```

Pseudocode 2: Interpolation-based model checking.

may abstract fewer facts leading to a non-spurious counterexample or a safe path since formula B is stronger.

The entire procedure is shown in the following algorithm.

Algorithm 2:
Interpolation-based model checking

- **Input:** An FSM $M = (I, T, S)$ and a property to verify P are given as input.
- **Output:** The algorithm returns TRUE if a counterexample has been found and FALSE if the circuit fulfills the property.
- **Description:** The algorithm, shown in Pseudocode 2 runs iteratively until either a fixed-point is reached or a counterexample is

found. In Line 2 it is checked whether the initial state violates the property. If the initial state already violates the property, TRUE is returned. Otherwise, the interpolation loop starts with checking paths of length 1 (Line 3). The outer `while`-loop from Line 4 to Line 19 runs as long as no counterexample and no fixed-point is found. In each iteration the algorithm checks whether a path that starts from the initial state can reach a property state in l steps where the initial states are specified in Line 5. This check corresponds always to the base case for a real, non-spurious counterexample. The inner `while`-loop from Line 6 to Line 18 is the interpolation loop. At first the partitions A and B are created. If the formula is satisfiable it is further checked whether the base case is considered. In this case a real counterexample has been found and TRUE is returned. Otherwise, a probably spurious counterexample has been found. Thus, l is increased and the outer loop completely restarts (Line 12 and 13). In contrast, if $A \wedge B$ is unsatisfiable an interpolant $\hat{I}$ (Line 15) is computed. In Line 16 it is checked whether a fixed-point is found, i.e., it is checked whether the disjunction of all previously computed interpolants (ϕ) is implied by the new interpolant. In the case, the circuit fulfills the property and FALSE is returned. Otherwise, if no fixed-point is found the new interpolant is added to ϕ and the inner loop restarts from Line 6. Eventually, a fixed-point is found, assuming that the circuit fulfills the property, the disjunction ϕ of all computed interpolants computes an over-approximation of the reachable states.

IMC is proven to be sound and complete in [McM03] and is further improved in, e.g., [DPK08, VG09]. Furthermore, the algorithm always terminates since either a counterexample is found or the completeness threshold is reached. Practically, It has been shown to be very effective on industrial benchmarks since it abstracts irrelevant facts for the proof of a certain property making the problem instances manageable.

2.4.3 Boolean Reasoning of Digital Circuits

In order to formally reason about digital circuits using a SAT solver a CNF has to be generated which is basically required for, e.g. BMC and IMC. Converting a digital circuit into a CNF can be done by, e.g., *Tseitin* encoding [Tse68] or *Plaisted-Greenbaum* encoding [PG86]. A digital circuit is translated into a CNF in linear time and space with respect to the size

of the circuit for both techniques. The proposed algorithms in this thesis exclusively use the common Tseitin encoding. A more detailed discussion and comparison of the mentioned techniques are presented in the elaborated work [JBH12].

2.5 Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) generates a set of input stimuli of a circuit according to a fault model to test the manufactured circuit also named as *post-production test*. This test ensures that the chip is correctly fabricated with respect to the underlying fault model. A defective chip will not be delivered to a customer. That means, each fabricated chip will be tested by the generated test patterns. Therefore, ATPG comes with special requirements. At one hand as much as necessary test patterns should be generated to keep the time of testing low and on the other hand the coverage of the test patterns should be as high as possible to detect most of the buggy chips.

ATPG has been proven to be NP-complete [IS75] and founds application in verification as well, e.g., [BRTF99, AV02].

Practically, ATPG is often performed on combinational circuits instead on sequential circuit to keep the complexity low. A sequential circuit is converted into a combinational circuit before by replacing the state elements by *pseudo* primary inputs and *pseudo* primary outputs. To test a sequential circuit based on combinational test pattern, *scan chains* are inserted into the circuit to justify arbitrary value at the state elements [WA73].

The test pattern are generated based on the commonly used fault model: *Stuck-At Fault Model* (SAFM) [BF76]. In SAFM a signal a constantly set to a Boolean value. SAFM covers various physical effects and it turned out that this fault model is relatively simple but practically very effective. There are two kinds of *stuck-at* faults denoted by $g_1@sa0$ and $g_1@sa1$, i.e., signal g_1 is constantly set to value 0 and 1, respectively. For each fault, ATPG generates a test pattern that is applied on the fabricated circuit to detect possibly inserted faults during the manufacturing process.

To test all signals of a circuit against the SAFM, a *fault list* is created. This list contains for each signal of the circuit two stuck-at faults, i.e, $\mathcal{F} = \{g_1@sa0, g_1@sa1, \ldots, g_n@sa0, g_n@sa1\}$ where n is the number of signals of the circuit.

For each item of the fault list, a test pattern is generated using ATPG. The complexity of ATPG is high since for each entry an NP-complete problem needs to be solved.

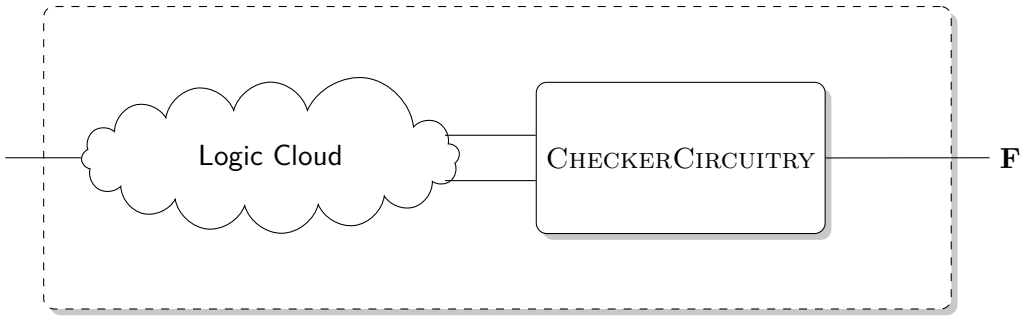


Figure 2.4: A schematic view of a circuit with a checker circuitry

.

Optimization techniques were proposed to reduce the size of the fault list and therefore the number of ATPG calls. Logical implications and equivalences are applied. Moreover, *fault simulation* tests whether further faults can be detected after obtaining a test pattern [JG03] which reduces the overall run time of ATPG.

Several ATPG algorithms have been proposed. A powerful ATPG engine based on Boolean satisfiability (SAT-ATPG) has been published in [Lar92] and was strongly further improved in [DEFT09, CPL+09, ED11]. Basically, a SAT problem is formulated for each item of the fault list. That means, a CNF of a decision problem whether a test pattern exists, is created, solved by a SAT-solver, and the result is interpreted into the ATPG domain.

2.6 Fault Tolerance Circuits

The thesis deals with analzing fault tolerant circuits. The technique that is implemented as hardening technique is transparent to the approaches. But examples of those techniques are provided in this section. Two kinds of hardening techniques are revisited in the following section. These techniques are also used in the experiments of this thesis.

2.6.1 Checker Circuitry

A checker circuitry is a part of a circuit that checks whether the computation is correct. This circuitry might be a parity checker, a hamming code, or a similar technique. In Figure 2.4 a schematic view of a circuit containing a checker circuitry is shown.

As a result the checker circuitry reports a detected fault via a fault signal denoted by **F**. The system that integrates the circuit can take some action once a fault is reported. For example, a reset sequence can be applied

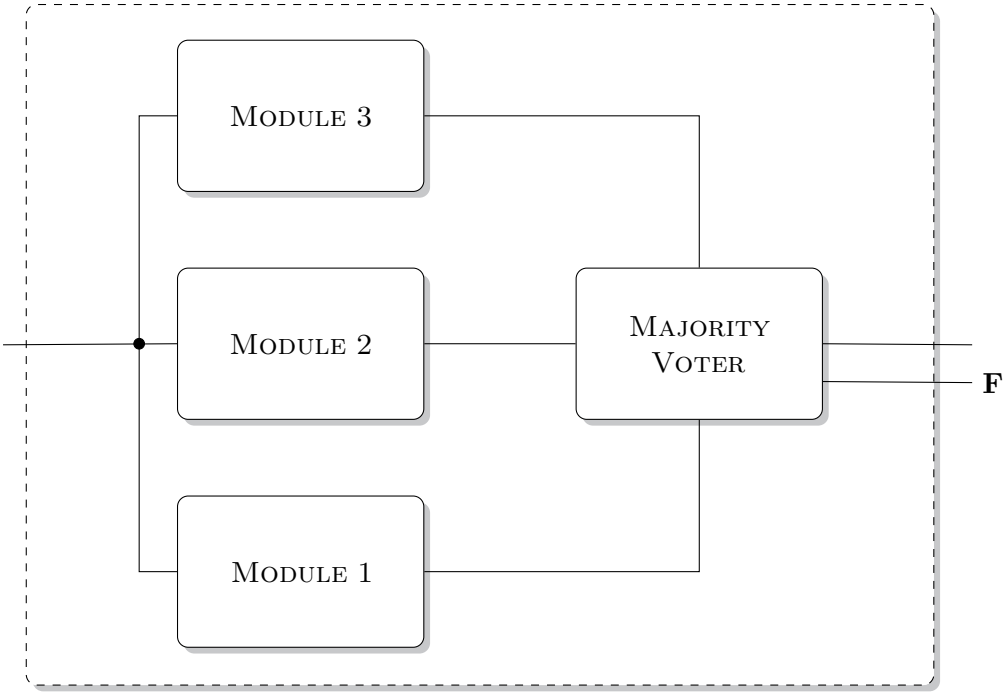


Figure 2.5: A schematic view of a system-level TMR implementation

to get the circuit into a consistent state. But those mechanisms are not further investigated in this thesis.

2.6.2 Triple Modular Redundancy - TMR

A *Triple Module Redundany* implementation is a commonly applied technique. The idea is to triplicate the circuit and to add a majority voter. A schematic overview of a *system-level* implementation is shown in Figure 2.5. The voter logic ensures that only correct values are propagated to the outputs assuming single faults.

Once the implementation is correctly implemented TMR ensures that single transient faults are completely corrected. That means, almost 100% of single transient faults are caught. Optionally, a TMR implementation reports also a corrected fault by the fault signal **F**.

In contrast to a system-level implementation, a *FF-based* implementation adds the triplication to each flip flop separately by triplicating the combinational logic of the input of the flip flop. This step is applied for each flip flip.

Chapter 3

Fault Model

A fabricated chip is delivered to the customer once the post-production test did not detect any misbehavior caused during the manufacturing process. However, after producing the chip various effects may influence the correct computation during operation. Recently, coping with soft errors became a major challenge for future technology scaling [Bor07, BBL+12]. Due to the increasing integration density the vulnerability of today's circuits against *transient faults* is significantly increased compared to older circuit generations. A transient fault temporarily modifies the functional behavior of a circuit [ALRL04]. Established techniques are available to catch and handle those faults in order to keep the circuit working properly. However, the correct implementation of those techniques needs to be verified. In order to verify certain behavior of a circuit in the presence of transient faults these effects need to be properly defined.

This chapter discusses the effects of transient faults modeled at logic level of a circuit and proposes a categorization of the impact of those faults for the circuit's component. A fault model at logic level is introduced to abstract the electrical effects caused by transient faults. Furthermore, effects of transient faults at Boolean level are lifted to word-level. That means, besides single transient faults, certain multiple transient faults are covered as well. Those multiple transient faults will occur more often in advanced technology generations as well [MZM10, MR08, Nic11].

3.1 Transient Faults

A *transient fault* may temporarily modifies a transistor's state in a digital circuit but does not damage the hardware physically – a *soft error* occurs. These faults are typically caused by the environment. High energy neutrons

and α-particles may cause a pulse for a very short duration. This may cause a bit flip in a circuit at internal signals, i.e., a logical value is flipped from 0 to 1 or from 1 to 0, respectively. Due to the affected logical value the circuit's function is temporarily changed. Over the time the fault effect might be observable at the output of the circuit. Consequently, the circuit does not operate as specified which may have dramatic consequences in safety-critical systems.

Transient faults are divided into two groups: *Single Event Upsets* (SEU) and *Single Event Transients* (SET) [Nic11]:

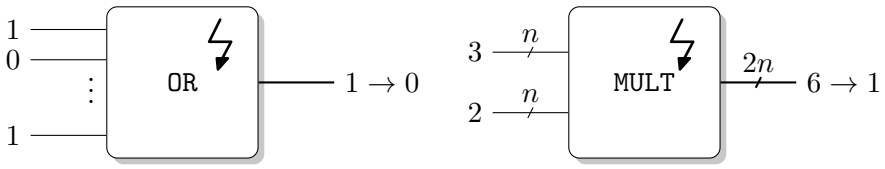
- **SEU** causes bit flips directly in the state elements, i.e., the stored value might be flipped,
- **SET** causes bit flips in combinational logic but might be propagated to the state elements when the fault arrives the state elements while it captures the values: Three masking effects may occur that the transient faults are not propagated to state elements:
  1. *Timing masking*: the fault is propagated too fast or too slow to the inputs of the state elements to capture the fault,
  2. *Electrical masking*: the amplitude is too small to be captured by the state elements,
  3. *Logical masking*: the path is not appropriately sensitized such that the fault does not arrive the state elements.

For a long time hardening techniques were only applied to state elements since SEU were more likely to occur than SET due to various factors such as feature sizes, frequency, and voltage. However, the probability that SET occurs increases, e.g., due to increasing operation frequency that increases the probability that a faulty value arrives the state elements.

Both categories SEU and SET are considered in this work under the term transient faults. However, if a separate analysis of SET and SEU is desired, this can be easily configured by restricting the analysis to the respective components. Analyzing faults in state elements or combinational logic is transparent to the techniques and models introduced in this work. Further, this thesis focuses on logical masking. Timing masking and electrical masking are considered in the work of, e.g., [MZM10] whose model complexity is significantly increased.

3.1.1 Transient Faults

The effect of a transient fault in a component's logic is modeled as non-deterministic value at a component's output.



(a) Transient fault: $1 \to 0$ (b) Transient fault: $6 \to 1$

Figure 3.1: Component Model: Transient faults at different levels of abstraction

Definition 3.1. *A* Complex Transient Fault *(CTF) is written as follows:* $b \to b'$ *with* $b \neq b'$ *and* $b, b' \in \mathbb{N}$*. That means, the value* b *is modified to value* b'*.*

Definition 3.2. *A* Simple Transient Fault *(STF) is written as follows:* $b \to b'$ *with* $b \neq b'$ *and* $b, b' \in \mathbb{B}$*. That means, a* bit flip *is described by* $0 \to 1$*, and* $1 \to 0$*. An STF is a special case of an CTF.*

Usually, the effects of a STF are single bit flips which are also commonly used in related works. In this work, a more general consideration is used covering a broad range of transient faults by allowing more bits to be flipped according to Definition 3.1.

Example 3.1. *In Figure 3.1(a) an* OR*-gate* g *with single-bit inputs and a single-bit output is shown. The STF* $1 \to 0$ *at component* g *affects the output, i.e., the output is inverted from* 1 *to* 0*.*

In contrast, Figure 3.1(b) shows multiplier MULT *with two* n*-bit inputs and one* $2n$*-bit output. The CTF* $6 \to 1$ *at the multiplier affects the output, i.e., the correct multiplication of* $2 \cdot 3 = 6$ *is modified to value* 1*.*

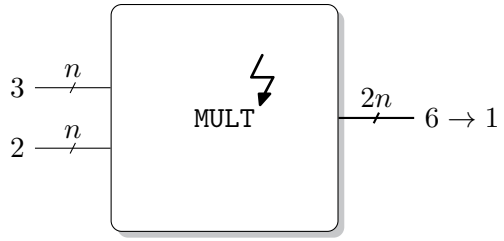
The space of all CTFs at a certain component is defined as follows:

Definition 3.3. *Given a component* $g \in V$ *of a circuit. All possible CTFs of component* g *are given by:*

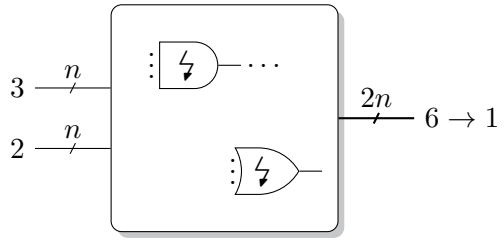
$$\mathcal{F}(g) = \bigcup_{\substack{b,b' \in \{0,\ldots,2^{\mathsf{bv}(g)}-1\} \\ b \neq b'}} (b \to b', g)$$

where bv *specifies the bit-width of component* g*.*

Example 3.2. *All CTFs of the* OR*-gate from Example 3.1 are given by* $\mathcal{F}(\texttt{OR}) = \{(0 \to 1, \texttt{OR}), (1 \to 0, \texttt{OR})\}$ *and all CTFs of the multiplier* MULT *are given by* $\mathcal{F}(\texttt{MULT}) = \bigcup_{\substack{b,b' \in \{0,\ldots,2n\} \\ b \neq b'}} (b \to b', \texttt{MULT})$*.*



(a) Transient fault: $6 \rightarrow 1$



(b) Multiple STFs: $6 \rightarrow 1$

Figure 3.2: CTFs and multiple STFs

In the related work, a different probabilities that a transient fault occur are taken into account, e.g., [HPB07, CRP+96, KMH05]. However, in this thesis all transient faults are equally distributed over all components yielding to a conservative fault model.

In the following, it is supposed that all CTFs of a component may non-deterministically occur. That means, a non-deterministic value is assumed according to the space of CTFs of the respective component. That means the assumed value is independent on the component's function.

Definition 3.4. *Given is a circuit $\mathcal{C} = (V, E)$. The set of all CTFs of the circuit is given by: $\mathcal{F}(\mathcal{C}) = \bigcup_{g \in V}(g, \mathcal{F}(g))$.*

The number of CTFs is denoted by $|\mathcal{F}(\mathcal{C})|$.

3.1.2 Component Model and Multiple Transient Faults

The space of transient faults depends on the abstraction level of the CUV.

As introduced in Section 2.3.1 a circuit is composed of components where a component is a more complex module or a primitive gate. For each primitive gate, there are two possible STFs, $0 \rightarrow 1$, and $1 \rightarrow 0$ as introduced in Definition 3.2.

Consider Figure 3.2 that illustates the `MULT`-module as a high level component (Figure 3.2(a)) and the corresponding gate level components

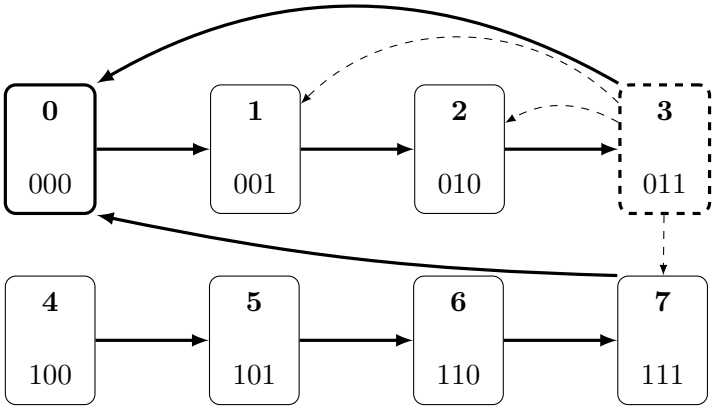


Figure 3.3: Modulo-3 counter with impact of a STF

(Figure 3.2(b)). Multiple STFs at gate level are modeled by a single CTF at the higher level. This modeleding allows to cover local multiple transient faults by a single CTFs. However, assembling a certain set of gates into a component arbitrary multiple STFs analysis can be performed. As a side effect the complexity is reduced.

There are related works that focuses on analyzing multiple transient faults, e.g., [MZM10, FFSD09]. While the work of [MZM10] is about analyzing (multiple) transient faults on electrical level which is typically more complex than on logical level, the work [FFSD09] analyze transient faults on logical level. The work of [FFSD09] is based on the author's Diploma thesis that analyzes multiple STFs of a digital circuit. This work considers multiple STFs over all components which results in a huge search space. For example, a circuit that consists of only 275 primitive gates the number of multiple STFs is approximately $6.09 \times 10^{10} = 60\,900\,000\,000$ while considering up to 4 simultaneously occurring transient faults. Despite this huge number up to 46% of the faults were completely classified within almost one hour. However, with increasing number of gates the problem becomes unmanageable and therefore simplifying the search space needs to be done. Due to the component model of this thesis local multiple STFs are covered by a single CTF.

Example 3.3. *In Figure 3.3 an FSM of a modulo-3 counter is shown. A state is marked by a rounded rectangle where in the upper part the state value is shown represented by a high level implementation and in the lower part the bit-encoding of the state value represented by a gate level implementation. Three single-bit state elements are required to stored the state information.*

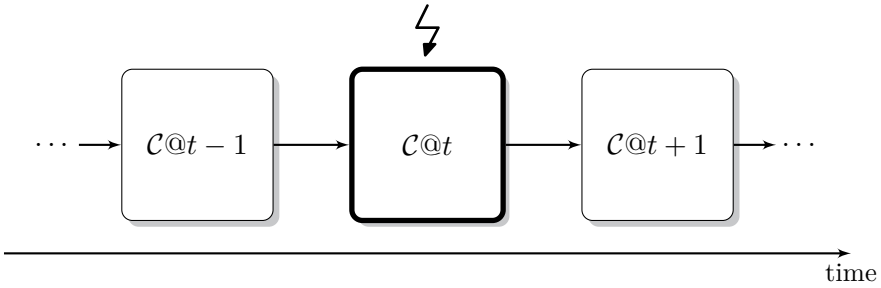


Figure 3.4: Transient fault at an arbitrary time frame

Arrows between the rectangle mark valid transitions. The states 0, 1, 2, *and* 3 *are reachable states. The remaining states are non-reachable states. Suppose the current state of counter is state* 3 *and a STF occurs. The dashed arrows mark possible impacts of the fault. Only a single bit is flipped by an STF. i.e., possible STFs are* $011 \rightarrow 010$, $011 \rightarrow 111$, $011 \rightarrow 001$ *since exactly one state bit is affected. Thus, the STF may cause a transition to state* 1, 2, *or* 7 *which are all invalid computations.*

In contrast, suppose an CTF occurs. Since all bits can be affected by an CTF all states can be reached by that fault. In context of STF, this can only be reached by more than one STF.

In the fault model of this thesis a CTF are modeled to occur only for a single but arbitrary time frame. Once a CTF occurs in time frame t, the CTF disappears immediately for all subsequent time frames $> t$, i.e., the affected component behaves as specified even though the CTF still affects the internal state of the circuit.

3.2 Problem Formulation of the Thesis

If a transient fault occurs the flipped bits have different effects of the circuit's behavior. These different kinds of behavior are distinguished by different terms named as *classes* and the computation of the impact is named as *classification*.

A schematic view of a circuit over different time frames is shown in Figure 3.4 in order to illustrate the situation to analyze. The notation $\mathcal{C}@t$ denotes the circuit in time frame t in an arbitrary state. Suppose the circuit is affected by a CTF at an arbitrary component g denoted by $\lightning$. An essential question about the impact of the CTF comes up:

> *Does any CTF at component g affects the primary outputs in any subsequent time frames?*

This intuitively stated question is formally and more elaborated presented later in this section. Answering that question is one of the main tasks of this thesis and means:

1. If the question is answered with "yes" for a certain component, the designer knows that this component is vulnerable against transient faults and can implement techniques for protection or can correct a maybe buggy implementation,

2. In contrast, if the answer is "no", the designer can safely conclude that the circuit correctly computes the desired function even in the presence of transient faults at the considered component.

Overall, providing an answer for all components forms a measure for the quality of the circuit in the presence of transient faults in terms of a ratio of vulnerable components and all components. However, assessing the quality in the presence of transient faults of a circuit is essential in the design process particularly for safety-critical applications as a requirement in certification processes (e.g., ISO 26262).

Consequently, the aim of the thesis is summarized as follows:

> *Quantify the fault tolerance as high as possible with the highest possible quality.*

3.3 Classes

The basic idea of robustness checking is to classify each component of a circuit into different classes based on the impact of transient faults. This corresponds to give an answer to the question stated in the section before.

Each class represents a certain behavior. The result is a partitioned circuit that highlights the different behavior. Therefore, the designer clearly identifies parts that need to be additionally protected or determines weaknesses of the implemented technique.

The different classes are formally introduced in the following: Given a circuit $\mathcal{C} = (V, E)$ and a component $g \in V$ that has to be classified. Further, the circuit might be equipped with a checker circuitry. The checker reports detected faults by a fault signal $\mathbf{F}$. If there is no such fault signal, $\mathbf{F}$ is implicitly assumed to be always equal to zero, i.e., $\mathbf{F}$ reports no fault.

Class 1 (Robust)**.** *The component g is classified as* **robust** *if for all scenarios and all CTFs one of the following conditions hold:*

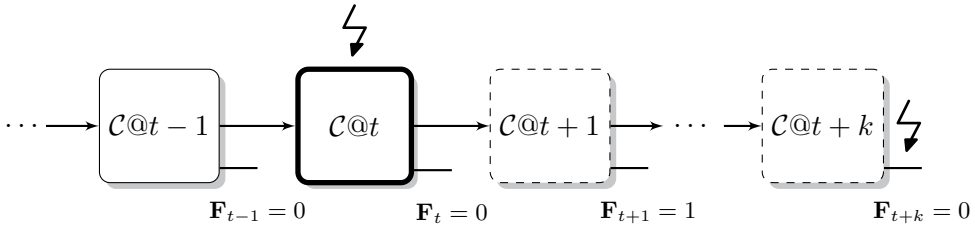


Figure 3.5: Robust classification of a component (Condition 1)

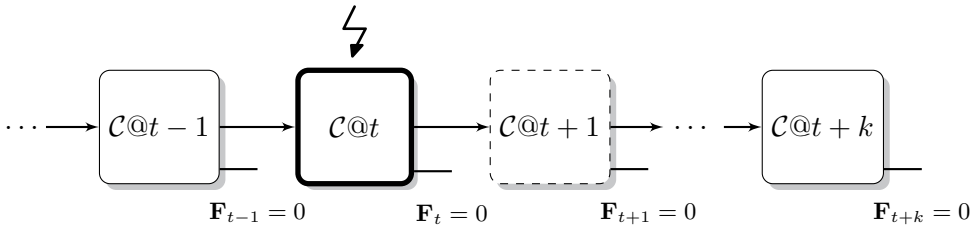


Figure 3.6: Robust classification for a component (Condition 2)

- **Condition 1:** *the fault signal* $\mathbf{F}$ *reports the fault before it may become observable at the primary outputs,*
- **Condition 2:** *the fault is corrected before it becomes observable at the outputs and the states matches the fault-free computation.*

All robust components are contained in the set $\mathbb{T}$.

Example 3.4. *Figure 3.5 illustrates a possible situation for* **Condition 1**. *A CTF occurs at time frame* t *and a component's outputs is affected. In this figure affected time frames are marked by dashed borders. However, the fault is detected through the internal checker circuitry and is reported by the fault signal, i.e.,* $\mathbf{F}_{t+1} = 1$. *Various recovery techniques are available triggered by the fault signal to get the circuit into a consistent state, e.g., by a reset sequence. That means, this behavior is covered by the implemented circuitry and the respective scenario and CTF is not critical.* **Condition 2** *is illustrated in Figure 3.6, the fault is simply corrected through the internal logic and the state of the circuit matches finally the fault-free computation, i.e., the circuit is getting back into a consistent state in time frame* $t + k$ *– marked by solid borders. If one of both conditions holds for all scenarios and all CTFs the component is classified as robust because no faulty computation is propagated to the primary outputs.*

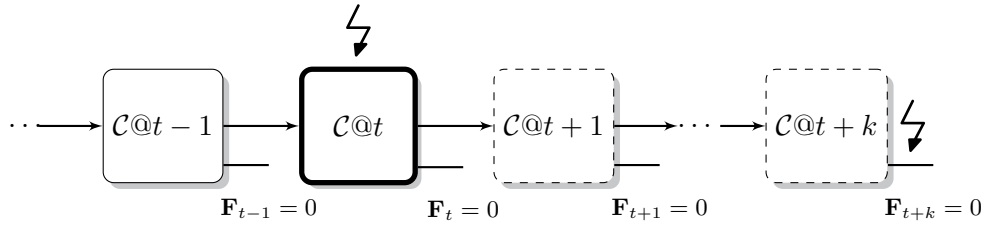


Figure 3.7: Non-robust classification of a component

Class 2 (Non-robust)**.** *The component g is classified as k-***non-robust** *if there is at least one scenario and at least one CTF $f \in \mathcal{F}_{\mathbb{N}}(g)$ at any time frame t that becomes observable on at least one primary output within k time frames before the fault signal reports a fault, i.e., $\mathbf{F}_{t+0} = 0, \ldots, \mathbf{F}_{t+k} = 0$. Thus, the fault becomes observable before it can be detected or corrected. All components that are k-non-robust are contained in the set $\mathbb{S}_k$.*

Remark. A component can be contained in the set $\mathbb{S}_k$ and $\mathbb{S}_{k'}$ with $k \neq k'$. That means, the sets are not necessarily disjoint. There is no requirement that k needs to be minimal. However, once a component is classified as k-non-robust, a further analysis with a higher value of k is typically not required.

Example 3.5. *A possible situation of Class 2 is illustrated in Figure 3.7. The circuit $\mathcal{C}$ operates normally until time frame t as a CTF occurs and the output of component g is affected (denoted by ϟ). Corrupted time frames are marked by dashed borders. A faulty value is propagated over k time frames as the fault becomes observable at time frame $t + k$ (denoted by ϟ). In all time frames between t and $t + k$ the fault signal does not report any fault. Consequently, the circuit violates the specification and the component g is classified as k-non-robust.*

The term k-non-robust is also denoted by non-robust if k is not necessary for the context and is related to the circuit in general.

If a component is not k-non-robust, i.e., no scenario and no CTF lead to a faulty behavior in k time frames but the states differ, *Silent Data Corruption* (SDC) occurs. That means, the state of the circuit is corrupted and differs from the fault-free computation. More formally defined in the following class:

Class 3 (Dangerous)**.** *The component g is classified as k-***dangerous***, if all scenarios and all CTFs at time frame t the fault is either correct / detected or the states in time frame $t + k$ are affected, i.e., the states differ from the*

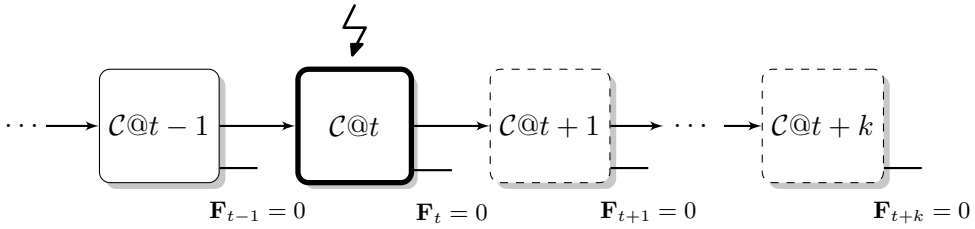


Figure 3.8: Dangerous classification of a component

fault-free computation while the fault signal does not report any fault. That means the circuit's output behavior is not modified within $t+k$ time frames. All components that are k-dangerous are contained in the set $\mathbb{D}_k$.

Example 3.6. *The situation of Class 3 is illustrated in Figure 3.8. A CTF is injected in time frame t. The state is affected in k further subsequent time frames where the fault signals do not report any fault.*

The classification of k-dangerous components is a temporary classification. Since a component might be k-dangerous but in further analysis the component may become $(k+m)$-non-robust or robust for any $m > 0$. The following scenarios may occur: 1) a fault becomes observable at the primary outputs in m additional time frame, i.e., $k+m$ and the component is classified as $(k+m)$-non-robust, or 2) the scenarios and CTFs are corrected or detected such that the component is robust. If a component is classified as k-dangerous further analysis is required. However, once a component is classified as k-non-robust or robust the classification is complete for this component. Therefore, the number of components classified as k-dangerous decreases with increasing k, i.e., more and more components are classified as non-robust or robust.

Lemma 3.1. *It holds $\mathbb{D}_k \supseteq \mathbb{D}_{k+1} \supseteq \ldots \supseteq \mathbb{D}_{k+m}$ for any m and k.*

Proof. By contradiction: Suppose it holds $\mathbb{D}_k \subset \mathbb{D}_{k+1}$ for any k. That means, there is a component g that is not k-dangerous but $(k+1)$-dangerous with $g \notin \mathbb{D}_k$ and $g \in \mathbb{D}_{k+1}$. A component that is not k-dangerous is either k-non-robust or robust. That means, either a CTF becomes observable after k time frames or all faults are corrected or detected which means that the classification is complete. Consequently, a component g that is not k-dangerous cannot be $(k+1)$-dangerous. □

Knowledge about k-dangerous components is valuable for the designer since the longer SDC holds in a system the more likely a second transient

Table 3.1: Best case complexity of the classification.

Classification	Time frame	Scenarios	CTFs
k-non-robust	one	one	one
k-dangerous	one	one	one
robust	all	all	all

fault occurs which may result in accumulated fault effects leading to a non-robust behavior. Moreover, a buggy implemented checker circuitry can be therefore detected using this kind of classification.

In order to distinguish the components of a circuit $\mathcal{C} = (V, E)$ based on their classifications the sets $\mathbb{S}_k$ for k-non-robust components, $\mathbb{T}$ for robust components, and $\mathbb{D}_k$ for k-dangerous components have been introduced. Components that are not yet classified are named as *non-classified*, i.e., before classification or due to the limitation of computational resources. Those components are contained in the set $\mathbb{U}$. The sets of classification are pairwise disjoint for all k, i.e., $(\bigcup_{i\in[0,k]} \mathbb{S}_k) \cap \mathbb{D}_k = \emptyset$, $\mathbb{D}_k \cap \mathbb{T} = \emptyset$, and $(\bigcup_{i\in[0,k]} \mathbb{S}_k) \cap \mathbb{T} = \emptyset$. Overall, it holds $V = \bigcup_{i\in[0,k]} \mathbb{S}_k \cup \mathbb{T} \cup \mathbb{D}_k \cup \mathbb{U}$.

3.3.1 Best Case Complexity

The classification of k-non-robust and k-dangerous components is reduced to an existential proposition, i.e., if there is at least *one* scenario and at least *one* CTF such that the output or the states are tampered, respectively, the components are classified as non-robust or dangerous and no further scenarios need to be checked. In contrast, the classification of robust components demands for a universal proposition, that states that under *all* scenarios and *all* CTFs the specification is kept for all time frames or the fault is reported. That means, it has to be proven that no scenario and no CTF violates the output behavior.

In the following the best case complexity for classifying the components in the respective classes is emphasized. The best case complexity of classifying k-non-robust, and k-dangerous components and the best case complexity of classifying robust components is significant different. The requirements to classify the components into the respective classes are summarized in Table 3.1.

The first column shows the classification. In the remaining columns: 1) *one* means that one time frame, one scenario, or one CTF needs to be

considered, 2) *all*, means that all time frames, all scenarios, or all CTFs need to be considering.

Consider the best case for k-robust components: Exactly a one scenario and one CTF need to be found for one time frame to classify a component to be non-robust. Analogously, this holds also for a k-dangerous component. In contrast, classifying a robust component requires that for all scenarios and all CTFs the fault is not observable at the primary outputs for all time frames. However, the number of scenarios exponentially grows with the number of primary inputs and the number of state elements. Moreover, as it will be later shown k is typically very large to provide a complete classification. Consequently, the effort of classifying robust components is significantly higher than for k-non-robust and k-dangerous components.

However, robust components can be easily derived from the classification of k-non-robust and k-dangerous components that significantly reduces the complexity in the best case. Since for a robust components it needs to be excluded that this component is neither k-non-robust nor k-dangerous which is in the best case easier than proving that all CTFs do not cause misbehavior over all time frames.

