Application-Dependent Hardware/Software Cross-Layer Fault Analysis

Christian Bartsch

Fachbereich Elektro- und Informationstechnik
Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
Kaiserslautern

Application-Dependent Hardware/Software Cross-Layer Fault Analysis

Vom Fachbereich Elektrotechnik und Informationstechnik
der Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
zur Verleihung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Christian Bartsch
geboren in Moers, Deutschland

D 386

Dekan	:	Prof. Dr. rer. nat. Marco Rahm
Vorsitzender der Prüfungskommission	:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
Gutachter	:	Prof. Dr.-Ing. Wolfgang Kunz
		Prof. Dr.-Ing. Görschwin Fey

Tag der Einreichung	:	20.03.2023
Tag der Verteidigung	:	06.07.2023

DANKSAGUNG

Diese Arbeit ist der „finale Schnitt“ meiner mehrjährigen Forschung. In dieser Zeit gab es zwar auch Tiefen, zum Glück aber deutlich mehr Höhen. Vor allem dank der Kollegen, die für eine herausragend schöne Atmosphäre am EIS-Lehrstuhl sorgen, und den freundlichen Nachbarn vom Echtzeit-Lehrstuhl werde ich diese schöne Zeit in guter Erinnerung behalten.

Besonderen Dank gebühren Professor Wolfgang Kunz und Professor Dominik Stoffel, die mich jederzeit unterstützt und mir bei Herausforderungen zur Seite gestanden haben.

Vielen Dank auch an den Vorsitzenden der Prüfungskommission, Professor Gerhard Fohler, sowie an den Zweitgutachter, Professor Görschwin Fey, ohne deren Zeit und Mühen die Verteidigung nicht hätte stattfinden können.

Ursprünglich sollte der „Final Cut“ bereits im Jahr 2001 mit dem Realschulabschluss stattfinden. Die Reise von diesem Punkt bis hier hin konnte ich nur deswegen schaffen, weil ich währenddessen eine Reihe von lieben Menschen in meinem Leben hatte. Diese möchte ich, wie es sich für einen finalen Schnitt gehört, in einem Abspann ehren; und damit diese Personen die gebührende Beachtung erhalten, kommt dieser Abspann hier an den Anfang.

Sophie und Wilhelm Altes

Birgit und Hans-Jürgen Bartsch

Peter Altes

Sebastian und Alexander Bartsch

Kristin Krüger

Die ganze Familie Altes

Marcel Tegtmeier, Michael Schröter und Stefanie

DANKE

ABSTRACT

Hardware devices fabricated with recent process technology are intrinsically more susceptible to faults than before. Resilience against hardware faults is, therefore, a major concern for safety-critical embedded systems and has been addressed in several standards. These standards demand a systematic and thorough safety evaluation, especially for the highest safety levels. However, any attempt to cover all faults for all theoretically possible scenarios that a system might be used in can easily lead to excessive costs. Instead, an application-dependent approach should be taken: strategies for test and fault resilience must target only those faults that can actually have an effect in the situations in which the hardware is being used.

In order to provide the data for such safety evaluations, we propose *scalable* and *formal* methods to analyse the effects of hardware faults on hardware/software systems across three abstraction levels where we:

1. perform a fault effect analysis at instruction set architecture level by employing fault injection into a hardware-dependent software model called *program netlist*,

2. use the results from the program netlist analysis to perform a deductive analysis to determine "application-redundant" faults at the gate level by exploiting standard combinational test pattern generation,

3. use the results from the program netlist analysis to perform an inductive analysis to identify all faults of a given fault list that can have an effect on selected objects of the high-level software, such as specified safety functions, by employing Abstract Interpretation.

These methods aid in the certification process for the higher safety levels by (a) providing formal guarantees that certain faults can be ignored and (b) pointing to those faults which need to be detected in order to ensure product safety.

We consider transient and permanent faults corrupting data in program-visible hardware registers and model them using the single-event upset and stuck-at fault models, respectively.

Scalability of our approaches results from combining an analysis at the machine and hardware level with separate analyses on gate level and C level source code, as well as, exploiting certain properties that are characteristic for embedded systems software. We demonstrate the effectiveness and scalability of each method on industry-oriented software, including a software system with about 138 k lines of C code.

TABLE OF CONTENTS

1 INTRODUCTION

Over the past decades information technology has advanced at a breathtaking pace. It has experienced significant improvements in every aspect, be it performance, size or power consumption. Digital systems that, just a few decades ago, used to be as large as a side table are nowadays as small as a credit card. Size has always been a limiting factor for the design of commercial products. However, the small size of modern digital systems allows them to be integrated almost everywhere. This enables new and diverse applications for digital systems and has sparked a gradual process where they perform more and more tasks in commercial products.

The task of most digital systems is to provide certain features for the product in which they are integrated in. For some features digital systems replace larger analog systems, like drive-by-wire in cars or fly-by-wire in avionics. For the majority of applications, digital systems are used to add new functionality to a product, for which smartwatches and smart home products are examples. This development has already changed the consumer market as is publicly observable. However, also industry is influenced by this transformation in many ways, affecting product design, manufacturing and the workplace.

This successive digitalisation has a large impact on our daily life and on how our society works. It creates an abundance of opportunities like feature-rich products and new types of jobs. However, it also brings challenges like new malicious attacks, cyber-crime and misbehaving products. These threats need to be addressed not just by society but also by engineers who design current and future generations of digital systems and products.

For example, a malfunctioning or failing digital system controlling the movements of a machinery or vehicle could harm persons close by. Imagine a metal press that does not recognise a hand or limb between ram and bolster or an aeroplane which incorrectly 'thinks' that it is flying too high.

Frequently, digital systems, like anti-lock braking system (ABS) and electronic stability program (ESP) in cars, are successfully used to increase the safety of products for humans. Yet, a failing safety function can make a product less safe than a product lacking such a safety function. This situation can occur when, for example, a failing safety function takes control of the product or when humans have forgotten how to operate the particular product without the failing safety function. Unfortunately, the miniaturisation successes in information technology make digital circuits more susceptible to faults, increasing their likelihood of failure. These safety-critical aspects of digital systems are the focus of this thesis.

Besides safety, another important aspect of digital systems is their security, in particular, their resistance against malicious attacks. Since the publication of attacks like Meltdown [70] and Spectre [63] the, once sidelined, security aspect as a potential cause for malfunctioning digital systems received a lot of attention in recent years. Such attacks can also threaten the product's safety. However, issues related to security are not in the scope of this thesis because their nature is typically quite different to those related to safety and, therefore, require a different set of methods and analyses than those presented in this thesis.

1.1 Embedded System

The *processor* is a commonly known example of a digital system. The task of a processor is to execute a software program and to provide interfaces for the software through which it can communicate with components located outside of the processor. The software execution is performed by *cores*, of which a processor can have one or more.

Processors are integrated in *computing platforms*. Figure 1.1 depicts a typical composition of a computing platform on an abstract level. The processor executes the software program that is located in the memory and uses an input/output-interface (I/O) to interact with the environment. The components of a computing platform are typically connected to a shared communication network, for example a bus, through which they communicate with each other. The combination of a hardware, like a computing platform, and a software program is denoted as *hardware/software system* (HW/SW system).

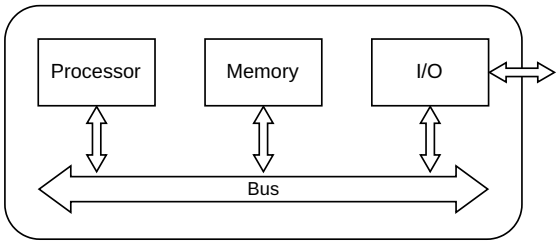


Figure 1.1: Computing Platform

The software that is executed on the computing platform stores intermediate values either in specific memory areas like *stack* or *heap* or in a component called *register file* that is located inside a processor core and has a very limited storage capacity. In contrast to other registers in the processor core, the registers in the register file are accessible by the software program for which they are denoted as *program-visible registers*.

A computing platform, together with connected peripheral devices like sensors or data storages, forms a computer system. Number and types of the peripheral devices depend on the use case of the particular system. The software that is executed by the processor, i.e., the *central processing unit* (CPU), instruments the system's hardware to provide the desired functionalities. The

part of a software program that directly interacts with the system's other components is denoted as *low-level software*, *hardware-dependent software* or *firmware*.

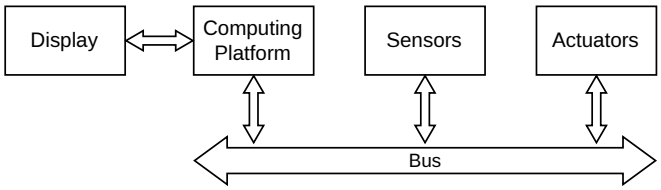


Figure 1.2: Embedded System

A computer system that provides a dedicated functionality to a larger system or product in which it is integrated is denoted as *embedded system*. The computing platform of the exemplary embedded system in Figure 1.2 uses a direct connection to communicate with a display and communicates with sensors and actuators via a bus. Today, with several billion units sold each year the vast majority of processors are used in embedded systems [104].