Lemma 3.2. *Given a component $g \in V$ such that g is neither k-non-robust nor k-dangerous, i.e., $g \notin \mathbb{S}_k$ and $g \notin \mathbb{D}_k$ then g is robust, i.e., $g \in \mathbb{T}$ for any k.*

Proof. If there exists no scenario and no CTF at component g that lead to faulty behavior in k time frames or to corrupted states in time frame k, then the circuit behaves as expected by detecting or correcting all CTFs, i.e., g is robust. □

Corollary 3.1. *Given the set of k-non-robust components $\mathbb{S}_k$ and k-dangerous components $\mathbb{D}_k$, then it holds: $\mathbb{T} \subseteq V \setminus (\mathbb{S}_k \cup \mathbb{D}_k)$.*

Proof. Follows directly from the proof of Lemma 3.2 by applying the proof for each robust component. □

3.4 Completeness

So far the classification of k-non-robust and k-dangerous components is based on an arbitrary value k. However, in order to compute all non-robust components all necessary sets $\mathbb{S}_0, \ldots, \mathbb{S}_{k_{\mathsf{cmpl}}}$ have to be determined where k_{cmpl} is a completeness threshold and is explained in the following.

It is assumed that the fault occurs at any arbitrary time frame t, i.e., all time frames where a fault can occur are covered. The *completeness threshold* (CT) k_{cmpl} describes the maximal value to cover all possible propagation

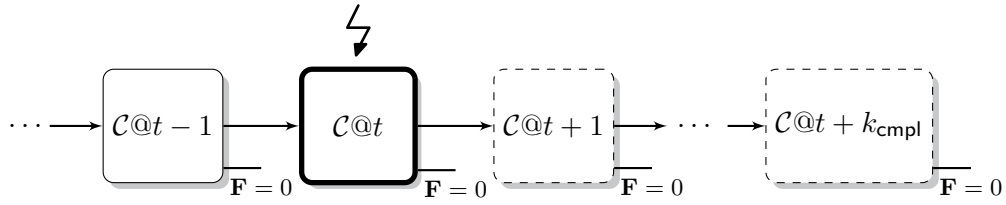


Figure 3.9: Unbounded dangerous components

paths that allows to fully determine all non-robust components . That means, for all values of $k \in [0, k_{\mathsf{cmpl}}]$ the set $\mathbb{S}_k$ has to be determined and the entire set of non-robust components $\mathbb{S}(k_{\mathsf{cmpl}})$ is determined by: $\mathbb{S}(k_{\mathsf{cmpl}}) = \mathbb{S}_0 \cup \mathbb{S}_1 \cup \ldots \cup \mathbb{S}_{k_{\mathsf{cmpl}}}$.

Checking all values of k from 0 up to the completeness threshold ensures that all possible scenarios to propagate a CTF are covered. An upper bound of the completeness threshold for an arbitrary circuit is given by the following lemma.

Lemma 3.3. *Given a circuit $C = (V, E)$ with n single-bit state elements*[1] *then the* completeness threshold *is bounded by:* $0 \leq k_{\mathsf{cmpl}} \leq 2^{2n}$.

Certainly, in practice the completeness threshold might be much smaller than the upper bound. But a computation of the value itself is a nontrivial task [CKOS04]. However, manually specified completeness threshold can be given by the designer who has special knowledge about the design. A technique to automatically obtain completeness is presented later in this thesis.

Once all necessary values of k has been considered such that all non-robust are determined, the vulnerable components of the circuit against CTFs are obtained. Even when all necessary values for k have been checked, SDC may occur which may consistute vulnerability of the circuit as well. That means, under all possible scenarios and all CTFs at a component for all $k \in [0, k_{\mathsf{cmpl}}]$ a fault is neither observable at the primary outputs nor reported by the fault signal but corrupting the states. This particular behavior is more formally defined as the following class:

Class 4. *A component g is classified as* **unbounded dangerous***, if and only if $g \in \mathbb{D}_{k_{\mathsf{cmpl}}}$.*

Example 3.7. *In Figure 3.9 a situation of Class 4. A CTF occurs in time frame t and affects all time frames up to k_{cmpl} time frames. The fault signal does not report any fault within the interval.*

[1] Note, a circuit with multi-bit state elements can be easily translated into a circuit exclusively consisting of single-bit state elements.

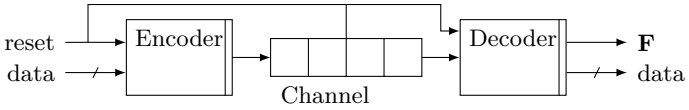


Figure 3.10: Transmission system

The existence of unbounded dangerous components in a design might be critical since the fault is manifested in the circuit state until a reset sequence is performed. The probability that a second transient fault may occur increases over the time, i.e, the longer the circuit is in the corrupted state the higher the probability that a second transient fault causes accumulated fault effects.

3.5 Observation Window

A circuit that contains checker circuitry occasionally comes with special constraints. For example, the fault signal should immediately report a fault after a bounded number of clock cycles when a transient fault occur. Thus, a bounded interval $[0, \bar{k}]$ may suffice to get accurate classifications with $\bar{k} \leq k_{\mathsf{cmpl}}$. This interval is called *observation window* and is manually specified by the designer. Often in practice the length of an adequate observation window is much smaller than the general completeness threshold, i.e., $\bar{k} \ll k_{\mathsf{cmpl}}$. Consequently, the manually specified observation window leads to a completeness threshold. However, if not other mentioned the observation window is set to k_{cmpl} by default. The influence of the choice of the observation window is illustrated by the following example.

Example 3.8 (from [FSFD11])**.** *A* $(7, 4)$*-Hamming-Code recognizes and repairs single faults [Ham50]. Figure 3.10 shows a transmission using an encoder for 4-bit* `data`*, a bit-wise serial channel, and a decoder. A failure in the transmitted code word is flagged by setting the fault signal* **F***. The circuit computing this transmission consists of 368 components. The timing is summarized like this:*

- *Encoding and transmission to the channel: 1 time step*
- *Transmission: 4 time steps (registers in the channel)*
- *Decoding, writing to the output, setting* **F***: 1 time step*

The classification of the components depends on the value $\bar{k}$*:*

- $\bar{k} < 6$*: The data from* $k = 0$ *did not arrive at the primary outputs, yet. Faults in the decoding logic are detected within 1 time step by setting the fault signal* $\mathbf{F}$*. The corresponding components are classified.*

  Faults in the channel change the state, but not all data has been decoded, yet. Faults in unprocessed registers are undetected, yet, and the components cannot be classified. While incrementing $\bar{k}$*, more and more components are classified.*

- $\bar{k} = 6$*: The input data reaches the primary outputs. Faults that can be detected are flagged and undetected faults in the encoder propagate to the primary outputs. All components are classified.*

- $\bar{k} > 6$*: Faults injected at* $t = 0$ *do not influence the state of the model after more than 6 time steps.*

3.6 Summary

The effects of transient faults are abstracted to logical level. A fault model is introduced that models transient faults as a non-deterministic output behavior of component. Based on the impact of transient faults at the components different behavior is observable at the circuit's outputs. Therefore, different classes that catches the respective behavior are introduced.

In order to overcome complexity issues the classification of robust components is translated into a easier problem according to the best case complexity. Furthermore, a completeness threshold has been introduced that needs to be reached to provide a general complete classification. However, this value might be very large. Practically a significantly smaller observation window has been introduced that is manually specified by the designer to overcome complexity issues.

Chapter 4

Robustness Measures

The classification presented in the previous chapter provides a partition of the circuit into the respective classes. However, in order to rate the quality of the circuit in the presence of transient faults two robustness measures in terms of a single value are introduced. This measures objectively and uniquely documents the quality of the circuit as it can be used, e.g., in certification. Moreover, different hardening techniques for the same circuit can be evaluated and compared based on these measures and the implementation with the best properties can be selected.

The two measures are briefly described:

- A *worst case* robustness measure is introduced that rate each component of a circuit based on the worst case that a scenario and CTF may occur.
- A *probabilistic* robustness measure is introduced that provides a differentiation between the non-robust components. This measure considers a better case of scenario and CTF than the worst case.

4.1 Worst Case Robustness Measure

At first the worst case robustness measure is introduced and defined in the following.

Definition 4.1. *Given a set of robust* $\mathbb{T}$*, non-robust* $\mathbb{S}(\bar{k})$*,* $\bar{k}$*-dangerous* $\mathbb{D}_{\bar{k}}$ *classified components. Furthermore, given a set of non-classified* $\mathbb{U}$ *with* $V = \mathbb{T} \cup \mathbb{S}(\bar{k}) \cup \mathbb{D}_{\bar{k}} \cup \mathbb{U}$ *for any* $\bar{k} \in [0, k_{\mathsf{cmpl}}]$*. The quality of the circuit in the presence of CTFs is given by the* Worst Case Robustness Measure *(*$\mathcal{WC}-\mathcal{RM}$*)* $R^{\bar{k}}_{\mathcal{WC}-\mathcal{RM}}$ *with* $R^{\bar{k}}_{lb} \leq R^{\bar{k}}_{\mathcal{WC}-\mathcal{RM}} \leq R^{\bar{k}}_{ub}$ *where the bounds are defined as follows:*

$$R_{lb}^{\bar{k}} = \frac{|\mathbb{T}|}{|V|} = 1 - \frac{|\mathbb{S}(\bar{k}) \cup \mathbb{D}_{\bar{k}} \cup \mathbb{U}|}{|V|} \quad \textit{(lower bound)}$$
$$R_{ub}^{\bar{k}} = \frac{|\mathbb{T} \cup \mathbb{D}_{\bar{k}} \cup \mathbb{U}|}{|V|} = 1 - \frac{|\mathbb{S}(\bar{k})|}{|V|} \quad \textit{(upper bound)}$$

That means, after classification - computing the respective sets - up to the observation window $\bar{k}$, $R_{\mathcal{WC}-\mathcal{RM}}^{\bar{k}}$ is caluclated by computing the bounds. The $\mathcal{WC}-\mathcal{RM}$ covers the worst case that any transient fault occurs since exactly a single scenario suffices to classify a component to be non-robust independent how likely the scenarios is. A more differentiated measure will be introduced later in this chapter. However, properties of the measure are additionally presented.

Lemma 4.1. *It holds* $R_{ub}^{\bar{k}} - R_{lb}^{\bar{k}} \geq R_{ub}^{\bar{k}+q} - R_{lb}^{\bar{k}+q}$ *for any* $\bar{k} \in [0, k_{\mathsf{cmpl}}]$ *and* $q \geq 0$.

Proof. Suppose all components are classified $\mathbb{U} = \emptyset$. It holds:

$$\frac{|\mathbb{T} \cup \mathbb{D}_{\bar{k}}|}{|V|} - \frac{|\mathbb{T}|}{|V|} \geq \frac{|\mathbb{T}' \cup \mathbb{D}_{\bar{k}+q}|}{|V|} - \frac{|\mathbb{T}'|}{|V|}$$
$$|\mathbb{D}_{\bar{k}}| \geq |\mathbb{D}_{\bar{k}+q}|$$

and the last inequality holds due to Lemma 3.1 for any $q \geq 0$. □

That means, while increasing the size of the observation window the gap of the robustness bounds decreases, i.e., the more accurate is the analysis. However, the accuracy – the gap of the bounds – of the classification depends on the choice of the observation window and varies from circuit to circuit. The higher the value the more fault tolerant is the CUV. That means, high values are expected for fault tolerant designs and low values for relatively unprotected circuits or buggy implementations.

Example 4.1. *Reconsider the Example 3.8 from page 48 of the previous chapter. The determined bounds of the robustness are shown in Table 4.1. Before the classification starts, all components are non-classified, i.e.,* $|\mathbb{U}| = 368$. *By considering more and more time frames from* 0 *up to* 6*, the components get classified. The robustness bounds meet each other after analyzing 6 time frames and the classification is therefore complete. That means,* $\bar{k} = 6$ *is sufficient.*

Table 4.1: Hamming model

$k \in [0,6]$	$\lvert\mathbb{T}$	$\lvert\mathbb{S}\rvert$	$\mathbb{D}_k$	$\lvert\mathbb{U}\rvert$	R_{lb}^k %	R_{ub}^k %
before class.	0	0	0	368	0.0	100.0
0	11	2	355	0	3.0	99.5
1	54	36	278	0	14.7	90.2
2	93	49	226	0	25.3	86.7
3	132	61	175	0	35.9	83.4
4	171	73	124	0	46.5	80.2
5	210	87	71	0	57.1	76.4
6	267	101	0	0	72.6	72.6

When analyzing a combinational circuits there are no dangerous components since a combinational circuit does not contain any state element. Consequently, some issues of the robustness measure are simplified. The size of observation window is always 1 since only one time frame suffices to classify all components. The robustness measure for combinational circuits $R_{\mathcal{WC}\text{–}\mathcal{RM}}$ with $R_{lb} \leq R_{\mathcal{WC}\text{–}\mathcal{RM}} \leq R_{ub}$ is simplified to:

$$R_{lb} = \frac{|\mathbb{T}|}{|V|} = 1 - \frac{|\mathbb{S}(0) \cup \mathbb{U}|}{|V|} \quad \text{(lower bound)}$$
$$R_{ub} = \frac{|\mathbb{T} \cup \mathbb{U}|}{|V|} = 1 - \frac{|\mathbb{S}(0)|}{|V|} \quad \text{(upper bound)}$$

Once all components are classified, i.e., $\mathbb{U} = \emptyset$, the bounds are equal.

4.2 Probabilistic Analysis

The introduced robustness measure $\mathcal{WC}-\mathcal{RM}$ from the previous Section 4.1 results in a *worst case* analysis, since a component is classified as non-robust if there is a single scenario and a single fault that violates the specification. That means, a single but suitable combination of both suffices to classify the component unless how likely the scenario and the fault are during normal operation. Therefore, non-robust components cannot be differentiated how vulnerable they are in practice: A component that has more scenarios than another component is more vulnerable against transient faults. In order to differentiate non-robust components a grading based on their number of scenarios that leads to an *Excitation and Propagation Probability* (EPP) of

faulty behavior which is introduced in the following. Having that grading, hot-spots can easily be highlighted and the designer is pin-pointed to those components with high probabilities, e.g., by visualizing the hot-spots. As a consequence, in order to keep cost constraints during the design process, only those components, containing the hot-spot are protected, because faults at those components are more likely to manipulate the output behavior. A trade-off between degree of fault tolerance and costs can be found using these gradings.

However, in order to obtain the grading more than a single scenario need to be computed. Diverse works have been published to tackle this problem [MZM10, HPB07]. For example, the work of [HPB07] considers all scenarios by building up a BDD that handles all scenarios. Therefore, the most precise possible grading is reached since the work considers all scenarios but at very high computational costs due to the high memory consumption of BDDs. Hence, the BDD-based approach is technically limited to very small circuits since the number of scenarios grows exponentially with the number of inputs of the circuit and the size of the observation window. Those techniques, considering all scenarios are referred to as *probabilistic analysis*.

However, in order to efficiently differentiate non-robust components a new notion that considers a bounded number of scenarios is introduced in the next section.

4.2.1 Excitation and Propagation Probabilities

A second robustness measure that constitutes a trade-off between the worst case analysis and the probabilistic analysis is presented in the following. A technique that considers more than a single scenario but potentially much fewer than all scenarios is introduced. This leads to a measure that is more accurate than the worst case measure and less accurate than the probabilistic measure while the complexity is significantly reduced. If the desired differentiation of non-robust components is reached and the corresponding hot-spots reflect enough information then no more scenarios need to be computed. Moreover, the new measure states the most general measure in this thesis since it embeds both extremes that are easily justified, i.e, this new measure can capture both, the worst case, and the probabilistic case. This measure has been published in [FFD10].

At first, EPP is formally introduced:

Definition 4.2. *Let $g \in \mathbb{S}(\bar{k})$ be a non-robust component and $\bar{k}$ the observation window. The function $\psi(g, \bar{k})$ denotes the number of scenarios that lead to a non-robust classification over the observation window $\bar{k}$.*

Definition 4.3. *Let $g \in \mathbb{S}(\bar{k})$ be a non-robust component and $\bar{k}$ the observation window. The* Excitation and Propagation Probability *(EPP) is given by:*

$$\mathsf{epp}(g, \bar{k}) = 1 - \frac{\psi(g, \bar{k})}{\Psi(\bar{k})}$$

with $\Psi(\bar{k}) = 2^{|\mathsf{in}(\mathcal{C})| \cdot \bar{k}}$ for an arbitrary observation window $\bar{k}$.

The epp function computes a ratio of scenarios that definitely yield faulty output and the number of all scenarios possible within a certain observation window $\bar{k}$. The following example demonstrates the meaning of epp:

Example 4.2. *Consider a combinational circuit $\mathcal{C} = (V, E)$ with four primary inputs, i.e., $|\mathsf{in}(\mathcal{C})| = 4$. Furthermore, let $a, b \in \mathbb{S}$ be two non-robust and $c, d, e \in \mathbb{T}$ be robust components. The worst case analysis yields $R_{\mathcal{WC}\text{–}\mathcal{RM}} = 3/5 = 60\%$. Further, assume that there are only two scenarios that excite and propagate a fault in component a,i.e., $\psi(a, 0) = 2$. Given the total number of $2^{|\mathsf{in}(\mathcal{C})|} = 2^4 = 16$ scenarios, the probability to excite and propagate the fault in a is only $\mathsf{epp}(g, 0) = 2/16 = 12.5\%$. Moreover, let any input trace be a scenario for a fault at component b, i.e., $\psi(b) = 16$ and the excitation probability at b is $\mathsf{epp}(b, 0) = 100\%$.*

The worst case analysis does not differentiate the two components because only a single scenario is computed. Both are simply classified as non-robust, even though b can be considered as a hot-spot while a is relatively save.

However, the number of scenarios grows exponentially with the number of inputs and the size of the observation window. Therefore, a maximum pre-defined portion of scenarios is introduced. The number λ is called *scenario ratio* and limits the number of considered scenarios with respect to all possible scenarios to reduce the complexity.

Definition 4.4. *Given an arbitrary scenario ratio λ with $0 < \lambda \leq 1$. The EPP limited by λ is given by:*

$$\overline{\mathsf{epp}}(g, \bar{k}, \lambda) = 1 - \frac{\min\left\{\psi(g, \bar{k}), \lceil \lambda \cdot \Psi(\bar{k}) \rceil\right\}}{\lceil \lambda \cdot \Psi(\bar{k}) \rceil}$$

Technically, once ψ exceeds the maximum number of scenarios the computation can be terminated rather than enumerating all scenarios to reduce computational costs. Based on this value a second robustness measure covering the notion of multiple scenarios is defined as follows in terms of a upper bound. Since there is no differentation of robust components the lower bound from Section 4.1 is used. However, it follows the upper bound using epp:

Definition 4.5. *The* parameterized robustness measure *denoted as* $\mathcal{P}-\mathcal{RM}$ *with respect to a scenario ratio* λ *with* $0 < \lambda \leq 1$ *is given by:*

$$R_{ub}^{\bar{k},\lambda} = \frac{1}{|V|} \sum_{g \in V} \overline{\mathsf{epp}}(g, \bar{k}, \lambda)$$

The higher the value of λ the more scenarios are considered and more and more components might be differentiated. However, increasing λ increases the computational effort as well. The more differentiations can be computed, the more the accuracy for the designer to find critical hot-spots. However, the influence of λ is emphasized as follows:

The parameter λ intuitively justifies the analysis between the worst case (λ close to zero) and the best case ($\lambda = 1$). Consequently, the higher λ the better the case. Thus, with increasing λ the upper bound of the $\mathcal{P}-\mathcal{RM}$ increases since scenarios become more or less likely.

Lemma 4.2. *Given two scenario ratios* λ *and* λ' *with* $\lambda \leq \lambda'$ *the relation* $R_{ub}^{\bar{k},\lambda} \leq R_{ub}^{\bar{k},\lambda'}$ *holds.*

Proof. Let $\lambda \leq \lambda'$. It has to be shown $R_{ub}^{\bar{k},\lambda} \leq R_{ub}^{\bar{k},\lambda'}$: It suffices to show:

$$\sum_{g \in V} \overline{\mathsf{epp}}(g, \bar{k}, \lambda) \leq \sum_{g \in V} \overline{\mathsf{epp}}(g, \bar{k}, \lambda')$$

$$|V| - \sum_{g \in V} \frac{\min\left\{\psi(g, \bar{k}), \lceil \lambda \cdot \Psi(\bar{k}) \rceil\right\}}{\lceil \lambda \cdot \Psi(\bar{k}) \rceil} \leq |V| - \sum_{g \in V} \frac{\min\left\{\psi(g, \bar{k}), \lceil \lambda' \cdot \Psi(\bar{k}) \rceil\right\}}{\lceil \lambda' \cdot \Psi(\bar{k}) \rceil}$$

$$\sum_{g \in V} \overbrace{\frac{\min\left\{\psi(g, \bar{k}), \lceil \lambda \cdot \Psi(\bar{k}) \rceil\right\}}{\lceil \lambda \cdot \Psi(\bar{k}) \rceil}}^{①} \geq \sum_{g \in V} \overbrace{\frac{\min\left\{\psi(g, \bar{k}), \lceil \lambda' \cdot \Psi(\bar{k}) \rceil\right\}}{\lceil \lambda' \cdot \Psi(\bar{k}) \rceil}}^{②}$$

$$\frac{1}{\lceil \lambda \cdot \Psi(\bar{k}) \rceil} \cdot \sum_{g \in V} ① \geq \frac{1}{\lceil \lambda' \cdot \Psi(\bar{k}) \rceil} \cdot \sum_{g \in V} ②$$

$$\frac{1}{\lceil \lambda \Psi(\bar{k}) \rceil} \geq \frac{1}{\lceil \lambda' \Psi(\bar{k}) \rceil}$$

The last equation holds since $\lambda \leq \lambda'$. □

However, configuring λ appropriately the same differentiation as the probabilistic analysis can be reached but with potentially fewer computational effort. Hence, adjusting the parameter λ a trade-off between accuracy and complexity can be found. Practically, λ can be increased until sufficient differentiation is provided finally decided by the designer.

Figure 4.1: Influence of the scenario ratio λ

4.2.2 Relation of $\mathcal{WC}-\mathcal{RM}$ and $\mathcal{P}-\mathcal{RM}$

The relation of both introduced measure needs to be investigated. $\mathcal{P}-\mathcal{RM}$ is the most general robustness measure defined and used in this thesis. The robustness measure $\mathcal{WC}-\mathcal{RM}$ is a special case of $\mathcal{P}-\mathcal{RM}$ as emphasized as follows.

Lemma 4.3. *Given an observation window* $\bar{k} \in [0, k_{\mathsf{cmpl}}]$ *and* $0 < \lambda \leq 1$ *with* $\lceil \lambda \cdot \Psi(\bar{k}) \rceil = 1$ *then it holds* $R_{ub}^{\bar{k}} = R_{ub}^{\bar{k},\lambda}$.

Proof. Follows directly from the proof of Lemma 4.2. □

Embedding the worst case analysis $\mathcal{WC}-\mathcal{RM}$ in $\mathcal{P}-\mathcal{RM}$ measure is easily done by simply adjusting λ close to zero such that $\lceil \lambda \cdot \Psi(\bar{k}) \rceil = 1$, i.e., exactly a single scenario is considered as presented previously in Section 3.3. In contrast, embedding the probability analysis is easily done by adjusting $\lambda = 1$, i.e., all possible scenarios are considered as it is performed in, e.g., [HPB07]. That means, the new measure embeds both kinds of analysis and constitutes a trade-off between accuracy and costs adjustable by λ.

Figure 4.1 shows a graphical interpretation of $\mathcal{P}-\mathcal{RM}$. The worst case analysis is performed when setting λ close to zero as introduced with Lemma 4.3. In contrast, probabilistic analysis is performed when setting $\lambda = 1$. The differentiation of non-robust components which are more vulnerable than other non-robust components increases with higher λ. However, the computational costs increases as well, when increasing λ since computing a scenario is not trivial.

Therefore, finding hot-spots, that means regions of the circuit that contains particularly non-robust components is possible using the new measure. The designer incrementally increases λ step by step until a sufficient accurate differentiation is obtained.

4.3 Class Models

In the previous chapter and in this chapter the fault model and the robustness measures have been presented, respectively. The fault model encompassed the effects of transient faults into classes: k-non-robust, k-dangerous, robust and unbounded dangerous components. In this chapter, two robustness measure have been presented.

In this section, meaningful combinations of fault model, measure, and circuits are introduced in the following.

In the chapter of this thesis various approaches are presented to classify the components of a sequential and combinational circuits. But there are theoretical and practical differences. For example, all approaches are able to handle sequential circuits except of one approach. Furthermore, theoretically all approaches are able to obtain EPP for each component. But from the efficiency point of view only one approach is able to compute EPP efficiently.

However, the first basic algorithm presented in the next section is able to handle all issues, from the classes of the fault model to the robustness measures. This algorithm state only a theoretical model and the further concrete approaches are derived from the model with certain properties. To provide a clear differentiation between the approaches unique terms are introdcued.

The most general class model of this thesis is provided by the EPP-based classification defined as follows:

Definition 4.6. *The class model* EPPModel *is defined over three models:*

1. *the fault model including k-non-robust, k-dangerous, robust, and unbounded dangerous components defined in Section 3.3,*
2. *the robustness measure $\mathcal{P}-\mathcal{RM}$ including differentiation between non-robust components, and*
3. *sequential circuits and combinational circuits as well.*

The EPPModel provides the most general analysis in this thesis which requires also the most computational effort. A more specialized class model is presented as next:

Definition 4.7. *The class* WCModel *is defined over three models:*

1. *the fault model including k-non-robust, k-dangerous, robust, and unbounded dangerous components defined in Section 3.3,*
2. *the robustness measure $\mathcal{WC}-\mathcal{RM}$ a special case of $\mathcal{P}-\mathcal{RM}$, and*

3. sequential circuits and combinational circuits as well.

The thesis focuses on providing approaches that handles WCModel. However, the approaches are theoretical able to handle EPPModel but due to efficiency issues the most approaches are reduced to the WCModel. A dedicated ATPG-based approach is able to compute EPPModel.

So far, both models consider sequential circuits that requires to analyze the circuit over a certain number of time frames. However, additionally combinational circuits are handled forming the following class model:

Definition 4.8. *The class model* CombModel *is defined over three models:*

- *non-robust and robust component defined in Section 3.3,*
- *the combinational case of the robustness measure $\mathcal{WC}-\mathcal{RM}$,*
- *combinational circuits.*

4.4 Summary

Knowing the quality of the circuit under the effects of transient faults is crucial for reliable circuits. In this section two measures have been proposed: 1) $\mathcal{WC}-\mathcal{RM}$ that considers the worst case of all possible scenarios and CTFs, and 2) $\mathcal{P}-\mathcal{RM}$ a more differentiated measure that provides a freely configurable accuracy that converges to a exact probabilistic analysis. In contrast to available probabilistic analysis a trade-off between accuracy and computational effort can be found.

Moreover, three fixed configuration combining the classes from the previous chapter and the introduced measures above are introduced. Based on this configuration approaches are proposed that classify the components of a circuit into the respective classes.

Chapter 5

Computational Model

While the previous chapters define what kind of behavior may occur in the presence of transient faults, this chapter introduces the algorithms to analyze the circuit and to classify the components into the introduced classes. A computational model that creates the fundamentals for the classification is introduced. Later in this thesis, various engines (classifiers) are presented covering a wide range of reasoning techniques. However, all engines assess the robustness along a basic classification technique. Before introducing the engines a theoretical algorithm forming the basic classification technique is introduced in this chapter. This algorithm provides a simplified view of how robustness checking works. Properties and requirements of the algorithm are discussed. In particular, approximation of reachability information is presented since an exact computation is often practically infeasible. Importantly, the influence of the approximation on the quality of the analysis is theoretically investigated. Later show, embedding approximation in robustness checking is very useful since a great trade-off between cost and quality can be reached.

5.1 Modelling CTFs in Circuits

In order to analyze the circuit's behavior under CTFs arbitrary values have to be injected at certain signals in the circuit since transient faults are modeled as non-deterministic values. The thesis focuses on the logic level and considers logical masking of transient faults. Therefore, faults in a circuit are modeled at logic level with common *fault injection techniques* as presented in the following.

All possible CTFs of a component are not directly modeled, e.g., as it is done in ATPG, since too many faults need to be modeled. Instead of

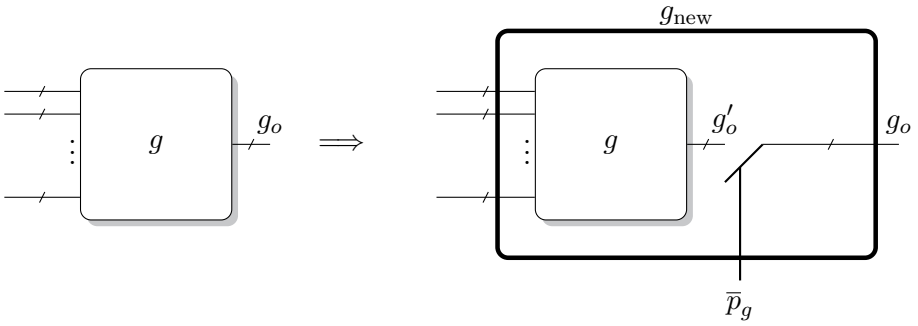


Figure 5.1: Component g_{new} encapsulate component g and logic to inject CTF

modeling all faults directly, a symbolic representation is chosen. That means, the presented model represents all possible faults at a component at once. This slightly increases the search space of a single problem instance but reduces the overall number of problem instances that need to be considered.

Given a circuit $\mathcal{C} = (V, E)$ and a set of components $\mathbb{U} \subseteq V$ to be classified. A *fault modelling* circuit $C^{\mathbb{U}} = (V', E')$ that models CTFs at components in $\mathbb{U}$ is constructed as follows: Each component $g \in \mathbb{U}$ is replaced by the construction illustrated in Figure 5.1. A new component g_{new} encapsulates the component g from the original circuit with a new free input p_g called *fault predicate*[1]. The component g_{new} behaves exactly as the component g when the fault predicate is set to zero (not activated). Otherwise, when the fault predicate is set to one (activated) an arbitrary value independent on g's function can be injected. That means, all possible CTFs according to $\mathcal{F}(g)$ are symbolically modeled. Overall, a component is virtually disconnected from its fan-out cone when the fault predicate is activated. The set of all fault predicates is denoted by $P^{\mathbb{U}}$ and contains one fault predicate per component, i.e, $|P^{\mathbb{U}}| = |\mathbb{U}|$.

However, since transient faults are considered only a single but arbitrary time frame is manipulated temporarily through fault injection. In contrast, a stuck-at-fault from SAFM manipulates a component permanently.

5.2 Models for Classifications

The basic classification techniques to classify components with respect to EPPModel is presented in the following. Since, the EPPModel is the

[1]Note, the fault predicate is not contained in the set of primary inputs of a circuit rather than in a dedicated set.

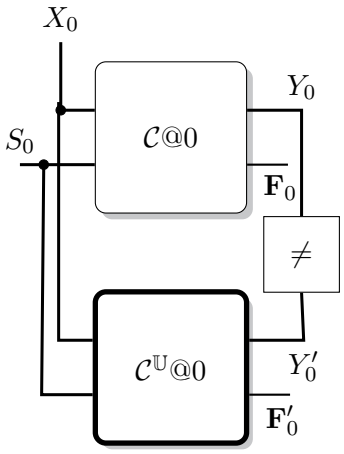


Figure 5.2: Model for classifying 0-non-robust components

most general model considered in this thesis the classification techniques is sufficiently general.

However, two models that allow to classify the components according to the proposed classes, k-non-robust and k-dangerous from Section 3.3 are presented: Given is a circuit $\mathcal{C} = (V, E)$, a non-empty set of components to be classified $\mathbb{U} \subseteq V$, and an observation window $\bar{k} \in [0, k_{\mathsf{cmpl}}]$.

5.2.1 Model for Classifying k-non-robust Components

Figure 5.2 illustrates the model for classifying 0-non-robust components. The circuit $C^{\mathbb{U}}$ is constructed to inject CTFs according to the components of $\mathbb{U}$. The circuits $\mathcal{C}$ and $\mathcal{C}^{\mathbb{U}}$ are modeled as a sequential equivalence check for one time frame. More precisely, the instances $\mathcal{C}$@0 and $\mathcal{C}^{\mathbb{U}}$@0 are stimulated by the same input stimuli denoted by X_0. Further, both circuits start at the same set of states S_0 (fault-free) which are called *injection states*. Here, all reachable states are allowed, i.e., $S_0 = S^*$. Since, non-robust components are classified, the primary outputs are checked for differences, i.e., $Y_0 \neq Y_0'$. Furthermore, the fault signals $\mathbf{F}_0$ and $\mathbf{F}_0'$ are highlighted. The fault signal $\mathbf{F}_0$ from the fault-free computation is assumed to be always zero, because if there is no internal fault, $\mathbf{F}_0$ must not report any fault. The set $P^{\mathbb{U}} = \{p_{g_1}, \ldots, p_{g_{|\mathbb{U}|}}\}$ contains all fault predicates with respect to $\mathbb{U}$. Exactly one fault predicate p_{g_i} is set to one, i.e., $p_{g_1} + \ldots + p_{g_n} = 1$ because single CTFs are considered.

Suppose the fault predicate $p_{g_i} \in P^{\mathbb{U}}$ is activated such that an arbitrary value according to $\mathcal{F}_{\mathbb{N}}(g_i)$ is injected at the output of component g_i. All remaining components behave as specified, since those fault predicates are

set to zero. If there exists a scenario $\tau = ((X_0), S_0)$, i.e., input stimuli X_0 for the primary inputs and assignments to the injection state S_0, such that $Y_0 \neq Y_0'$ is true and the fault signal does not report a fault, i.e., $\mathbf{F}_0' = 0$, the component g_i is classified as 0-non-robust and is stored in the set $\mathbb{S}_0$.

This model is denoted as $\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, 0)$, because it classifies non-robust components of $\mathbb{U}$ of circuit $\mathcal{C}$ while considering all reachable states S^* while no additional time frame is unrolled for propagation.

In general not all non-robust components can be classified by considering an observation window of length one. A model arguing over an observation window of length two is shown in Figure 5.3. Two new instances of $\mathcal{C}$ are appended to the existing instances, respectively. Note, than circuit $\mathcal{C}^{\mathbb{U}}$ is used only in the first time frame since transient faults are modeled. Next states S_1 and S_1' are connected to the new instances where both copies are stimulated by the same input stimuli, denoted by X_1. If there exists a scenario $\tau = ((X_0, X_1), S_0)$ with an activated fault predicate p_{g_i} that leads to differing outputs Y_1 and Y_1' before the fault signal reports a fault, then the component g_i is classified as 1-non-robust. This model is denoted as $\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, 1)$ because one additional time frame is unrolled for propagation. Modeling an observation window of length k is analogously constructed and denoted by $\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k)$ that classifies k-non-robust components.

This model is used to classify non-robust components and to compute the sets $\mathbb{S}_0, \ldots, \mathbb{S}_{k_{\mathsf{cmpl}}}$. Additionally, this model is also used to compute multiple scenarios as required to compute the grading based robustness measure $\mathcal{P}-\mathcal{RM}$. To compute the grading of non-robust components the function $\psi(g, k)$ needs to be realized that returns the number of those scenarios up the scenario ratio λ described in Section 4.2. The function is simply realized by counting the scenarios that lead to the faulty behavior. Once the scenario ratio is reached, i.e., the number of scenarios exceeds the pre-defined ration, the computation stops.

Using this model non-robust components are classified by incrementally extracting all k-non-robust components and increasing k by one until the maximal observation window $\bar{k}$ has been explored. However, after determining k-non-robust components stored in the set $\mathbb{S}_k$, k-dangerous components are determined using the following model. Components that are already classified as k-non-robust components are not considered here anymore. That means, the remaining components need to be analyzed: $\mathbb{U}' = \mathbb{U} \setminus \mathbb{S}_k$.