Some processors are designed for specific tasks. For example, the instruction types and internal structure of *digital signal processors* (DSPs) make them particularly performant in audio or image processing. Processors that do not have a specialised design are denoted as *general-purpose processors*.

The combination of a general-purpose processor with other digital components provides a large degree of flexibility w.r.t. the implementation of functionalities. This design freedom is a major advantage of embedded systems.

1.2 Safety in Embedded Systems

New manufacturing technologies for digital circuits are constantly being developed to satisfy an ever increasing demand for improvements in performance, area and power consumption. A result of this research effort are ever more complex transistor structures and manufacturing processes. Modern integrated transistors, for example, now have a complex three-dimensional structure that cannot be simplified to the planar model that was used some decades ago. Some steps in the evolution of field effect transistors (FET), a widely used transistor for digital circuits, are depicted in Figure 1.3, with increasing complexity from left to right.

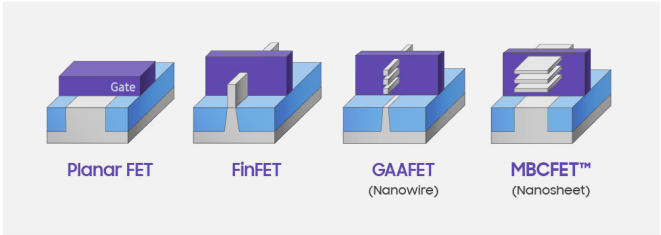


Figure 1.3: Transistor Designs [30]

The larger complexity and smaller distance between their structural elements have made transistors and their interconnections more susceptible to faults and lead to an increased probability of errors, malfunctions and failures.

The consequences of a malfunctioning embedded system depend mostly on its application. For example, a malfunctioning embedded system which provides infotainment in a car may be annoying but it would not directly affect the safety of the passengers. However, embedded systems are frequently employed in safety-critical tasks. For example, in application domains like avionics or autonomous driving they control the speed and direction of travel and identify obstacles to prevent collisions. In other application domains like production automation they verify that nothing, especially no body part, is close or between a moving mechanical system.

For such safety-critical tasks the requirement of *functional safety* has become a key concern in industry and, not rarely, defines the "economic operating point" of new products. This stimulated the development of several international standards like IEC 61508 to define strict measures and requirements for every phase of the development cycle to ensure functional safety in products.

Examples of industry specific safety standards are:

- ISO 26262 for automotive
- IEC 61511 for industrial processes
- IEC 61513 for nuclear power plants
- IEC 62061 & ISO 13849 for machinery
- EN 5012x for railways
- DO-178 & DO-254 for aviation
- IEC 62304 for medical devices

Yet, it is often not clear how the objectives formulated in these standards can actually be achieved in practice by the chosen architectures and design methods, and how this can be documented. For instance, to increase the safety of an embedded system, a large spectrum of design measures with varying costs is available both at the hardware and at the software level.

1.3 Motivation

Size and complexity of modern digital systems allow faults to manifest themselves in numerous ways at various locations. Implementing protection against all possible types of faults is costly and sometimes infeasible. We need to focus on those faults which can actually violate the functional safety of the system.

Safety standards demand the provision of guarantees that faults that are ignored can never compromise the functional safety of a system. In order to provide such guarantees the complete behaviour of the fault affected digital system has to be considered. The most prominent approaches for such safety

analyses are exhaustive simulation and formal verification. Of those, the former tends to require an infeasible amount of time to analyse the effects for every possible input sequence, while for the latter the computational complexity of the analysis quickly becomes impractical.

The need for a formal method that can provide the aforementioned guarantees and that is still scalable motivates this thesis. We achieve this goal by exploiting the following characteristics of embedded systems.

Embedded systems typically provide clearly defined, limited tasks which implies that, for the most part, the software does not undergo major changes during a system's lifetime. This is true especially for low-level software that constitutes an important component of a chip's general infrastructure, for example, by controlling the communication between application software and hardware, by implementing important functions for chip management and, more and more often, by replacing traditionally hardware-implemented control functions of the system. Such software components are usually part of the system's firmware. They are tightly coupled with the hardware and, in many cases, play a particularly important role for the overall system safety. This application-specific nature of embedded systems, typically, leads to software which utilizes only a fraction of the processors capabilities.

The small changes to embedded systems software allows the implementation of cost-efficient safety measures tailored to a concrete software program. The limitation to a particular system software restricts the hardware utilization and, as shown in our experiments, limits the number of locations from where errors can propagate through the system to make it malfunction.

In this thesis we focus on faults that manifest themselves in processor cores and do not explicitly consider faults originating from other components of the embedded system. The main objective of this thesis is to show how formal methods can be used to identify all faults that can never affect the safety of a HW/SW system of realistic size. This thesis contributes a method to generate a computational model on which the effects, and the absence of any effect, of hardware faults on the software behaviour can be formally proven. Further, it shows how fault effects can be analysed across multiple abstraction layers in a formally sound way. Note that by showing that only certain faults can have a specific effect, e.g., affecting a safety function, formal methods also prove that every other fault can never cause the considered effect and can, therefore, be ignored in the corresponding safety evaluation.

1.4 Overview and Scientific Contributions

As motivated in the previous section, the main objective of this thesis is the development of methods to formally analyse the effects of hardware faults on the behaviour of a HW/SW system. In order to obtain realistic results, we employed fault models that are commonly used in the field of testing and analysed the fault effects on industry-oriented software programs. This research effort yields a toolbox of methods to formally analyse the propagation of fault effects across multiple abstraction levels, namely the gate level, the instruction

set architecture (ISA) level and the higher software level (C level). We achieved a degree of automation where almost all steps of the methods presented in this thesis are fully automated. We point out the few steps where manual effort is required in the corresponding sections.

As the main scientific result of this thesis we present a cross-layer technique providing formal guarantees on fault effects propagating from the gate level to the C level. We demonstrate the new approach for systems of realistic size that were so far deemed intractable for any formal analysis of this kind. The main elements of this contribution are the following:

1. We develop a set of methods which allow us to improve the scalability of generating our basic computational model, called *program netlist (PN)*. This contribution makes it possible to use PNs in our fault effect analyses even for large firmware and driver systems. We discuss these methods in Section 4.3.

2. In order to increase the scalability of the fault analysis methods, we develop the dependency analysis presented in Section 4.4. This method allows a cone-of-influence reduction on a PN and decreases the complexity of any PN-based formal analysis.

3. The ISA-level fault models used in this thesis are designed in such a way that they have a sound representation on the gate level. We achieve this by formulating stuck-at and single-event upset (SEU) fault models for program-visible registers. We provide a detailed discussion on this, together with several fault description styles, in Section 5.

4. The formal method presented in Chapter 6 (*FEA*), can analyse the effects of hardware faults on the program behaviour at the ISA level by injecting faults in program netlists. The result of FEA are two sets of faults. In the first set are faults which, for certain input sequences, have an effect on the considered program behaviour. The faults in the second set never have an effect on the considered program behaviour.

5. In Chapter 7 we show how the knowledge about the absence of fault effects obtained from FEA can be used to deductively identify faults on the gate level which, due to the constraints applied by the executed software program, also never have any effect on the behaviour of the HW/SW system. We denote these faults as *application redundant faults*. This method (*FTEA*) employs automated test pattern generation for this purpose and identifies gate-level faults that can never have an effect on the considered program behaviour.

6. In order to analyse fault effects also on higher software abstraction levels, we developed the formal method *FPA*. FPA takes the ISA-level fault effects found by FEA, inductively transfers them to faults at the higher software level (C level) and applies Abstract Interpretation to identify all other software objects, like safety critical functions, that can be affected

by a particular fault. The result of FPA is a 1-to-n relation between hardware faults on the ISA level and their effects on the C level. We present this method in Chapter 8.

PUBLICATIONS

We already published several aspects of this thesis as shown by the following bibliography. Key elements of this thesis are presented in [14, 17, 18].

[12] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *International Workshop on Resiliency in Embedded Electronic Systems*. 2015.

[13] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*. Ed. by R. Wimmer. Albert-Ludwigs-Universität Freiburg, 2016, pp. 10–21. DOI: 10.6094/UNIFR/10634.

[14] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *2016 17th Latin-American Test Symposium (LATS)*. 2016, pp. 153–158. DOI: 10.1109/LATW.2016.7483356.

[15] C. Bartsch, C. Villarraga, et al. "Efficient SAT/Simulation-based model generation for low-level embedded software". In: *17th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2014, Böblingen, Germany*. Ed. by J. Ruf, D. Allmendinger, et al. Cuvillier, 2014, pp. 147–157.

[16] C. Bartsch, C. Villarraga, et al. "Safety across the HW/SW interface - Can formal methods meet the challenge?" In: *International Symposium on Integrated Circuits, ISIC 2016, Singapore, December 12-14, 2016*. IEEE, 2016, pp. 1–3. DOI: 10.1109/ISICIR.2016.7829707.

[17] C. Bartsch, C. Villarraga, et al. "A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems". In: *Journal of Electronic Testing* 33.1 (2017), pp. 77–92. DOI: 10.1007/s10836-017-5643-3.

[18] C. Bartsch, S. Wilhelm, et al. "Compositional Fault Propagation Analysis in Embedded Systems using Abstract Interpretation". In: *2021 IEEE International Test Conference (ITC)*. 2021.

[19] C. Bartsch, S. Wilhelm, et al. "Combining Fault Effect Analysis and Fault Propagation Analysis to Determine Source Code Level Effects of Hardware Faults". In: *Embedded World*. WEKA FACHMEDIEN GmbH, 2022.

[95] S. G. Sørensen, C. Bartsch, et al. "Generation of Formal CPU Profiles for Embedded Systems". In: *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, Oct. 2022.

[107] C. Villarraga, B. Schmidt, et al. "An Equivalence Checker for Hardware-Dependent Software". In: *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2013, pp. 119–128.

[108] C. Villarraga, B. Schmidt, et al. "Software in a Hardware View: New Models for HW-dependent Software in SoC Verification and Test (Invited Paper)". In: *Proc. International Test Conference (ITC'14)*. 2014.

2 BACKGROUND

In this chapter we review the various state-of-the-art concepts and methods from different fields which are used in this thesis and which are relevant for a deeper understanding of our contributions.

2.1 Abstraction Levels

Abstraction is a method where specific information details are intentionally left out of consideration. The application of abstraction creates a more general concept, i.e., an abstract model of a concrete object. For example, the abstract term *transistor* is used when the information about the transistor type, e.g., bipolar or field effect transistor, is irrelevant for a particular discussion.

The goal of abstraction is to remove complexity by considering only those aspects which are relevant for a specific purpose. For example, when only the result value of an addition needs to be analysed time or individual steps the processor needs to perform the computation are irrelevant and, therefore, can be ignored.

When an abstract concept is further abstracted every abstraction step creates a new, higher, *abstraction level*. In the following sections we provide a survey on those abstraction levels for hardware and software that are used in this thesis.

2.2 Abstraction Levels for Software

There exist several abstraction levels at which the behaviour of software programs can be described. In this thesis, only the lowest three of the software abstraction levels are of relevance, the *machine level* 2.2.1, the *assembly level* 2.2.2 and the *C level* 2.2.3.

2.2.1 Machine Level

Machine language is the actual language read in and executed by the processor. Its alphabet consists of only two letters, the values of a binary digit (*bit*) zero and one. A software program written at machine level, i.e., in machine language, is called *machine code* or *machine program*. The machine level is, therefore, the lowest abstraction level for a software program.

In general, machine code is a sequence of bit values situated in the memory of a computing platform (cf. Figure 1.1). For some processor architectures the machine code is stored together with the data in the same memory while for others the machine code is stored separate from the software-accessible data.

In such architectures, the memory with the machine code, typically, cannot be written by the software and is, therefore, denoted as *read-only memory* (ROM). Memory whose contents can be read and modified by a software program in any order is denoted as *random access memory* (RAM). In most systems writing and overwriting machine code is strictly forbidden during runtime, because it is extremely error-prone. The smallest addressable unit of a memory is typically a *byte* which is a group of 8 adjacent bits. The locations of individual bytes are identified by their respective memory addresses.

When a processor executes a software program it reads an instruction, i.e., a group of consecutive bytes denoted as *instruction word*, from memory, decodes it and executes the instruction as specified by the architecture the processor implements. This specification is part of the instruction set architecture discussed in Section 2.3.3.

A segment of the instruction word, denoted as *opcode*, specifies the operation the processor is to perform. Other bits define the operands to be used by the operation. Possible operand types of an instruction are numbers, memory addresses or registers in the register file. Opcode and operands are written in binary format and may even be distributed over the instruction word. For example the bits for operand "imm" in Figure 2.1 are split into two blocks of adjacent bits located at two different positions in the instruction word.

31					0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
00000000	01110	00010	010	01000	0100011

Figure 2.1: Example machine code instruction

Consider the example instruction for a RISC-V processor in Figure 2.1. The numbers at the top of the table are indices for the left-most and the right-most bit, providing the necessary information how the bits of the instruction word have to be interpreted. The highest index is used for the *most significant bit* (MSB) and the lowest index for the *least significant bit* (LSB). The first row shows the fields of a RISC-V instruction format that is used to specify different variants of store instructions, according to the RISC-V instruction set manual [109]. The instruction consists of the following instruction fields:

- An opcode field to identify the instruction as store instruction.
- A func3 field telling the processor how many bytes have to be written to memory.
- The fields rs1 and rs2, of which each contains a number addressing a program-visible register used in the store operation.
- Two fields for the imm operand used to compute the memory location where the values have to be stored.

The last row of Figure 2.1 shows an example instruction word split into the corresponding instruction fields. This particular instruction performs a store

operation (opcode) and stores 4 bytes (func3) of register number 14 (rs2) to the memory starting with the address that is computed by adding a constant value (imm) to the value in register 2 (rs1).

2.2.2 Assembly Level

The machine code is what is actually executed by the processor, but its bits and bytes are very difficult for humans to read and understand. Machine programs are, therefore, expressed in an abstract representation called *assembly language*. Programs in such a language can be translated to machine code using an *assembler*. In assembly language, instructions are represented by a textual description that hides the details of bit values for the individual fields in the instruction word.

Typically, an instruction specified at assembly level starts with a *mnemonic* describing the operation to be performed followed by a comma-separated list of operands.

At assembly level, numbers are usually written in decimal or hexadecimal format, and rarely in binary format. Program-visible registers are addressable via aliases and not just by their number. In some processors the alias provides information about the register's intended usage, e.g., for RISC-V processors `t0` refers to a register for temporary values and intermediate results.

```
1 target:
2 li t0, 0
3 ...
4 j target
```

(a) Symbol

```
1 mv   t0, t1
2 addi t0, t1, 0
```

(b) Pseudo-Instructions

Figure 2.2: Example features of the assembly level

Memory addresses can also be written in hexadecimal format, however, such explicit use of memory addresses is rare. Instead, a descriptive symbol denoted as *label* is used that implicitly points to a specific memory address. The RISC-V assembly code in Figure 2.2a shows in line 4 a jump instruction with the label "target" as operand. This label is defined in line 1 and will be translated to the memory address of the load immediate instruction in line 2 during the translation from assembly code to machine code.

Another assembly-level feature that is not supported at the machine level are pseudo-instructions. Their purpose is to improve development and readability of assembly code by providing a simplified version of (a set of) semantically equivalent assembly instructions. For example, both RISC-V assembly instructions in Figure 2.2b copy the contents of register `t1` to register `t0` and both are translated to the same machine instruction. However, the first instruction, move, requires less assembly level operands and is more expressive.

Except for some special cases, an assembly instruction can be directly translated to a machine instruction. A translation in the other direction, i.e., from

machine instruction to assembly instruction, is also possible. In theory, this task can be difficult to accomplish as shown by [20]. In practice, however, and in particular for the software considered in this thesis, disassembly is usually possible using standard techniques and tools.

For low-level programs used in embedded systems the translation from assembly level to machine level is straightforward, so that analysis results obtained at assembly level can be mapped to the machine level in a sound way.

```
1 is_even_number: xori a0, a0, -1
2                 lui  t0, 1
3                 and  a0, a0, t0
4                 ret
```

Figure 2.3: Example assembly code

The RISC-V assembly code function in Figure 2.3 checks whether the provided value in register `a0` is an even or odd number and returns the answer to the calling function by writing the corresponding binary value of *true* or *false* to register `a0`. For the computation the number's bit values are inverted by using the xor-operation in line 1 and, then, all bits, except the LSB, are set to zero in line 2. The value of the LSB in `a0` tells whether the given number was even or not.

Even without an explicit comparison it should be clear that the assembly code in Figure 2.3 is easier to understand than its equivalent machine code. However, as we will point out in the following section, the next higher abstraction level allows to write the same function in a much conciser way.

2.2.3 C Level

As program complexity increased, the need for higher abstraction levels grew, resulting in the development of new programming languages. Nowadays, there exist many programming languages which support various styles of programming allowing to freely choose the degree of abstraction. Characteristic for these programming languages is their use of abstracter concepts. For example, they utilize statements and variables instead of instructions and memory locations, respectively. The two major advantages are the use of statements and variables being more intuitive for humans and the size of programs written at higher abstraction levels being an order of magnitude smaller because a single statement can represent multiple machine instructions.

A popular example is the C programming language, which is still frequently used for programming embedded systems, especially for device drivers. These low-level programs provide an abstract interface to peripheral devices for application software written on a higher abstraction level. In the rest of this thesis we will use the term *C level* to refer to software abstraction levels above assembly.

The translation process from C level to assembly level is denoted as *compilation* and is performed by a software program called *compiler*.

A major disadvantage of this abstraction level is that knowledge obtained from C level cannot be transferred to lower abstraction levels in a sound way. The reason for this is the large degree of freedom in the ways C statements can be translated to assembly instructions. The assembly code produced by one compiler can be completely different to the assembly code of another compiler although they have translated the same C-code. This aspect becomes worse in the presence of optimisations, as they are regularly performed by compilers. For example, the value of a variable can be placed in memory, in the register file or can be optimised away.

```
1 bool is_even_number(int number)
2 {
3     return (number % 2) == 0;
4 }
```

Figure 2.4: Example C code

The code in Figure 2.4 is functionally equivalent to the assembly code in Figure 2.3. It is shorter and uses the modulo operation which may involve the more complex integer division in the compiled program, whereas the optimised assembly code employs simple logic operations.