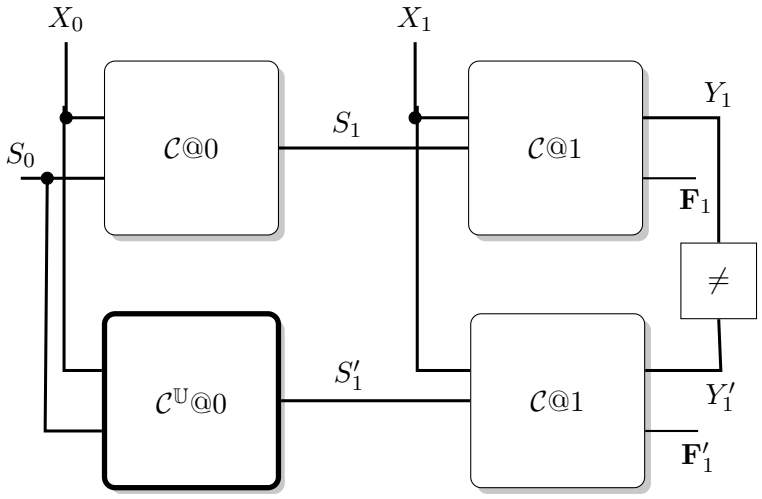


Figure 5.3: Model for classifying 1-non-robust components

5.2.2 Model for Classifying k-dangerous Components

Figure 5.4 illustrates the model for classifying 0-dangerous components. Similar as in the model for k-non-robust components the fault modeling circuit $C^{\mathbb{U}'}$ is constructed according to $\mathbb{U}'$.

The model is almost identical to the model of non-robust components with the exception that the states S_1 and S_1' are checked for difference rather than the primary outputs.

Given the set of fault predicates $P^{\mathbb{U}'} = \{p_{g_1}, \ldots, p_{g_{|\mathbb{U}'|}}\}$ and allowing only a single fault injection. Suppose the fault predicate $p_{g_i} \in P^{\mathbb{U}'}$ is activated. If there exists an scenario $\tau = ((X_0), S_0)$, such that $S_1 \neq S_1'$ holds, that means that the state is affected by the injected CTF and the fault signal $\mathbf{F}_0'$ does not report any fault the component g_i is classified as 0-dangerous. The model is denoted by $\mathcal{D}(\mathcal{C}, S^*, \mathbb{U}', 0)$ analogously to $\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, 0)$, and $\mathcal{D}(\mathcal{C}, S^*, \mathbb{U}', k)$ analogously to $\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k)$ where an observation window of size k is considered.

Both introduced models classify components against the class model EPPModel enabling to compute $\mathcal{WC}-\mathcal{RM}$ and $\mathcal{P}-\mathcal{RM}$. However, EPPModel is the most general model, the models WCModel and CombModel are easily derived by adapting the model. The WCModel is derived while considering only a single scenario. Further, the CombModel is derived by simply considering only one time frame since there are no state elements in combinational circuits.

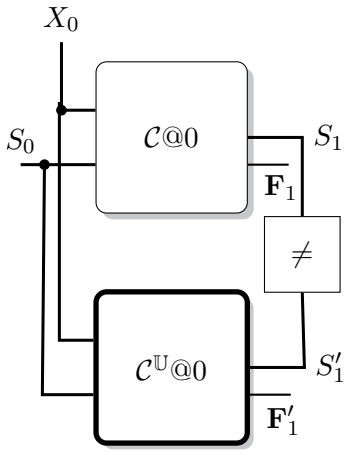


Figure 5.4: Model for classifying 0-dangerous components

5.3 Basic Algorithm

Having both models defined a basic algorithm for classifying components is presented in the following. Let $\mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k))$ and $\mathsf{sol}(\mathcal{D}(\mathcal{C}, S^*, \mathbb{U}, k))$ be two functions exactly classifying k-non-robust and k-dangerous components, respectively according to the observation window of size k. However, the concrete realization of both functions is not necessarily specified at this point since both functions are later implemented by the engines. For now, both functions are theoretical models in the following algorithm in order to provide a basic algorithm to compute the circuit's robustness unless how they are practically implemented.

Algorithm 3:
The basic algorithm to assess the circuit's robustness.

- **Input:** The circuit $\mathcal{C} = (V, E)$ and a set of components to be classified $\mathbb{U} \subseteq V$ and an maximum observation window $\bar{k} \in [0, k_{\mathsf{cmpl}}]$ are given as input.
- **Output:** The classification of the components $\mathbb{U}$ into robust, non-robust and k-dangerous components, i.e., a tuple $(\mathbb{T}, \mathbb{S}, \mathbb{D}_{\bar{k}})$ is returned.
- **Description:** The algorithm, shown in Pseudocode 3 works iteratively by incrementing the size of the observation window until $\bar{k}$

```
1  begin
2      k = 0;
3      𝕊 = 𝕋 = ∅;
4      while true do
5          𝕊_k = sol(𝒩(𝒞, S*, 𝕌, k));
6          𝕊 = 𝕊 ∪ 𝕊_k;
7          𝔻_k = sol(𝒟(𝒞, S*, 𝕌 \ 𝕊_k, k));
8          𝕋 = 𝕋 ∪ 𝕌 \ (𝕊_k ∪ 𝔻_k);
9          𝕌 = 𝕌 \ (𝕊_k ∪ 𝔻_k);
10         if k = k̄ or 𝕌 = ∅ then
11             return (𝕋, 𝕊, 𝔻_k)
12         end
13         k++;
14     end
15 end
```

Pseudocode 3: Basic classification algorithm.

and classifies for each window k-non-robust and k-dangerous components. The algorithm starts with settings $k = 0$ and the respective sets of non-robust and dangerous components are initialized to empty sets.

In Line 3 k-non-robust components are classified and are added to the entire set of non-robust components in Line 6. The k-dangerous components are classified in Line 7 where the robust components are easily derived based on Lemma 3.1 and added to the entire set of robust components in Line 8. Already classified components are in Line 9 excluded from further analysis.

The iteration stops if either the maximum observation window $\bar{k}$ is reached or there are no more components to be classified ($\mathbb{U} = \emptyset$). Finally, the components partionioned into the classes are returned.

5.4 Handling Reachability Information

So far, complete reachability information is assumed on S_0 for fault injection. But computing all reachable states is a hard problem itself. A high accuracy of the reachable states comes inherently with high computation costs.

Approximations are frequently used when an exact result is computationally too expensive. The classification of the components requires to cover all possible scenarios. In particular, covering the exact set of reachable states is a nontrivial requirement. This requirement is completely abstracted in Algorithm 5.3. However, the explicit computation of this set becomes very hard. Several methods have been proposed during the last years, e.g., [SVD08, CCK03, CMB06]. For example, BDD-based fixed-point computation is often used [CGP01]. But when considering larger systems this technique reaches its limit due to the *state explosion problem* [CGP01]. However, approximation may practically often suffice while providing exact results. There are two ways to approximate the exact set of reachable states: 1) over-approximation and 2) under-approximation. In the experiments it turned out that exploiting both approximations of reachable states is very effective in order to provide a high quality analysis within a reasonable run time.

5.4.1 Influences of Approximations

Approximations of the state space are frequently used within formal verification. But the result of the each verification step need to be properly interpreted as well as in robustness checking. Exploiting approximations in robustness checking has a direct influence on the classifications and therefore to the quality of robustness checking which needs to be investigated. These theoretical analysis is presented in the following.

Over-approximation of Reachable States

Classifying the components based on a over-approximation of the reachable states means that CTFs might be injected into states that are non-reachable. Consequently, scenarios are covered that are not possible during operation. This may lead to spurious classifications and may yield a lower value for the robustness. This is more formally emphasized as follows:

Theorem 5.1. *Let* $\hat{S}$ *be an over-approximation of the reachable states, i.e.,* $S^* \subseteq \hat{S}$*, and* $k \in [0, k_{\mathsf{cmpl}}]$*. Then it holds:* $\mathbb{S}_k \subseteq \hat{\mathbb{S}}_k$ *with* $\hat{\mathbb{S}}_k = \mathsf{sol}(\mathcal{N}(\mathcal{C}, \hat{S}, \mathbb{U}, k))$ *and* $\mathbb{S}_k = \mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k))$*.*

Proof. Sketch: Since $S^* \subseteq \hat{S}$, i.e., more states are considered for injecting and propagating of a CTF there might be a scenario that leads to a classification of a component to be k-non-robust that would not be found by considering S^*. □

That means, the classification based on an over-approximation of reachable states over-approximates k-non-robust components stored in $\mathbb{S}_k \subseteq \hat{\mathbb{S}}_k$. The components of $\hat{\mathbb{S}}_k \setminus \mathbb{S}_k$ are called *spurious* k-non-robust components since the components might be misleadingly classified.

Example 5.1. *Consider reachable states in a* Triple Modular Redundancy *(TMR) circuit consisting of three equal modules and a majority voter.*

Each of the three modules are in the same reachable state assuming that the circuit operates fault-free, i.e., $S_0 = S^*$. Therefore, all CTFs within the modules are masked by the majority voter and these components are classified as robust.

A simple over-approximation and implied consequences of the classifications are explained: Suppose all three modules can be in different states may computing therefore different outputs. A CTF within a module cannot be properly masked since the correct majority cannot be computed. Consequently, all components of the modules might be classified as spurious non-robust.

The analogous observation is presented for k-dangerous components.

Theorem 5.2. *Let $\hat{S}$ be an over-approximation of the reachable states, i.e., $S^* \subseteq \hat{S}$ and, $k \in [0, k_{\mathsf{cmpl}}]$. Then it holds: $\mathbb{D}_k \subseteq \hat{\mathbb{D}}_k$ with $\mathbb{D}_k = \mathsf{sol}(\mathcal{D}(\mathcal{C}, S^*, \mathbb{U}, k))$ and $\hat{\mathbb{D}}_k = \mathsf{sol}(\mathcal{D}(\mathcal{C}, \hat{S}, \mathbb{U}, k))$.*

Proof. Analog to proof of Theorem 5.1. □

These components are called spurious k-dangerous components and are stored in the set $\hat{\mathbb{D}}_k$.

Having both determined, an over-approximated set of k-non-robust components and an over-approximated set of k-dangerous components implies an under-approximation of the robust components. That means, using an over-approximatin of the reachable state space, classification of k-non-robust and k-dangerous components yields an under-approximation of robust components, i.e., $\mathbb{T} \subseteq \mathbb{U} \setminus (\hat{\mathbb{S}}_k \cup \hat{\mathbb{D}}_k)$. This observation has a consequence on the robustness measure. Suppose the spurious k-non-robust and spurious k-dangerous components are determined. Since a subset of robust components is computed an under-approximation of the lower bound for the robustness is given by:

$$\check{R}_{lb}^{\bar{k}} = \frac{|\hat{\mathbb{T}}|}{|\mathbb{U}|} \leq R_{lb}^{\bar{k}} \tag{5.1}$$

For practical purposes of robustness checking using an over-approximation of reachable states yields a safe lower bound for the robustness. That means, some components may be misleadingly classified as non-robust, which would be robust under exact assumptions (S^*). However, the classified robust components are definitely robust components. Refinement methods are available (e.g, [CGJ+00]) to narrow down down this problem that can also be applied for robustness checking.

Under-approximation of Reachable States

An under-approximation of the reachable states contains fewer states than the exact set of reachable states. Consequently, states might be missed for injection and propagation. For example, a function of an ALU cannot be activated since the respective state is not contained in the under-approximation. Hence, non-robust components might be classified as robust yielding an over-approximation of the upper bound for the robustness. More formally:

Theorem 5.3. *Let $\check{S}$ be an under-approximation of the reachable states with $\check{S} \subseteq S^*$ and $k \in [0, k_{\mathsf{cmpl}}]$, then it holds: $\check{\mathbb{S}}_k \subseteq \mathbb{S}_k$ with $\check{\mathbb{S}}_k = \mathsf{sol}(\mathcal{N}(\mathcal{C}, \check{S}, \mathbb{U}, k))$ and $\mathbb{S}_k = \mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k))$.*

Proof. Sketch: Since $\check{S} \subseteq S^*$, i.e., states are missed for injection and propagating of CTFs and therefore scenarios might be missed to classify a component to be non-robust. □

The under-approximation of k-non-robust components yields a safe upper bound for the robustness. Given $\check{\mathbb{S}}_k$ for the classified k-non-robust components yielding an over-approximation of the upper bound of the robustness and is given by:

$$\hat{R}^{\bar{k}}_{ub} = 1 - \frac{|\check{\mathbb{S}}_{\bar{k}}|}{|V|} \geq R^{\bar{k}}_{ub} \tag{5.2}$$

However, the quality of the computed bounds directly depends on the strength of the approximation of the reachable states. The experiments will show that relevant classes of circuits can be exactly classified even when the approximation is very coarse.

5.5 Classification by Means of Model Checking

The classification of the components can be naïvely translated into a model checking problem. Industrial-strength model checkers are highly-optimized and can handle complex circuits [Par10] such that robustness checking can

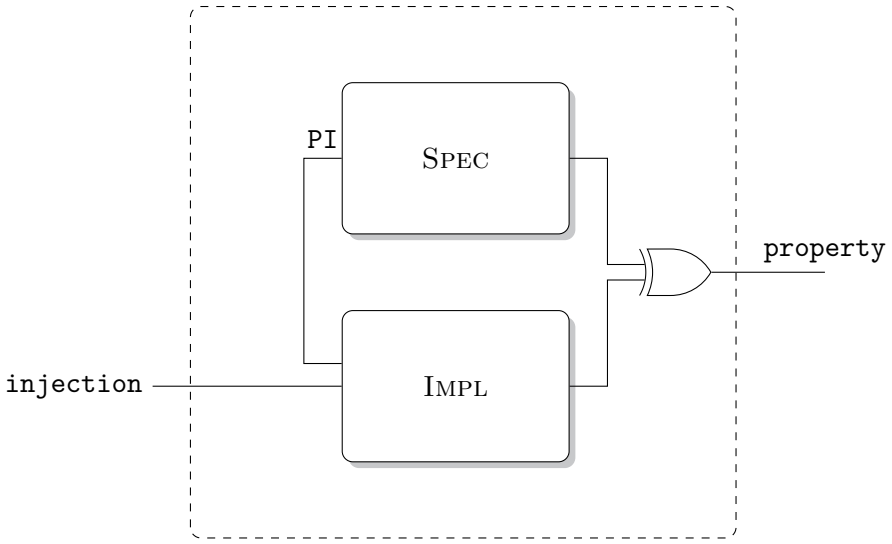


Figure 5.5: Classification based on model checking

benefit from this strength. A naïve approach to classify the components of a circuit is to translate the problem into a model checking problem by generating a circuit model with fault injection and a temporal property for each component. Consequently, $|\mathbb{U}|$ properties need to be generated and verified. For each property the model checker determines the respective solution in terms of property holds or fails which corresponds to classified into the respective class.

The implementation effort is kept very low since only the properties need to be generated instead of developing dedicated algorithms. In the following section the classification of non-robust components is translated into a model checking problem. Therefore, a model that needs to verified and a CTL property are introduced to determine non-robust components.

5.5.1 Problem Formulation

The problem formulation is very similar to that model presented in Section 5.2. Figure 5.5 depicts the problem formulation that is generated for each component. Given is a circuit $\mathcal{C} = (V, E)$ and a component $g \in V$ that has to be classified. The term Spec represents the original circuit $\mathcal{C}$ and the term Impl represents the circuit $\mathcal{C}$ with fault injection logic at component g as introduced in Section 5.1. Due to considering CTF that occurs at an arbitrary single time frame, additional logic is inserted into Impl that ensures that single CTFs are injected. The input `injection`

controls whenever a fault is injected, i.e., the time frame chosen by the model checking. If this input is set to one, a CTF at component g is injected.

Both circuits are stimulated with the same input illustrated by `PI`. The output `property` reports when at least one output pair of SPEC and IMPL becomes unequal by setting the signal to 1.

Having this construction the following CTL property is used to classify non-robust components:

$$\mathbf{AG}(\texttt{injection} \implies \mathbf{AG}\neg\texttt{property})$$

The property states that globally for all paths the injection of a CTF implies that globally for all paths the outputs are equal. If this property holds for component g the component is classified as robust. Otherwise, if the property fails the component is classified as non-robust.

5.5.2 Discussion

This approach comes with several drawbacks that is shown in long run time which is also be demonstrated by the experiments later in this thesis. However, the run time depends on the property. There may exist similar constructions that may yield better run times. But in general, domain specific knowledge about the underlying problem cannot be easily exploited. Industrial model checker are usually closed-source application and therefore adding certain features is difficult if not impossible. Exemplary, coarse approximations of the reachable state often suffice to get accurate results. Embedding those approximations is impossible if dedicated parameters of the verification tool are hidden or not avaliable.

5.6 Low-Level Optimization Techniques

Classifying the components of a circuit corresponds to solving several complex problems since the number of scenarios that need to be searched grows exponentially with the number of inputs and considered time frames. To reduce the overall computational effort optimizations are presented. Even on this level of algorithm the classification can be improved.

Exploiting structural knowledge is exploited used in formal verification to reduce the overall's verification run time, e.g., [XWMB12, BIMM12, BKA02, FSF11].

In the following a low-level technique is described that has been published in [FHD+11].

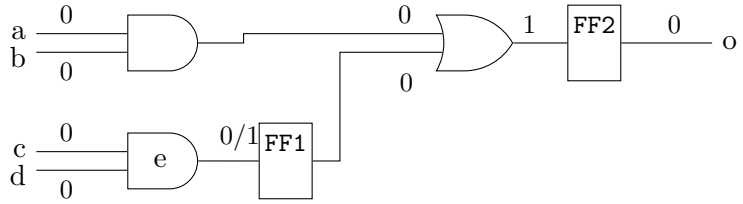


Figure 5.6: Example circuit in DAG-based representation with weighted edges

5.6.1 Shortest Path Analysis

In order to check whether a fault is observable at the primary outputs, the affect values need to be propagated over the state elements for several time frames. A light-weighted pre-process has been published in [FHD+11] and is presented below.

In Figure 5.6 a sequential circuit $\mathcal{C} = (V, E)$ is shown based on a DAG-based representation with weighted edges. Each edge $e = (v, v') \in E$ is weighted as follows: if v' is a state element then the edge is weighted with one, otherwise the edge is weighted with zero. Assume a CTF occurs at component e. To propagate the fault to the primary output o at least two time frames need to be considered since the shortest path to a primary output includes two state elements, `FF1` and `FF1`. This observation is more formally emphasized in the following.

Definition 5.1. *Given a circuit $\mathcal{C} = (V, E)$ and a component $g \in V$. The* Minimal Propagation Path MPP *of component g is the shortest path from g to a primary output with respect to all primary outputs.*

MPP can be computed for each component based on, e.g., Dijkstra's shortest path algorithms [Dij59] which can be done very fast since the complexity is $O(n \log n + m)$ where m is the number of edges and n the number of nodes of the circuit graph.

Lemma 5.1. *Given a circuit circuit $\mathcal{C} = (V, E)$ and a component $g \in V$. If the component g is k-non-robust then it holds: $k \geq \mathsf{MPP}(g)$.*

Proof. Suppose the component g is k-non-robust. To propagate a CTF at g to at least one primary output at least k time frames need to be considered otherwise the fault cannot be propagated to the outputs. □

This idea is exploited during the classification to reduce the number of problems to be solved. The most engines rely on SAT solvers thus the number of SAT calls is significantly reduced.

Recall the function $\mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k))$ from Section 5.3 that computes k-non-robust components. To reduce the search space of this function components that do no match the necessary condition of Lemma 5.1 can be excluded from the classification for the current value of k. That means, $\mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathsf{filter}_{\mathsf{MPP}}(\mathbb{U}, k), k))$ with $\mathsf{filter}_{\mathsf{MPP}}(\mathbb{U}, k) = \{g \in \mathbb{U} \mid \mathsf{MPP}(g) \leq k\}$ is called.

Chapter 6

RobuCheck - An Integrated Robustness Checker

The fundamentals for assessing the robustness of a circuit have been introduced in the previous chapters. However, the implemented engines classifying the components of a circuit have not been presented so far. In this chapter, various engines or *classifiers* showing different strengths are introduced. Integrating different engines establish a powerful verification tool. One engine may classify one circuit very effectively while another engine may take long run times and vice versa. All these engines are tightly integrated in RobuCheck [FFSD10]: a powerful push-button tool that implements an highly-optimized flow [FHD+11] that automatically assesses the robustness of a circuit. Comparable to modern model checker tools various engines are orchestrated within an unified powerful verification tool. Before introducing precisely the insights of RobuCheck the integrated engines are presented in separate sections.

Mainly three corners of robustness checking need to be addressed to effectively classify the circuit. Improving these corners will improve the overall verification run time. The more effective the approaches the better robustness checking scales well for larger circuits while providing high quality results.

- Sine computing the exact reachable states is hard, approximations are used. However, bad approximations may not yield high quality results. Consequently, a focus is on determining suitable approximations that allows to provide accurate results.

- Robustness checking is performed during the design process once a designer has implemented hardening techniques and the robustness

needs to be assessed. The performance of the classification directly influences the run time of the robustness checker and inflences therefore the outcome of the designer.

- A sparse result in terms of broad robustness bounds that was computed very fast is not useful for the designer. Those useless results are often caused by incomplete computations coming from bad approximations. That means, completeness and good approximations need to be strongly considered. Proving the absence of faulty behavior typically needs to reach the completeness thresholds. But reaching this value is often infeasible since computing this value itself is usually difficult. However, completeness can alternatively guaranteed that finally implies completeness and moreover provides exact results.

This thesis focuses mainly on engines that are based on formal methods falling back to formal reasoning engines as a SAT solver. While these formal engines provide a complete analysis, since they cover all possible scenarios, a simulation-based engine provides a fast but incomplete analysis and can handle very large circuits. However, a simulation-based engine is an essential engine for an effective flow of robustness checking since pre-processing may significantly reduce the search space which would increase the performance of the classification. Therefore, beside mainly formal-method based engines, additionally a simulation engine is presented in this chapter as well.

All engines are briefly introduced in the following and afterwards detailed introduced in the respective sections:

BMC-classifier: The first approach is based on *Bounded Model Checking* and is named as *BMC-classifier*. This classifier basically checks a series of safety properties that are dedicated to classify k-non-robust and k-dangerous components. This classifier was initially proposed in [FSFD11]. For that, properties are introduced that allow to effectively classify all components within one problem instance. Theoretically, the approach is complete once the respective completeness threshold is known.

ATPG-classifier: The second approach is introduced along SAT-based sequential ATPG and is named as *ATPG-classifier*. Sequential SAT-based ATPG is essentially adapted for robustness checking, i.e., in particular to handle CTFs. The ATPG-classifier was briefly proposed in [FSFD11]. The main difference of the ATPG-classifier to the BMC-classifier is that the components are classified separately - stepwise. That means, for each component a dedicated problem instance is

created and solved. This approach is very close to the naïve model checking approach presented in Section 5.5. But ATPG a priori shrinks the problem instance since certain parts of the circuit can be safely ignored which comes in the model checking only with significant additional costs. Consequently, this may significantly reduce the size of the problem instance and therefore the size of the search space as well.

Both engines consider the circuit theoretically over an arbitrary number of time frames and provide therefore a complete classification. However, the engines reach their limits when large circuits are considered that consist of several thousand components. The problem instances get too large and intractable to solve them efficiently by a formal reasoning engine like a SAT solver.

Abstraction and *Decomposition* techniques have been proposed for model checking in general - for BDD-based model checking as well as for SAT-based checking [GS05, CGP01, CGJ$^+$00, CLM89, EMA10, XWMB12] – to enhance the ability to verify larger circuits effectively, i.e., to overcome complexity issues. Both techniques have been proposed in [FFA$^+$12, FF10] are further improved in this thesis. Both techniques are implemented in two additional classifiers which are briefly described below.

ITP-classifier: The BMC-classifier and ATPG-classifier rely basically on unrolling the transition relation up to the completeness threshold, i.e., up to the reachability diameter of the underlying automaton (introduced in Definition 2.20 on page 24). The completeness thresholds can be very large even for smaller circuits as shown in Section 3.4. Thus, the transition relation must be unrolled several thousand times. This is practically not manageable due to limited computational resources such as run time and space. In order to tackle this problem, Craig interpolation [Cra57, McM03] is exploited in order to provide still a complete and sound classification while computing an approximate state space by interpolants. Craig interpolants were firstly exploited in McMillan's work of SAT-based model checking with interpolation [McM03]. This approach is effectively able to prove properties on industrial-sized circuits in particular without unrolling the transition relation up to the completeness threshold. Interpolants are used to automatically derive an approximation of the state space while abstracting irrelevant facts to prove the respective property. However, this kind of model checking is adapted for robustness checking. Furthermore, a step beyond McMillan's approach, a fixed-computation

using interpolation on the property is introduced allowing for a complete classification for unbounded observation windows as well as a new kind of computing approximations using interpolations. This allows to classify unbounded dangerous components.

COMP-classifier: Applying robustness checking on, e.g., complex processor designs, composed of several high level modules, suffers from very long run times. Since the complexity significantly increases for complex designs other methods are required. A *compositional approach* [FF10] named as *COMP-classifier* is introduced in this thesis that reduces the problem into smaller sub-problems. This classifier considers suitable subcircuits of a circuit in a separate problem instance. All components of a subcircuit are locally classified and eventually composite with the entire circuit if necessary. Since classifying the components locally on subcircuits results in smaller problem instances that consequently speeds up the classification significantly because the surrounding logic is ignored. The approach is very powerful for certain kinds of circuits but strongly depends on the choice of the subcircuit. Guidelines of choosing suitable subcircuits are provided.

SIM-classifier: Furthermore, a simulation-based approach named as *SIM-classifier* is introduced as well. The SIM-classifier performs random simulation and randomly injects CTFs. This classifier is not only used as a pre-processing step but is also tightly integrated in the classification process of the formal engines. A common simulation-based engine is adapted to robustness checking in terms of the fault model and component model.

After a precise introduction of all classifiers the differences are discussed. Due to the diversity in terms of different manner of classification a wide range of circuits can be effectively classified. Further, the classifiers are different in terms of time and space complexity presented in a separate section. In the last section of this chapter, the highly-optimized flow of RobuCheck is presented.

6.1 BMC-classifier

A theoretical algorithm for classifying the components of a circuit has been presented in the previous chapter in Section 5.3 but serves only as theoretical model. In this section, a formal methods based classifier leant on *Bounded Model Checking* (BMC) named as *BMC-classifier* is introduced.

The basic idea of this classifier is to check two kinds of safety properties covering the computational model to classify k-non-robust and k-dangerous components. BMC is then used to determine solutions of the BMC formula using a SAT solver. More precisely, it is checked whether the properties hold on the circuit that corresponds to whether components are k-non-robust or k-dangerous. Finally, as already introduced in the basic algorithm the robust components are easily derived from the classification of k-non-robust and k-dangerous components.

At first, the problem formulation, i.e., the two properties are introduced and translated in a series of SAT problems from an instantiated BMC formula. After introducing the properties, a universal algorithm is presented to classify k-non-robust and k-dangerous components, respectively.

In general, the BMC-classifier is complete, i.e., all components are completely classified. In the experiments it turned out that the BMC-classifier is very effective for relevant classes of circuits. However, in practice the BMC-classifier cannot cover all reachable states since the computation of those states is computationally very expensive. Due to the limited computational resources certain scenarios might be missed and the circuit may only partially classified. However, in order to overcome those complexity issues approximation techniques for the reachable state space are introduced. Moreover, at the end of this section, characteristics of the approach are discussed.

6.1.1 Problem Formulation

BMC is adapted for robustness checking in the following. Two kinds of properties (formulas) are introduced: 1) a property for k-non-robust components and 2) a property for k-dangerous components. In the context of BMC the formulas can be seen as safety properties which are later explained more elaborated. Reconsider the general BMC formula introduced in Section 2.4.1:

$$\mathsf{BMC}(l) = I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1}) \wedge P(s_l)$$

where I is a predicate describing the initial state, T is the transition relation, and P is the negated property. If the formula is satisfiable for any l then there is a path of length l from an initial state s_0 to state s_l that satisfies the negated property P (violates the desired specification) and therefore the circuit is buggy.

To perform robustness checking using BMC a formula for P is required: a property P has to be defined which injects CTFs, propagates the affected

states, and forces the difference of the primary outputs and states, respectively. Remember, faults on k-non-robust components are observable at the primary outputs and faults on k-dangerous components corrupt the states after k time frames. The two properties classifying k-non-robust and k-dangerous components are introduced in the following that cover the notion of the computational model presented in Section 5.2. Finally, an algorithm is introduced that classifies the components with respect to the introduced properties. This algorithm can be embedded into the theoretical algorithm presented in Section 5.3 resulting in a practically useful classifier.

Property for k-non-robust Components

The property for classifying k-non-robust components is divided into three parts: 1) injection, 2) propagation, and 3) difference of primary outputs. The property is denoted by $P_N(\mathbb{U}, l, k)$ where $\mathbb{U} \subseteq V$ is the set of components that needs to be classified, l describes an index of states for fault injection (number of steps from the initial state), and k is the number of steps from injection into state s_l to the time frame $l+k$ where the primary outputs are checked. The entire number of steps computed by the property corresponds to the size of the observation window, i.e., k exactly describes the size of the observation window and is also used in this way.

Part 1 In the first part fault injection is performed: As described before in Section 5.1 the circuit is copied and logic to inject CTFs is introduced at the components of $\mathbb{U}$, i.e., the fault modeling circuit $C^{\mathbb{U}}$ with new inputs $P^{\mathbb{U}}$ is obtained. The fault modeling basically follows the construction from SAT-based debugging [SVAV05] where values are injected to determine possibly faulty gates in a buggy circuit. However, in this modeling arbitrary values are particularly injected as faults. The set $P^{\mathbb{U}}$ contains all fault predicates enabling the fault injections. Since single faults are considered, exactly one arbitrary predicate is activated. This constraint is given by ϕ_{one} where

$$\phi_{\text{one}} \Leftrightarrow \sum_{p \in P^{\mathbb{U}}} p = 1.$$

After modeling both circuits $\mathcal{C}$ and $\mathcal{C}^{\mathbb{U}}$ the FSMs with transition relations T and $T^{\mathbb{U}}$ are derived, respectively. The formula to inject CTFs is then given by:

$$\mathsf{inj}(\mathbb{U}, l) \Leftrightarrow T(s_l, s_{l+1}) \wedge T^{\mathbb{U}}(s_l, s'_{l+1}) \wedge \phi_{\text{one}}$$

Both transition relations are stimulated by the same input omitted here to keep the formulas simple. The fault injection is done into the same state s_l

representing the fault injection state. The successor states s_{l+1} and s'_{l+1} might be unequal since a fault is injected. Consequently, the formula inj corresponds to the model from Section 5.2.1 unrolled for one time frame.

Part 2 The second part propagates the correct and affected states over $k-1$ time frames after fault injection:

$$\mathsf{prop}(l,k) \Leftrightarrow \bigwedge_{i=l+1}^{l+k-1} T(s_i, s_{i+1}) \wedge T(s'_i, s'_{i+1})$$

Both equal transitions relations are stimulated by the same inputs as well and perform a transition from a current state to a successor state. Note, the original transition T relation is taken for propagation instead of $T^{\mathbb{U}}$ since a fault is only injected into one time frame.

Part 3 The third part defines the constraint particularly for determining k-non-robust components. The following formula forces the primary outputs to be different in the last time frame $l+k$:

$$N(l,k) \Leftrightarrow Y_{l+k} \neq Y'_{l+k}$$

The conjunction of these three parts leads to:

$$P_N(\mathbb{U}, l, k) = \mathsf{inj}(\mathbb{U}, l) \wedge \mathsf{prop}(l,k) \wedge N(l,k) \tag{6.1}$$

The state s_l for injecting a fault is not constrained in the formula and can be arbitrarily chosen. But k-non-robust components are only classified based on reachable states. Thus, states on s_l must be reachable states only. This is realized while a BMC problem with the introduced property $P_N(\mathbb{U}, l, k)$ is created. The instantiated BMC problem denoted by NBMC is given by:

$$\mathsf{NBMC}(\mathbb{U}, l, k) = \overbrace{I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1})}^{\text{prefix}} \wedge \overbrace{P_N(\mathbb{U}, l, k)}^{\text{suffix}} \tag{6.2}$$

In each circuit instance for fault-free and faulty computation the fault signal **F** is forced to be zero since scenarios are computed without flagging the fault signal, i.e., when faults are not reported. In order to keep the formulas simple this constraint is omitted.

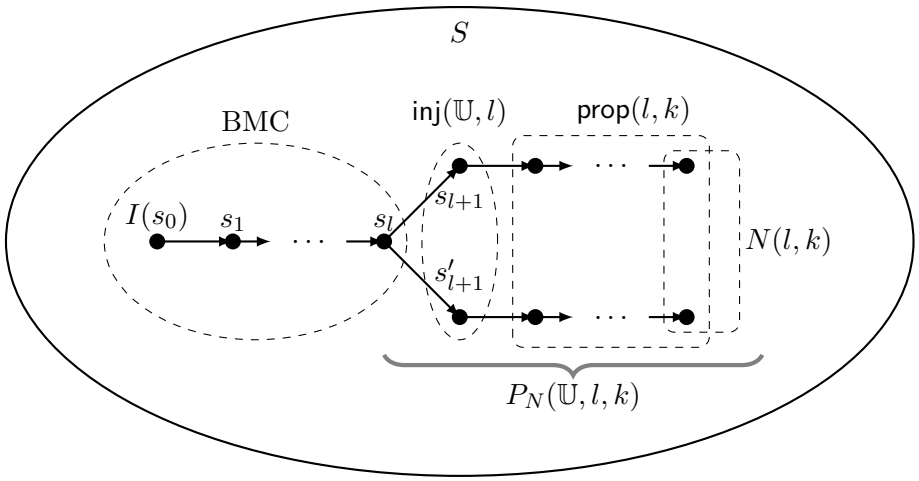


Figure 6.1: Illustration of formula $\mathsf{NBMC}_l(\mathbb{U}, k)$

The *prefix* of the formula computes states reachable from the initial state $I(s_0)$ to the state s_l in l steps which corresponds to l time frames. Therefore, the parameter l adjusts how deep in the state space the fault is injected and plays an important role when considering completeness aspects later in this chapter.

The intention of the formula NBMC is illustrated in Figure 6.1. The big circle denotes the entire state space S. The dashed objects represent the prefix of the NBMC formula and the introduced parts before:

1. BMC marks the path from the initial state to a reachable state in l steps,
2. $\mathsf{inj}(\mathbb{U}, l)$ performs the fault injection into the state s_l. After fault injection, the states may differ illustrated by divergent arrows,
3. $\mathsf{prop}(l, k)$ propagates the possibly different states over $k - 1$ time frames, and finally,
4. $N(l, k)$ forces the primary outputs computed after k steps to be different

A SAT solver is used to determine solutions for the formula that directly corresponds to k-non-robust components of the instantiated NBMC for any $l \in [0, l_{\mathsf{cmpl}}]$ and $k \in [0, k_{\mathsf{cmpl}}]$. As introduced in Section 2.4.1 the parameter l_{cmpl} is the completeness threshold that ensures that all reachable states are covered. The $\mathsf{NBMC}(\mathbb{U}, l, k)$ formula is translated into a CNF $\Phi_N(\mathbb{U}, l, k)$ and is then checked for satisfiability.

Lemma 6.1. *The CNF $\Phi_N(\mathbb{U}, l, k)$ is satisfiable if and only if there is a component $g \in \mathbb{U}$ that is k-non-robust for some l and k.*

If Φ_N is satisfiable the satisfying assignment is extracted in particular the assignments of the fault predicates $P^{\mathbb{U}}$. Suppose the fault predicate $p_g \in P^{\mathbb{U}}$ has been activated by the SAT solver, i.e., a CTF at component g has been injected. Thus, the respective component g is classified as k-non-robust. By determining all solutions of the CNF according to the fault predicates $P^{\mathbb{U}}$ all k-non-robust components are determined with respect to l and k and are stored in the set $\mathbb{S}_k^l$.

Enumerating all k-non-robust components is easily realized by iteratively calling the SAT solver for determining a solution. If a solution is found, i.e., the CNF is satisfiable, the activated fault predicate $p_g \in P^{\mathbb{U}}$ is extracted, a unit clause $\{\neg p_g\}$ is added to the CNF, i.e., $\Phi_N = \Phi_N \cup \{\neg p_g\}$, and Φ_N is again checked by the SAT solver. This prevents the SAT solver to deliver the same solution again. Eventually, the CNF formula becomes unsatisfiable. Thus, all k-non-robust components have been determined with respect to the BMC instance adjusted by l and k - the set $\mathbb{S}_k^l$ is entirely computed. If not all components of $\mathbb{U}$ are classified the value of l is increased by one and components are further determined until l_{cmpl} is reached. These steps are performed for each $k \in [0, k_{\mathsf{cmpl}}]$.

Property for Dangerous Components

This section introduces the property for determining k-dangerous components denoted by P_D analogously to P_N. The formula consists of three parts as well as the property for k-non-robust components. However, the difference between classifying k-dangerous components and k-non-robust components is that in the last time frame the states are forced to be different instead of the primary outputs. That means, it is checked whether a CTF corrupts the states after k time frames. Thus, only the third part is different and is given by:

$$D(l, k) \Leftrightarrow s_{l+k} \neq s'_{l+k}$$

which forces the state to be different rather then the primary outputs. The conjunction of all formulas $\mathsf{inj}(\mathbb{U}, l)$, $\mathsf{prop}(l, k)$ and $D_(l, k)$ yields the property for k-dangerous components:

$$P_D(\mathbb{U}, l, k) = \mathsf{inj}(\mathbb{U}, l) \wedge \mathsf{prop}(l, k) \wedge D(l, k)$$

and the BMC problem denoted by DBMC is given by:

$$\mathsf{DBMC}(\mathbb{U}, l, k) = \overbrace{I(s_0) \wedge \bigwedge_{0 \leq i < l} T(x_i, s_i, s_{i+1}, y_i)}^{\text{prefix}} \wedge \overbrace{P_D(\mathbb{U}, l, k)}^{\text{suffix}} \tag{6.3}$$

This formula is translated into a CNF $\Phi_D(\mathbb{U}, l, k)$ and is checked for satisfiability by a SAT solver where solutions accordingly correspond to k-dangerous components.

Lemma 6.2. *The CNF $\Phi'(\mathbb{U}, l, k)$ is satisfiable if and only if there is a component $\mathbb{U}$ that is k-dangerous for some l and k.*

Determining all k-dangerous components is performed as well as for the k-non-robust components. The result is stored in set $\mathbb{D}_k^l$.

In both properties, the parameter k exactly specifies the size of the observation window as introduced in Section 5.2 about classification. The parameter l comes from the BMC formula that specifies the number of unrolled time frames after the initial state. Intuitively, l describes how deep in the states space a CTF is injected. To provide a complete classification with respect to a fixed value of k, all reachable states have to be checked for the state s_l - the injection state. This can be achieved by checking all $l \in [0, l_{\mathsf{cmpl}}]$ where l_{cmpl} is the completeness threshold introduced in Section 2.4.1 implemented by the algorithm in the next section.

6.1.2 Algorithm

In Section 5.3 a basic algorithm has been presented that uses two functions virtually to classify the components: 1) $\mathsf{sol}(\mathcal{N}(\mathcal{C}, S^*, \mathbb{U}, k))$ to classify k-non-robust, and 2) $\mathsf{sol}(\mathcal{D}(\mathcal{C}, S^*, \mathbb{U}, k))$ to classify k-dangerous components with respect to the exact state space S^* with $\mathbb{U} \subseteq V$ components to be classified and $\mathcal{C} = (V, E)$ the circuit under verification. The algorithm introduced in this section provides an implementation for both functions to completely classify the components. That means, both abstract functions can be replaced by the new classifier in order to provide a real implementation.

The classification is performed with respect to a fixed size of observation window, i.e., $k \in [0, k_{\mathsf{cmpl}}]$. Basically, the algorithm determines k-non-robust and k-dangerous components, respectively, as long as not all reachable states are considered, i.e., $l \leq l_{\mathsf{cmpl}}$, and at least one component remains to be classified.

Algorithm 4:

```
1  begin
2      l = 0;
3      while 𝕌 ≠ ∅ ∧ l ≤ l_cmpl do
4          Φ = toCNF(class(𝕌, l, k));
5          A_k^l = ∅;
6          while SAT?(Φ) do
7              m = model(Φ);
8              forall p_g ∈ P^𝕌 do
9                  if m[p_g] then
10                     A_k^l = A_k^l ∪ {g};
11                     Φ = Φ ∪ {¬p_g};
12             end
13         end
14         𝕌 = 𝕌 \ A_k^l;
15         l = l + 1;
16     end
17     return A_k^0 ∪ A_k^1 ∪ ... ∪ A_k^l
18 end
```

Pseudocode 4: BMC-classifier.

BMC-classifier.

- **Input:** A circuit $\mathcal{C} = (V, E)$ under analysis, a set of components to be classified $\mathbb{U} \subseteq V$, and the size of the observation window $k \in [0, k_{\mathsf{cmpl}}]$ are given as input. Further, the parameter class specifying the formula used for classification, i.e., $\mathsf{class} \in \{\mathsf{NBMC}, \mathsf{DBMC}\}$ to classify k-non-robust or k-dangerous components, respectively.
- **Output:** The set of k-non-robust S_k or k-dangerous $\mathbb{D}_k$ components is returned, respectively.
- **Description:** The code shown in Pseudocode 4 and is explained in the following. The algorithm starts with setting $l = 0$. The outer `while`-loop from Line 3 to Line 16 iterates as long as there is at least one component to be classified and the completeness threshold is not yet reached, i.e., not all reachable states have been discovered. In each iteration the CNF of the respective formula class based on the current value of l and the remaining components to classify $\mathbb{U}$ is constructed. Note, k is an input parameter and is fix during the entire run of this algorithm. The set A_k^l, that stores classified components according to l and k, is initialized with the empty set in Line 5. In the `while`-loop from Lines 6 to 13 all solutions for the CNF and therefore classified components are determined. That means, if the CNF Φ is satisfiable, the model is extracted in Line 7. Next, the `for`-loop searches for the activated fault predicate p_g (Line 9). The respective component g is added to the set of classified components A_l^k and a unit clause $\{\neg p_g\}$ is added to the CNF Φ to block this solution. Eventually, Φ becomes unsatisfiable, i.e., all components with respect to l and k are classified and the inner loop ends. All determined components are excluded from further classifications (Line 14). Finally, the value of l is incremented by one and the outer iteration restarts. At the end, the classification terminates if either all components are classified or the completeness threshold has been reached. All classified components are returned (Line 17).

Given a fixed k, the algorithm (BMC-classifier) is first called using the formula for determining k-non-robust components (NBMC) and afterwards

with the remaining components as presented in the basic classification algorithm to classify k-dangerous components (DBMC) from Section 5.3. Details regarding the implementation and efficiency issues are presented at the end of this section.

6.1.3 Completeness

Once the completeness threshold l_{cmpl} is reached for a given property, it is safely concluded that all reachable states and therefore all possible scenarios have been explored as introduced in Section 2.4.1.

However, in general the completeness threshold might be very large and the BMC-classifier may iterate several thousand times and therefore calls the SAT solver several thousand times which is not efficient for large circuits. But, completeness is theoretically guaranteed.

Approximation techniques overcome this problem for a wide range of relevant circuit instances by providing still exact computation. The embedding of those approximations into the BMC-classifier is presented in the next section.

An alternative to a common SAT-based model checker might be a BDD-based approach. A BDD-based approach could be used in order to compute the exact reachable states as it is commonly done in *Symbolic Model Checking* (SMC) [CMCHG96]. SMC iteratively computes symbolically the image of the transition relation until a fixed-point is reached that finally ensures that all reachable states are covered. However, even when the final BDD is compact, during the iteration the BDD might be very large consisting of hundreds of thousands nodes which requires a huge amount of memory making BDDs for large circuits unmanageable. The work of [FSFD11] makes use of BDD-based fixed-point computation in robustness checking and shows that exploiting approximations results in much better performance.

However, there are several approximation techniques that can be embedded into the BMC-classifier. The basis for embedding those approximations are presented in the next section. The concrete realization of the approximation is transparent to the approach. Data from a *testbench* or constrained random-simulation can be used here as well.

6.1.4 Embedding Reachability Information

Once the completeness threshold l_{cmpl} is known for a circuit a complete classification with respect to a observation window k is provided. However, computing this value is as hard as model checking itself [CKOS04].

Various techniques on top of BMC have been proposed in order to overcome this problem, see, e.g., [SVD08, CCK03, CMB06]. However, all

exact techniques have high run times while several approximation techniques provide a trade-off between accuracy and run time. Even when an approximation is used exact results can be obtained as shown later in the experimental evaluation. Handling approximate reachability information within the BMC-classifier is presented in the following.

While in Section 5.4 the theoretical influence of the approximations used in robustness checking has been investigated this section embeds approximations into the BMC-based approach.

Over-approximation

Assume $\hat{\omega}(s_0)$ is a predicate over the state variable s_0 describing a set of states with $\forall s \in S^* \implies \hat{\omega}(s)$, i.e., $\hat{\omega}$ computes an over-approximation of the set of the reachable states S^*. This predicate is embedded into the formulas for determining k-non-robust and k-dangerous components by simply replacing the prefix from the BMC instance by the predicate:

$$\hat{\mathsf{NBMC}}(\mathbb{U}, k, \hat{\omega}) = \hat{\omega}(s_0) \wedge P_N(\mathbb{U}, 0, k) \tag{6.4}$$

$$\hat{\mathsf{DBMC}}(\mathbb{U}, k, \hat{\omega}) = \hat{\omega}(s_0) \wedge P_D(\mathbb{U}, 0, k) \tag{6.5}$$

Both formulas depend on the state variable s_0. Therefore, l is constantly set to 0. Based on the influence of approximations described in Section 5.4 both formulas determine an over-approximation of k-non-robust components stored in the set $\hat{\mathbb{S}}_k$ and k-dangerous components stored in the set $\hat{\mathbb{D}}_k$, respectively since $\hat{\omega}$ is an over-approximation of reachable states and may contain states that are not reachable from the initial state in any number of steps.

Exemplary, a very simple over-approximation is to construct a predicate as $\omega(s\_1) = 1$, i.e., all states are allowed as reachable states.

Under-approximation

In contrast to over-approximation, assume $\check{\omega}(s_0)$ is a predicate over the state variable s_0 and it holds: $\forall s \in S.\check{\omega}(s) \implies s \in S^*$, that means $\check{\omega}$ is an under-approximation of the reachable states S^*. The formula for determining k-non-robust is analogously to the over-approximation given by:

$$\check{\mathsf{NBMC}}(\mathbb{U}, k, \check{\omega}) = \check{\omega}(s_0) \wedge P_N(\mathbb{U}, 0, k)$$

States might be missed for fault injection where a fault cannot be properly propagated to the primary outputs certain parts of the circuit are not

properly sensitized. Consequently, the solutions of the formula under-approximates k-non-robust components.

Those under-approximations can be computed very differently. Considering simply a certain number of transitions from the initial state computes an under-approximation as introduced in [FSD09].

Definition 6.1. *Given a* Reachability Window Parameter *(RWP)* $\bar{l} \in [0, l_{\mathsf{cmpl}}]$ *the predicate* $\mathsf{SPB}(\bar{l})$ *describes any state along any path reachable from the initial state up to a length of* $\bar{l}$ *with:*

$$\mathsf{SPB}(\bar{l}) = I(s_{-\bar{l}}) \wedge \bigwedge_{-\bar{l} \leq i < 0} I(s_i) \vee T(s_i, s_{i+1})$$

The approximation is called States along any Bounded Path *(SBP).*

The parameter $\bar{l}$ influences certainly the space of reachable states considered as states for fault injection and therefore the accuracy of the classification. In SPB the higher the value of $\bar{l}$ potentially more states are discovered and more components are properly classified.

Once the classification is completed for a certain $\bar{l} \ll l_{\mathsf{cmpl}}$, i.e., all components are classified an under-approximation is sufficient to fully classify the circuit without considering the entire state space. As it will be shown later in the experiments, checking a small number of various reachability windows is sufficient to fully classify relevant circuit classes.

6.1.5 Incremental Satisfiability

The BMC-classifier relies on numerous satisfiability checks and therefore depends inherently on the SAT solver's performance to solve a CNF. In a naïve way for each single classification the problem formulation needs to be translated into CNF and is then checked by a SAT solver. However, this can be significantly improved by the technique proposed by [Sht01] where incremental satisfiability for BMC is introduced. This technique is adapted for the BMC-classifier as well.

During the solve process today's SAT solvers generate conflict clauses pruning the search space which increases the performance considerably. However, *learnt information* in terms of conflict clauses can be partially transfered to similar SAT instances by keeping the current instance and add only new necessary facts. In particular, when the BMC-classifier increases the parameter l by one only the clauses of the new unrolled time frame are required to be added to the solver instead of rebuilding the entire formula. Learnt information about the previous run are kept within the SAT solver's internal databases. Besides saving the run time of rebuilding

the entire formula mainly exploiting the learnt information improves the performance as also shown in [Sht01]. This kind of computation is known as *incremental satisfiability*. Overall, instead of resolving the entire CNF from one classification to a further classification, learnt information is exploited by keeping the SAT solver's database during the entire classification.

6.2 ATPG-classifier

The BMC-classifier presented in the previous section handles all components in one monolithic problem instance. In contrast, the ATPG-classifier considers one component within one problem instance as it is similar done on sequential SAT-based ATPG.

SAT-based sequential ATPG computes test patterns for stuck-at faults at the circuit's signals. This approach is adapted for robustness checking. While classical ATPG engines consider stuck-at faults the ATPG-classifier for robustness checking needs to handle CTFs. Further, ATPG is often reduced to the combinational case resulting in over testing but reduces the problem instances significantly. Therefore, ATPG for robustness checking is inherently harder to solve since reachability information need to be properly taken into account. Moreover, problem instances are usually harder to solve since the underlying circuits contain many redundant logic to tolerate transient faults. This kind of instances that are unsatisfiable are mostly hard for the SAT solvers. In contrast, the problem instances of ATPG are usually satisfiable instances since most of the faults are testable which corresponds to satisfiable instances.

6.2.1 Problem Formulation

Technically, the ATPG-classifier is very close to the BMC-classifier with the exception that only a single component is considered in a single problem instance. To classify all components of a circuit, for each component a separate problem instance is created and solved.

Since only one component is considered in a single problem instance parts of the logic can be safely ignored as it is done in classical ATPG as well. A *Cone-Of-Influence* (COI) reduction is applied that determines logic that influences the analysis.

Figure 6.2 illustrates the general idea which logic is relevant when classifying a component. The fanout cone and the transitive fanin cone of the affected component need to be modeled for a single time frame. The remaining logic can be safely ignored. This reduces the size of the problem instances significantly. Consequently, classifying a single component may

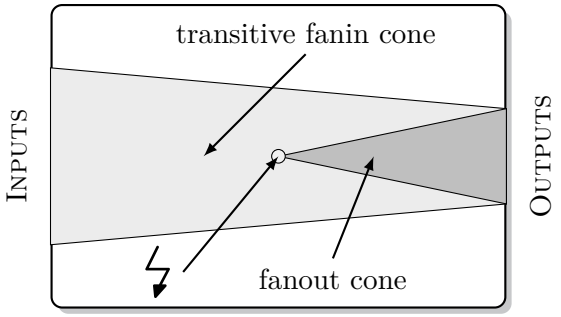


Figure 6.2: Fanout and transitive fanin cone of an affected component

perform much faster than the monolithic model of the BMC-classifier since the search space is accordingly smaller.

6.2.2 Using ATPG to compute EPP

Computing a differentiation of non-robust components requires to compute more than a single scenario according to the techniques introduced in Section 4.2. Recall the parameter λ limits the number of scenarios to computed in order to overcome complexity issues.

ATPG is suitable to compute more than a single scenario. While COI is applied certain inputs are not relevant and are considered as *don't care* inputs which significantly reduces the run time.

Further improvements are achieved by performing *Minimal Assignment Analysis* (MAA) as for example introduced in [RS04, ED07]. Those techniques can be used to reduce the overall number of SAT calls which consequently reduces the overall run time.

6.2.3 Comparison to Blackbox Model Checker

This classifier is very close to the naïve model checking approach introduced in Section 5.5. This approach converts the model and the CTL property into a problem instance. That means, similar as in the ATPG approach each component is analyzed separately.

However, ATPG-classifier can exploit more problem domain knowledge as for example the COI reduction is applied. The model checker has no knowledge about the component to be classified and perform reductions of the problem instance without this specific knowledge. Consequently, reductions are usually less effective in the same time as required by COI in ATPG. Of course, COI can also be performed before translation the

classification into the a model checking problem. But in the worst case the entire circuit needs to be modeled such that COI is obsolete.

6.3 ITP-classifier

Completeness of robustness checking can be guaranteed by both proposed classifiers - the BMC-classifier and the ATPG-classifier - from the previous sections by checking all necessary combinations of $l \in [0, l_{\mathsf{cmpl}}]$ and $k \in [0, k_{\mathsf{cmpl}}]$. Based on Lemma 3.3 about the completeness threshold the number of those combinations may increase exponentially with the size of the state elements of a circuit. Consequently, checking the number of combinations, where each check implies a model checking problem, is infeasible even for smaller circuits. Up to small values for l_{cmpl} and k_{cmpl} both engines are very effective, in particular for those circuits where a bounded observation window is sufficient and approximate reachability information leads to high accuracy. But providing classifiers that effectively conclude completeness while avoiding to check all combinations even when the completeness thresholds l_{cmpl} and k_{cmpl} are large, is necessary for the remaining hard to classify circuits.

In practice, often it suffices to partially check all combinations since some states or scenarios might not be relevant to classify a component into the respective class. The classifier named as *ITP-classifier* introduced in this section and published in [FFA+12] automatically determines which combinations are necessary such that finally an exact classification is performed. Consequently, an alternative termination criterion is provided by exploiting interpolations.

The approach of McMillan's approach of interpolation-based model checking has been presented in Section 2.4.2. Interpolation adapted for robustness checking has been firstly presented in [FFA+12] which is detailed presented in this section. Moreover, a new model checker is presented in thesis based on a new approximation technique.

Three new techniques based on interpolation are introduced to conclude completeness before checking all combinations in this thesis. Combining all these techniques within the ITP-classifier, provides an fully automatic approach to derive suitable over-approximations by interpolation. The basic ideas of the three techniques are briefly listed below and detailed explained in separate sections.

1. The classical SAT-based interpolation procedure as proposed by McMillan [McM03] is adapted for robustness checking. That means,

for each property determining k-non-robust and k-dangerous components with respect to a certain observation window $\bar{k}$, the interpolation procedure is started that guarantees that all necessary states for injection have been discovered. Recall the parameter l specifies the number of time frames after the initial state firstly introduced in Section 6.1. The integrated approach of the ITP-classifier avoids to check all values for $l \in [0, l_{\mathsf{cmpl}}]$ by over-approximating the reachable state space. This provides an effective and complete classification with respect to a bounded observation window as motivated in Section 3.5 for certain circuit classes.

2. The approach of McMillan computes an over-approximation of the state space by joining a set of interpolants. The computation of those interpolants follows a certain partitioning of the underlying BMC formula into part A and B. A newly introduced partitioning allows to compute a single interpolant that over-approximates the reachable states instead of few interpolants.

   Once McMillan's interpolants lead to a too strong over-approximation showed by spurious counterexamples the interpolants need to be completely discarded and the computation restarts. The new interpolants proposed in this thesis can be reused even when the approximation is too coarse. Joining all interpolants using the new technique lead to more and more accurate approximation. Eventually, the new interpolants converge to the exact reachable states.

   Beside the application in robustness checking in this thesis the new approximation contributes to general model checking as well. Additionally, a simple model checker integrates this new techniques is introduced in this section as well.

3. Finally, providing completeness with respect to an unbounded observation window requires to consider all necessary values of $k \in [0, k_{\mathsf{cmpl}}]$, i.e., all propagation paths need to be discovered.

   Due to the high computational cost of naïvely checking all values, an over-approximation of the propagation path is introduced. A derived fixed-point computation on the property may potentially guarantee completeness without checking all values up to k_{cmpl}. Technically, interpolation is performed directly on the property, i.e., fault-free and faulty computations are over-approximated. This certain kind of interpolation requires that at least all reachable states are considered for fault injection adequately provided by both previously mentioned techniques (1.& 2.).

In this section the ITP-classifier is introduced consisting of three separate parts. Each part is introduced in a separate section. Finally, the ITP-classifier is explained including all those three parts in terms of a algorithm.

6.3.1 Adaption of Interpolation-based Model Checking

The first classifier exploiting interpolation is introduced that simply adapts interpolation-based model checking from [McM03] for robustness checking. For a bounded observation window the new interpolation classifier introduced in this section effectively ensures that all reachable states are discovered for fault injection before potentially reaching the completeness threshold l_{cmpl}, i.e., leading to a complete classification with respect to a bounded observation window.

Interpolants compute an over-approximation by abstracting *facts* in terms of parts of the state space that are irrelevant for the current reasoning. This may lead to a fast convergence to a fixed-point than explicitly unrolling the transition relation up to the completeness threshold.

A safety property holds on a circuit if it is proven that the property holds for all reachable states of the circuit's automaton. BMC has been introduced to verify the property on states by iteratively unrolling the transition relation. All reachable states are checked when the transition relation is unrolled l_{cmpl} times. This may become very expensive for larger circuits. Interpolation-based model checking [McM03] often terminates before reaching this completeness threshold which significantly reduces the verification time. Interpolants abstract some facts which are irrelevant to prove the property and therefore speeds up the convergence to a fixed-point. Therefore, in order to prove a property only relevant states are considered.

In robustness checking a series of safety properties are checked introduced in the BMC-classifier in Section 6.1. More precisely, for each $k \in [0, \bar{k}]$, a new property needs to be checked to classify k-non-robust and k-dangerous components. Mapping robustness checking to interpolation-based model checking avoids considering all values of $l \in [0, l_{\mathsf{cmpl}}]$ for each $k \in [0, \bar{k}]$ where $\bar{k}$ is the maximum size of the observation window setted by default to k_{cmpl} (Section 3.5). That means, for each k an earlier termination might be reached instead of completely unrolling the transition relation up to l_{cmpl} times which significantly improves the run times.

In the following, interpolation-based model checking [McM03] is adapted for robustness checking based on the BMC-based engine.

Problem Formulation

Recall the BMC-classifier and the underlying formula $\mathsf{NBMC}(\mathbb{U}, l, k)$ as introduced in Section 6.1:

$$\mathsf{NBMC}(\mathbb{U}, l, k) = I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1}) \wedge P_N(\mathbb{U}, l, k)$$

The property $P_N(\mathbb{U}, l, k)$ is forced to inject faults into state s_l on the components of $\mathbb{U} \subseteq V$, propagates the faults over k time frames and finally forces the primary outputs to be different. Consider the following partition (A, B) of the formula $\mathsf{NBMC}(\mathbb{U}, l, k)$:

$$A := I(s_0) \wedge T(s_0, s_1)$$
$$B := \bigwedge_{i=2}^{l} T(s_{i-1}, s_i) \wedge P_N(\mathbb{U}, l, k)$$

Suppose all k-non-robust components have been determined with respect to the values of l and k and only the remaining components $g \in \mathbb{U}$ not shown to be non-robust (at least one, i.e.,$|\mathbb{U}| \geq 1$) are considered to classify further. In this case, $A \wedge B$ is unsatisfiable, since no fault injection is observable within k time frames. One of the two following reasons leads to the unsatisfiable formula:

1. The formula B is unsatisfiable itself. That means, each fault injection does not entirely depend on the set of states for injection, i.e., in any case the fault is either unobservable at the primary outputs or is reported by the fault signal. Consequently, B is unsatisfiable, since B exactly constrains the part of fault injection, propagation, and forces unequal primary outputs. In this case, determining the interpolant is skipped and all components are classified with respect k, since more states are not necessary to classify the remaining components.

2. Otherwise A and B are satisfiable, respectively, an interpolant σ with $\sigma = \mathsf{itp}(A, B)$ is computed. The interpolant σ is defined over the state variables expressed by s_1, i.e, the common variables of A and B with $\mathsf{Var}(\sigma) \subseteq \mathsf{Var}(A) \cap \mathsf{Var}(B)$. The interpolant is an over-approximation of the image of the transition relation, i.e., the states of the first transition are over-approximated ($A \implies \sigma$) but still contains sufficient facts to contradict the property ($B \wedge \sigma$ is unsatisfiable). That means, faults injected into the states from the

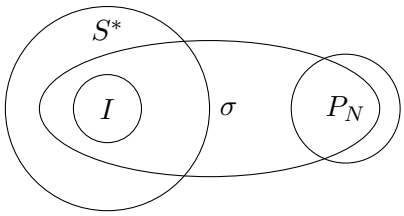


Figure 6.3: Over-approximation of the interpolants leading to spurious counterexamples

over- approximated image represented by the interpolant are either unobservable or reported by the fault signal.

However, the interpolant is added to A such that $A = (I(s_0) \vee \sigma(s_0)) \wedge T(s_0, s_1)$ where the variable s_1 of σ is replaced by s_0 and the procedure restarts. This virtually unrolls the transition relation.

For each interpolant it is checked whether a fixed-point is reached. A fixed-point is reached when the disjunction of all previously computed interpolants implies the new interpolant, i.e., $\sigma_1 \vee \ldots \vee \sigma_n \implies \sigma$ where $\sigma_1, \ldots, \sigma_n$ are previously computed interpolants and σ is the new interpolant. Once the disjunction becomes true at least all reachable states has been covered since the disjunction of the interpolants and the initial state yield an over-approximation of the reachable state space, i.e., $\forall s \in S^* \,.\, (I \vee \sigma \vee \ldots \vee \sigma_n)(s)$ is true. The classification is complete with respect to k.

Once the instance $A \wedge B$ becomes satisfiable where A is extended by the interpolant a probably spuriously classified non-robust component has been determined due to the over-approximation by the interpolant. States for injection are beside reachable states also non-reachable states and therefore the classification might be spurious. This case is shown in Figure 6.3 where the interpolant σ intersects states with the property P_N.

In this case the procedure restarts by increasing the value of l, which either allows for classifying non-robust components based on reachable states only or makes the interpolants weaker by strengthening B, i.e., abstracting fewer facts.

Finally, a fixed-point will be eventually found and the classification is complete with respect to the current value of k [McM03]. After reaching a fixed-point, k is increased by one up to $\bar{k}$ and the procedure restarts while discarding all interpolants.

```
1  begin
2      S_k = ∅;
3      l = 0;
4      while U \ S_k ≠ ∅ ∧ l ≤ l_cmpl do
5          S_k = S_k ∪ sol(NBMC(U, l, k));
6          if l == 0 then l = l + 1, continue;
7          R = I;
8          B := ⋀_{i=2}^{l} T(s_{i−1}, s_i) ∧ P_N(U \ S_k, l, k);
9          while !SAT?(A ∧ B) do
10             A := R(s_0) ∧ T(s_0, s_1);
11             σ = itp(A, B);
12             if σ ⟹ R then
13                 return S_k
14             end
15             R = R ∨ σ;
16         end
17         l = l + 1;
18     end
19 end
```

Pseudocode 5: Interpolation-model checking adapted for robustness checking.

Algorithm

In this section an algorithm adapting the previously described procedure is introduced. This algorithm states the first part of the ITP-classifier denoted by *ITP-classifier-1*.

Algorithm 5:
ITP-classifier-1.

- **Input:** A circuit $\mathcal{C} = (V, E)$, set of components to be classified $\mathbb{U} \subseteq V$, and a fixed $k \in [0, \bar{k}]$ are given as input.
- **Output:** The set of k-non-robust components are returned.
- **Description:** The code shown in Pseudocode 5 and is described in the following. The algorithm starts with initializing the set of k-non-robust classified components to the empty set and starts with $l = 0$. The `while`-loop from Line 4 to Line 18 iterates as long as at least one component needs to be classified and the completeness threshold l_{cmpl} is not reached. At first k-non-robust components with respect to the current value of l are classified and added to the set $\mathbb{S}_k$. The interpolation procedure starts when at least l is set to 1. In case of $l = 0$ the output loop restarts with an incremented l (Line 6).

  When $l \leq 1$ the interpolation procedure starts by construction the set $R = I$ to be the first set states and B a fixed part of the partition. The `while`-loop from Line 9 to Line 16 computes interpolants and checks whether a fixed-point is reached. The loop terminates once a spurious classification is performed.

  In Line 10 the partition A is created and an interpolant of (A, B) is computed. If a fixed-point is found, i.e., if the condition in Line 12 evaluates to true, then the set of all k-non-robust components are determined and returned.

  Otherwise, the interpolant is added to the previously computed interpolants in Line 15.

  If the loop terminates at Line 17 a spurious classification is performed and in this case l is incremented and the outer loop restarts.

6.3.2 Adequate Over-Approximation

ITP-classifier-1 computes an over-approximation of the reachable states based on interpolation by partitioning the NBMC into the formulas A and B accordingly. The new approach computes an over-approximation based on

interpolation by an opposite partitioning. This over-approximation is then used in robustness checking and a separate model checker - both introduced later in this section.

The effectiveness of robustness checking strongly depends on the states considered while fault injection. An automatically determined and suitable over-approximation leads to an effective classification.

Several properties of new approach are presented later in this section covering the strength and weaknesses in comparison to McMillan's interpolation approach.

Computing an Over-approximation

Initially, a new term to distinguish different kinds of approximations of reachable states is introduced.

Definition 6.2. *Given a transition system $M = (I, S, T)$ and a predicate δ defined over the state variables of M. Then δ is called* adequate approximation *if δ contains only non-reachable states, i.e., $\forall s \in S^* \,.\, \delta(s) = 0$.*

Once an adequate approximation δ is computed an over-approximation of the reachable states has been obtained, i.e., $\bar{\delta}$ computes an over-approximation since it holds: $\forall s \in S^* \,.\, \bar{\delta}(s) = 1$.

In the following an interpolation-based approach is presented to compute adequate approximations matching the previous definition. Reconsider the BMC formula from page 26 of Section 2.4.1

$$\mathsf{BMC}(l) = I(s_0) \wedge \bigwedge_{0 \leq i < l} T(s_i, s_{i+1}) \wedge P(s_l)$$

with a safety property P that has to be verified with respect to a circuit. Further, reconsider the state approximation $\mathsf{SPB}(\bar{l})$ from Section 6.1.4 that computes any state along any path reachable from the initial state in $\bar{l}$ number of steps. This formula is used to check the property as follows:

$$\mathsf{BMC}_{\mathsf{reach}}(\bar{l}) = \mathsf{SPB}(\bar{l}) \wedge P(s_0)$$

The difference to the classical BMC formula is that intuitively all states up to $\bar{l}$ steps are constrained for fault injection rather than only states reachable in $\bar{l}$ steps.

Suppose the formula is unsatisfiable for any $\bar{l} \in [0, l_{\mathsf{cmpl}}]$, i.e., no state reachable from the initial state along any path of length $\bar{l}$ intersects the property states. An interpolant from the following partition (A, B):

$$A = P(s_0) \quad \text{and} \quad B = \mathsf{SBP}(\bar{l}) \tag{6.6}$$

is computed. Let $\delta = \mathsf{itp}(A, B)$ be an interpolant and reconsider the Theorem 2.14 of Craig interpolation. The interpolant δ computes states that are not reachable from the initial states in $\leq l$ steps since $B \wedge \delta$ is unsatisfiable. But the interpolant contains states that fulfill the property. Those states might be non-reachable states or reachable states in more than l steps since $A \implies \delta$. However, since δ may not contain exclusively non-reachable states δ might not be an adequate approximation since the property may fail in general. In theory at least one adequate approximation exists but in practice often more than one adequate approximation can be obtained.

Theorem 6.1. *Given a transition system $M = (I, S, T)$ and a property that holds on the circuit. There exists an $\bar{l} \in [0, l_{\mathsf{cmpl}}]$ with $\mathsf{BMC}_{\mathsf{reach}}(\bar{l})$ is unsatifiable then $\delta = \mathsf{itp}(A, B)$ is an adequate approximation.*

Proof. Setting $\bar{l} = l_{\mathsf{cmpl}}$ then $\mathsf{SBP}(\bar{l})$ models all reachable states and $\mathsf{BMC}_{\mathsf{reach}}(\bar{l})$ is unsatisfiable since the property holds. An interpolant $\delta = \mathsf{itp}(A, B)$ is computed. Formula B models all reachable states and $\delta \wedge B$ is unsatisfiable therefore δ contains only non-reachable states since $A \implies \delta$. □

Thereby, it is proven that an adequate approximation is computed when $\bar{l}$ is set to l_{cmpl}. However, in practice an adequate approximation is often computed when $\bar{l}$ is much lower than l_{cmpl}. In order to verify that an interpolant is an adequate approximation, model checking is performed by simply checking whether $\bar{\delta}$ is an invariant. As the experiments show, this check can be done within very low run times.

Once, an adequate approximation has been obtained this approximation can be used in robustness checking as well as in model checking.

Algorithm

A separate algorithm presented in Algorithm 6.3.2 determines adequate approximation denoted as *ITP-classifier-2*.

Algorithm 6:
Compute adequate approximation ADQ.

- **Input:** The algorithm gets a transition system $M = (I, S, T)$, a property P that has to be verified, and fixed value $l \in [0, l_{\mathsf{cmpl}}]$ as inputs.

```
1  begin
2      A = P(s_0);
3      B = BMC_reach(l̄);
4      if SAT?(A ∧ B) then
5          return ∅;
6      δ = itp(A, B);
7      if δ̄ is invariant on M then
8          return δ
9      return ∅;
10 end
```

Pseudocode 6: Computing adequate approximations: ADQ.

- **Output:** As a result either an empty set or an adequate approximation is returned.

- **Description:** The code is shown in Pseudocode 6. At first, the algorithm creates the partition (A, B) accordingly and initially check whether $A \wedge B$ is satisfiable (Line 4). In case of satisfiability the empty set is returned. Otherwise, an interpolant is computed and checked whether the negation of the interpolant is invariant using a black-box model checker. If an adequate approximation has been obtained the interpolant is returned otherwise the empty set is returned.

The algorithm determines adequate approximations that can be used in model checking to verify a property. As already introduced, BMC is used to verify a circuit with respect to a property. However, completeness is guaranteed when reaching the completeness threshold ensures that all reachable states has been covered. However, if a property holds on at least all reachable states and some non-reachable states then the circuit has been successfully verified with respect to the property. Moreover, if the property holds on all reachable states and some non-reachable states the property holds also on the circuit. The general idea is to exploit the adequate approximation in model checking. Once the property holds on all states of the adequate approximation the property does hold on the circuit since at least all reachable states has been discovered.

Consider the following adapted BMC formula where $\Delta = \{\delta_1, \ldots, \delta_n\}$ is a set of adequate approximations.

$$\hat{\mathsf{BMC}}(\Delta) = \bigwedge_{\delta \in \Delta} \bar{\delta}(s_0) \wedge P(s_0) \tag{6.7}$$

If the formula is unsatisfiable the property holds on the circuit since the property holds on at least all reachable states. In contrast, if the formula is satisfiable the property might be falsified on reachable states or non-reachable states. However, further refinement checks are required, i.e., computing more adequate approximations. However, this is not further investigated in this work. Several well established techniques can be applied in context of *Counterexample-Guided Abstraction Refinement* (CEGAR) [CGJ+00] to overcome this issue.

6.3.3 Model Checking with Adequate Approximations

BMC proves a property by unrolling the transition relation up to the completeness threshold. Reaching this value ensures that all reachable states has been discovered. This is usually computational expensive. In contrast, to find a bug, i.e., disproving a property, it often suffices to reach a small values that is pratically easier to solve.

The following model checking works similar as a bounded model checker. While checking whether the current states intersects the property states a new steps that checks whether the property holds on an over-approximation is added.

A simple model checker named as SimpMC that exploits adequate approximation is presented in this section.

Algorithm 7:
SimpMC - A Model Checker that exploits adequate approximations.

- **Input:** The algorithm gets a transition system $M = (I, S, T)$, a property P that has to be verified.
- **Output:** The algorithm either returns `TRUE` meaning there there is a counterexamples or returns `FALSE` meaning that the property holds on the circuit.
- **Description:** The code of the algorithm is shown in Pseudocode 7. The algorithm starts with initializing the set of adequate approximations Δ to the empty set.