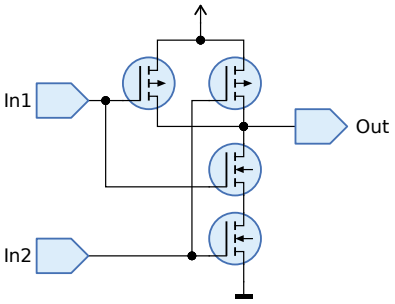
2.3 Abstraction Levels for Hardware

Similar to the software domain, the behaviour of hardware can be described at several abstraction levels, of which the *gate level* 2.3.1, the *register transfer level* 2.3.2 and the *instruction set architecture level* 2.3.3 are of relevance in this thesis. In this context, we focus on hardware located inside a processor, particularly the processor core. Digital circuits can be classified as combinational or sequential. A digital circuit that only uses its current input values to compute output values is called *combinational circuit*, while a circuit that uses previously computed values, alone or in combination with the current input values, to compute output values is called *sequential circuit*. The previously computed values constitute the *state* of a sequential circuit. A sequential circuit implements a *transition function* that defines how the next values, i.e., the next state, is computed depending on the previous state and the current input. The state of a sequential circuit is usually stored in memory elements like flipflops and registers. A combinational circuit does not have such storage elements.

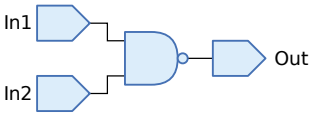
2.3.1 Gate Level

The gate level is the first abstraction level above the transistor level. At the gate level transistors and electrical information such as voltage, resistance and capacitance, of a digital circuit are hidden. Only the logic functions implemented by the transistor circuitry are represented. Every gate models a Boolean function abstracting the corresponding transistor circuit. Instead of two discrete

voltage levels used at the transistor level, the inputs and outputs of gates use Boolean values which can also be interpreted as bit values.



(a) Transistor level



(b) Gate level

In1	In2	Out
0	0	1
0	1	1
1	0	1
1	1	0

(c) Truth table

Figure 2.5: Gate level abstraction for NAND

Figure 2.5 illustrates the abstraction from transistor level (Figure 2.5a) to gate level (Figure 2.5b) at the example of the negated AND-operation. Both circuits implement the same logic function, i.e., for all input scenarios they produce the same output behaviour, as depicted in Table 2.5c.

The transistor circuit in Figure 2.5a uses the complementary metal-oxide-semiconductor (CMOS) technology commonly used to manufacture digital circuits by using two types of metal-oxide-semiconductor field-effect transistors (MOSFET). One MOSFET type the PMOS transistor, sets the output voltage to *high*, i.e., a logical one, and the other transistor type, the NMOS transistor, sets it to *low*, i.e., a logical zero.

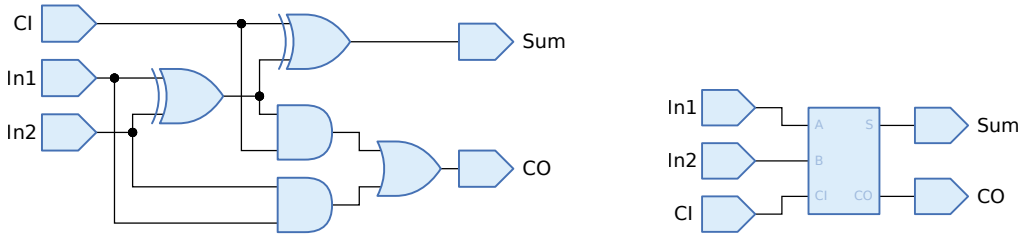
When compared with the abstraction level of assembly from the software domain, the gate level provides similar advantages and disadvantages. It reduces the overall complexity of a digital circuit and is easier to understand for humans while allowing the knowledge transfer of analyses to and from the transistor level in a sound way. However, performing a comprehensive analysis for circuits of realistic size on gate level can be too complex to pursue. This is particularly true when the full behaviour, i.e., the complete semantics, of the circuit has to be considered, as is required for fault effect analyses.

2.3.2 Register Transfer Level

The next higher abstraction level is the *register transfer level* (RT level or RTL). Abstraction from gate level to RTL is achieved by hiding the computation of individual Boolean values of the gate level behind modules and statements.

This abstraction is illustrated in Figure 2.6, where the gate-level full adder in Figure 2.6a and its RT level representation in Figure 2.6b are functionally equivalent. Both circuits take three Boolean values, In1, In2 and carry-in (CI), as inputs and compute sum and carry-out (CO).

An advantage of the RT level is that it supports the aggregation of multiple Boolean/bit values into bitvectors. Figure 2.7 shows a textual description of a similar circuit by using the hardware description language (HDL) Verilog. It defines an addition module that takes two 32bit signals interprets them as signed numbers and assigns the results of the addition to the output signal.



(a) Gate level (b) RT level

Figure 2.6: RT level abstraction of addition

Popular HDLs for the design of digital circuits at RT level are Verilog and VHDL[1].

```
1  module signed_addition(
2      input  logic [31:0] in1,
3      input  logic [31:0] in2,
4      output logic [31:0] sum);
5
6      always_comb
7      begin
8          sum = signed(in1) + signed(in2);
9      end
10 endmodule
```

Figure 2.7: 32-bit signed addition at RT level

As can be seen in Figure 2.7, the concepts used for designing hardware at RT level are similar to those used for writing software at C level. Likewise, a disadvantage of this abstraction level is that most knowledge obtained from analyses at RT level cannot be soundly mapped to lower abstraction levels, because statements at RT level can be implemented in many ways at the gate level.

2.3.3 ISA Level

The *instruction set architecture* (ISA) specifies the behaviour of a processor during software execution and defines what machine code a processor supports including the instruction types and their formats. The ISA provides, therefore, the interface between hardware and software. By using the analogy from Section 2.2.1: the machine language provides the letters and the ISA defines words and grammar of the machine language that a processor understands.

ISA specifications are abstract and do not contain implementation details like exact instruction timing in processor clock cycles. They ensure that software written for a specific ISA can be executed on every processor that implements this ISA.

1 **V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage

A processor description on ISA level contains mainly a component to decode instructions, a file of program-visible registers and an implementation-independent description of the processor's behaviour for each instruction. The state of a processor core at the ISA level is denoted as *architectural state* (AS) and includes the state of the program-visible registers, the program counter and any program-visible status information.

The advantage of the ISA level is its high degree of hardware abstraction while retaining the ability to be combined with software on machine level. *Instruction set simulators* use this to provide high-performance execution of software programs compiled for an ISA that may be different from the ISA of the processor on which the simulator runs.

Since the ISA level is implementation-independent, knowledge obtained from analyses at ISA level cannot be translated to the gate level in a sound way in most cases. An exception is knowledge regarding program-visible registers as they have a representation on the gate level.

The relevant ISAs for this thesis are SuperH2 [86], originally developed by Hitachi, and RISC-V [109] developed by University of California, Berkeley.

In this thesis we exploit that the ISA level provides a direct connection to gate level via the register file and an interface to the machine level.

2.4 Verification of Hardware & Software

An important part during the design process of hardware and software is to ensure that the design meets its specification, e.g., it is free of bugs and resilient against faults. The basic idea is to formalise the design specification by deriving mathematical models, denoted as *properties* or *assertions*. A design, then, complies with its specification only if every property holds for all possible values at the *primary inputs*. When the property set is complete, i.e., it covers every aspect of the specification, then compliance with the specification is also a necessary condition for the properties to hold.

Primary inputs of a design or a design component are those inputs which are not controlled by the design or component itself but by the environment, i.e., by connected external systems or devices which provide values to these inputs. Examples of primary inputs in software are the register *ao* in Figure 2.3 and the variable *number* in Figure 2.4. Examples of primary inputs in hardware are *In1* and *In2* in Figure 2.5 and *CI*, *In1* and *In2* in Figure 2.6. Analogous to primary inputs are the *primary outputs* of a design which are used by the design to send values to its environment.

Verification is a complex and time-intensive task where its procedural details, like the selection of analysis methods and the number and order of the analysis steps, depend entirely on the individual design and its application case. That is why there exists a wide range of methods and tools, developed for different system types and different aspects of hardware and software that have to be verified. These methods employ one or a combination of the following concepts.

2.4.1 Simulation and Emulation

Simulating a hardware or software design means that its behaviour, which it would have in the real world, is imitated by a simulation tool that executes a computational model of the design. The simulation tool executes the computational model, e.g., RT code of hardware or assembly code of software, by interpreting its statements. The basic idea of the interpretation process is to simulate the behaviour of a single statement and then determine the successor statement whose behaviour will be simulated next.

Simulation has the advantages that it is very fast and that it scales linearly with the number of statements, allowing fast simulation runs of large hardware and software designs. A simulation run can start from reset or a user defined start statement in the design and either ends when no successor statement can be found or when a user defined end statement was simulated.