```
1  begin
2      Δ = ∅;
3      foreach l ∈ [0, l_cmpl] do
4          if SAT?(BMC(l)) then
5              return TRUE;
6          end
7          δ = ADQ(M, P, l);
8          if δ ≠ ∅ then
9              Δ = Δ ∪ {δ};
10         if !SAT?(B̂MC(Δ)) then
11             return FALSE;
12     end
13     return FALSE;
14 end
```

Pseudocode 7: SimpMC exploiting adequate approximations.

The outer `foreach`-loop from Line 3 to Line 12 iterates over all values $l \in [0, l_{\mathsf{cmpl}}]$. In Line 4 it is checked whether there is a counterexample when unrolling l time frames. If the formula is satisfiable, then there is a counterexample and consequently `TRUE` is returned.

Otherwise, if the formula is unsatisfiable there is no path of length l from the initial state to the property state. Thus, the algorithm of computing adequate approximations ADQ from Algorithm 6.3.2 is called. If the algorithm obtained an adequate approximation, i.e., $\delta \neq \emptyset$, the adequate approximation is added to the set Δ.

In Line 10 it is checked whether the property holds on the over-approximation of the reachable states represented by the adequate approximations. If the property holds, `FALSE` is returned. Otherwise, the loop restarts.

Eventually, the loop terminates which means that there is no counterexample of length $l \in [0, l_{\mathsf{cmpl}}]$. Consequently, the property holds on the circuit and `TRUE` is returned.

In the worst case the algorithm iterates until l_{cmpl} is reached. However, in many cases SimpMC terminates before, while proving the property based

on adequate approximation. Consequently, SimpMC might be an additional piece of an formal verification framework.

SimpMC is completely new in this thesis and has not yet been published. The model checker is evaluated later in this thesis against a state-of-the-art model checker.

6.3.4 Classification with Adequate Approximations

In context of robustness checking adequate approximation can be exploited as well. In Section 6.1 the BMC-classifier has been introduced. Moreover, embedding approximations into robustness checking are also introduced in Section 6.1.4. Reconsider the Formulas 6.4 from Section 6.1.4:

$$\hat{\mathsf{NBMC}}(\mathbb{U}, k, \hat{\omega}) = \hat{\omega}(s_0) \wedge P_N(\mathbb{U}, 0, k)$$
$$\hat{\mathsf{DBMC}}(\mathbb{U}, k, \hat{\omega}) = \hat{\omega}(s_0) \wedge P_D(\mathbb{U}, 0, k)$$

where $\hat{\omega}$ represents an arbitrary over-approximation. In order to embed a set of adequate approximation with $\Delta = \{\delta_1, \ldots, \delta_n\}$ the formula $\hat{\omega}$ is constrained as follows:

$$\hat{\omega}(s_0) = \bigwedge_{\delta \in \Delta} \bar{\delta}(s_0)$$

The adequate approximation can be used to over-approximate k-non-robust and k-dangerous components and finally to derive a subset of robust components as introduced in Section 5.4.

The better the approximations the more exact is the computation of k-non-robust and k-dangerous components and finally the subset of robust components. Eventually, the computation of adequate approximation yields the exact set of reachable states and consequently the classification is always complete.

6.3.5 Proving Unbounded Dangerous Components

There are certain conditions that a fault becomes not observable at the primary outputs for any subsequent time frame but corrupts the states, i.e., the fault persists in circuit's states. This behavior was introduced under the term *unbounded dangerous* in Section 3.3. Components that lead to this behavior are hard to classify since all possible propagation paths need to be discovered to exclude that no fault becomes observable. That means, technically that an observation window up to the completeness threshold

k_{cmpl} needs to be analyzed. But this is practically impossible due to the exponentially increasing search space for each additional time frame. However, alternative solutions are required to overcome this problem.

A proof procedure classifying components that are either robust or unbounded dangerous based on interpolation is introduced in the following. This proof procedure potentially avoids to consider an observation window up to k_{cmpl} which significantly reduces the run times. As already exploited in the previous approach over-approximation is used in this approach as well. To avoid to unroll the transition relation up to k_{cmpl} the behavior of fault-free and faulty computation are over-approximated. Once this over-approximated behavior leads to a fixed-point it is proven that the considered components are either robust or unbounded dangerous.

Problem Formulation

The basic idea is to perform a fixed-point computation on the property to classify non-robust components based on interpolation. The interpolants compute an over-approximation of the fault-free and faulty computation. The computation may terminate earlier than considering the completeness threshold k_{cmpl} once a fixed-point is reached.

Reconsider the formula to classify k-non-robust components $\mathsf{NB\hat{M}C}(\mathbb{U}, k, \hat{\omega})$ for not yet classified components $\mathbb{U} \subseteq V$, $\mathbb{U} \neq \emptyset$, any $k \in [1, k_{\mathsf{cmpl}}]$ and any over-approximation $\hat{\omega}$.

A partition (A, B) is defined as follows:

$$A := \bigwedge_{\delta \in \Delta} \bar{\delta}(s_0) \wedge T(s_0, s_1) \wedge T^{\mathbb{U}}(s_0, s_1') \wedge \phi_{\text{one}} \wedge \mathsf{prop}(0, 1)$$
$$B := \mathsf{prop}(1, k) \wedge N(0, k)$$

The formula A performs fault injection in components of $\mathbb{U}$ into states constrained by the over-approximations of Δ and propagates the fault-free and faulty computation for one additional time frame. The formula B propagates the remaining $k-1$ time frames and forces the primary outputs to be different in the last time frame.

Suppose all solutions of the formula has been determined and at least one component needs to be classified, i.e., $\mathbb{U} \neq \emptyset$. Consequently, $A \wedge B$ is unsatisfiable and an interpolant is computed with $\hat{I} = \mathsf{itp}(A, B)$. The interpolant $\hat{I}$ is defined over the common variables s_1 and s_1', i.e., over states of the fault-free and faulty computation and it holds $A \implies \hat{I}$ and $B \wedge \hat{I}$ is unsatisfiable based on Definition 2.14 of Craig interpolants. However, the interpolant computes an over-approximation of the fault-free and faulty computation. Conjoining this interpolant $\hat{I}$ to the formula A

```
1  begin
2      A = ω̂ ∧ T(s_0, s_1) ∧ T^𝕌(s_0, s'_1) ∧ φ_one ∧ prop(0, 1);
3      A' = prop(0, 1);
4      B = prop(2, k) ∧ N(0, k);
5      while !SAT?(A ∧ A' ∧ B) do
6          δ = itp(A ∧ A', B);
7          if δ ⟹ A then
8              return 𝕌
9          end
10         A = A ∨ δ;
11     end
12     return ∅;
13 end
```

Pseudocode 8: ITP-classifier 3.

yields $A' = A \vee \hat{I}(s_1, s'_1)$. Once an interpolant is computed a fixed-point test is performed by checking whether the disjunction of the previously computed interpolants implies the new interpolant. In this case a fixed-point is reached and the components of $\mathbb{U}$ are considered as unbounded dangerous $\mathbb{D}_{k_{\mathsf{cmpl}}} \subseteq \mathbb{U}$. If the formula becomes satisfiable the procedure restarts while incrementing the value of k and discarding all interpolants. The reason why the instance becomes satisfiable varies: 1) either a real fault on at least one component becomes observable, 2) the approximation of $\hat{\omega}$ might be to coarse, or 2) the computed interpolants $\hat{I}$ are to coarse. In every case the procedure is restarted while k is incremented. A detailed algorithm follows in the next section.

Algorithm

An algorithm that performs fixed-point computation on the property is introduced in the following:

Algorithm 8:
Fixed-point computation on the property: FIXEDPROP.

- **Input:** The algorithm gets a transition system $M = (I, S, T)$, and a set of components that have to be proven as unbounded dangerous or

robust. Moreover, an arbitrary over-approximation $\hat{\omega}$ is given as input.

- **Output:** The algorithm returns either the set of components that are unbounded dangerous or robust, or the algorithm returns the empty set which means that the approximations are too weak or at least one component is non-robust.

- **Description:** The code of the algorithm is shown in Pseudocode 8 and described in the following.

  The algorithm starts with initializing the dedicated partitions as introduced in the previous section. The `while`-loop iterates as long as the approximation on the property does not get do coarse, i.e., until the formula becomes satisfiable. In the loop, an interpolant is computed. In Line 7, it is checked whether a fixed-point is reached. If a fixed-point is found, the components of the set of $\mathbb{U}$ are classified as unbounded dangerous or robust components which is returned in Line 8. Otherwise, if the no fixed-point is found, the approximation is joined to the set A that virtually unrolled the transition relation. Once the approximation gets to coarse, i.e., the loop terminates and an empty set is returned.

6.3.6 Complete Algorithm of the ITP-classifier

All three parts that forms the ITP-classifier has been presented in the previous sections. In this section an algorithm that integrates all three parts that provides a complete and effective classifier is presented.

Algorithm 9:
ITP-classifier

- **Input:** The algorithm gets a transition system $M = (I, S, T)$ derived from circuit $\mathcal{C} = (V, E)$, and a set of components to be classified $\mathbb{U} \subseteq V$.

- **Output:** The algorithm returns non-robust and robust components, respectively.

begin
  $k = l = 0$;
  $\mathbb{S} = \mathbb{D} = \mathbb{T} = \Delta = \emptyset$;
  while $\mathbb{U} \neq \emptyset$ **do**
    $\mathbb{S} = \mathsf{sol}(\mathsf{NBMC}(\mathbb{U}, l, k))$;
    $\mathbb{U} = \mathbb{U} \setminus \mathbb{S}$;
    if $\mathbb{U} = \emptyset \vee (l = l_{\mathsf{cmpl}} \wedge k = k_{\mathsf{cmpl}})$ **then**
      return $(\mathbb{T}, \mathbb{S})$
    $\delta = \mathrm{ADQ}(M, P_N(\mathbb{U}, l)$;
    if $\delta \neq \emptyset$ **then**
      $\Delta = \Delta \cup \{\delta\}$;
    end
    if *necessary states are covered with respect to* k **then**
      $l = 0$;
    else
      $l = l + 1$;
      continue;
    end
    $\hat{\mathbb{S}}_k^l = \mathsf{sol}(\hat{\mathsf{NBMC}}(\mathbb{U}, l, k))$;
    $\hat{\mathbb{D}}_k^l = \mathsf{sol}(\hat{\mathsf{DBMC}}(\mathbb{U} \setminus \hat{\mathbb{S}}_k^l, l, k))$;
    $\check{\mathbb{T}}_k^l = \mathbb{U} \setminus (\hat{\mathbb{S}}_k^l \cup \hat{\mathbb{D}}_k^k)$;
    $\mathbb{U} = \mathbb{U} \setminus \check{\mathbb{T}}_k^l$;
    $\mathbb{T} = \mathbb{T} \cup \check{\mathbb{T}}_k^l$;
    if $\mathbb{U} = \emptyset$ **then return** $(\mathbb{T}, \mathbb{S})$;
    if $k = 0$ **then**
      $k = 1$;
      continue;
    end
    $W = \textsc{FixedProp}(M, \hat{\mathbb{D}}_k^l, \bigwedge_{\delta \in \Delta} \bar{\delta}(s_0))$;
    $\mathbb{T} = \mathbb{T} \cup W$;
    $k = k + 1$;
  end
end

Pseudocode 9: ITP-classifier.

- **Description:** The code of the algorithm is shown in Pseudocode 9 and described in the following.

  The algorithm starts with initializing required indices l and k to be zero, and set of classification and adeqaute approximations to be empty.

  The `while`-loop iterates as long as not all components are classified. At first step of the loop, non-robust components are classified with respect to l and k. This classified components are excluded from further classification in Line 6. Furthremore, in Line 7 it is checked whether components are not yet classified or the respective completeness thresholds are reached. In this case, all complete analysis has been performed and the respective sets of classification are returned.

  Otherwise, an adequate approximation is determined with respect to k. If an adequate approximation has been determined, this approximation is added to the entire set of adequate approximations Δ.

  Line 13 checks whether all necessary are covered with respect to the current value of k. This is realized by the ITP-classifier-1 but the detailed pseudocode is ommitted here. If all necessary states are covered the algorithm proceeds with Line 19. Otherwise, l need to be incremented by one to cover more states.

  Line 19 to Line 23 determine an over-approximation of k-non-robust and k-dangerous components that finally implies a subset of robust components. The subset of robust components are added to the entire set of robust components. If no more components need to be classified robust and non-robust are returned in Line 24.

  In the remaining code, the fixed-point computation on the property for the potentially unbounded dangerous components $\hat{\mathbb{D}}_k^l$ is performed. This is technically possible if k is greater than 0 which is checked in Line 25.

  If k is greater than 0, the algorithm FixedPoint is called with the transition relation M, the potential dangerous components $\hat{\mathbb{D}}_k^l$, and an over-approximation determined by the adequate approximations.

  This algorithm either returns a set of unbounded dangerous or robust components or an empty set. However, this set is added to the entire set of robust components.

At the end the computation for the current value of k is complete since all necessary states are discovered. Consequently, k is increamented by one and the `while`-loop iterates.

6.4 COMP-classifier

When a problem instance gets too large, partitioning is done in various fields in computer science. In formal hardware verification the search space increases often exponentially with respect to the size of the input. For example the *state explosion problem* in symbolic model checking prevents the verification of larger systems to being effective. *Divide-and-conquer*-based methods are available to verify larger system. Examplarily [BCM$^+$90, CGJ$^+$00, CLM89] successfully applied this techinques.

The engines introduced so far consider the entire circuit within one monolithic problem instance. In this section a *compositional* approach is introduced. Basically, a set of subcircuits of a circuit is given and the classification is locally performed on the subcircuits and if necessary the classification is composite with the entire circuit. A new notion of non-robust components is introduced since the classification locally on the subcircuit may not hold on the entire circuit. Consequently, it is required to validate the classification with respect to the entire circuit, performed by additional checks - *justification check* and *propagation check*. The validation can be performed differently, e.g., simulation yields approximate results and formal-based techniques yield exact results, respectively, but are computational expensive. Based on this approach, the *COMP-classifier* results which has been published in the author's work [FF10].

The compositional approach considers combinational circuits. The classification locally on the subcircuit can be performed by any engine that at least classifies the circuit with respect to the CombModel. In fact, the BMC-classifier, ATPG-classifier, and ITP-classifier can be effectively used to perform the local classification.

However, the performance of the COMP-classifier depends strongly on the choice of the subcircuits. An idea of how choosing good subcircuits are presented later in this section.

6.4.1 General Approach

Given a combinational circuit $\mathcal{C}$ divided into a set of not necessarily disjoint subcircuits $P = (\mathcal{C}_1, \ldots, \mathcal{C}_n)$ with $\bigcup_{\mathcal{C}_i \in P} \mathcal{C}_i = \mathcal{C}$. The components of the

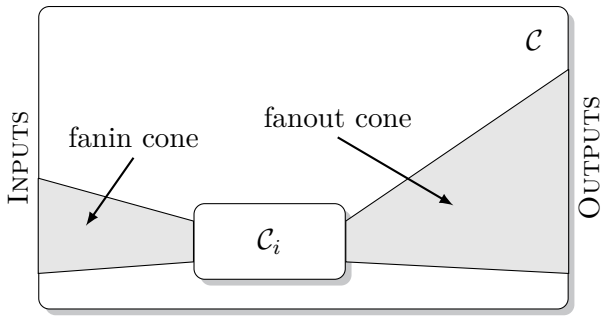


Figure 6.4: General idea of locally classifying subcircuits

circuit $\mathcal{C}$ are classified by two steps for each subcircuit $\mathcal{C}_i \in P$: 1) locally perform the classification of the components of $\mathcal{C}_i$ by any classifier that classifies components with respect to CombModel, and 2) finally composite the result of the classification while considering the entire circuit. In Figure 6.4 the basic idea is illustrated. Consider the subcircuit $\mathcal{C}_i$ which is locally classified by any classifier while omitting the fanin cone and fanout cone. Therefore, the problem instance of classifying $\mathcal{C}_i$ is reduced according the size of the subcircuit. Consequently, the local classification may perform much faster but may require additional checks.

6.4.2 Local Classification

The compositional approach considers combinational circuits. Therefore, the class model CombModel is applied. Hence, there are only non-robust, or robust components to be classified. Due to the local classification on the subcircuit, a new notion of non-robust components is necessarily introduced since surrounding logic of the subcircuit is ignored.

Definition 6.3. *Let $C_i = (V_i, E_i)$ with $\mathcal{C}_i \in P$ be the current subcircuit to be analyzed. A component $g \in V_i$ is called* locally non-robust *if g is classified as non-robust on the subcircuit C_i.*

This new kind of locally classified components is analogously introduced for robust components as follows:

Definition 6.4. *Let $C_i = (V_i, E_i)$ with $\mathcal{C}_i \in P$ be the current subcircuit to analyzed. A component $g \in V_i$ is called* locally robust *if g is classified as robust on the subcircuit C_i.*

Locally non-robust components $g \in V_i$ are stored in the set $\mathbb{S}_{\mathcal{C}_i}$ where locally robust components $g \in V_i$ are stored in the set $\mathbb{T}_{\mathcal{C}_i}$ and it holds $\mathbb{S}_{\mathcal{C}_i} \cup \mathbb{T}_{\mathcal{C}_i} = V_i$.

The robustness of a subcircuit $\mathcal{C}_i$ is given by:

$$R_{\mathcal{C}_i} = \frac{|\mathbb{T}_{\mathcal{C}_i}|}{|V_i|}$$

which corresponds to the robustness measure for combinational circuits from Section 4.1.

However, if the circuit is equipped with a fault signal $\mathbf{F}$ a certain condition for this signal must hold for the compositional approach in order to exploit the following lemma.

Locally reported faults by fault signal $\mathbf{F}_i \in V_i$ in subcircuit $\mathcal{C}_i$ must be reported by the fault signal $\mathbf{F} \in V$ at the entire circuit $\mathcal{C}$ as well. This assumption is named as *fault signal implication* and can be verified by a model checker. Based on this, the following lemma provides a powerful property that can be easily exploited that reduces the run times significantly.

Lemma 6.3. *Given a component $g \in V_i$ of subcircuit $\mathcal{C}_i = (V_i, E_i)$. If g is locally robust then g is also robust under the* fault signal implication*. Thus, $g \in \mathbb{T}_i \implies g \in \mathbb{T}$.*

Proof. If there is no scenario and no CTF that leads to faulty behavior at the primary outputs and fault signal does not repot any fault on the subcircuit, then there is also no scenario and no CTF at the entire circuit. □

Components that are locally robust must not be further analyzed which lowering the run time since additional checks are not necessary. Once all components are locally classified a lower bound for the robustness is provided by:

$$R_{lb}^{\text{comp}} = \frac{\sum\limits_{C_i \in P} |\mathbb{T}_i|}{|V|} \leq \frac{|\mathbb{T}|}{|V|}$$

where $\mathbb{T}$ is the set of robust components determined by analyzing the entire circuit.

In contrast, the classification of locally non-robust components needs to be further validated on the circuit performed by composing partially relevant logic parts of the circuit as introduced in the following section.

6.4.3 Composite Classification

Since the classification locally on the subcircuits ignores the surrounding logic, each classification of non-robust components needs to be composite

with the entire circuit. Scenarios that show faulty behavior at the subcircuit level may not be possible at the entire circuit level. However, the respective scenarios need to be validated against the entire circuit in terms of whether the scenarios and the faulty output are justifiable and observable at the entire circuit's outputs, respectively. This validation is divided into two checks: 1) check for *justification* of the scenarios, and 2) check for *propagation* of the faulty output. Once the checks are performed it is accordingly decided whether the component is non-robust or robust based on the entire circuit. Both checks are introduced in the following.

Justification

Each classification of a locally non-robust component $g \in \mathcal{C}_i$ comes with a set of pairs consisting of a scenario and a faulty output:

$$T_g = \{(X_1, Y_1), \ldots, (X_m, Y_m)\},$$

where X_i is an assignment to the subcircuit's inputs and Y_i is an assignment of the subcircuit's outputs. The checker for justification denoted as $\mathsf{j-checker}$ returns a subset of pairs where the inputs of the subcircuit is justifiable within the entire circuit:

$$\mathsf{j-checker}(T_g, \mathcal{C}) := \{(X_i, Y_i) \in T_g \mid X_i \text{ is justifiable in } \mathcal{C}\} \subseteq T_g$$

However, if the checker determines that none of the scenarios can be justified the respective component g is classified as robust. Otherwise, that means if there is at least one scenario that is justifiable the propagation of the faulty output needs to be checked.

Propagation

Given a set of faulty outputs according to the justfiable inputs with $T'_g = \mathsf{j-checker}(T_g, \mathcal{C})$. The propagation checker denoted by $\mathsf{p-checker}(T'_g, \mathcal{C}) \in \{False, True\}$ returns whether a faulty output is observable at the circuit's primary outputs as follows:

$$\mathsf{p-checker}(T'_g, \mathcal{C}) = \begin{cases} \text{FALSE} & \text{if } Y_i \text{ are not observable } \forall (X_i, Y_i) \in T'_g \\ \text{TRUE} & \text{otherwise} \end{cases}$$

Overall, if finally the $\mathsf{p-checker}$ returns FALSE for the component g, then g is classified as robust. Otherwise, the $\mathsf{p-checker}$ returns TRUE, i.e., faulty behavior is observable, the component is classified as non-robust.

6.4.4 Flow of Validation

There are several opportunities of how the j−checker and the p−checker can be realized and how they interact. Two types of interactions are presented:

One possible interaction is that the j−checker and the p−checker interact incrementally. That means, once a component has been classified as locally non-robust, a single scenario and one faulty output has been obtained that needs to be checked for justification. Instead of enumerating all those scenarios and faulty outputs, the single scenario is first checked for justification using the j−checker. If the scenario is justifiable, the corresponding single faulty output is checked for propagation using the p−checker. If the faulty output is observable at the primary output on the circuit the respective component is classified as non-robust. Otherwise, of the faulty output is not observable the check for justification repeats with the next scenario and fault output.

However, in the best case a single component can be classified as non-robust without enumerating and checking all scenarios and faulty outputs.

The more general interaction is to first compute a set of a certain size of scenarios during the classification and perform the validation on this set. The effectiveness of the approach depends on the circuit and the respective partitioning.

6.4.5 Realization of the Validation

The validation in terms of j−checker and p−checker can be realized differently. They can be divided into formal and non-formal methods. Consequently, the results might be exact or approximate which has a direct influence on the validation and finally the classification. Therefore, various realizations and the consequences for the classification are presented.

Basically, the j−checker considers the fanin cone of the inputs of the respective subcircuit and checks whether there the scenarios can be justified. In contrast, the p−checker considers the fanout cone of the subcircuit' s output and checks whether the faulty output is observable at the circuit's primary outputs.

Justification by Simulation

The check for justification by simulation may perform very fast. The result might not be exact since simulation checks the scenarios non-exhaustively. The simulation applies a certain number of stimuli to check whether the scenario is justifiable. If under this number of stimuli the scenario cannot be justified the j−checker returns that the scenario is not justifiable which

is certainly an approximate result. Consequently, the j−checker based on simulation returns a subset of the scenarios that are justifiable.

Justification by SAT techniques

The check for justification based on SAT techniques is to translate the problem into a SAT-problem yielding an exact method for determining justifiable scenarios. Necessary logic of the circuit is translated into a CNF and the respective value of the scenarios are appropriately constrained. The resulting CNF is satisfiable iff the scenario is justifiable. Otherwise, the CNF is unsatifiable and the scenario not justifiable. Consequently, the j−checker based on SAT techniques determines an exact set of justifiable scenarios.

Approximate Propagation by SAT techniques and Simulation

Propagation can also be realized by simulation and SAT techniques. The check of propagation based on simulation similar realized as the j−checker yields an over-approximation of robust components due to the non-exhaustive search. A SAT-based check leads to an equivalence check that is significantly harder to solver but deliver exact results. If this formal-based kind of check is performed, i.e., that includes all relevant circuit logic the resulting problem becomes very complex and may require long run times.

A more light-weighted check is used to reduce the complexity. An approximate check for propagation by SAT techniques is presented by considering the fanout cone of the subcircuit's outputs while omitting the off-path inputs of the cone. A equivalence check is then performed. The result is an over-approximation of whether faulty outputs are justifiable.

6.4.6 Influence of Choosing Subcircuits

The choice of the subcircuits significantly influences the effectiveness of the approach. There are various opportunities to compute a set of subcircuits. Available partitioning algorithms, e.g., [KL70], determine a disjoint set of subcircuits. However, this disjoint partition might not be appropriate for the compositional approach since including checker circuitry within each subcircuit may significantly decrease the run time. Both cases, one including the checker and one excluding the checker are described:

- The local classification of a subcircuit that includes the checker circuitry may output the components as locally robust that are robust on the entire circuit as well because the checker circuitry detects

Table 6.1: Accuracy of combining approximate and exact validation

Justification	Propagation	Robustness
simulation	simulation	upper bound
exact (SAT)	simulation	upper bound
simulation	exact (SAT)	upper bound
exact (SAT)	exact (SAT)	exact
exact (SAT)	approximation (SAT)	lower bound

all internal faults. Therefore, the classification is already completed on the subcircuit and does not require any validation of scenarios and faulty outputs on the entire circuit. Thus, the compositional approach is very effective since no additional validation is required and the search space for each classification based on the subcircuit is considerably reduced.

- In contrast, the classification of a subcircuit that excludes the checker circuitry might be computationally very expensive. Suppose the components of a subcircuits are classified as locally non-robust. Consequently, the j−checker and the p−checker validate whether the component is non-robust or robust on the entire circuit, respectively. Suppose the j−checker determines that all scenarios are justifiable. This number of scenarios might be exponential with respect to the number of the inputs of the subcircuit. Consequently, the j−checker has to check a very large number of scenarios where each check is computationally complex. Afterwards, the p−checker determines that all faulty outputs are not observable at the primary outputs of the entire circuit since the fault signal reports all faults and it is concluded that the component is robust on the entire circuit.

Overall, both cases demonstrate how the choice of the subcircuits influences the effectiveness of the compositional approach. In one situation the classification has been completed on the subcircuit itself and in the other situation the classification has been completed after a huge series of additional checks.

6.4.7 Comparison of Accuracy

In Table 6.1 the combination in terms of accuracy of different techniques used for justification and propagation is listed.

Exact computation of the robustness is guaranteed when realizing the check for justification and propagation by using exact SAT techniques. A lower bound of the robustness is provided when considering an abstracted propagation that ignores certain parts of the circuit based on SAT techniques. In the remaining cases an upper bound of the robustness is provided once simulation is used since a non-exhaustive exploration of the search space is performed that may miss some corner cases.

In this thesis, an exact check for justification using SAT techniques and an approximate propagation check using SAT is used that provides a lower bound of robustness.

6.5 SIM-classifier

Simulation is extensively used in the field of functional verification for very complex circuits. Random simulation does not exhaustively analyzes the entire search space an only roughly computed approximation is provided in almost short run times. In a tightly integrated verification flow random simulation is an integral part while verifying complex industrial circuits.

In the context of robustness checking, random simulation is used in two ways. First, random simulation is used as a pre-processing step in order to approximate k-non-robust and k-dangerous components before starting the formal engines. Additionally, random simulation is tightly integrated within the introduced SAT-based approaches that guides the simulation into a particular search space.

In this section random simulation for robustness checking is introduced. Similar as in the ATPG-classifier the classification of the components is performed step by step. Once a component is classified a next component is analyzed. Usually, several thousand of inputs are generated and simulated per component. Randomly chosen input values are used to stimulate the circuit over a certain number of time frames. However, beside generating input stimuli, fault injection has to be additionally performed for each component in order to model CTFs. Due to the component model introduced in Section 2.3.1, fault injection becomes more complex. Since not only Boolean values are allowed. Depending on the bit size of the respective component, numerous faults are possible and need to be appropriately discovered during simulation.

Due to the nature of simulation, all those computations generating input stimuli and generating fault injections cannot be performed exhaustively because of the limited computational resources. Consequently, random simulation approximates the classification and is almost a non-formal approach. However, the entire flow may benefit from random simulation, because some

components are *easily* classified, for example due to their structural and functional properties. But when considering circuits that contain almost only robust components, random simulation states an overhead, since robust components cannot be proven because of the incomplete analysis.

However, random simulation is used to classify components into k-non-robust and k-dangerous components, respectively. If a single scenario and a single CTF lead to misbehavior the component is classified. Based on this the *SIM-classifier* results and is more detailed presented in the following section.

6.5.1 Algorithm

Random simulation is used to compute an approximation of k-non-robust and k-dangerous components which may boost the whole performance of robustness checking. The more components are classified by simulation the fewer components need to be checked by the formal engines which potentially yields better run times.

Algorithm 6.5.1 the pseudo-code of the SIM-classifier is presented.

Algorithm 10:
SIM-classifier.

- **Input:** A circuit $\mathcal{C} = (V, E)$ to be analyzed, a set of components to be classified $\mathbb{U} \subseteq V$, $\bar{l}$ the number of time frames from the initial state, and $\bar{k}$ the size of the observation window are given as inputs.
- **Output:** The approximate set of classified components according to $\bar{l}$ and $\bar{k}$.
- **Description:** In Figure 6.5 a schematic view of random simulation for robustness checking is shown. At first the injection state is computed by applying $\bar{l}$ randomly chosen stimuli started from the initial state. This computed state is used for fault injection. Before performing fault injection for each component the fault-free state and primary outputs need to be computed that constitutes the reference values. After determining the reference values the components of $\mathbb{U}$ are analyzed. For each component $g \in \mathbb{U}$: A randomly chosen value according to $\mathcal{F}(g)$ is set to g's output and the circuit is simulated by the same stimuli as used as computing the reference values. Once the primary outputs differ from the reference values the component is

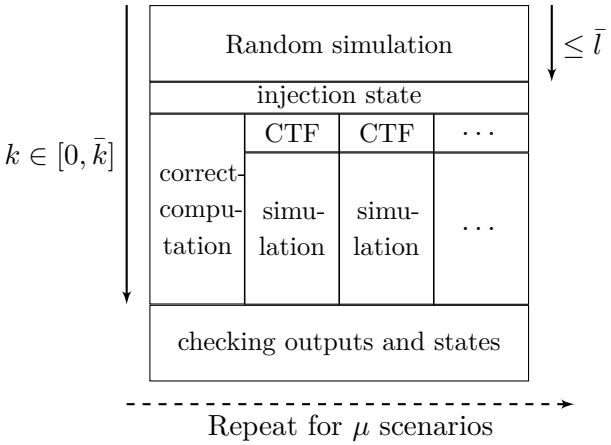


Figure 6.5: General idea of the SIM-classifier

classified as k-non-robust. After $\bar{k}$ simulation steps the primary output is equal to the reference value the states are checked for inconsistency. If the states differ the component is classified as k-dangerous and is further checked when considering new randomly chosen stimuli.

In each step, by checking the reference value it is checked whether the fault signal is set to one, what means that under current stimuli the fault is detected and therefore not classified as k-non-robust or k-dangerous.

Finally, if there are components remaining non-classified the entire simulation is repeated until μ stimuli has been applied.

Under-approximation

The random simulation computes an under-approximation of reachable states while simulating simuli due to its incomplete analysis. Consequently, an under-approximation of k-non-robust and k-dangerous components is returned according to Theorem 5.3. However, the parameter $\bar{l}$ and $\bar{k}$ influence the number of classifications as well as in the formal-based approaches. But the simulation engine cannot classify robust components due to the non-exhaustive search. Therefore, the SIM-classifier provides an upper bound of robustness as presented in Section 5.4.1.

Over-approximation

In contrast to usual random simulation that considers a partial set of reachable states, considering an over-approximation of reachable states is very useful in robustness checking as illustrated as follows. Formal classifiers exploit an over-approximation of reachable states in order to provide a proof procedure for robust components. Here, spurious k-non-robust and spurious k-dangerous components are classified and a subset of robust components are finally driven. In order to improve the overall performance for classifying spurious k-non-robust and spurious k-dangerous components in the formal classifiers, random simulation is used with an over-approximation of reachable states for the injection state. Algorithm 6.5.1 is easily adapted in order to realize this classifications. Computing the injecting state is skipped rather than a randomly chosen state is used. Therefore, the parameter $\bar{l}$ is not required anymore.

As a result the simulation engine based on an over-approximation provides only a meaningful classification when it is used as a pre-processing step before a formal classifier since the classification is spurious but excludes spurious components for the formal classifier. That means, neither a lower bound nor an upper bound of $\mathcal{WC}-\mathcal{RM}$ is provided because the classifications are spurious and robust components cannot be derived. However, the entire flow benefits from this kind of classification as it will be detailed evaluated in the experimental section.

6.5.2 Integration in the Classifiers

A large portion of the run times of the classifiers is caused by classifying k-dangerous components. For different values of k this steps is repeated several times while often a large amount of all components are k-dangerous even when k is small.

Random simulation is used to classify k-dangerous components before the formal techniques are called. All components that are classified by simulation can be skipped during the formal analysis which may significantly reduce the search space. That means, the random simulation roughly approximate k-dangerous components. Two heuristics related to the SIM-classifier and formal classifiers are newly introduced in this thesis in Section 6.7.3.

6.6 Comparison of the Classifiers

All classifiers has been presented in the previous sections. In this section the differences of these classifiers are more elaborated discussed.

Table 6.2: Complexity of the formal-method based classifiers.

	ATPG-classifier	BMC-classifier	ITP-classifier
Size	$O(2\|V\| \cdot \bar{k} + \|V\| \cdot \bar{l})$	$\Omega(2\|V\| \cdot \bar{k} + \|V\| \cdot \bar{l})$	$\Omega(2\|V\| \cdot \bar{k} + \|V\| \cdot \bar{l})$
Instances	$\geq \|\mathbb{U}\|$	≥ 1	≥ 1
Space	$2^{\|X\| \cdot (\bar{l}+\bar{k})}$	$2^{\|X\| \cdot (\bar{l}+\bar{k}) + \|\mathbb{U}\|}$	$2^{\|X\| \cdot (\bar{l}+\bar{k}) + \|\mathbb{U}\|}$

The five presented classifiers are mainly divided into four formal classifiers and one simulation approach. The SIM-classifier states the non-formal approach while the remaining classifiers are based on formal methods.

All classifiers analyze any sequential circuit except of the COMP-classifiers. The COMP-classifier analyze combinational circuits and is able to handle large circuits effectively.

6.6.1 BMC, ATPG, and ITP-classifier

The BMC-classifiers, ATPG-classifier, and ITP-classifiers are compared. As already mentioned the BMC-classifier and ATPG-classifier are theoretically able to completely analyze a circuit up to the respective thresholds. However, practically only bounded observation windows can be effectively handled. For many practical relevant cases a bounded analysis suffices since general conditions provided by the designer lead both classifiers to being effective verification tools.

However, the problem instances of the BMC-classifier and the ATPG-classifier vary: The BMC-classifier includes all components at once in single problem instance rather than the ATPG-classifier that considers only one component in single problem instance. Consequently, the ATPG-classifier needs to create and solve the problem instances for each component, separately, but the size those instances might be significantly smaller than the BMC-classifier's instances since only relevant logic need to be considered.

Moreover, the ATPG-classifier is also effectively able to compute EPP for each component.

The most powerful classifier is the ITP-classifier which generally requires no additional parameter as for example the size of the observation window as required in the BMC-classifier and ATPG-classifier. The ITP-classifier automatically determines completeness potentially before reaching the completeness thresholds.

Table 6.2 lists the complexity of the ATPG-classifier, the BMC-classifier, and the ITP-classifier. The first row specifies the size of the SAT instance

while considering the full observation window $\bar{k}$ and the full reachability window $\bar{l}$. The difference of the ATPG-classifier compared to the BMC-classifier and ITP-classifier is that the ATPG-classifier models in the worst case the entire circuit. Usually, the cone-of-influence reduction yields smaller instances. In contrast, the SAT instances of the BMC-classifier and the ITP-classifier is always since they model always the entire circuit.