The disadvantage of simulation is that every simulation run requires a concrete value for every primary input of the design. For sequential designs a concrete value must be provided for every primary input and every time point, e.g., a value per clock cycle for hardware or a value per function call for software. This dependency on concrete values causes the simulation runtime to scale exponentially with the number of primary inputs when a simulation run for every combination of input values has to be performed. Such an *exhaustive simulation* is, therefore, rarely feasible for designs of medium and larger size.

Emulation is based on the same principle as simulation. Its major difference is that the imitation process is executed on a separate piece of hardware for which the design and the emulation environment is highly optimised. The runtime per statement during emulation is lower than during simulation providing a significant speed advantage for a complete emulation run over an equivalent simulation run. Nonetheless, emulation shares the problem of exponential scaling w.r.t. the number of primary inputs with simulation. This issue drastically overcompensates the linear speed advantage of emulation such that *exhaustive emulation* does not provide a feasible alternative to an infeasible exhaustive simulation.

Simulation- and emulation-based verification methods check the assertions during each simulation run and notify the verification engineer when an assertion fails. However, analyses based on non-exhaustive simulation can never cover the complete design behaviour and can, therefore, never guarantee the absence of behaviour violating any specification or safety goal. In order to attenuate this disadvantage modern verification methods perform a number of simulation runs until the probability of specification violations is determined to be sufficiently low. For safety-critical applications, like avionics or nuclear power plants, this presents a serious drawback, because the computation of this probability depends on, possibly flawed, assumptions on the real-world. Examples of such assumptions are values deemed to be impossible or less likely for certain primary inputs, or that the rate at which faults appear is below a certain limit.

2.4.2 Symbolic Techniques

Hardware circuits or software programs can also be verified by using symbolic techniques. In the hardware domain the general term for such techniques is *symbolic simulation*, while *symbolic execution* is used in the software domain [27, 28].

Symbolic simulation-based methods and symbolic execution-based methods share a common methodology from which they inherit its advantages and disadvantages. In the following, we discuss these similarities.

Symbolic techniques use symbolic values which mathematically represent *sets* of data values in the design's state variables and at its primary inputs [53, 91]. This solves the limitation of classic simulation and emulation where only single, individual concrete values can be considered in each simulation/execution run. By using symbolic values, at least in principle, the complete design behaviour for all input values can be considered in a single symbolic simulation/execution run. A holding assertion, therefore, is valid for the complete design behaviour.

A disadvantage is that the complexity of the mathematical models increases with every interpreted statement. For data processing designs, which typically have a large portion of non-branching statements, the complexity growth with each statement may be manageable even for designs of medium size.

However, the complexity can increase significantly with each control statement [25] like branches at assembly level or if-statements at C level. The reason is that control statements can have more than one successor which increases the number of execution paths. In the worst case, the number of execution paths grows exponentially with the number of control decisions. This is known as the *path explosion* problem [98]. Applying symbolic simulation or symbolic execution to larger control-centric designs, e.g., designs with a hardware- or software-implemented finite state machine, is, therefore, rarely feasible.

Symbolic techniques employ several optimisation strategies to keep the computational model compact [6, 81]. By example of symbolic execution, one optimisation strategy is to include only reachable program paths in the model. Such paths are identified in an analysis applied on an intermediate mathematical model of the program. Another optimisation strategy is to merge identical program paths [6]. In order to facilitate the execution of both optimisations, symbolic techniques use *control flow graphs* which model the control behaviour of a hardware or software design. We discuss control flow graphs in detail in Section 2.7.

2.4.3 Formal Verification

Formal verification methods translate the full design into a mathematical model and combine this with the mathematical model of a property. By applying mathematical and logical proofs formal methods try to prove the correctness of the combined model. If this succeeds, it formally shows that the design complies with the considered property for its complete behaviour. If this fails, a counterexample can be generated showing a behaviour where the consid-

ered property fails, i.e., where the design violates the considered aspect of its specification. The advantage of formal verification is that it considers all possible values at the primary inputs and can analyse data- and control-centric designs. A disadvantage is that it can suffer from an exponential complexity in terms of runtime or memory usage of the formal proof engines, making the analyses of medium to large designs infeasible. In practice, timeouts and limited computing resources cause formal verification methods to eventually terminate inconclusively. A well-known example for the complexity problem is the formal verification of hardware-implemented multiplication by using SAT solvers [111]. A SAT solver is a computer program solving the Boolean satisfiability problem which belongs to the class of NP-hard[2] problems.

The methods presented in this thesis employ only formal techniques and formal verification tools guaranteeing that the complete behaviour of an analysed hardware/software system is considered.

Bounded Model Checking

The formal verification of a sequential hardware design requires to analyse the design's behaviour over several time points, i.e., clock cycles, starting from system reset. In *bounded model checking* (BMC) [24] this is done by unrolling the hardware, e.g., duplicating the hardware, and connecting one hardware instance with another in such a way that every instance models the hardware behaviour for a specific clock cycle. The number of hardware instances grows linear with the number of clock cycles to be analysed. Unfortunately, the analysis complexity grows exponentially with the number of hardware instances. In practice this is observed by a sharp increase in the computational resource consumption. This limits the application of BMC in particular and formal verification in general to verification runs that cover only a small number of clock cycles after reset. For example, only a dozen clock cycles of unrolling is feasible for a processor of medium size, like in-order RISC-V processors with 5 pipeline stages.

Interval Property Checking

A solution to analyse the hardware behaviour further away from reset than just a few clock cycles is *interval property checking* (IPC) [78]. The fundamental model of IPC, a finite unrolling of the circuit, is the same as in BMC, however IPC does not need to start from reset. In general, IPC starts from *any-state* where registers are not constrained and are allowed to have an arbitrary value. However, starting from any-state can lead to large over-approximations where the analysis includes behaviour that is unreachable in practice. This can easily lead to the situation that an unreachable state causes a property to fail, denoted as *false negative* or *false counterexample*. In practice, false counterexamples are reduced and excluded by adding *invariants* to the properties. Invariants are formulated as logic constraints that are universally valid for the design under

2 non-deterministic polynomial time hardness

verification and that restrict the set of starting states by excluding unreachable behaviour.

An application case for IPC is to decompose the specification of a sequential circuit into properties of manageable time length, each representing a specific behaviour. We re-use the basic idea of this concept in Section 8.1 where we analyse the behaviour of a large software system.

Static Software Analysis

In the domain of software verification static and dynamic software analyses are widely used techniques. Static software analysis-based approaches consider the behaviour of single statements or complete software programs without executing them [88]. In contrast, dynamic software analysis-based approaches require the software program under consideration to be executed, e.g., via simulation or emulation, making this approach non-formal. Static software analysis is employed in a variety of verification tasks that includes but is not limited to the following:

- A simple expression checking algorithm verifying that software code complies with a specific set of design rules [10].
- A sophisticated formal analysis that mathematically proves the functional correctness or behavioural properties of a software program [34, 106].

The formal methods in the software domain presented in this thesis belong to the category of static software analysis.

Abstract Interpretation

The semantics of a programming language is a formal description of the behaviour of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program on all possible inputs. Yet in general, the concrete semantics is not computable. Even under the assumption that the program terminates (cf. halting problem [105]), it is too detailed to allow for efficient computations. A solution is to introduce a formal abstract semantics that approximates the concrete semantics of the program in a well-defined way and still is efficiently computable. This abstract semantics can be chosen as the basis for a static software analysis. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster.

Abstract Interpretation is a formal method for sound semantics-based static software analysis [32]. It supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound in the sense that it computes an over-approximation of the concrete program semantics. Abstract Interpretation always provides full data and control coverage.

As of today, Abstract Interpretation-based static analysers have evolved to become standard methods for determining non-functional software quality

properties [54, 56]. On the one hand, this includes source code properties, such as compliance with coding guidelines, compliance with software architectural requirements, as well as absence of runtime errors and data races [58]. On the other hand, also low-level code properties are covered such as absence of stack overflows and violation of timing constraints [57, 59].

2.5 Verification of Hardware/Software Systems

A HW/SW system is a digital system where one part of the system is implemented in hardware while the other part is implemented in software. Only the combination of both parts can provide the required functionalities. Embedded systems, as defined in Section 1.1, are widely deployed types of HW/SW systems.

The challenge to fully analyse and verify such systems is considerable due to the large complexity emerging from a combination of hardware and software. The traditional approach to analyse such systems is to separate the hardware part from the software part and to analyse each part separately. The advantage of this approach is that the complexity of each analysis is significantly smaller than the analysis of the complete HW/SW system. However, this comes at the cost that the interaction between hardware and software is not considered. This may allow for certain development faults and error propagation paths to remain hidden, endangering the correct operation of the HW/SW system.

The methods proposed in this thesis show how the effects of hardware faults on the behaviour of embedded systems can be formally analysed across HW/SW boundaries.

2.6 Faults

In every stage of the design and manufacturing process as well as during its operational lifetime, a fault can lead to an error in a HW/SW system and cause it to malfunction. Be it a flawed specification, bugs introduced in the design phase, manufacturing variations or physical wear-out, faults can originate from several source types and manifest themselves in many ways.