The next row specifies the number SAT instances to be solved in the best case. The ATPG-classifier classifies the components in separate problem instances. Therefore, at least $|\mathbb{U}|$ instances need to be solved. In contrast, the BMC-classifier and the ITP-classifier model all components at once in a single instance. Consequently, in the best case only a single instance need to be created that classifies all components to be non-robust.

Moreover, in the last row the size of the search space is roughly specified. To classify a single component with the ATPG-classifier, values for the primary inputs over $\bar{k} + \bar{l}$ time frames need to be searched. Suppose a circuit with only 5 inputs and an observation window and an reachability window of 10 yields already a huge search space: $2^{5 \cdot 20} \approx 1.26 \times 10^{30}$. In contrast, the BMC-classifier and ITP-classifier additionally need to decide in which component a fault is injected. The size of the search space is significantly increased when considering only 50 components to be classified: $2^{5 \cdot 20 + 50} \approx 1.42 \times 10^{45}$.

Although the search space of the BMC-classifier and the ITP-classifier is significantly higher learnt information are transfered across the classifications since the instances are solved incrementally. Exploiting learnt information in the ATPG-classifier is possible for a single component over various time frames.

6.7 RobuCheck

RobuCheck is a unified push-button tool that integrates all proposed classifiers presented in this thesis into a highly-optimized flow for robustness checking. RobuCheck has been firstly published in [FFSD10] as a static verification tool and has been strongly improved in [FHD+11]. Moreover, concurrent classification to exploit multi-core processor architecture is firstly introduced in this thesis. The back-end of RobuCheck relies on the verification framework WoLFram [SKF+09] which provides basic functionality for analyzing Boolean circuits.

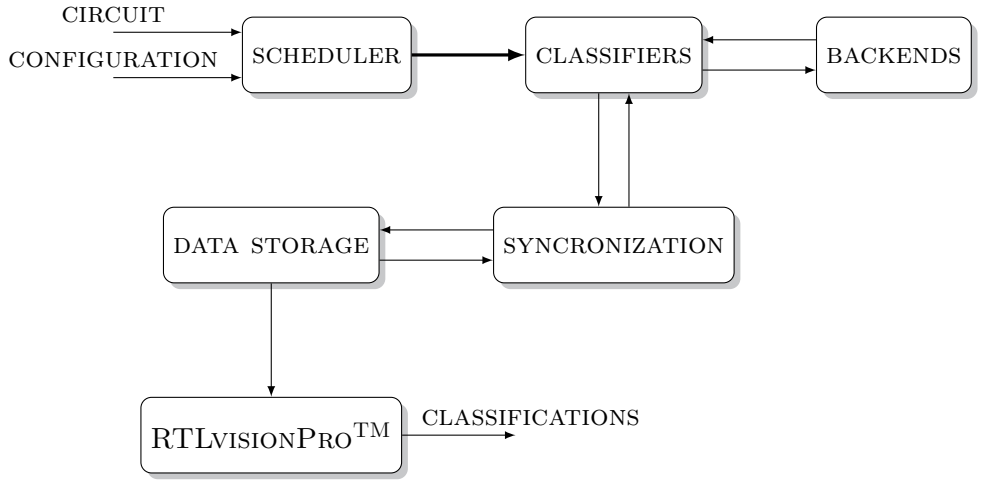


Figure 6.6: System overview of RobuCheck

6.7.1 System Overview

RobuCheck automatically computes the robustness by analyzing any part of a digital circuit. The input of RobuCheck is a digital circuit in VHDL or Verilog format and a configuration file that specifies certain parameters of the analysis.

In Figure 6.6 RobuCheck's system overview is shown. The input in terms of a circuit and a configuration file is given to the scheduler. The scheduler reads the configuration file that specifies a *Classification Process* (CP) ρ which is later described.

The Scheduler calls the classifiers in the configured order. The classifies communicates with the Backends, i.e., SAT solvers and simulation engines. Each classifier has access to shared memory that stored the classified components. This shared memory can be concurrently accessed protected by synchronization mechanism. Once a classifier determines a classification, the data storage is filled with that information such that the remaining classifiers may benefit from this information, i.e., the classifiers can skip this already classified component which reduces the overall run time.

The result of the classification can be visualized with RTLVisionPro™ [Con09]. RTLVisionPro is a strong visualization engine of hierarchical schematic view, source code browsing, and cross-probing between schematic view, and source code. This engines is integrated in RobuCheck via *Tool Command Language* (TCL). The *Graphical User Interface* (GUI) is shown in Figure 6.7. In particular, the visualization engine is used to

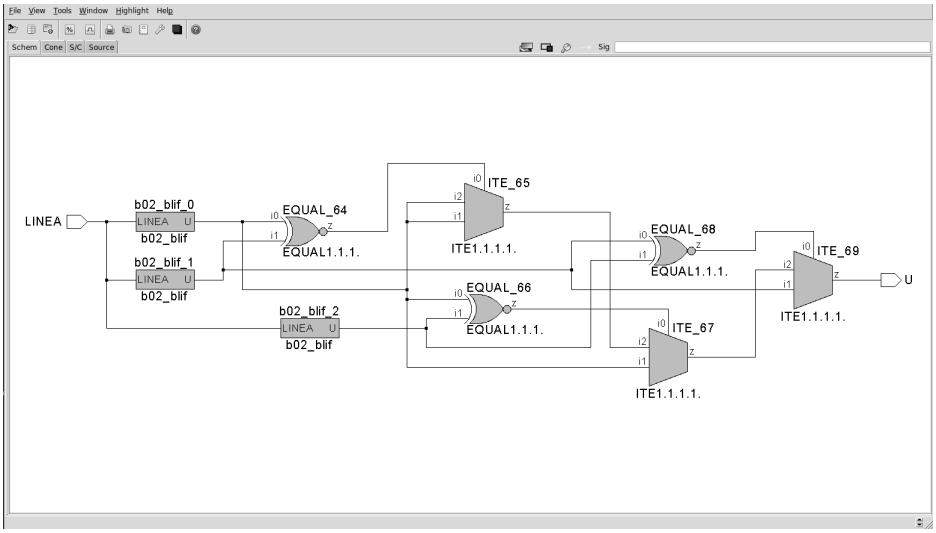


Figure 6.7: Graphical User Interface of RTLVisonPro^{TM}

differentiate between the classification. For example, k-non-robust components are highlighted with red-coloured components, robust components are highlighted with green-coloured components, and, k-dangerous components are highlighted with yellow-coloured components. The more fine-grained analysis that consider *Excitation and Propagation Probabilities* (EPP) as introduced in Section 4.2 yields a differentiation between non-robust components. A fine-grained gradation of the red-coloured components provides a visual differentiation as well, i.e., the more vulnerable the component the darker the red.

6.7.2 Technical Details

RobuCheck is implemented in C++ and uses several thirdparty libraries as for example external SAT solvers. Technical internals are described in the following.

Classification Process

A *Classification Process* (CP) consists of several parameters:

- **classifier (cls):** The name of the classifier with

$$\mathsf{cls} \in \{\text{BMC}, \text{ATPG}, \text{ITP}, \text{COMP}, \text{SIM}, \text{SUB-CP}\}$$

  The SUB-CP specifies that a list of CPs are started in parallel. The remaining names corresponds to the proposed engines before.

- **observertion window $\bar{k}$:** The parameter $\bar{k} \in [0, k_{\mathsf{cmpl}}]$ specifies the maximum considered observertion window.
- **reachability window $\bar{l}$:** The paramter $\bar{l} \in [0, l_{\mathsf{cmpl}}]$ specifies the maximum number of time frames from the initial state to inject a fault.
- **optimization lookup table olt:** A lookup table specified which optmizations are enabled.
- **list of CPs $\rho_1, \ldots, \rho_n$:** A list of sub-CPs that are exectued in parallel.
- **pre-process CP $\rho_{\mathbf{pre}}$:** The pre-process CP specifies a separate CP that is exectued before.
- **reserved flags REV:** A list of flags dedicated to the classifiers are stored in Φ_{flags}.

The CPs are stored in the configuration file using the *Extensible Markup Language* (XML). A CP configures which classifiers are used with the required parameters and in which order they are executed. Since a CP internally supports the list of CPs that are exectued in parallel (SUB-CP), very powerful configurations can be setted up.

CP for BMC-classifier and ATPG-classifier The BMC-classifier and ATPG-classifiers gets beside the circuit two additional parameters: size of the observation window $\bar{k}$ and the size of the reachability window $\bar{l}$.

CP for ITP-classifier The ITP-classifier does not depend on certain parameters since completeness is automatically determined. Therefore, the parameter $\bar{l}$ and $\bar{k}$ are set to the maximal possible value. However, in certain cases a limitation can be configured.

CP for COMP-classifier The COMP-classifier analysis combinational circuits. Therefore, unrolling the transition relation is not required. Hence, the parameters $\bar{l}$ and $\bar{k}$ are constantly setted to zero. The general idea of the COMP-classifier is to classify the circuit based on a set of subcircuits as detailed described in Section 6.4. The subcircuits are stored in a separate file specified in the flags Φ_{flags}.

CP for SIM-classifier The SIM-classifier depends additionally on the parameter how many traces have to be simulated. This number of traces is specified in Φ_{flags}[traces] as a positive number.

SAT solver

WoLFram encapsulates several solvers for Boolean satisfiability that can be arbitrarily selected. RobuCheck focuses on three different solvers: MiniSAT [ES03], PicoSAT [Bie08], and Lingeling [Bie10].

The most frequently used SAT solver is MiniSAT in version 2.2 which is integrated in RobuCheck over MiniSAT's C++ API. That means, performance related features like incremental satisfiability can be adequately exploited via its API. PicoSAT and Lingeling are used in the context of interpolation as described in the following.

Interpolation

The ITP-classifier requires to compute Craig interpolants. Typically, the performance of interpolation-based algorithms are based on the strength of the back-end interpolation procedures. In order to further improve the performance of the ITP-classifier as originally published in [FFA+12], a powerful flow has been newly introduced in this thesis to obtained interpolants.

As presented in Section 2.2.4 interpolants can be computed in two ways: *proof-based* along the resolution proof of a SAT solver, or *model-based* using any SAT solver that provides satisfying assignments. Both techniques have been implemented in RobuCheck and are integrated as a first-come first-served approach explained later. Recall, an interpolant $\hat{I}$ is logical formula describing the relation of a unsatisfiable formula pair (A, B). Each technique used in this work starts with the formulas A and B given as CNF.

To the best of the author's knowledge the integration of both interpolation approach has not been proposed so far in the literature. The integration is firstly done in this thesis. Due to the significant difference of both approaches the integration in terms of concurrent computation is very useful.

Proof-based *McMillan's Interpolation System* (MIS) computes interpolants along a resolution proof of an unsatisfiable formula. Figure 6.8 depicts the implemented flow to compute interpolants based on resolution proofs. RobuCheck starts the computation by generating a trace of the proof of the unsatisfiable formula $A \wedge B$. The SAT solver PicoSAT has been chosen to generate such traces. However, the trace generated by PicoSAT does contain all information to generate interpolants such as pivot variables. The tool tracecheck checks the trace for correctness, i.e., whether the resolution inference rule is correctly applied to derive the empty clause. While checking the trace this tool generates a resolution proof containing

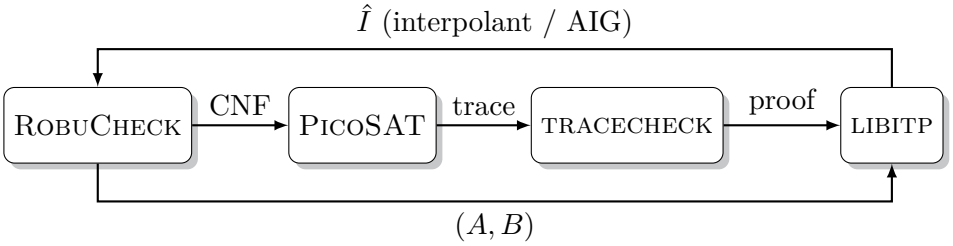


Figure 6.8: Proof-based interpolants computed with PicoSAT

sufficient information to generate an interpolant. The library libitp gets the formula pair (A, B) and the resolution proof as input and generates an interpolant based on McMillan's interpolation system (MIS) (see Section 2.2.4). Finally, libitp returns the determined interpolant represented as AIG to RobuCheck.

Interpolants are often very large consisting of several hundreds of thousand of nodes but containing often a huge amount of redundant logic. Since interpolants are represented in AIG several tools supporting AIG files and minimizing the graph can be easily applied, i.e., ABC [Gro12].

Model-based In contrast to the proof-based approach to generate interpolants the model-based approach does not require a resolution proof rather than enumerating satisfying assignments of A and B. The basic idea of the approach is to compute satisfying assignments of formula A and create a DNF with that assignments that finally builds the interpolant. Minimizing the assignments is crucial for the performance of this approach. But while minimizing the assignments the definition of Craig interpolant need to be taken into account, i.e., the assignment abstracts A but contradicts B.

The computed interpolant is a DNF, i.e., $\hat{I} = p_1 \vee p_2 \vee p_n$ where p_i is a satisfying assignment of A with $\mathsf{Var}(p) = \mathsf{Var}(A) \cap \mathsf{Var}(B)$. Due to the exponential grow with the number of variables of A the naïve approach of enumerating all assignments might be very slow. Generalizing the assignments speeds up the computation by a considerable factor. This is performed in an extra step and is also known as *cube enlargement*. Three techniques has been newly implemented to minimize the assignments. The original paper [CIM12] implements other minimization algorithms.

Generalize p with respect to A: MIN-1 The Pseudocode 10 is a greedy algorithm enumerate assignments and generates an interpolant in terms of a DNF. The general algorithm is revisited in Section 2.2.4. Let p be an assignment of A' where A' is formula A excluding all already

```
1 begin
2     for l ∈ Var(p) do
3         if SAT?(A ∧ p \ {l} ∪ {l̄}) then
4             return MIN-1(p \ {l});
5     end
6     return p;
7 end
```

Pseudocode 10: Generalize assignment p: MIN-1.

```
1 begin
2     U = Var(UCORE(p ∧ B));
3     for l ∈ Var(p) do
4         if l ∉ U then
5             p = p \ {l};
6     end
7     return p;
8 end
```

.

Pseudocode 11: Generalize assignment p: MIN-2.

computed assignments. Pseudocode 10 checks whether a variable $l \in \mathsf{Var}(p)$ is necessary for the minterm such that $p \models A'$ by checking whether the assignment p is still satisfiability by inverting the Boolean value of l. That means, if $p \setminus \{l\} \cup \{\bar{l}\} \models A'$ holds then the variable l can be safely removed from the minterm, i.e., $p' = p \setminus \{l\}$. The algorithm MIN-1 is recursively called with the new obtained minterm p'. Consequently, at most $|p|$ SAT calls are performed. Finally, the algorithm returns the generalized minterm.

Generalize p' with respect to B (UNSAT-core): MIN-2 According to Craig's interpolation theorem an assignment $p \models A$ contradicts B, i.e., $p \not\models B$. That means p can be minimized as long as it is unsatisfiable with B.

The Pseudocode 11 exploits the variable of the unsat core of $p \wedge A$. For each variable occuring in p it is checked whether the variable is contained in the unsat core. If the variable $l \in \mathsf{Var}(p)$ is not contained in the unsat core ($l \notin U$) l is not used to derive a final conflict of the SAT solver. Therefore the variable l is not necessary in p and can be safely removed from p. After checking all variables the minimized p is returned.

```
1 begin
2     for l ∈ p do
3         if !SAT?(p \ {l} ∧ B) then
4             return MIN-3(p \ {l});
5     end
6 end
```

Pseudocode 12: Generalize assignment p: MIN-3.

Generalize p' with respect to B: MIN-3 Similar as the previous approach MIN-2 the last approach MIN-3, shown in Pseudocode 12, checks whether each variable is necessary to derive a conflict by additional SAT calls.

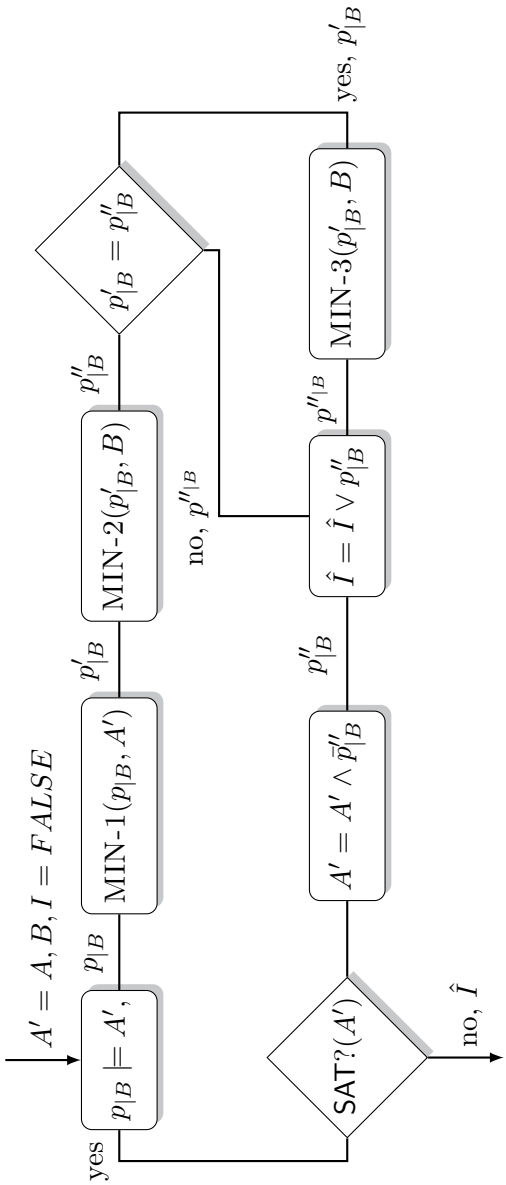


Figure 6.9: Model-based interpolant computation

Overall flow The overall flow of the model-based technique is shown in Figure 6.9. The flow starts on the incoming edge with initially $A' = A$, B, and an empty interpolant $\hat{I} = FALSE$. The first step is to compute an assignment $p_{|B}$ with $p_{|B} \models A'$ as minterm which is defined over the common variables of A and B. In the following the assignment is minimized by the presented techniques: MIN-1, MIN-2, and MIN-3. After minimizing MIN-1 a new minterm $p'_{|B}$ is obtained that may contain fewer variables. Furthermore, the next steps tries again to apply minimization according to the formula B by MIN-2 that exploits the unsatisfiable core. This step is a very light-weighted step since only a fast lookup whether a variable is contained in the unsatisfiable core is performed. However, depending on the SAT solver's internal heuristics this steps is more or less effective. Therefore, in the next step it is checked whether the minimizing MIN-2 deleted at least one variable. If not, the minimization MIN-3 is additionally called to try reduce the minterm more aggressive. The reduced minterm $p''_{|B}$ is added to the interpolant $\hat{I}$ and to avoid the recomputation of the same assignment, $p''_{|B}$ is blocked in A'. Finally, if at least one further assignment exists a new iteration starts. Eventually, no more assignment exists and a new interpolant has been computed.

Technically, the DNF of the interpolant is compactly represented by a BDD. However, after computing the complete interpolant all prime implicants are added to an AIG which is returned as interpolant. As SAT solver Lingeling [Bie10] is used that fully supports incremental satisfiability and returned *failed literals* that are used in MIN-2.

Concurrent Computation Both approaches to compute interpolants are integrated in RobuCheck. The approaches are concurrently started, each in a separate process in order to exploit multi-core systems. Once one process successfully returned the other process is terminated and the interpolant of the fastest determined results is returned. This kind of computation is very useful when an advance selection of the fastest approach is difficult. Overall, for each computation of an Craig interpolant the fastest approach is performed.

6.7.3 Simulation Heuristics

The SIM-classifier is used as a pre-process to roughly approximate non-robust components. However, the SIM-classifier is also used within the formal engines. The classifiers proves robust components by determining over-approximations of k-non-robust and k-dangerous components. Typically a large number of k-dangerous components are classified multiple

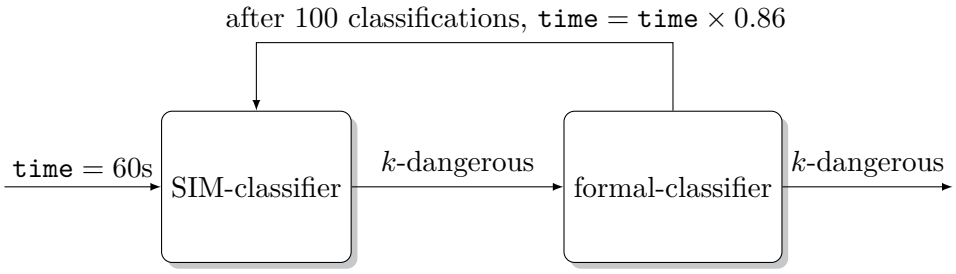


Figure 6.10: Integrated flow of the SIM-classifier into formal-methods based classifiers

times. In order to reduce this costly computation the SIM-classifier is tightly integrated into the formal classifiers to approximate k-dangerous components.

Heuristic #1: Repeated Simulation

Before formally classifying k-dangerous components, the SIM-classifier is called first by exactly analyzing the observation window of size k. As a results the SIM-classifier determines a subset of k-dangerous components.

To provide an effective and complete classification the run time SIM-classifier is limited by a dynamic parameter. The determined k-dangerous components are not further analyzed by the formal classifiers which prunes the search space of the underlying SAT instance and reduces costly SAT calls. Moreover, after a certain number of formal classifications the SIM-classifier is called with more restrictive computational resources.

Figure 6.10 illustrates the integration of the SIM-classifier and the formal methods based classifier denoted by *formal-classifier*.

At the beginning for each k-dangerous classification, the maximal run time of the SIM-classifier is limited to $60s$. After termination the results are passed to the formal-classifier. After 100 classifications by the formal-classifier, the SIM-classifier is called again. But the maximal run time of the SIM-classified is reduced by a factor of 0.86. Eventually, the formal-classifier classifies all remaining k-dangerous components since the SIM-classifier naturally does not exhaustively explore the search space. Thereby, maximal run time converges to 0.

The concrete values for the heuristics has been obtained during preliminary experiments. This particular values yields the best results. But, however, otherwise values are configurable and may results in better run times for another training set.

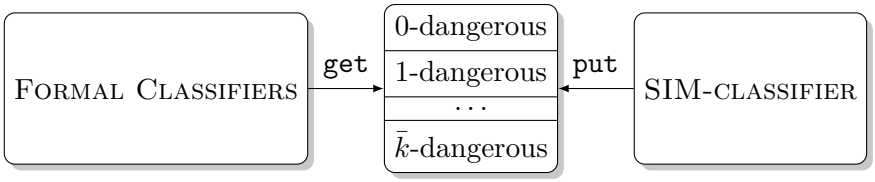


Figure 6.11: Collecting k-dangerous components.

This flow is integrated within the formal methods based classifies analyzing sequential circuits, i.e., the BMC-classifier, ATPG-classifier, and ITP-classifier.

Heuristic #2: Collecting k-dangerous Components

This heuristic is dedicated to multi-core architectures since it runs in parallel to the formal classifiers. An extra classifier is created on top of the SIM-classifier to collect only k-dangerous components for a certain interval $k \in [0, \bar{k}]$.

In Figure 6.11 the idea of this heuristic is illustrated. The SIM-classifier approximates k-dangerous components and stores this information via `put` in a storage. Components that a classified as k-dangerous are in particular also further considered for different values of k even to reduce multiple classifications by the formal classifiers. This storage can be accessed by the formal classifier via `get`. The storage is able to handle concurrent access.

Overall, while the SIM-classifier computes k-dangerous components the formal classifier can concurrently access this classification to prune the search space.

6.8 Summary

This chapter introduced the classifiers that implement the theoretical fundamentals of performing robustness checking. The BMC-classifier, ATPG-classifier, and ITP-classifier formally analyze sequential circuits. The COMP-classifier handles larger combinational circuits by decomposing the problem formulation into smaller but potentially easier to solve instances. Moreover, the SIM-classifier a simulation-based approach has been introduced that handles sequential circuits and provides a roughly determined approximation of k-non-robust and k-dangerous components.

Furthermore, RobuCheck that integrates all these classifiers has been presented. In particular for the ITP-classifier a new back-end for determining interpolant has been proposed. This back-end concurrently computes

interpolants based on two different approaches while the fastest approach delivers the interpolant.

Moreover in this chapter, a new model checker SimpMC exploiting the inverse interpolants has been proposed.

The SIM-classifier is particularly tightly integrated into the formal classifier to determine k-dangerous components concurrently and consecutively by Heuristic #1 and Heuristic #2, respectively.

Chapter 7

Experiments

In the previous chapters robustness checking has been theoretically introduced in terms of a tool called RobuCheck. In RobuCheck formal and non-formal classifiers are integrated into to a highly-optimized flow. This chapter presents the results of the evaluation of RobuCheck on a set of academic and industrial benchmarks.

- At the beginning the internal interpolation engine of RobuCheck is evaluated on a set of benchmarks. Basically, the proof-based approach and model-based approach is evaluated on unsatisfiable SAT instances coming from the SAT competition.
- In a next evaluation, the new model checker SimpMC is evaluated on a set of benchmarks from the *Hardware Model Checking Competition* and compared against a state-of-the-art model checker.
- As next, the proposed algorithms to assess the circuit's robustness are evaluated. The evaluation starts with the BMC-classifier, ATPG-classifier, and the ITP-classifier. Each of this three classifiers are evaluated separately in terms of accuracy and run time.
- Furthermore, the SIM-classifier is evaluated and compared against a formal classifier.
- After a separate evaluation of the classifier, the concurrent classification joining all three classifiers is evaluated in two different setups. In these setups the SIM-classifier is additionally turned on.
- The COMP-classifier is evaluated on combinational ISCAS'85 circuits and on complex arithmetic circuits.

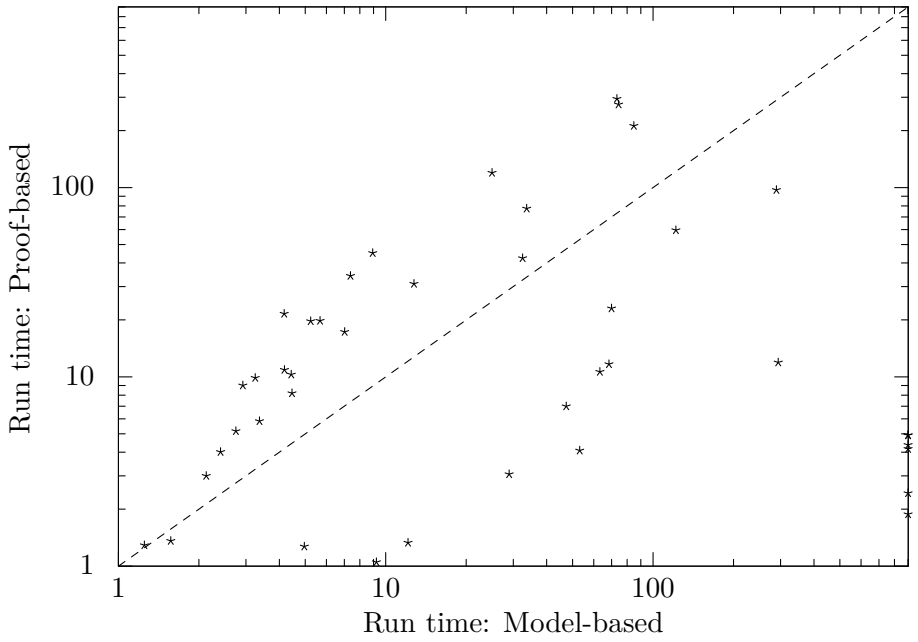


Figure 7.1: Run time of model-based vs. proof-based approach

- The more differentiated robustness measure considering the worst case analysis and probabilistic analysis is evaluated.
- As last evaluation, benchmarks from IBM are taken to evaluated the ITP-classifier on industrial circuits.

7.1 Interpolation: Model-based vs. Proof-based

The back-end of RobuCheck implements two approaches to compute interpolants: a proof-based approach and a model-based approach. Both have been revisited in Section 2.2.4 where details of the implementation have been introduced in Section 6.7.2 in particular with new reduction techniques.

Randomly selected unsatisfiable SAT problems were taken from the *SAT competition benchmark set 2011*[1]. Overall, 45 instances were used to compare both approaches in terms of run time and size of the obtained interpolants.

[1] Available under `http://www.satcompetition.org`

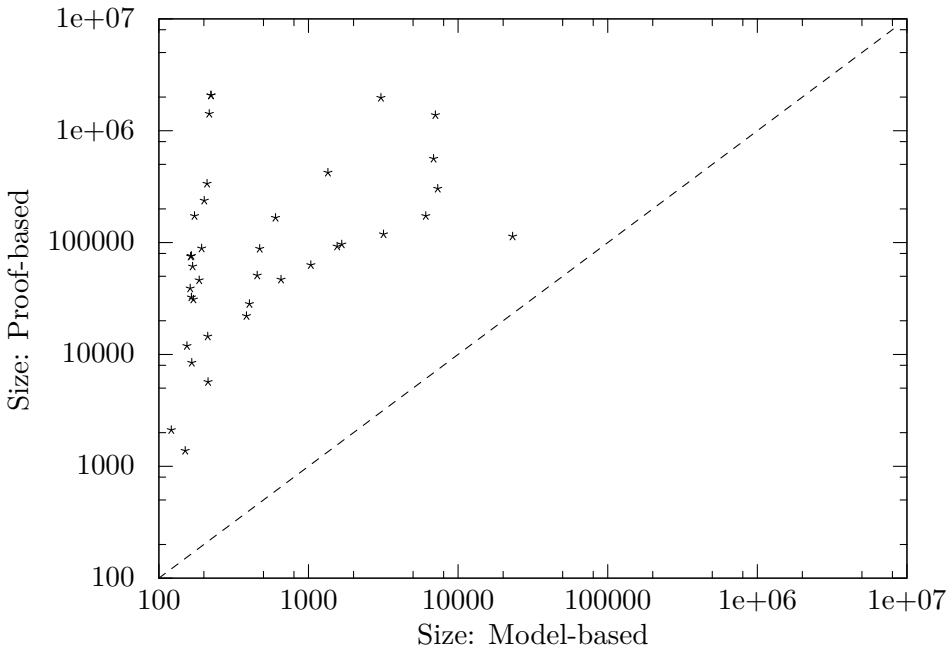


Figure 7.2: Size of the interpolants computed by model-based and proof-based approach.

For each instance a partition into A and B is randomly created. The number of common variables of A and B ranges from 63 to 126 variables and the size of the entire CNF ($A \cup B$) ranges from 20,000 clauses to 97,000 clauses. A time out was set to 900 CPU seconds. The experiments were conduced on a AMD Opteron^{TM}CPU with six cores running at 2.8Ghz with 32GB main memory.

Figure 7.1 shows a scatter plot of the run time in CPU seconds for both approaches. A single point represents a single instance. If a point is below the diagonal line the proof-based approach was faster than the model-based approach and vice versa. Overall, both areas are almost equal, i.e., both approaches perform similar.

Figure 7.2 shows a scatter plot of the size of the obtained interpolants for both approaches. The size is provided by the number of *And-Inverter-Graph* (AIG) nodes that correspond to the number of `AND`-gates.

All obtained interpolants using the model-based approach are smaller than the obtained interpolants using the proof-based approach. While the proof-based approach generates very large interpolants where the largest contains more than 2 million nodes, generates the model-based approach

an interpolant with 223 nodes for the same benchmark. The discrepancy of the obtained interpolants is significant. But the model-based approach runs out of time for some cases where the proof-based approach provides an interpolant within the time limit.

However, the run time of the model-based approach significantly depends on the number of common variables of the partition (A, B) since the number of assignments that are computed grows exponentially with that number.

Additionally, the model-based approach was evaluated where the minimization techniques have been turned off. For all instances the model-based approach ran out of time. Consequently, minimizing the assignments is crucial for the model-based approach.

Overall, the interpolants are concurrently computed by both approaches where the fastest approach delivers the interpolant.

7.1.1 Future Work

The proof-based approach computes the interpolants along the resolution proof. The size of the proof might be very large and thus the time to compute the proof and afterwards the interpolant may take very long time. The interpolant is completely computed when the entire proof is traversed. The model-based approach enumerates assignments where the number of these assignments grows exponentially with the number of the common variables. The interpolant is completely computed when all assignments are conjoined to a DNF.

Both approaches are time consuming relative to the entire run time of the verification task. While the proof-based approach delivers the interpolant after the entire proof is traversed the model-based approach can deliver parts of the interpolant even before completion. This part can be used even before the entire interpolant is computed to checked whether a spurious counterexample or a spurious classification is performed which can be safely decided even in case of an incomplete interpolant. Consequently, in this case the further enumeration of assignments can be aborted and therefore the run time might be reduced.

7.2 Simple Model Checker - SimpMC

In Section 6.3.3 a simple model checker SimpMC exploiting interpolants to over-approximate the reachable states is introduced. Beside RobuCheck this model checker has been implemented in C++ on top of WoLFram [SKF+09]. In this section SimpMC is evaluated on a randomly selected

subset of the *Hardware Model Checking Competition* (HWMCC) benchmarks from 2012[2].

Overall, 90 problem instances were used and a time out of 900 CPU seconds was set. To compare SimpMC against a different model checker, the powerful model checker IIMC[3] was used. IIMC implements several verification algorithm in particular the newly introduced *IC3* [Bra11] approach in a very sophisticated flow consisting of various optimization techniques. IC3 won the third place in the HWMCC 2010 with a relatively outdated SAT solver Zchaff [MMZ+01a]. In contrast, SimpMC implements only a single verification algorithm based on interpolation.

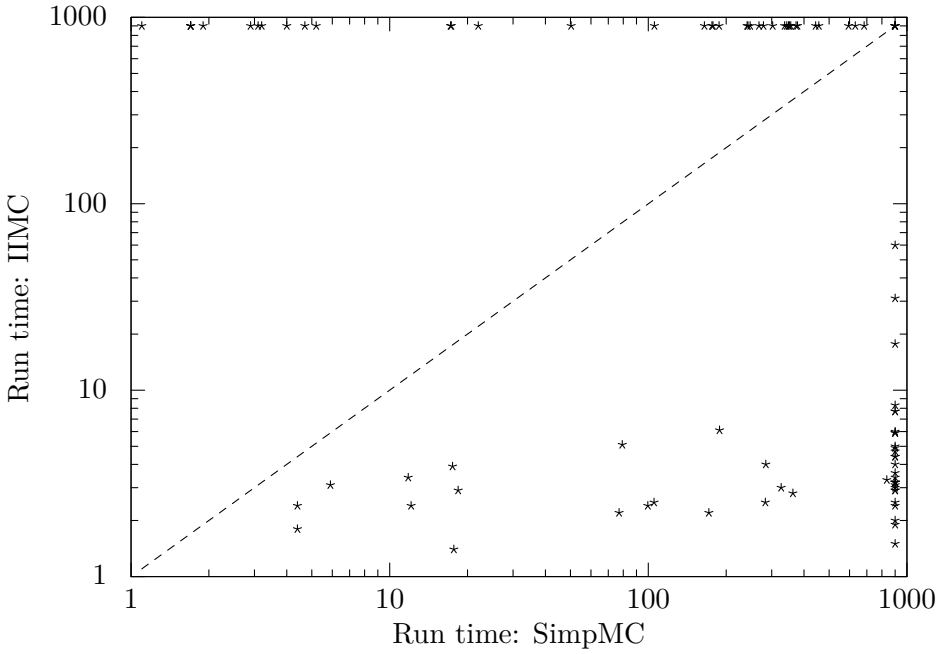


Figure 7.3: Run time of SimpMC vs. IIMC.

Figure 7.3 shows a scatter plot of the run time for both tools. In 49 instances, SimpMC is faster than IIMC where IIMC runs in a time out for 29 out of these 49 instances, i.e., SimpMC outperforms IIMC by a considerably factor. For the remaining 51 instances SimpMC could not terminate within the time out where IIMC solved additionally 38 instances. Overall, SimpMC solved 49 instances faster than IIMC although SimpMC

[2] Available under `http://fmv.jku.at/hwmcc12/`
[3] Available under `http://ecee.colorado.edu/wpmu/iimc/`

does not implement any sophisticated flow. Consequently, SIMPMC states a powerful additional verification engine within an orchestrated verification tool.