In this section we provide an overview on terminology, categories and standardisation measures to increase the fault resilience of HW/SW systems.

2.6.1 Terminology

Table 2.1 presents definitions of terms [8] used in this thesis some of which may be used interchangeably in daily life due to their ambiguous meaning.

Only an *active* fault produces an error. A fault that is present but does not produce an error is *dormant* [8], for example, a fault that sets a value that is already zero to zero. In order to activate a fault an input sequence denoted as *activation pattern* or *activation condition* is required. Some activation conditions are trivial, because they are always fulfilled, while others require very complex combinations of internal state and input sequences.

Failure	A failure is a deviation of the output behaviour from the intended behaviour of a component or system.
Error	An error is the incorrect output value of a failing component or system, i.e., an incorrect state or an incorrect signal value.
Fault	A fault is the "adjudged or hypothesized cause of an error".
Fault Model	A fault model describes the effects of a fault on a component or system at a certain abstraction level.

Table 2.1: Terminology

Faults can be categorised into several fault classes, each representing a specific aspect of a fault. In Table 2.2 we review the fault classes that are relevant for this thesis. In general, any combination of fault classes in Table 2.2 are possible. In this thesis we focus on random faults that appear during the operational lifetime of a system, i.e., faults that are both operational and natural w.r.t. the fault classes. The examples provided in Table 2.2 reflect this focus. Development faults like bugs are, in general, out of scope but when they degrade the safety this degradation can be, for some cases, highlighted by our methods. In Sections 5.1 and 5.2 we present fault models for internal permanent and external transient faults, respectively.

Traditional analysis methods have a strong focus on *single faults*, i.e., they assume that during a considered time interval during system operation only a single fault occurs. However, due to shrinking transistor sizes and complexer transistors, the probability of faults has increased significantly, raising the need to consider *multiple faults*. We address this development by explaining how we inject multiple faults in our computational model and providing experimental data of fault analyses for multiple faults.

When the effects of a hardware fault manifest themselves as errors, in the digital domain they appear at first at the lowest digital abstraction level, i.e., the gate level. Like correct values, erroneous values are passed on to every successor in the fanout of the affected component allowing them to affect several other values. This is how errors can propagate through a digital circuit. During propagation, errors may become visible at higher abstraction levels and they may appear in the software after "crossing" the HW/SW boundary. Error propagation continues until all erroneous values are corrected or until they cannot propagate further, i.e., cannot affect more components.

Like faults, also errors can be categorised as shown in Table 2.3. In this thesis we consider both soft and hard errors. Some faults may not affect the behaviour of a component or system because the propagation of its errors to any primary output depends on a complex combination of internal state and input values. While in theory such complex combinations might be possible, in practice the application, e.g., the software running on a processor, could prevent the propagation of these errors to any primary output. In situations where a local analysis cannot provide formal guarantees that an error cannot have more effects on the system than found in the local analysis we also denote such errors as *latent errors*.

Phase of occurrence	
Development fault	Faults introduced into the system during development processes, for example by design engineer mistakes, bugs in tools or material impurities during production.
Operational fault	Faults that occur after shipment when the system is in use.
Phenomenological cause	
Human-made fault	Faults that result from human actions, like design mistakes or due to a wrongly operated system.
Natural fault	Faults that are "caused by natural phenomena without human participation" [8].
System boundaries	
Internal fault	Faults originating from mechanisms inside the system, like electromigration, ageing or thermal processes due to operational activity.
External fault	Faults originating from outside the system, like radiation, unstable power supply or thermal processes due to environmental temperature.
Persistence	
Permanent fault	Faults that are always present, for example a short circuit of wire with the supply voltage.
Transient fault	Faults that have only a temporary presence after which the affected component returns to its fault-free operation. For example, value changes caused by a particle strike.
Intermittent fault	Transient faults and elusive permanent faults with an unknown activation pattern are grouped together as intermittent faults. Typically, the presence of such faults seems to be temporary, however, they reappear after some time.

Table 2.2: Fault Classes [8]

In order to increase the fault resilience of embedded systems, i.e., to identify all faults that can cause the HW/SW system to fail, the fault effects on the system's behaviour have to be analysed such that appropriate measures can be taken. The state-of-the-art approach for such an analysis is to use fault models that describe fault behaviour as observed in the real-world, integrate them into a system model and analyse the behaviour of the fault injected system for single faults and multiple faults.

Persistence of the causing fault	
Soft Error	Errors that are caused by intermittent or transient faults [8].
Hard Error	Errors that are caused by permanent faults.
Protection	
Detected Error	An error is detected when a signal value, e.g., by a safety measure, indicates its presence [8].
Latent Error	An error is latent when it is not (yet) detected [8].
Corrected Error	An error is corrected when the error-free state is restored, e.g., due to logic masking or by an error-correcting algorithm.

Table 2.3: Error Categories

2.6.2 Fault Models

Nowadays, a large number of fault models are in use [113]. In this section we focus on those models that are used by our methods during the experimental evaluation. We want to point out that an extension to support other fault models is straightforward and can be easily implemented.

A permanent fault can occur when a physical connection between a component's output signal and the supply voltage or ground is created, e.g., by electromigration, causing the affected bit to be permanently set to the corresponding binary value. This fault behaviour is commonly modelled as a *stuck-at* fault which sets the value of a signal to either zero (stuck-at-0) or one (stuck-at-1).

As mentioned in Table 2.2, an example scenario for a transient fault is when an energetic particle strikes a transistor in such a way that it creates a temporary electrostatic charge that produces a current spike through the transistor. When such a particle strike changes the output value of a storage element like a register, it is called *single-event upset* (SEU). When a particle strike, instead, changes a value of the combinational part of a system, i.e., a signal value by affecting a transistor of a gate, it is called *single-event transient* (SET) [74] or *digital SET* (DSET) [39] when a demarcation between the digital and analog domain is desired. In this thesis we only consider digital signals and digital fault effects. The effect of an SEU or SET fault is modelled by using a *bitflip*, i.e., the corresponding binary value is changed to its opposite, and only the location of the bitflip determines the name of the event.

In general, the probability that an SET effect is observable at a primary output of an embedded system can be considered relatively low. A reason is that the manifestation of an SET in circuits implemented in CMOS technology is much lower than for SEUs [39]. Another reason is that in circuits like embedded systems which are beyond a certain size and complexity an SET needs to affect a register or other memory elements in order to propagate to a primary output. This requires the following three conditions to be fulfilled [38]. 1) the

SET must have an open logic path, i.e., no logic masking, to a memory element, 2) it arrives at the memory element with a sufficient amplitude and duration and 3) it arrives in the time window where the clock allows the SET effect to be latched-up by the memory element. In sum, one could argue that SETs are neglectable. However, the increasing susceptibility of digital circuits towards SETs due to transistor and frequency scaling as reviewed by [42] requires the consideration of SETs in safety analyses.

The methods presented in this thesis can identify gate-level faults like SETs that can never have a certain effect in the following way. We use the analysis presented in Chapter 6 to identify all combinations of single-bit and multiple-bit faults in program-visible registers that can never cause the considered effect at ISA-level. Note that a gate-level fault can affect one or more registers on higher abstraction levels. We, then, use these results in the analysis of Chapter 7 to deduce the corresponding gate-level faults that can never have the considered effect.

In logic circuits, different faults may be equivalent in the sense that they have the same fault effect on the primary outputs and have identical controllability and observability conditions [26]. For example, both stuck-at-0 faults in Figure 2.8 require a logic 1 at the inputs of the AND-gate for a fault effect to be visible at the output and have the same effect on the output of the AND-gate.

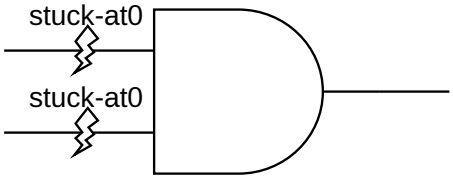


Figure 2.8: Fault equivalence class example

A benefit of fault equivalence classes is that for simulation-/emulation-based methods only the effects of a single fault of an equivalence class needs to be computed, which can significantly reduce the number of required analysis runs. Formal methods benefit from fault equivalence classes because they can reduce the necessary amount of injected fault logic. In order to analyse fault effects on the primary outputs of a component only a single fault per equivalence class needs to be considered, which can significantly reduce the complexity of the formal analysis. We exploit this in our experimental evaluation in Section 8.4.1. In addition to that, well designed fault resilience methods can protect against all faults of the same equivalence class.

Unfortunately, the diverse high-level effects of a single low-level fault in combination with the large complexity of HW/SW systems make *fault abstraction*, i.e., the development of sound high-level fault models that precisely describe the effects of low-level faults, infeasible. However, when the details of high-level effects are not needed in an analysis and the knowledge about any possible effects is sufficient, e.g., when assessing the fault resilience of a HW/SW system, then, formally sound abstraction can be achieved, as we show in Chapter 8.

2.6.3 Fault Injection

In order to analyse fault effects on the behaviour of an embedded system, the corresponding fault model has to be integrated into the system model. For this task *mutants* and *saboteurs* [47] are widely utilized.

```
Register[1] = Register[2] + Register[3];
```

Figure 2.9: Addition

Mutation-based methods inject faults by replacing a component of a system model with another component, the mutant. The mutant behaves like the original component until the fault is activated. The code in Figure 2.9 shows an addition of two program-visible registers for which we create the mutant depicted in Figure 2.10. When the fault's *activation_condition* is fulfilled, e.g., a certain amount of time has passed or a register that the instruction reads contains a specific value, the mutant in Figure 2.10 does not perform the addition but, instead, writes a specific value to the destination register one.

```
if activation_condition
    Register[1] = 'hBADEAFFE;
else
    Register[1] = Register[2] + Register[3];
end
```

Figure 2.10: Example mutant

The greater goal of a fault analysis, e.g., the certification of fault resilience, typically, requires the analysis of a large number of fault effects and not just the effects of one or a few faults. The fault activation condition can be utilised for this to selectively enable or disable the activation of specific faults for a fault analysis run. The idea is to instrument the system model with several or even all faults of a fault list and let the verification engineer decide for each analysis run which faults are allowed to activate during the analysis.

```
1 // Saboteur start
2 if activation_condition
3     Register[2] ^= fault_mask;
4 end
5 // Saboteur end
6 Register[1] = Register[2] + Register[3];
```

Figure 2.11: Example saboteur

Saboteurs are components which are added to a system model to change internal signals of the system. In order to fulfil their task saboteurs are inserted

between a component that writes and another component that reads a particular signal. Like mutants, saboteurs only inject faults when activated. The code in Figure 2.11 contains the addition of Figure 2.9 prepended by a saboteur. If the fault's *activation_condition* is fulfilled, the saboteur inverts the bits of register 2 as specified by the `fault_mask`, i.e., every zero bit in the `fault_mask` inverts the bit at the corresponding position in register 2.

Mutants and saboteurs are very much alike. If we consider only one abstraction level, their main difference is that mutants change the functionality of a component while saboteurs affect the communication between components. However, when multiple abstraction levels are considered, then, a saboteur in one module may look like a mutant at a higher abstraction level. Similarly a mutant module may, in fact, be a module with an injected saboteur at a lower abstraction level.

In this thesis we employ saboteur-based fault injection to analyse the effects of a faulty architectural state in a processor core. In our experimental setup we exploit the fault activation condition to analyse several fault effects in a single analysis run. For example, by leaving `fault_mask` in Figure 2.11 unconstrained a formal method would, then, prove a property for all possible values of `fault_mask`, i.e., for all single bitflips and multiple bitflips in register 2.

2.6.4 Safety Standards

A major concern of industrial enterprises is to ensure the safety of their products. Their need for a thorough and widely accepted certification process yielded several safety standards, most of which focus on a single industry branch and/or specific aspects of a product or its development process.

For example, the main purpose of standards published by AUTOSAR[3] [7] is to ensure compatibility between products of different manufacturers, but it also provides rules for software development and design rules for software to address safety concerns. A comprehensive set of software design rules is the main goal of the programming standard MISRA-C[4] [77]. AUTOSAR and MISRA-C are widely applied in the software domain. However, their focus is on the reduction of development faults in the software and not on an increased resilience of embedded systems against operational faults in the hardware.

Widely applied standards in the domain of safety-critical embedded systems covering operational faults in the hardware are, for example, DO-178C/DO-254 for aviation and industry specific standards derived from IEC 61508 like ISO 26262 for the automotive industry.

The standards require a risk assessment that combines severity and probability of possible accidents. The result of the risk assessment determines the safety level requirement for an embedded system. This typically leads to the requirement that the number of faults that can affect safety-critical regions over a period of time must stay below a certain limit.

3 **AUT**omotive **O**pen **S**ystem **AR**chitecture
4 **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation

For cost reasons, the set of faults which the system is made resilient against must be selected carefully. Fortunately, in most applications a protection against all faults is not necessary because masking mechanisms inherent to the system prevent a fault from propagating [29, 31].

Single-Point Fault	A fault that can affect safety-critical system parts where it can be neither detected nor corrected.
Residual Fault	A fault that can affect safety-critical system parts and escapes safety measures.
Multiple-Point fault	A fault that cannot affect safety-critical system parts or a fault that can but is detected or corrected by safety measures, but still could violate a safety goal in combination with one other independent failure.
Detected Multiple-Point fault	A multiple-point fault that is detected by safety measures.
Perceived Multiple-Point fault	A multiple-point fault that is perceived by the driver.
Latent Multiple-Point fault	A multiple-point fault that cannot be detected by safety measures or perceived by the driver.
Safe Fault	A single-point fault or multiple-point fault that cannot affect safety-critical system parts, e.g., because of logic masking.

Table 2.4: ISO 26262 Terminology (cf. Section 7.4.3.2 of [51])

For example, ISO 26262 requires hardware faults to be classified as "safe faults", "single-point faults or residual faults", "detected or perceived multiple-point faults" or "latent multiple-point faults" (cf. Table 2.4).

Unfortunately, ISO 26262's definition of latent fault, which is the short name of a latent multiple-point fault used in the standard, conflicts with the commonly used definition of latent fault presented in [8] (cf. Table 2.1). Since categorising multiple-point faults is out of the scope of this thesis we use the term *latent fault* according to the definition provided in Table 2.1 in the remainder of this thesis.

Safety mechanisms have to be implemented that prevent faults from leading to single-point failures, reduce residual failures or prevent faults from being latent (cf. Section 7.4.3.3 and Section 7.4.3.4 of [51]).

Some parts of a product may have a larger relevance w.r.t. the safety of the whole product than other parts. ISO 26262 addresses this by defining four different *automotive safety integrity levels* (ASIL). Each ASIL requires that certain quantitative target values for the single-point and latent fault metrics are met (cf. Section 8.4.5 and Section 8.4.6 of [51]). The values computed for both metrics can be improved by increasing the fraction of identified "safe faults" (cf. Section C.2.2 and Section C.3.2 of [51]).

Due to their growing maturity, formal verification methods receive growing acceptance by safety norms. In DO-333 [84], the Formal Method Supplement to DO-178C [83], defines formal methods as "mathematically based techniques for the specification, development, and verification of software aspects of digital systems". The importance of soundness is emphasized: "an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true."

Also ISO 26262, in its software part, recommends formal verification and static analysis by Abstract Interpretation as verification methods for higher criticality levels, e.g., for verification of software architectural design, software unit verification, or software integration verification (cf. Table 4, Table 7, and Table 10 of [52]).

In its hardware part, ISO 26262 highly recommends the employment of inductive (for all ASIL) and deductive (for ASIL C and D) methods to analyse the safety of a hardware design. In industry, *failure mode effect analysis* (FMEA) [37] and *fault tree analysis* (FTA) [110] are widely used techniques to qualitatively analyse fault effects, including in the embedded system domain [62, 72]. FTA is a top-down method to deductively identify all low-level faults that can lead to a failure on higher abstraction levels. FMEA is a bottom-up method that starts with a low-level set of faults and inductively computes their effects on higher abstraction levels.

The methods presented in this thesis have similar goals as FMEA and FTA. Like FMEA, one goal of this thesis is to employ a low-level analysis to compute the fault effects on higher abstraction levels. And like FTA, another goal of this thesis is to employ a high-level analysis to identify corresponding faults on lower abstraction levels. The difference is that the scope of FMEA and FTA include physical aspects, e.g., failures in the power supply, and in practice are applied on a very high abstraction level while this thesis focuses on the lower-level logic of a digital circuit. In contrast to FMEA and FTA which require substantial manual effort and do not have a definition of completeness, this thesis is largely automated and can provide guarantees that all possible fault effects of the considered fault model are found by our analyses. Note that our methods can be easily integrated into other methods analysing fault effects, including FMEA and FTA. For example, a product-wide FMEA can take the results of our method as an input and propagate the computed fault effects from the digital computing platform further into the rest of the system.

2.7 Control Flow Graph

A *control flow graph* (CFG) explicitly models all possible outcomes of control decisions, denoted as *control flow*, of a software program. Control decisions are made by control instructions like jump or branch instructions. Definition 1 formally defines a CFG.

Corollary 1 follows directly from Definition 1. In this thesis we use a slightly modified CFG definition by re-defining that a single node represents only one instruction.

Definition 1. *A CFG $C = (V, E, r)$ is a triple where V is a non-empty set of vertices, $E \subseteq V \times V$ is a set of directed edges and r with $r \in V$ is the root node. A CFG node $v \in V$ represents a basic block of a software program. Depending on the considered abstraction level, a basic block is a non-branching sequence of statements or instructions. The root node r represents the start of the considered software program. There is an edge $e \in E$ with $e = (v_i, v_j)$ iff there exists an execution of the program where the processor transitions from the last instruction of v_i to the first instruction of v_j [3].*

Corollary 1. *There exists a path $p = (e_1, e_2, ..., e_n)$ with $e_i \in E$ from the root node r to a node $v \in V$ of a control flow graph C, if and only if, the corresponding instruction sequence is reachable from the software's start.*

Most static software analysis tools use CFGs. Some of them perform their analysis entirely on a CFG. Others, like the tools used in this thesis, use a CFG to perform some preprocessing before the actual analysis starts. For example, in Section 4.1 we discuss how we generate a formal computational model for a HW/SW system based on a CFG and in Section 8.1 we employ CFGs to find a feasible decomposition for large software systems.