7.3 Robustness Checking

This section provides the obtained results of performing robustness checking using ROBUCHECK.

7.3.1 Benchmarks

A subset of circuits of the ITC99 benchmark suite[4] was used as benchmarks. More precisely, the circuits `b08` to `b15` and further derived fault-tolerant circuits were used.

Based on the original circuits techniques to protect the circuit against transient faults have been implemented:

- A system-level and FF-based *Triple Modular Redundancy* (TMR) implementation are revisited in Section 2.6.2. Both techniques were applied to the benchmark circuits where the system-level circuits have no fault signal and the FF-based circuits have a fault signal. The system-level implementations are marked by `-tmr-sys` and the FF-based implementations are marked by `-tmr-ff`.

- As a further hardening technique a *parity checker* has been implemented on each considered ITC'99 circuit. A checker circuitry has been generated that computes the parity over the flip flops and the primary outputs. The primary outputs are buffered with extra flip flops. A wrong parity is reported by a fault signal. The implementation of these circuits are relative robust since each transient fault that flips odd numbers of flip flops and primary outputs is detected. The parity circuits are marked by `-par`. The circuits have been optimized by SIS [SSL+92].

The characteristics of all circuits are shown in Table 7.1. The first column *Circuit* denotes the name of the circuit. The remaining columns list the characteristics: $|X|$ lists the number of primary inputs, $|Y|$ lists the number of primary outputs, $|V|$ lists the number of components, and $|FF|$ lists the number of flip flops. Overall, 32 ITC'99 and derived circuits are used.

[4] Available under `http://www.cerc.utexas.edu/itc99-benchmarks/bench.html`

Table 7.1: Characteristics of the benchmark circuits.

Circuit	$\|X\|$	$\|Y\|$	$\|V\|$	$\|FF\|$
b08	9	4	240	21
b09	1	1	203	28
b10	11	6	279	17
b11	7	6	894	31
b12	5	6	1363	121
b13	10	10	415	53
b14	32	54	11395	245
b15	36	70	11035	449
b08-tmr-sys	9	4	765	63
b09-tmr-sys	1	1	619	84
b10-tmr-sys	11	6	902	51
b11-tmr-sys	7	6	2743	93
b12-tmr-sys	5	6	4148	363
b13-tmr-sys	10	10	1345	159
b14-tmr-sys	32	54	34703	735
b15-tmr-sys	36	70	33771	1347
b08-tmr-ff	9	4	1458	63
b09-tmr-ff	1	1	1543	84
b10-tmr-ff	11	6	1463	51
b11-tmr-ff	7	6	3766	93
b12-tmr-ff	5	6	8141	363
b13-tmr-ff	10	10	3094	159
b14-tmr-ff	32	54	42788	735
b15-tmr-ff	36	70	48588	1347
b08-par	9	5	618	26
b09-par	1	2	597	30
b10-par	11	7	730	24
b11-par	7	7	1876	38
b12-par	5	7	3990	128
b13-par	10	11	1273	64
b14-par	32	55	20203	300
b15-par	36	71	25656	520

Unless otherwise stated, the benchmarks were carried out on AMD Opteron^{TM} CPU running at 3.0GHz with 64GB main memory.

7.3.2 Formal classifiers

The BMC-classifier, ATPG-classifier, and ITP-classifier are based on formal methods analyzing sequential circuits. These classifiers are evaluated on the benchmarks described above. All components have to be classified, i.e., $\mathbb{U} = V$.

Quality

The BMC-classifier and ATPG classifier uses approximate reachability information. An over-approximation of the reachable states leads to a lower bound of robustness ($R^{\bar{k}}_{lb}$) and an under-approximation of the reachable states leads to an upper bound of the robustness ($R^{\bar{k}}_{ub}$). In order to obtain bounds of the robustness both classifiers are called twice with different approximations leading to four runs for each circuit.

As over-approximation simply the entire set of states is considered. This is realized by leaving the initial value of the state elements unconstrained. The under-approximation is realized by configuring the reachability window to 10 time frames, i.e., $\bar{l} = 10$. An observation window of size 10 was considered, i.e., $\bar{k} = 10$. The ITP-classifier generally considers an unlimited observation window and computes the set of states by interpolants.

The BMC-classifier and ATPG-classifier use a single core of CPU. Thus, run time is measured in CPU second. The ITP-classifier computes interpolants concurrently on two CPU cores according to the proposed concurrent computation of interpolants. Thus, the run time of the ITP-classifier is measured in wall clock seconds. For each benchmark the run time was limited to 8 hours.

Under unlimited computational resources the BMC-classifier and ATPG-classifier deliver identical results since the computational model and the considered set of states are equal. However, the differences are shown in the following.

The ITP-classifier computes fully automatic suitable approximations based on interpolation. Thus, reachability window and observation window are configured to be unlimited.

Table 7.2: Determined robustness bounds of ITC'99 circuits.

	BMC-classifier		ATPG-classifier		ITP-classifier		Virtual Best	
Circuit	$\hat{R}_{lb}^{\bar{k}}$	$\check{R}_{ub}^{\bar{k}}$	$\hat{R}_{lb}^{\bar{k}}$	$\check{R}_{ub}^{\bar{k}}$	$R_{lb}^{k_{\text{cmpl}}}$	$R_{ub}^{k_{\text{cmpl}}}$	R_{lb}	R_{ub}
b08	0.00%	48.33%	0.00%	48.33%	0.00%	0.00%	0.00%	0.00%
b09	0.00%	79.31%	0.00%	79.31%	0.00%	0.49%	0.00%	0.49%
b10	0.00%	1.79%	0.00%	1.79%	1.79%	1.79%	1.79%	1.79%
b11	0.11%	8.84%	0.11%	8.84%	6.26%	7.83%	6.26%	7.83%
b12	0.00%	51.58%	0.00%	51.58%	0.00%	39.47%	0.00%	39.47%
b13	0.72%	54.70%	0.72%	100.00%	6.02%	48.19%	6.02%	48.19%
b14	0.00%	100.00%	0.00%	96.42%	0.09%	13.19%	0.09%	13.19%
b15	0.00%	100.00%	0.00%	99.95%	0.43%	28.45%	0.43%	28.45%
b08-tmr-ff	98.83%	99.45%	98.83%	100%	98.83%	98.83%	98.83%	98.83%
b09-tmr-ff	99.81%	99.87%	99.81%	100%	99.81%	99.81%	99.81%	99.81%
b10-tmr-ff	98.43%	98.43%	98.43%	100%	98.43%	98.43%	98.43%	98.43%
b11-tmr-ff	99.50%	100.0%	0.0%	100%	99.50%	99.50%	99.50%	99.50%
b12-tmr-ff	99.79%	100.0%	0.0%	100%	99.84%	99.84%	99.84%	99.84%
b13-tmr-ff	99.03%	99.32%	99.03%	100%	99.29%	99.29%	99.29%	99.29%
b14-tmr-ff	0.0%	100.0%	0.0%	100%	0.00%	99.75%	0.00%	99.75%
b15-tmr-ff	0.0%	100.0%	0.0%	100%	0.00%	99.71%	0.00%	99.71%
b08-par	74.11%	97.57%	74.11%	97.57%	74.11%	74.11%	74.11%	74.11%
b09-par	84.25%	98.16%	84.25%	98.16%	86.93%	87.10%	86.93%	87.10%
b10-par	82.60%	84.79%	82.60%	84.79%	83.84%	83.84%	83.84%	83.84%
b11-par	80.44%	86.14%	80.44%	86.14%	83.32%	83.64%	83.32%	83.64%
b12-par	6.57%	94.61%	83.98%	84.79%	6.57%	99.85%	83.89%	99.85%
b13-par	89.32%	95.05%	89.32%	95.05%	89.32%	94.66%	89.32%	94.66%
b14-par	0.0%	94.74%	1.91%	99.99%	0.00%	100.00%	1.91%	94.74%
b15-par	0.0%	99.73%	2.50%	99.97%	0.00%	100.00%	2.50%	99.73%

Table 7.3: Robustness of hard TMR circuits.

	BMC-classifier				ITP-classifier		
Circuit	R_{lb}	R_{ub}	l	k	$R_{lb}^{k_{\mathsf{cmpl}}}$	$R_{ub}^{k_{\mathsf{cmpl}}}$	Run time
b08-tmr-sys	1.2%	99.4%	14	5	99.4%	99.4%	68
b09-tmr-sys	0.3%	99.6%	19	4	99.6%	99.6%	44
b10-tmr-sys	1.5%	97.8%	16	4	97.8%	97.8%	778
b11-tmr-sys	0.6%	99.4%	13	2	99.4%	99.4%	373
b12-tmr-sys	0.3%	99.8%	19	2	99.8%	99.8%	395
b13-tmr-sys	2.3%	99.0%	7	3	99.0%	99.0%	213

Table 7.2 the determined robustness bounds are listed for the BMC-classifier, ATPG-classifier, and ITP-classifier. The first column lists the circuit name. For each classifier a lower bound and an upper bound of the robustness is provided. The last two columns lists the virtually best robustness bound, i.e, the best lower bound and the best upper over all classifiers.

For a large portion of the circuits tight bounds are determined, i.e., the original ITC'99 circuits have a relatively low robustness value and the fault-tolerant implementations have high robustness values as expected. Furthermore, the BMC-classifier and the ATPG-classifier delivers often the same results except for the `tmr-ff` circuits where the ATPG-classifier did not classify any component leading to an upper bound of 100%. In contrast, the ITP-classifier deliver equal or even tighter bounds in all cases except circuit `b14-par` and `b15-par`. Even for the large circuits `b14` and `b15` the ITP-classifier delivers good results since a systematic computation of the reachable states is performed that includes only relevant facts to classify the components. Only the ATPG-classifier completed a few components.

The BMC-classifier and ATPG-classifier could marginally classify more components than the ITP-classifier for the circuits `b14-par` and `b15-par`. A more detailed analysis of both circuits show that the interpolation engine did not complete the computation of the interpolant. The resolution proof of the proof-based approach was getting too large to complete and the number of common variables of the formula pair was to high to complete the generation process by the model-based approach.

Furthermore, a significant result is that the ATPG-classifier did not work very well on the -tmr-ff circuits.

Outstanding results are reached by the ITP-classifier for the `-tmr-sys` circuits listed in Table 7.3. Since most of the components of these circuits

Table 7.4: Run times of the formal classifiers.

	BMC-classifier	ATPG-classifier	ITP-classifier		
Circuit	Run time	Run time	Run time	l	k
b08	5.4	22.8	68.4	18	17
b09	3.1	16.7	*time out*	21	47
b10	27.9	20.2	745.4	6	12
b11	175.9	2349.5	*time out*	12	21
b12	335.1	4427.9	*time out*	42	12
b13	19.6	14.2	*time out*	30	20
b14	*time out*	*time out*	*time out*	4	3
b15	*time out*	*time out*	*time out*	19	2
b08-tmr-ff	751.9	37060.0	1146.3	11	18
b09-tmr-ff	322.0	31904.6	99.9	10	2
b10-tmr-ff	5190.1	30511.9	163.7	5	5
b11-tmr-ff	*time out*	*time out*	1171.8	2	3
b12-tmr-ff	*time out*	*time out*	21863.5	13	12
b13-tmr-ff	30213.8	49447.9	334.1	4	2
b14-tmr-ff	*time out*	*time out*	*time out*	2	0
b15-tmr-ff	*time out*	*time out*	*time out*	2	0
b08-par	250.3	201.7	1303.3	18	18
b09-par	262.0	182.6	*time out*	61	83
b10-par	1856.2	385.6	133.6	9	6
b11-par	35968.8	18116.2	*time out*	15	13
b12-par	34151.2	32186.0	*time out*	2	1
b13-par	29056.1	1065.0	*time out*	28	18
b14-par	*time out*	*time out*	*time out*	2	0
b15-par	*time out*	*time out*	*time out*	2	0

are unbounded dangerous and are therefore hard to classify. Once a fault is injected the modified state persists for all subsequent time frames but is masked by the majority voter, i.e., the circuit state is corrupted by the specification of the circuit is still kept. Consequently, classifying these components requires that the corresponding proof procedure of the ITP-classifier needs to unroll the circuit up to the completeness threshold k_{cmpl}. However, the classifications are completed by effectively computing suitable approximations before reaching this value. The ITP-classifier provides exact results while the BMC-classifier provides only marginal results leading to very low accuracy.

Run time

Table 7.4 shows the run times of the formal classifiers. The run times for the BMC-classifier and the ATPG-classifier are accumulated over two runs since the lower bound and the upper bound are determined separately. A *time out* is written if at least one run was out of time as well as if the ITP-classifier was out of time.

The search space of the BMC-classifier and ATPG-classifier is bounded by the maximal reachability window and observation window of size 10. The ITP-classifier automatically determines this value but are additionally listed by l and k.

The run times of the BMC-classifier and the ATPG-classifier against the ITP-classifier needs to be compared separately since the approaches are different. The size of search space of the ITP-classifier is higher than size of the BMC-classifier and ATPG-classifier since reachability window and observation window is left open for the ITP-classifier. In the following the BMC-classifier and ATPG-classifier are referred to as static classifiers.

In some cases the run time of the ITP-classifier is higher than for the static classifiers but the obtained bounds are more accurate. Examples for this case are `b10, b13` (compare Table 7.4 and Table 7.2).

However, the difference of the run times of the BMC-classifier and the ATPG-classifier is significant. For example, the ATPG-classifier requires 1065 seconds and the BMC-classifier requires 29056 seconds to classify the circuit `b13-par`. That means in this case the ATPG-classifier outperforms the BMC-classifier considerably. In contrast, the ATPG-classifier is significantly outperformed by the BMC-classifier exemplary in case of circuit `b12`. That means, there are significant cases where both perform differently but it is a priori unknown which classifier would perform better.

The reached reachability windows and observation windows of the ITP-classifier are very different. For example the classification of the circuit `par_b09` could not be completed although a reachability window of 61 and a observation window of 83 were considered. A detailed analysis of this case shows that exactly a single component has not been completely classified which is also reflected by the very tight bounds in Table 7.2. A further detailed analysis shows that in the most cases a considerably portion of the components are classified while considering small reachability windows and observation windows.

Overall, the ITP-classifier generally provides a higher accuracy, i.e., the gap of the bounds is significantly lower and in some cases exact results are provided in terms of equal bounds. For example all components of the

circuit `b12-tmr-ff` are classified by the ITP-classifier in lower run times where the static classifiers completed only a few classifications.

Moreover, as a serious point, the ITP-classifier has typically significant higher memory consumption than the BMC-classifier and ATPG-classifier. This is caused by the interpolation engine computing interpolants from the resolution proof as illustrated in Section 7.1. However, the memory consumption can be drastically reduced by minimizing the resolution proof which directly corresponds to the size of the interpolants. However, available techniques can be easily applied within RobuCheck.

Conspicuously, the run times of the parity circuits are very high for all classifiers. The parity is computed over many `XOR`-gates. The corresponding SAT instance in terms of a CNF is defined over clauses based on conjunctions and disjunctions. The SAT solver has no direct knowledge about those `XOR`-gates. Usually, `XOR` dominated SAT instances are hard to solve which is addressed by several works, e.g., [Soo12] to improve the performance for those instances. There are SAT solver that particularly consider these kinds of instances, e.g., CryptoMiniSAT[5].

7.3.3 SIM-classifier

Results of the SIM-classifier are presented in this section. Recall the SIM-classifier performs random simulation to classify the components by randomly computed input stimulus. The number of simulation traces has been configured to be unlimited. But the run-time of the SIM-classifier was limited to 8 hours. That means, the SIM-classifier terminates once all components are completely classified or runs out of time. Recall, the SIM-classifier explores partially the search space that means may miss corner cases.

Upper Bound

Table 7.5 lists the obtained results of the SIM-classifier. The column $|\mathbb{S}|$ specifies the number of classified non-robust components. The column *Traces* specifies the number of simulated traces. The column R_{ub} denotes the obtained robustness in terms of an upper bound. Additionally the best obtained robustness bound by the formal classifiers are listed as well from Table 7.2. The last column denotes the difference of best obtained robustness bound and the bound obtained by the SIM-classifier.

The differences of the bounds are significant in certain cases although the number of simulated traces is huge. For circuit `b14` the difference is

[5] Available under `http://www.msoos.org/cryptominisat2/`

Table 7.5: Robustness obtained by the SIM-classifier.

Circuit	$\lvert\mathbb{S}\rvert$	Traces	R_{ub}	Best R_{ub}	Diff.
b08	238	26,408,000	0.83%	0.00%	0.83%
b09	120	146,825,000	40.89%	0.49%	40.40%
b10	271	66,569,000	2.87%	1.79%	1.08%
b11	774	270,678,000	13.42%	7.83%	5.59%
b12	633	230,749,000	53.56%	39.47%	14.09%
b13	289	1,238,000	30.36%	48.19%	**-17.83**%
b14	3680	73,878,000	67.71%	13.19%	54.52%
b15	2466	31,492,000	77.65%	28.45%	49.20%
b08-tmr-ff	13	115,674,000	99.11%	98.83%	0.28%
b09-tmr-ff	3	121,234,000	99.81%	99.81%	0.00%
b10-tmr-ff	21	120,123,000	98.56%	98.43%	0.13%
b11-tmr-ff	19	37,554,000	99.50%	99.50%	0.00%
b12-tmr-ff	12	28,545,000	99.85%	99.84%	0.01%
b13-tmr-ff	22	54,085,000	99.29%	99.29%	0.00%
b14-tmr-ff	2	11,018,000	$<$ 100.00%	99.75%	0.25%
b15-tmr-ff	0	6,513,000	100.00%	99.71%	0.29%
b08-par	135	114,533,000	78.16%	74.11%	4.05%
b09-par	34	198,235,000	94.30%	87.10%	7.20%
b10-par	104	108,784,000	85.75%	83.84%	1.91%
b11-par	221	181,717,000	88.22%	83.64%	4.58%
b12-par	123	82,258,000	96.92%	99.85%	**-2.93**%
b13-par	94	86,921,000	92.62%	94.66%	**-2.04**%
b14-par	245	33,399,000	98.79%	99.74%	**-0.95**%
b15-par	137	21,295,000	99.47%	99.73%	**-0.26**%

considerably. The formally best obtained robustness bound is 13.19% where the SIM-classifier determined 67.71%. Thus, a significant difference. In particular, the SIM-classifier cannot prove lower bounds since it requires to prove the absence of faulty behavior which is naturally not provided by simulation. However, this is necessary when showing the correctness of fault-tolerant circuits. But the SIM-classifier is useful as a pre-processor: In case of the circuit `b13` the SIM-classifier completed significant more components than the best formal classifier, i.e, the difference of the bounds is 17.84%. For circuits `b15-tmr-ff`, the SIM-classifier did not complete any classification. For the larger parity circuits, the SIM-classifier classifies a few more components than the best formal classifier.

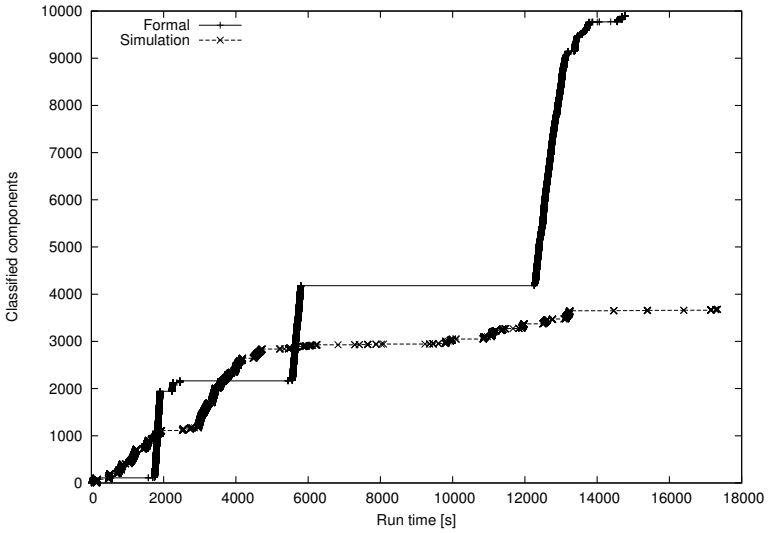


Figure 7.4: Formal vs. simulation

Overall, the SIM-classifier provides a roughly approximate classification. In the following the progress of the classification is illustrated.

Classification Progress

Figure 7.4 illustrates the classification progress. The x-axis denotes the run time and the y-axis the number of completely classified components. The circuit `b14` has been taken as example.

The SIM-classifier comes into a saturation approximately after 5000 CPU seconds. Only a small portion of components are subsequently classified. The formal-classifier systematically explores the search space and classifies more components. However, an integration of both approaches seems to be powerful since the area between both curves can be exploited to skip the classifications. That means once one classifier has completely classified a component the classification can be transfered to the remaining classifier. One can further see, that the formal classifier is even more powerful than the SIM-classifier on this complex benchmark.

Heuristics

In Section 6.7.3 two heuristics have been introduced. In Heuristic #2 the SIM-classifier collects k-dangerous components concurrently while the formal classifiers classifies formally the components. While classification over different time frames components are classified as k-dangerous multiple

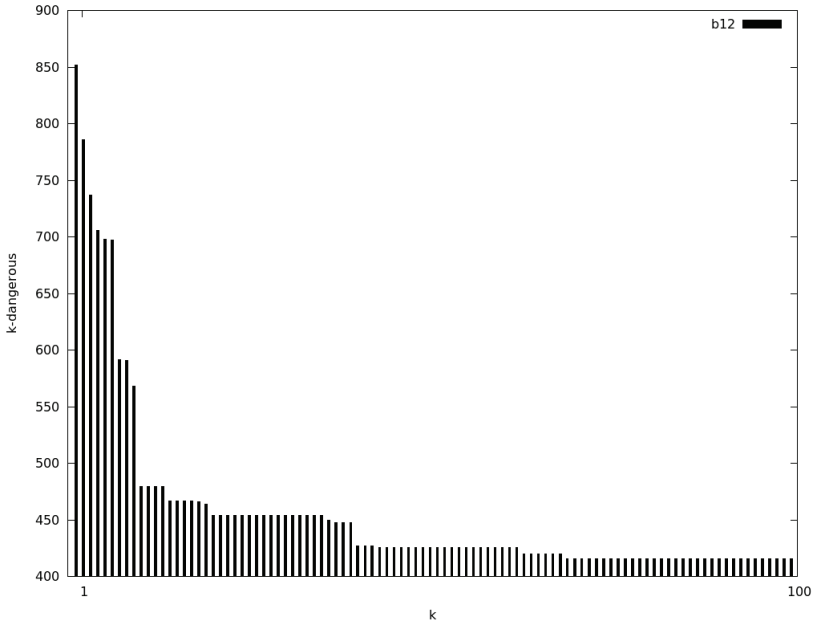


Figure 7.5: Collect k-dangerous components for circuit `b12`

times for various values of k. The heuristic presented in Section 7.3.4. In this heuristic, the SIM-classifier collects k-dangerous components for different values of k.

In Figure 7.5 a histogram of collected k-dangerous is shown for ITC'99 circuit `b12`. This circuit is composed of 1363 components. The x-axis denotes the range of k and y-axis denotes the number of components classified as k-dangerous for various k, respectively. The SIM-classifier ran for only one minute. The observation window has been configured to the interval $[0, 100]$.

For a small value of k there are many k-dangerous components (850 out of 1363), i.e., a fault is observable at the state elements. With increasing k the fault is either propagated to the outputs or masked out, i.e, fewer components are k-dangerous. Note the k-dangerous components might also be k-non-robust components which is not uniquely classified due to the incomplete simulation. But the formal classifiers can access this storage to skip costly k-dangerous classifications which increases the overall performance.

7.3.4 Concurrent Classification

Running the classifiers concurrently are presented in the following evaluation. Due to the global classification storage of RobuCheck the classifier may benefit from classification of other classifiers.

The underlying hardware of the benchmark system has six CPU cores. The following configuration has been chosen:

- BMC-classifier and ATPG-classifier: Both classifiers consider over-approximation and under-approximation of reachable states which yields non-robust and robust components. The observation window has been configured to maximal $\bar{k} = 10$ time frames and in case of an under-approximation $\bar{l} = 0$ time frames has been configured for the reachability window. This yields overall four classifiers.
- ITP-classifier: The ITP-classifier uses internally two CPU cores to compute interpolants as presented in Section 6.7.2.

Overall, all six cores are used within this evaluation. The overall run time was drastically reduced from 8 hours to only 1 hour.

Two different setups are evaluated denoted by *Setup #1* and *Setup #2*. In the first setup Heuristic #1 from Section 6.7.3 has been activated and in the second setup Heuristic #2 from Section . Additionally, in both setups the heuristic of the *Minimal Propagation Path* from Section 5.6 has been turned on to skip costly classifications by a simple structural analysis.

In Table 7.6 the determined robustness bounds of both setups are listed. The first column denotes the name of the circuits. The lower and upper bounds are shown in the remaining columns, respectively.

Even in the limited computational resouce to only one hour run time the accuracy of this evaluation is very high for both setup. Often the results are equal to the result obtained by running the classifiers separately for 8 hours.

However, in some cases the results of Setup #1 and Setup #2 differ. Significant differences are observed for, e.g, circuit `b15` and circuit `b12-tmr-ff`. The progress of the classification is illustrated in Figure 7.6 over the complete run time for both circuits. The x-axis denotes the run time and the y-axis the number of non-classified components.

As it is shown in Figure 7.6 Setup #2 classifies more components per time for both circuits. However, the only difference of the configuration is Heuristic #1 for Setup #1 and Heuristic #2 for Setup #2. That means, in this configuration Heuristic #2 that collects k-dangerous component. concurrently is more effective for these circuits. There are also cases where Setup #1 is better but these cases are less significant.

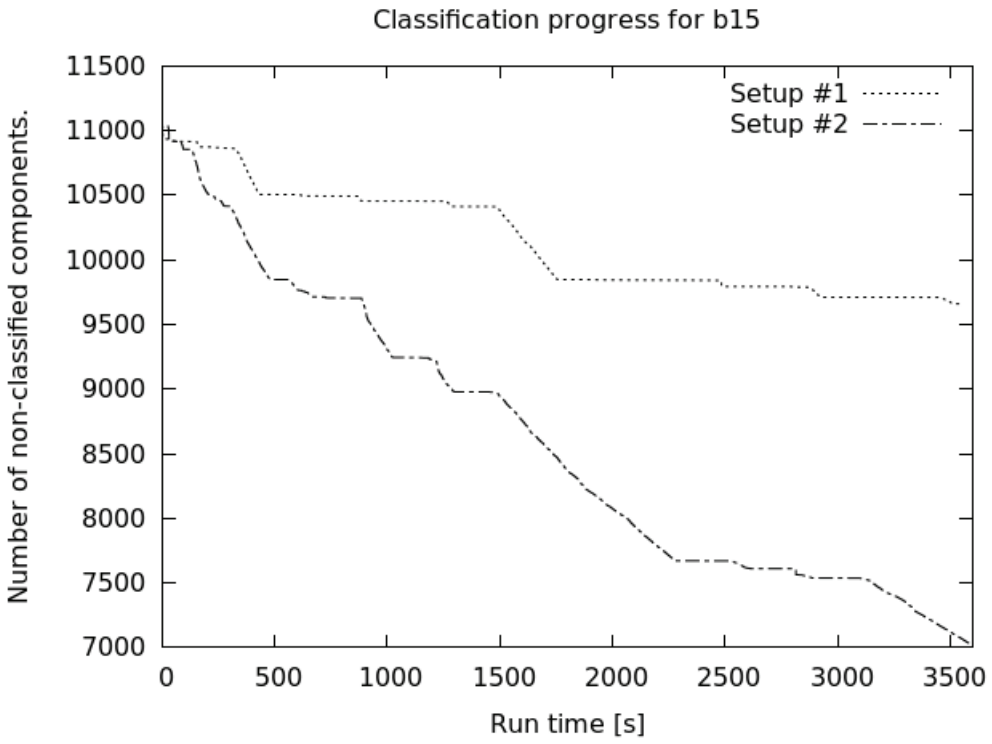
Table 7.6: Determined bounds by Setup #1 and Setup #2.

	Setup #1		Setup #2	
Circuit	$R_{lb}^{\bar{k}}$	$R_{ub}^{\bar{k}}$	$R_{lb}^{\bar{k}}$	$R_{ub}^{\bar{k}}$
b08	0.00%	0.00%	0.00%	0.00%
b09	0.00%	0.49%	0.00%	0.49%
b10	1.79%	1.79%	1.79%	1.79%
b11	0.11%	14.99%	6.49%	8.05%
b12	0.00%	46.37%	0.00%	46.37%
b13	0.96%	53.49%	1.45%	56.39%
b14	0.00%	63.47%	0.01%	79.93%
b15	0.43%	87.94%	0.43%	64.05%
b08-tmr2	98.83%	98.83%	98.83%	98.83%
b09-tmr2	99.81%	99.81%	99.81%	99.81%
b10-tmr2	98.43%	98.43%	98.43%	98.43%
b11-tmr2	99.50%	99.50%	99.50%	99.50%
b12-tmr2	23.62%	99.84%	41.56%	99.84%
b13-tmr2	99.29%	99.29%	99.03%	99.29%
b14-tmr2	0.00%	99.74%	0.00%	99.74%
b15-tmr2	0.00%	99.71%	0.00%	99.71%
b08-par	74.43%	74.43%	74.11%	74.11%
b09-par	86.93%	87.10%	86.93%	87.10%
b10-par	83.97%	83.97%	83.70%	83.70%
b11-par	80.54%	85.13%	73.40%	86.35%
b12-par	36.07%	94.76%	31.68%	99.82%
b13-par	89.32%	94.66%	89.32%	94.66%
b14-par	0.20%	99.97%	0.60%	99.74%
b15-par	2.50%	99.97%	0.01%	99.73%

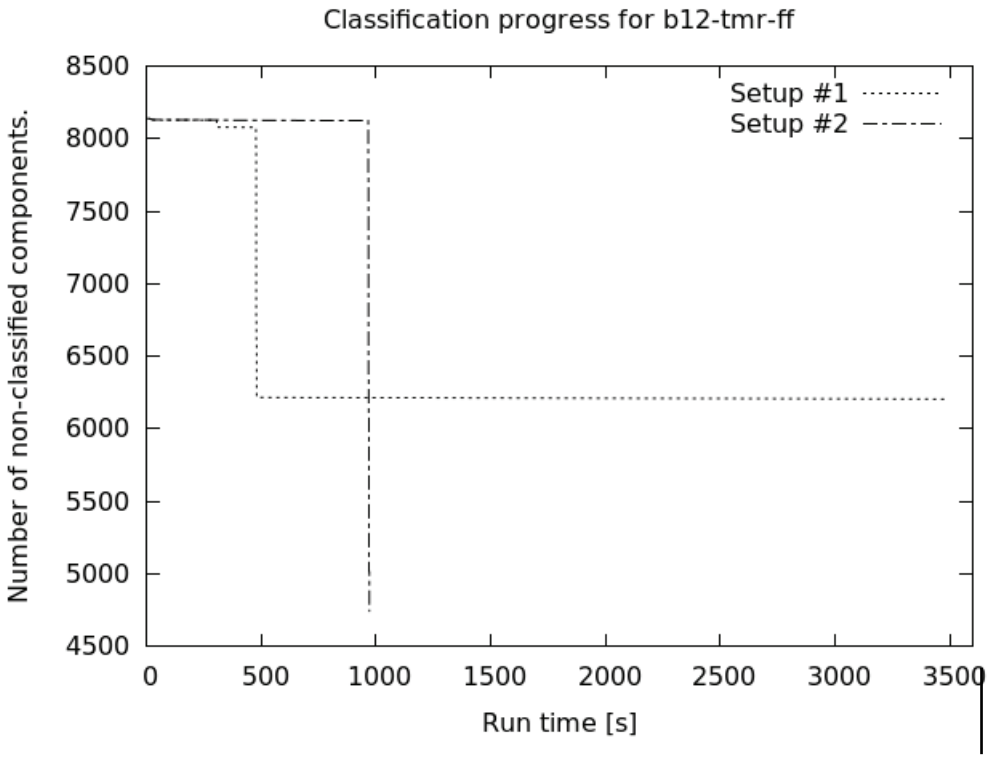
7.3.5 Probabilistic Analysis

In Section 4.2 a technique was proposed that computes a differentiation of non-robust components. The results of the evaluation of this technique is presented in the following.

Combinational circuits were taken from the LGsynth93 benchmark suite and sequential circuits from the ITC'99 benchmark suite, respectively. For every circuit a parity checker was implemented as introduced in Section 2.6.1. A time out was set to 5000 CPU seconds. Exceeding this limit is denoted by *time out*.



(a) b15



(b) b12-tmr-ff

Figure 7.6: Progress of `b15` and `b12-tmr-ff` by Setup #1 and Setup #2

Table 7.7: Results for combinational circuits.

			Worst Case		EPP-based (100 scenarios)				EPP-based (10,000 scenarios)				
Circuit	$\|X\|$	$\|V\|$	R_{lb}	$\|\mathbb{S}\|$	R_{ub}^{λ}	λ	Run time	MAA	R_{ub}^{λ}	λ	$> \lambda\Psi(1)$	Run time	MAA
par_5xp1	7	391	88.44%	49	**95.35%**	100.00%	5.96	8.55					
par_9sym	9	655	98.69%	9	**99.71%**	97.66%	3.73	9.6	**99.71%**	100%	0	3.89	9.64
par_apex7	49	720	77.48%	275	**77.48%**	$< 0.01\%$	465.14	32.41	**77.48%**	$< 0.01\%$	204	*time out*	170.75
par_cm42a	4	81	85.71%	15	**92.02%**	100.00%	0.17	0.21					
par_cm82a	5	62	73.17%	22	**86.97%**	100.00%	0.2	0.26					
par_cmb	16	136	63.16%	70	**90.27%**	0.76%	5.55	5.52	**95.91%**	15.25%	6	76.64	44.89
par_comp	32	385	41.36%	285	**41.36%**	$< 0.01\%$	300.79	864.64	–	–		*time out*	*time out*
par_con1	7	65	81.11%	17	**95.01%**	100.00%	0.26	0.36					
par_cordic	23	2866	97.65%	69	**97.65%**	0.01%	524.98	*time out*	–	–		*time out*	*time out*
par_cu	14	166	76.92%	51	**79.63%**	3.05%	11.77	2.86	**92.88%**	61.03%	3	108.04	29.95
par_duke2	22	976	76.23%	255	**76.42%**	0.01%	384.39	620.65	–	–		*time out*	*time out*
par_e64	65	1409	88.45%	193	**89.59%**	$< 0.01\%$	553.68	740.04	–	–		*time out*	*time out*
par_f51m	8	318	91.76%	29	**95.09%**	100.00%	4.62	6.82					
par_frg1	28	393	92.74%	35	**92.74%**	$< 0.01\%$	25.15	13.85	**92.74%**	$< 0.01\%$	35	822.3	324.25
par_rd84	8	1157	82.81%	204	**96.14%**	100.00%	48.73	111.83					
par_rot	135	1932	76.17%	583	**76.17%**	$< 0.01\%$	4550.92	170.28	**76.17%**	$< 0.01\%$	583	*time out*	556.64
par_sao2	10	502	82.16%	96	**94.41%**	48.83%	19.39	46.65	**96.96%**	100%	0	21.24	50.74
par_sqrt8ml	8	447	60.80%	187	**91.92%**	100.00%	19.46	54.26					
par_squar5	5	273	80.87%	57	**91.04%**	100.00%	1.2	1.71					
par_t481	16	1752	99.11%	16	**99.11%**	0.76%	62.64	450.69	**99.11%**	15.26%	16	1304.48	*time out*
par_table5	17	1455	70.58%	448	**75.07%**	0.38%	858.44	1963.05	–	–		*time out*	*time out*

Table 7.7 shows the results for the combinational circuits. The first three columns describe properties of the circuit: the name, the number of primary inputs and the number of components in the circuit. Note that for combinational circuits SDC cannot occur, i.e. all components are classified as robust or as non-robust. Consequently, there is only a single value for the robustness of such circuits as introduced in Section 4.2. The results of the worst case analysis, the new measure using 500 scenarios and the new measure using 10,000 scenarios are given in the following columns. The parameter λ has been adjusted accordingly. For the worst case analysis, the robustness value and the number of non-robust components are shown in columns R_{ub} and $|\mathbb{S}|$, respectively. For the new measure, the robustness value R_{ub}^{λ}, the parameter λ, the overall run time t in CPU seconds without *Minimal Assignment Analysis* (MAA) (Section 6.2.2), and the run time (MAA) with MAA are shown in the respective columns. Additionally, column $> \lambda\Psi$ gives the number of components having more than 10,000 scenarios. Blank cells denote that no computation with 10,000 scenarios was required as all components had less than 500 scenarios. The run times are longer for the new measure as usually more than a single scenario has to be considered before the classification of a non-robust component is completed. For small circuits the use of MAA increases the run time (e.g. for `par_5xp1`). In these cases the SAT solver efficiently enumerates multiple solutions. In contrast, when the number of inputs increases, MAA often yields shorter run times as a single solution of the SAT solver is generalized to many scenarios (e.g. for `par_apex9` and for `par_rot`). The robustness value for the new measure is typically larger than the one for the worst-case analysis. As soon as a single scenario exists a component is classified as "completely non-robust" in the worst-case analysis. For the new measure non-robust components are graded by the number of scenarios. As long as the number of scenarios is below the predefined limit, a component contributes to the circuit's robustness. For example, this case occurs for the circuits `par_cmb` and `par_cu`. In such cases a fine grain differentiation between non-robust components is available. The designer decides whether further protection is required for some of these components. If the number of scenarios always exceeds the predefined limit, the robustness values are identical for the worst case analysis and the new measure. For example, this occurs in case of `par_rot` and `par_t481`. All non-robust components must be considered as hot spots and further hardening techniques have to be taken to handle transient faults.

A more detailed evaluation for the combinational circuit `par_cu` is shown by the histogram in Figure 7.7. The x-axis depicts the number of scenarios. The y-axis gives the number of components that had a certain number of

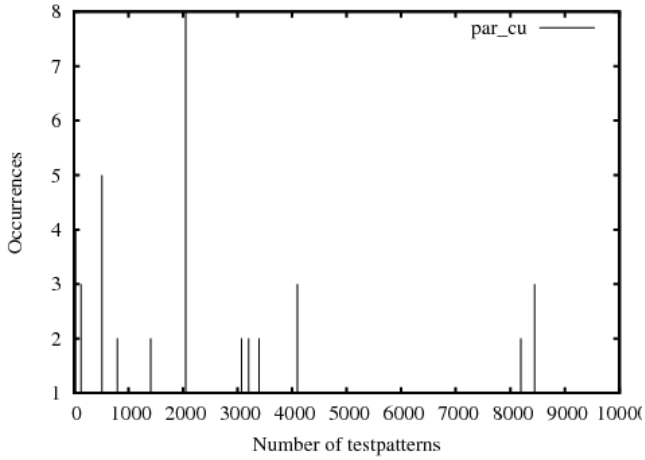


Figure 7.7: Histogram of circuit `par_cu`

scenarios. In total there are $2^{14} = 16,384$ input scenarios. The number scenarios considered was limited to 10,000 as in the previous experiment. The worst-case analysis yields 51 non-robust components. For most of these components there exist less than 10,000 scenarios . A very fine grain differentiation between non-robust components is determined.

7.3.6 COMP-classifier

The COMP-classifier implements a compositional reasoning. That means, a set of subcircuits are locally classified and the results are composed with the entire circuit if necessary. In this evaluation combinational circuits are considered.

The COMP-classifier has been compared against a monolithic approach. The BMC-classifier has been chosen as monolithic approach considering only one time frame since combinational circuits are analyzed.

Two different kinds of circuits have been evaluated that are described in the following.

- Two circuits from the ISCAS'85 benchmark suite have been taken. All *Fanout-Free Regions* (FFR) of the circuits have been hardened with TMR. Each FFR has a fault signal. All local fault signals are propagated to a global fault signal.
- An *Arithmetic Logic Unit* (ALU) that consists of an multiplier and an adder circuit. Operands can be selective shifted or negated. The bit-width of the operands can be arbitrarily chosen. For this evaluation

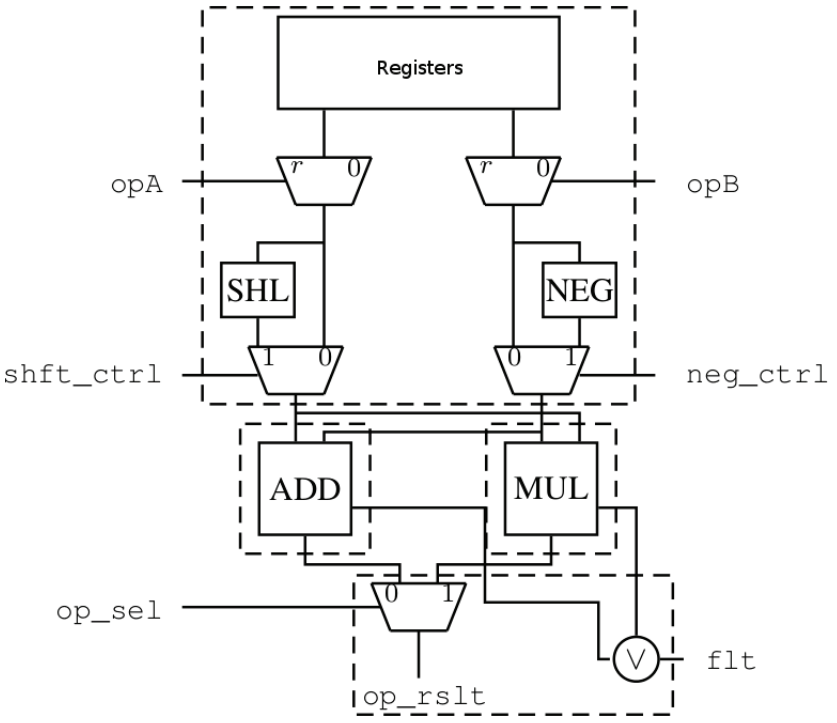


Figure 7.8: Arithmetic logic unit

the bit-width 8 and 16 has been generated. Figure 7.8 shows the the ALU.

Results of the circuits are shown in Table 7.8. The first three columns show the circuit name, size of the circuit $|V|$, and number of subcircuits n. The columns *R(C)* and *Run time (C)* show the robustness and the run time of the COMP-classifier. The columns *R (M)* and *Run time(M)* show the robustness and the run time of the BMC-classifier.

The run time of both approaches was limited to 5000 CPU seconds. A time out is denoted by *time out*.

ISCAS'85 Circuits

For robustness checking each FFR has been chosen as subcircuit. As shown in Table 7.8 this leads to 929 subcircuits for the c5313-red circuit, and 1408 subcircutis for the c7552-red circuit. The first circuit is fully classified by the COMP-classifier and the BMC-classifier while the COMP-classifier outperforms the BMC-classifier by a factor of two. The larger circuit was only fully classified by the COMP-classifier. The BMC-classifier was able to

Table 7.8: Monolithic vs. Compositional

Circuit	$\|V\|$	n	R (C)	Run time (C)	R (M)	Run time (M)
c5315-red	20295	929	89.68%	2147.83s	72.50%	1071.52s
c7552-red	30795	1408	$< 95.86\%$	*time out*	70.91%	3069.89s
ALU-8	1204	4	74.75%	818.81s	73.69%	18.57s
ALU-16	2884	4	-	*time out*	70.97%	165.99s

classify approximately 1000 components within 5000 seconds such that an upper bound of the robustness is provided. The COMP-classifier provides a lower bound of the robustness. The most components are correctly as robust classified by the COMP-classifier. In some cases, the approximation of the propagation is too optimistic.

Arithmetic Logic Unit

The subcircuits have been chosen according the dashed rectangle shown in Figure 7.8. Knowledge about the functionalities of the respective components are exploied to derive good subcircuits. Transient faults are tolerated by the adder module `ADD` and the multiplier module `MULT`. A modulo-3 check has been implemented in both arithmetic circuits. Detected transient faults are reported by the local fault signals and propagated to the top level fault signal.

The 8-bit ALU has been fully classified by both approaches while the COMP-classifier outperforms the BMC-classifier by a factor of 43. The accuracy of the COMP-classifier is in this case very high despite the optimistic approximation. The 16-bit ALU was not fully classified. The BMC-classifier does not complete any classification. The COMP-classifier was not able to fully classify the multiplier subcircuit. The listed robustness is computed after the COMP-classifier ran out of time. Based on the good choice of the subcircuit the COMP-classifier provides a useful lower bound of the robustness.

Based on the observation that the ALU is part of a complex processor design it is illustrated how the COMP-classifier can be applied to determine the robustness of complex circuits. Functional tests coming from contrained random simulation are often available that can be used to sensitize paths within the COMP-classifier.

Table 7.9: Robustness checking by CADENCE SMV.

Circuit	$R_{lb}^{k_{\mathsf{cmpl}}}$	$R_{ub}^{k_{\mathsf{cmpl}}}$	Run time
b08	0%	0%	2652s
b09	0%	47.52%	*time out*
b10	1.79%	1.79%	5309s
b11	0%	100%	*time out*
b12	0%	100%	*time out*
b13	0%	100%	*time out*
b14	0%	100%	*time out*
b15	0%	100%	*time out*

7.3.7 Robustness Checking by Means of Model Checking

As presented in Section 5.5 robustness checking can be translated into a model checking problem which can be solved by state-of-the-art available model checkers. This section presents the results when using a model checker to perform robustness checking.

For this evaluation CADENCE SMV[6] has been used. Preliminary experiments turned out that the model checker NUSMV[7] did not return any proper result in reasonable run times, i.e., no classification was completed.

CADENCE SMV supports the input format SMV. In an SMV file the model and the CTL property are specified. Technically, for each component ROBUCHECK generates an SMV file according to the model from Section 5.5.

The original ITC'99 circuits and the derived system-level TMR circuits are checked using CADENCE SMV. The experiments were conducted on an AMD Opteron$^{\text{TM}}$CPU with six cores running at 2.8Ghz with 32GB main memory. The run time for each classification was limited by 600 seconds. The overall run time of the classification was limited to 8 hours. A time out is denoted by *time out*. The model checker was run using the default configuration. The reachability window and observation window are automatically chosen by the model checker that internally guarantee completeness.

For all TMR circuits, the model checker ran out of time without any classification. Even for the small circuits `b08-tmr-sys` with 795 components the classification did not succeed.

The results for the original ITC'99 circuits are listed in Table 7.9. The first column denotes the name of the circuit. The second the third column

[6] Available under `http://w2.cadence.com/webforms/cbl_software/index.aspx`
[7] Available under `http://nusmv.fbk.eu/`

denotes the determined robustness bounds. The last column lists the wall clock time in seconds.

Only two circuits are fully classified leading to equal robustness bounds. However, the best classifier integrated in RobuCheck outperforms the model checker by the factor of 492 for `b08` and 254 for `b10`. Despite the low size of both circuits circuits, `b08` with 240 components, and `b10` with 279 components, the run times are relatively long.

Overall, dedicated verification approaches exploits problem domain knowledge that leads to signifanctly increased performance.

7.3.8 IBM Benchmarks

In the author's work [FFA+12] parts of the ITP-classifier have been published as a joint work with IBM. In this paper the ITP-classifier was evaluated on IBM benchmarks which are presented in the following.

Table 7.10 lists the results of the IBM benchmarks. The benchmarks are grouped: data-path circuits denoted by D1 to D13 and circuits taken from a multi-processor control unit denoted by D14 to D30. The benchmarks were conducted in an Intel i5 processor running at 3.1GHz with 4GB main memory. The first four columns specify the characteristics of the circuits. The column *Classified* denotes the number of components that have been fully classified. The column l and k denotes the reached values for l and k. The run times are shown in the last column provided in CPU seconds. The analysis of the components is restricted to flip flops.

A significant portion of the flip flops were classified for the most circuits. If RobuCheck ran out of time or out of memory less than 100% were classified. In those cases the underlying problem instance may get too large and consequently the search space becomes too complex. The experiments were conducted by an older version of RobuCheck using only the proof-based approach to obtain interpolants. A newer version of RobuCheck that integrates the model-based approach may classifier more components since this approach consumes usually much fewer memory. Overall, RobuCheck was able to produce a significant outcome for industrial circuits.

Table 7.10: IBM Benchmarks

Circuit	$\|X\|$	$\|Y\|$	$\|FF\|$	Classified [%]	l	k	Runtime
D1	204	259	1430	9.30%	2	0	2728
D2	228	65	1424	17.49%	5	1	783
D3	727	293	1395	7.74%	3	0	220
D4	700	497	1038	70.52%	7	1	678
D5	364	142	940	100.00%	2	1	60
D6	105	60	699	99.86%	2	57	1699
D7	284	262	513	84.99%	18	3	3797
D8	112	56	456	100.00%	6	2	144
D9	268	99	447	89.26%	8	1	8281
D10	734	194	435	87.36%	2	1	611
D11	155	120	394	100.00%	2	1	11
D12	53	37	322	100.00%	2	1	19
D13	124	67	222	48.20%	5	3	37
D14	119	112	878	81.55%	5	3	1492
D15	140	55	804	88.56%	31	0	710
D16	29	24	555	15.86%	61	1	1201
D17	377	25	506	70.16%	6	6	1504
D18	176	154	464	70.26%	55	1	2044
D19	252	131	451	56.54%	7	7	1714
D20	327	102	428	67.06%	3	44	9050
D21	173	256	412	88.35%	8	4	486
D22	135	206	247	90.28%	2	33	23170
D23	218	96	231	95.24%	2	86	3631
D24	119	57	231	96.54%	2	2	580
D25	227	114	216	95.37%	4	95	7589
D26	70	51	210	17.62%	131	0	3697
D27	103	63	207	91.30%	7	6	598
D28	130	79	195	95.90%	2	80	35888
D29	100	37	126	100.00%	5	5	353
D30	139	94	123	100.00%	4	5	59

Chapter 8

Conclusion and Future Work

The vulnerability of transient faults increases significantly with continuously shrinking feature sizes. Tolerating transient faults is possible based on strong hardening techniques such as *Triple Modular Redundancy* or *Error-Correcting-Codes*. The implementation of those techniques might be buggy itself. Thus, fault tolerance in terms of robustness needs to be verified. This task has been introduced in this thesis under the term robustness checking.

The thesis starts by introducing an adequate fault model to handle transient faults at logic level. Transient faults are considered as a non-deterministically value change of a component's outputs. Not only Boolean values are considered, a more complex component model allows to cover local multiple transient faults as well. Usually, models that consider multiple faults are huge but in this thesis local multiple transient faults become manageable.

A transient fault may affect the circuit's function differently. According to the different behavior a categorization of each component in terms of k-non-robust, k-dangerous, unbounded dangerous, and robust is introduced. This step is called classification.

Moreover, objective and unique measures are introduced to document the circuit's robustness. The first measure consider the worst case that a scenario and a transient fault occur. This leads to a conservative measurement. The second measure consider a configurable probability that a pre-defined number of scenarios and transient faults occur. This measure reflects more the behavior of a circuit during operation since not every scenario may occur in practice. However, the computational effort of computing both measures is very different. A trade-off between accuracy and run time can be found.

A basic algorithm has been introduced that follows the introduced computational model to classify the components into the respective classes.

The accuracy of the classification strongly depends on the considered set of states for the fault injection. But exact reachability information is hard to compute. Therefore approximation techniques are embedded into the classification and the influences of the classification are formally emphasized and considered in the respective measures. Here, new techniques to compute approximations have been introduced based on Craig interpolation. Moreover, a new model checker results that significantly outperforms a state-of-the-art model checker for a subset of relevant benchmarks.

Overall, various approaches adapted from the well-known *Bounded Model Checking* (BMC), *Automatic Test Pattern Generation* (ATPG), *Interpolation*-based BMC, *Compositional* verification and *Random simulation* have been introduced to classify the circuit's components. All these approaches are integrated into a highly-optimized flow of robustness checking within the verification tool called RobuCheck. RobuCheck is able to formally classify the components of any kinds of combinational and sequential circuits and highlights the obtained results in a strong visualization engine that pin-points the designer directly to vulnerable components.

The demand for a dedicated verification flow has been demonstrated by translating the problem of robustness checking into a model checking problem. Those model checking instances have been solved by a industrial-strength model checker. However, this approach provides poor accuracy. The proposed algorithms of this thesis outperform this model checking approach considerably while providing significant higher accuracy.

The integrated classifiers in RobuCheck can be called concurrently or consecutively that allows to particularly configure the verification for different circuit classes. Overall, the evaluation of academic and industrial benchmarks shows the effectiveness of RobuCheck. Even when the computational resources in terms of run time is drastically limited high accuracy is reached by RobuCheck. Even more, by using sophisticated approximation techniques the accuracy is often very high and even in case of hard circuit the ITP-classifier provides exact results.

Possible future work directions are to lift the models of robustness checking to a higher level of abstraction in order to handle more complex circuits. Here the first steps are made with the compositional approach. Moreover, the problem formulation of the ITP-classifier can be lifted to *Satisfiable Modulo Theory* (SMT) to provide more compact problem instances and finally to more compact interpolants. This may increase the performance and decrease the memory consumption considerably.

List of Symbols

$T(s, s')$	transition from state s to state s'
I	predicate of the initial state
P	predicate describing the property
$\bar{k}$	maximal size of the observation window $\bar{k} \in [0, k_{\mathsf{cmpl}}]$
k	size of the observation window $k \in [0, \bar{k}]$
k_{cmpl}	completeness threshold to cover all propagation paths
l	number of time frames considered after the initial state
l_{cmpl}	completeness threshold to cover all reachable states
$\bar{l}$	maximal size of the reachability window $\bar{l} \in [0, l_{\mathsf{cmpl}}]$
$\mathbb{S}_k$	set of k-non-robust components
$\mathbb{D}_k$	set of k-dangerous components
$\mathbb{T}$	set of robust components
$\hat{\mathbb{S}}_k$	set of spurious k-non-robust components
$\hat{\mathbb{D}}_k$	set of spurious k-dangerous components
$\check{\mathbb{S}}_k$	subset of k-non-robust components
$\check{\mathbb{D}}_k$	subset of k-dangerous components
$\sigma, \hat{I}$	Craig interpolants
A, B	Boolean formulas
$R_{lb}^{\bar{k}}$	lower bound of the robustness with respect to $\bar{k}$
$R_{ub}^{\bar{k}}$	upper bound of the robustness with respect to $\bar{k}$
S_0	states for fault injection
S^*	set of all reachable states
$\hat{S}$	over-approximation of reachable states
$\check{S}$	under-approximation of reachable states

List of Figures

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[AK84] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *IEEE Computer*, 17(8):67–80, 1984.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Lanwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing*, 1(1):11–33, 2004.

[Ash01] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001.

[AV02] J.A. Abraham and V.M. Vedula. Verifying properties using sequential ATPG. In *Internatioal Test Conference*, pages 194 – 202, 2002.

[Bau02] R. Baumann. Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121, 2002.

[Bau05] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305 – 316, sept. 2005.

[BBC+09] S. Baarir, C. Braunstein, R. Clavel, E. Encrenaz, J.-M. Ilie, R. Leveugle, I. Mounier, L. Pierre, and D. Poitrenaud. Complementary formal approaches for dependability analysis. In *DFT '09*, pages 331 –339, oct. 2009.

[BBL+12] S. Baeg, J. Bae, S. Lee, C.S. Lim, S.H. Jeon, and H. Nam. Soft error issues with scaling technologies. In *Asian Test Symp.*, pages 68–68, 2012.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[BCG+10] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, and B. Jobstmann. Robustness in the presence of liveness. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 410–424. 2010.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, pages 428 –439, 1990.

[BCT07] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *Automated technology for verification and analysis*, pages 162–176, 2007.

[Ber68] E. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, 1968.

[BF76] M.A. Breuer and A.D. Friedman. *Diagnosis and reliable design of digital systems*. Digital system design series. Computer Science Press, 1976.

[Bie08] A. Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[Bie10] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2010.

[Bie12] A. Biere. Aiger package. Technical report, Johannes Kepler University, Linz, Austria, 2012.

[BIFH+11] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Reducing the size of resolution proofs in linear time. *International Journal of Software Tools Technology Transfer*, 13(3):263–272, June 2011.

[BIMM12] J. Baumgartner, A. Ivrii, A. Matsliah, and H. Mony. IC3-guided abstraction. In *Int'l Conf. on Formal Methods in CAD*, pages 182–185, 2012.

[BKA02] J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer Aided Verification*, pages 151–165, 2002.

[Bor07] S. Borkar. Thousand core chips: A technology perspective. In *Design Automation Conf.*, pages 746–749, 2007.

[Bra11] A. R. Bradley. SAT-based model checking without unrolling, 2011.

[BRTF99] Vamsi Boppana, SreerangaP. Rajan, Koichiro Takayama, and Masahiro Fujita. Model checking based on sequential atpg. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 418–430. Springer Berlin Heidelberg, 1999.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677 –691, aug. 1986.

[CCK03] P. Chauhan, E.M. Clarke, and D. Kroening. Using SAT based image computation for reachability analysis. Technical Report 2197, Carnegie Mellon University, 2003.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. 2000.

[CGP01] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[CIM12] H. Chockler, A. Ivrii, and A. Matsliah. Interpolants without proofs. In *Haifa Verification Conference*, 2012.

[CKOS04] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. 2004.

[CLM89] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[CMB06] M.L. Case, A. Mishchenko, and R. K. Brayton. Inductively finding a reachable state space over-approximation. In *Int'l Workshop on Logic Synth.*, pages 172–179, 2006.

[CMCHG96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. 1996.

[CMR+02] P. Civera, L. Macchiarulo, M. Rebaudengo, M.Sonza Reorda, and M. Violante. An fpga-based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing*, 18:261–271, 2002.

[Con09] Concept Engineering GmbH. *RTLvision PRO*. http://www.concept.de, 2009.

[Coo71] S.A. Cook. The complexity of theorem proving procedures. In *3. ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CPL+09] A. Czutro, I. Polian, M. Lewis, P. Engelke, S. M. Reddy, and B. Becker. Tiguan: Thread-parallel integrated test pattern generator utilizing satisfiability aaalysis. In *International Conference on VLSI Design*, pages 227–232, 2009.

[Cra57] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):pp. 250–268, 1957.

[CRP+96] H. Cha, E.M. Rudnick, J.H. Patel, R.K. Iyer, and G.S. Choi. A gate-level simulation environment for alpha-particle-induced transient faults. *IEEE Trans. on CAD*, 45(11):1248 –1256, nov 1996.

[DB98] R. Drechsler and B. Becker. *Graphenbasierte Funktionsdarstellung*. B.G. Teubner, Stuttgart, 1998.

[DEFT09] R. Drechsler, SS. Eggersglüß, G. Fey, and D. Tille. *Test Pattern Generation using Boolean Proof Engines*. Springer, 2009.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DJBT81] J. A. Darringer, W.H. Joyner, C. L. Berman, and L Trevillyan. Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272 –280, july 1981.

[DKPW10] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. 2010.

[DLL62] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:506–521, 1960.

[DPK08] V. D'Silva, M. Purandare, and D. Kroening. Approximation refinement for interpolation-based model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 68–82, 2008.

[ED07] S. Eggersglüß and R. Drechsler. Improving test pattern compactness in SAT-based ATPG. In *Proceedings of the 16th Asian Test Symposium*, ATS '07, pages 445–452. IEEE Computer Society, 2007.

[ED11] S. Eggersglüß and R. Drechsler. Efficient data structures and methodologies for SAT-based ATPG providing high fault coverage in industrial application. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(9):1411 –1415, sept. 2011.

[EMA10] N. Een, A. Mishchenko, and N. Amla. A single-instance incremental sat formulation of proof- and counterexample-based abstraction. In *Int'l Conf. on Formal Methods in CAD*, pages 181–188, 2010.

[Eme95] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science*, pages 995–1072. Elsevier, 1995.

[EMS07] N. Een, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up SAT. In *SAT*, pages 272–286, 2007.

[ES03] N. Een and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FD08] G. Fey and R. Drechsler. A basis for formal robustness checking. In *Int'l Symp. on Quality Electronic Design*, pages 784–789, 2008.

[FF10] S. Frehse and G Fey. Kompositionelle Formale Robustheitsprüfung. In *Zuverlässigkeit und Entwurf*, pages 73–74, 2010.

[FFA+12] S. Frehse, G. Fey, E. Arbel, K. Yorav, and R. Drechsler. Complete and effective robustness checking by means of interpolation. In *Int'l Conf. on Formal Methods in CAD*, pages 82–90, 2012.

[FFD10] S. Frehse, G. Fey, and R. Drechsler. A better-than-worst-case robustness measure. In *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 78–83, 2010.

[FFSD09] S. Frehse, G. Fey, A. Suflow, and R. Drechsler. Robustness check for multiple faults using formal techniques. In *Digital System Design, Architectures, Methods and Tools, 2009. Digital System Design, Conference on*, pages 85 –90, aug. 2009.

[FFSD10] S. Frehse, G. Fey, A. Sülflow, and R. Drechsler. Robucheck: A robustness checker for digital circuits. In *EUROMICRO Symp. on Digital System Design*, pages 226–231, 2010.

[FHD+11] S. Frehse, F. Haedicke, M. Diepenbeck, G. Fey, and R. Drechsler. Hochoptimierter Ablauf zur Robustheitsprüfung. In *Zuverlässigkeit und Entwurf*, pages 35 – 42, 2011.

[FRF12] S. Frehse, H. Riener, and G. Fey. Hardware-software-cosynthese zur verbesserung der fehlertoleranz. In *Zuverlässigkeit und Entwurf*, pages 90–96, 2012.

[FSD09] G. Fey, A. Sülflow, and R. Drechsler. Computing bounds for fault tolerance using formal techniques. In *Design Automation Conf.*, pages 190–195, 2009.

[FSF11] A. Finder, A. Sülflow, and G. Fey. Latency analysis for sequential circuits. In *European Test Conf.*, pages 129–134, 2011.

[FSFD11] G. Fey, A. Sulflow, S. Frehse, and R. Drechsler. Effective robustness analysis using bounded model checking techniques.

Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 30(8):1239 –1252, aug. 2011.

[FSSFa10] G. Fey, A. Sülflow, and R. Drechsler S. Frehse and. Automatische formale Verifikation der Fehlertoleranz von Schaltkreisen. *it-Information Technology*, 42(4):216–223, 2010.

[FWD10] S. Frehse, R. Wille, and R. Drechsler. Efficient simulation-based debugging of reversible logic. In *Int'l Symp. on Multi-Valued Logic*, pages 156 –161, 2010.

[GOSM08] M. Gössel, V. Ocheretny, E. Sogomonyan, and D. Marienfeld. *New Methods of Concurrent Checking*. Frontiers in Electronic Testing Series. Springer Science+Business Media B.V., 2008.

[Gro12] ABC Group. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/ alanmi/abc/, 2012.

[GS05] A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *Computer Aided Verification*, pages 112–124, 2005.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Jour.*, 26(2):147–160, 1950.

[Hel63] L. Hellerman. A catalog of three-variable or-invert and and-invert logical circuits. *Electronic Computers, IEEE Transactions on*, EC-12(3):198 –223, june 1963.

[HFF+11] F. Haedicke, S. Frehse, G. Fey, D. Grosse, and R. Drechsler. metasmt: Focus on your application not on solver integration. In *International Workshop on Design and Implementation of Formal Tools and Systems*, pages 22 – 29, 2011.

[HH08] M. Hunger and S. Hellebrand. Verification and analysis of self-checking properties through ATPG. *11th IEEE International On-Line Testing Symposium*, 0:25–30, 2008.

[HHC+09] M. Hunger, S. Hellebrand, A. Czutro, I. Polian, and B. Becker. Atpg-based grading of strong fault-secureness. In *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, pages 269 –274, june 2009.

[HLGD12] F. Haedicke, H.M. Le, D. Grosse, and R. Drechsler. CRAVE: An advanced constrained random verification environment for SystemC. In *System on Chip (SoC)*, pages 1 –7, oct. 2012.

[HPB07] J.P. Hayes, I. Polian, and B. Becker. An analysis framework for transient-error tolerance. In *VLSI Test Symposium, 2007. 25th IEEE*, pages 249 –255, may 2007.

[Hua95] G. Huang. Constructing Craig interpolation formulas. In *Annual International Conference on Computing and Combinatorics*, pages 181–190, 1995.

[IS75] O.H. Ibarra and S.K. Sahni. Polynomially complete fault detection problems. *Computers, IEEE Transactions on*, C-24(3):242 – 249, march 1975.

[JBH12] M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on cnf. *Journal of Automated Reasoning*, pages 1–37, 2012.

[JFWD10] J.C. Jung, S. Frehse, R. Wille, and R Drechsler. Enhancing debugging of multiple missing control errors in reversible logic. In *Great lakes symposium on VLSI*, pages 465–470, 2010.

[JG03] N. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2003.

[Kai11] R. Kaivola. Intel core$^{\mathrm{tm}}$ i7 processor execution engine validation in a functional language based formal framework. In *PADL*, page 1, 2011.

[KGN+09] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *CAV*, pages 414–429, 2009.

[KK07] I. Koren and C.M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[KMH05] S. Krishnaswamy, I.L. Markov, and J.P. Hayes. Logic circuit testing for transient faults. In *Test Symposium, 2005. European*, pages 102 – 107, may 2005.

[KPJ+06] 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 *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1 –6, march 2006.

[KPMH07] S. Krishnaswamy, S.M. Plaza, I.L. Markov, and J.P. Hayes. Enhancing design robustness with reliability-aware resynthesis and logic simulation. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 149 –154, nov. 2007.

[Kra97] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *The Journal of Symbolic Logic*, 62(2):457–486, 1997.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In *IN ECAI-92*, pages 359–363. Wiley, 1992.

[KSMS11] H. Katebi, K. A. Sakallah, and J.P. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *SAT*, SAT'11, pages 343–356, 2011.

[Lar92] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

[LT93] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18 –41, july 1993.

[McM03] K.L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, pages 1–13, 2003.

[MMZ+01a] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, Design Automation Conf., pages 530–535, New York, NY, USA, 2001. ACM.

[MMZ+01b] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conf.*, pages 530–535, 2001.

[MR08] J. A. Maestro and P. Reviriego. Study of the effects of mbus on the reliability of a 150 nm sram device. In *Design Automation Conf.*, pages 930–935, 2008.

[MSS99] J.P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. on CAD*, 48(5):506–521, 1999.

[MZM06] N. Miskov-Zivanov and D. Marculescu. Circuit reliability analysis using symbolic techniques. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(12):2638 –2649, dec. 2006.

[MZM10] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(10):1614–1627, 2010.

[Nic11] M. Nicolaidis. *Soft Errors in Modern Electronic Systems*. Frontiers in Electronic Testing. Springer, 2011.

[NVI12] NVIDIA. NVIDIA's next generation CUDA^{TM}compute architecture: Fermi. Technical report, NVIDIA GmbH, 2012.

[Par10] V. Paruthi. Large-scale application of formal verification: from fiction to fact. In *Int'l Conf. on Formal Methods in CAD*, pages 175–180, 2010.

[PCZ+08] A. Pellegrini, K. Constantinides, Dan Zhang, S. Sudhakar, V. Bertacco, and T. Austin. Crashtest: A fast high-fidelity FPGA-based resiliency analysis framework. In *Computer Design, ICCD 2008. IEEE International Conference on*, pages 363 –370, 2008.

[PD11] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, February 2011.

[PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, September 1986.

[Pnu77] A. Pnueli. The temporal logic of programs. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:46–57, 1977.

[Pud97] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *The Journal of Symbolic Logic*, 62(3):981–998, 1997.

[RF12] H. Riener and G. Fey. Model-based diagnosis versus error explanation. In *Formal Methods and Models for Codesign (MEMOCODE), 2012 10th IEEE/ACM International Conference on*, pages 43 –52, july 2012.

[RFF12] H. Riener, S. Frehse, and G. Fey. Improving fault tolerance utilizing hardware-software-co-synthesis. In *Design, Automation and Test in Europe*, pages 939–943, 2012.

[Rin12] J. Rintanen. Planning as satisfiability: Heuristics. *Artificial Intelligence*, 193(0):45 – 86, 2012.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2004.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985.

[SFFD09] A. Sülflow, S. Frehse, G. Fey, and R. Drechsler. Anwendungsbezogene Analyse der Robustheit von Digitalen Schaltungen. In *Zuverlässigkeit und Entwurf*, pages 45–52, 2009.

[SFWD12] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. Revkit: An open source toolkit for the design of reversible circuits. In *Reversible Computation. Workshop on Reversible Computation*, volume 7165 of *Lecture Notes in Computer Science, LNCS*, pages 64–76, 2012.

[Sht01] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME*, volume 2144 of *LNCS*, pages 58–70, 2001.

[SKF+09] A. Sülflow, U. Kühne, G. Fey, D. Grosse, and R. Drechsler. Wolfram- a word level framework for formal verification. In *Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, RSP '09, pages 11–17, Washington, DC, USA, 2009. IEEE Computer Society.

[SKK+02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389 – 398, 2002.

[SKS+11] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1 –1:29, may-june 2011.

[SLM07] 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, pages 1442–1447, San Jose, CA, USA, 2007. EDA Consortium.

[Soo12] M. Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In Daniel Le Berre, editor, *POS-10*, volume 8 of *EPiC Series*, pages 2–14. EasyChair, 2012.

[SSL+92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[SVAV05] A. Smith, A. Veneris, M.F. Ali, and A. Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(10):1606 – 1621, oct. 2005.

[SVD08] S. Safarpour, A. G. Veneris, and R. Drechsler. Improved SAT-based reachability analysis with observability don't cares. *JSAT*, 5(1-4):1–25, 2008.

[TM96] D. E. Thomas and P. R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 3rd edition, 1996.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[VG09] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *Int'l Conf. on Formal Methods in CAD*, pages 1–8, 2009.

[VLP+05] D. W. Victor, J. M. Ludden, R. D. Peterson, B. S. Nelson, W. K. Sharp, J. K. Hsu, B.-L. Chu, M. L. Behm, R. M. Gott, A D. Romonosky, and S. R. Farago. Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. *IBM J. Res. Dev.*, 49(4/5):541–553, July 2005.

[vN56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 34:43–99, 1956.

[WA73] M. J. Y. Williams and J. B. Angell. Enhancing testability of large-scale integrated circuits via test points and additional logic. *IEEE Trans. on Comp.*, C-22(1):46–60, 1973.

[Weg87] I. Wegener. *The complexity of Boolean functions*. Wiley-Teubner, 1987.

[Wei12] G. Weissenbacher. Interpolant strength revisited. In *SAT*, Lecture Notes in Computer Science. 2012.

[WGF+09] R. Wille, D. Große, S. Frehse, G. W. Dueck, and R. Drechsler. Debugging of toffoli networks. In *Design, Automation and Test in Europe*, pages 1284–1289, 2009.

[WGF+11] R. Wille, D. Große, S. Frehse, G.W. Dueck, and R. Drechsler. Debugging reversible circuits. *Integration, the VLSI Journal*, 44(1):51 – 61, 2011.

[Xil13] Xilinx. *Xilinx TMRTool*. http://www.xilinx.com, 2013.

[XWMB12] J. Xu, M. Williams, Hari Mony, and Jason Baumgartner. Enhanced reachability analysis via automated dynamic netlist-based hint generation. In *Int'l Conf. on Formal Methods in CAD*, pages 157–164, 2012.

[Yeh96] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293 –307 vol.1, feb 1996.

[ZBD07] C. Zhao, X. Bai, and S. Dey. Evaluating transient error effects in digital nanometer circuits. *Reliability, IEEE Transactions on*, 56(3):381 –391, sept. 2007.

[ZFWD11] H. Zhang, S. Frehse, R. Wille, and R. Drechsler. Determining minimal testsets for reversible circuits using Boolean satisfiability. In *AFRICON, 2011*, pages 1 –6, sept. 2011.

[ZKKSV06] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli. SAT sweeping with local observability don't-cares. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 229 –234, 0-0 2006.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, page 10880, 2003.