2.7.1 CFG Generation

Control flow graphs can be generated for software programs on several abstraction levels, including machine level, assembly level and C level. Generating CFGs for software programs like firmware and drivers on C level may, generally, require a complex code analysis. However, as a consequence of standardization efforts, e.g., as given by the MISRA-C or AUTOSAR standards, statements following control decisions are often provided explicitly in the C code of standard-compliant software programs. The CFG generation for C code compliant with such standards is in practice, therefore, not overly complex.

However, sometimes the C code is not available, for example, because the software is a third-party *intellectual property* (IP). Analysing this software then makes a CFG generation based on machine code or assembly code necessary, which can be a very complex task to pursue as shown by [61, 103]. An important factor for the complexity is the number of indirect jumps and branches where the successor instruction depends on the concrete value in a program-visible register. Computing all possible values of such a register is, in general, very complex. But without the successor information for all instructions, the generated CFG would not comply with Corollary 1 causing it to be *incomplete*. An analysis based on an incomplete CFG yields incomplete results, since not the entire software behaviour can be considered.

For the generation of our computational model that is used by the methods presented in this thesis we employ formal methods to complete a CFG extracted from assembly code. We also exploit properties that are characteristic for firmware and drivers to reduce the CFG completion complexity.

2.8 Automated Test Pattern Generation

In the domain of digital circuits, "testing" refers to a class of well established techniques to detect hard error-producing faults. A widely employed technique is to apply carefully selected values to the primary inputs of a digital circuit, i.e., a *test pattern*, and compare the values on the primary outputs with the expected values [26].

Test pattern-based testing is applied after manufacturing and before shipping of a digital circuit and aims at detecting permanent faults. Test pattern-based testing is also applied after shipping by using *built-in self-test* (BIST) [97]. Like post-production, pre-shipping testing, BIST addresses permanent faults and faces the problem that the implementation of self-tests to detect all possible faults can create an infeasibly large area or performance overhead.

The main objective of testing is to detect all faults of a given fault list with a most compact set of test patterns. Compactness is achieved by removing unfavourable test patterns from the generated set, e.g, when a test pattern detects a subset of the faults detected by another test pattern. A test pattern detects one or more faults by activating them and setting the conditions for the produced errors to be propagated to a primary output. If the actual output values produced by a circuit are the same as the expected ones, then the circuit under test does not contain a fault that is covered by the applied test pattern. However, if the received response does not exactly match the expectation, then the applied test pattern has detected a fault.

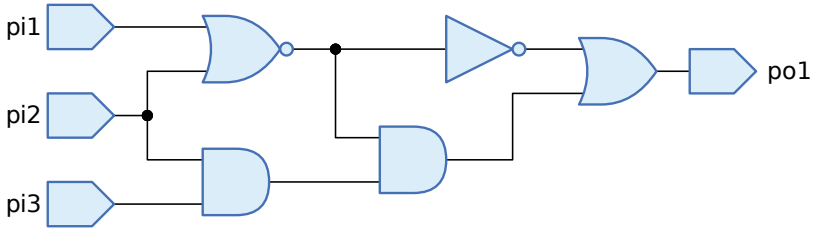


Figure 2.12: Example gate-level circuit

Lets consider the gate-level circuit in Figure 2.12, which has three primary inputs, one primary output and a couple of logic gates. Lets assume further that a test pattern to detect a stuck-at-0 fault at the input of the OR-gate as indicated by the arrow in Figure 2.13 should be found. In order to activate the fault the corresponding signal value must be set to one, the opposite of the fault value, and the other input of the OR-gate must be set to zero so that the input where the fault is located determines the output value of the OR-gate.

Figure 2.14 shows a possible test pattern, i.e., a value assignment for the primary inputs, to achieve this. For the fault-free case the output value of the OR-gate which is also the primary output of the considered component is one (cf. Figure 2.14) while in the event of the considered stuck-at-0 fault it would be zero.

A complete set of test pattern contains at least one test pattern for every fault in a circuit. The generation process for such a set is done in an automated

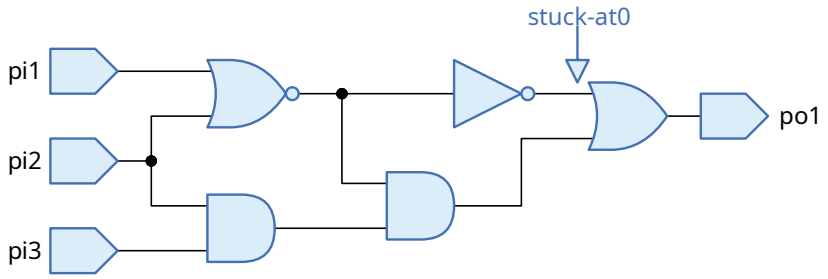


Figure 2.13: Gate-level circuit for test

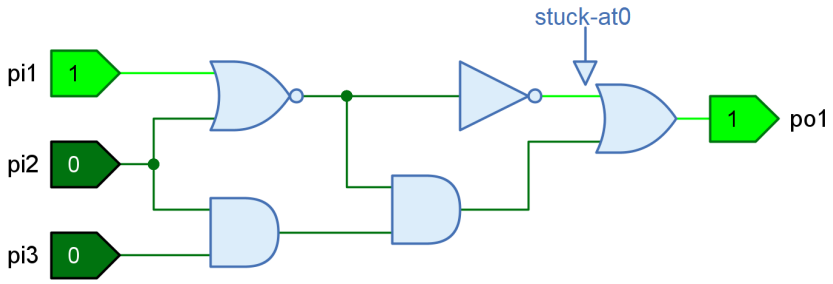


Figure 2.14: Gate-level circuit under test

fashion by software tools and, therefore, called *automated test pattern generation* (ATPG) [26, 65] .

In general, fault detection requires value assignments at the primary inputs of a component such that a considered fault is activated, i.e., an error appears, and the produced error gets propagated to a primary output.

If a fault cannot be activated, then the signal where the fault is located cannot be controlled. These type of faults are denoted as *ATPG uncontrollable* [46]. Sometimes no assignment of values to the primary inputs can propagate the produced error of an activated fault to any primary output. These type of faults are denoted as *ATPG unobservable* [46]. ATPG uncontrollable and ATPG unobservable faults are denoted as untestable faults, because they cannot be detected by test patterns.

Figure 2.15 shows an example of an uncontrollable fault. In order to detect the stuck-at-0 fault at the output of the AND-gate pointed by arrow the gate output has to be set to one. However, this is not possible, because this output is always zero, for all value combinations at the primary inputs. This stuck-at-0 fault is, therefore, not ATPG detectable.

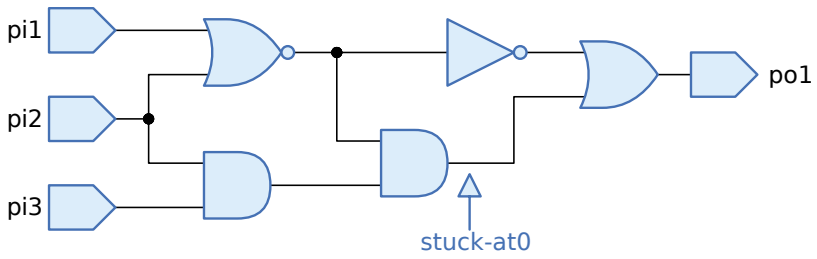


Figure 2.15: Gate-level circuit - uncontrollable signal

An example for an unobservable fault is depicted in Figure 2.16. Here, a test pattern for the stuck-at-0 fault at an input of an AND-gate should be found. The input is controllable and can be set to one, the opposite value of the fault, by applying a zero to the primary inputs one and two. In order to allow the value of the first input to determine the output value of the AND-gate the second input of the gate must be set to one. However, this requires the primary input two to be set to one which creates a conflict. This conflict prevents the propagation of the erroneous value at the first AND-gate input to the primary output of the circuit which makes the considered fault unobservable.

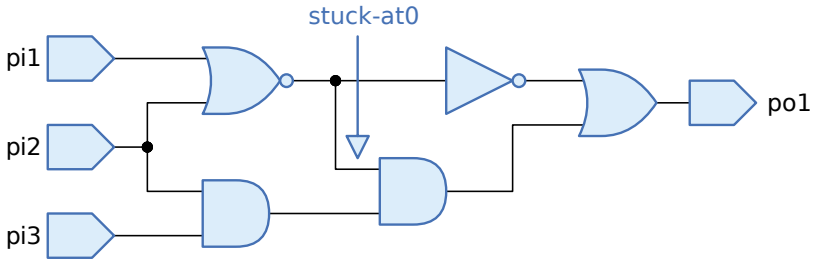


Figure 2.16: Gate-level circuit - unobservable fault

Uncontrollable and unobservable faults can indicate the potential for logic optimisations. For example, untestable faults may be replaced by constant logic signals of the value that they are stuck at because they do not change the input-output behaviour of a circuit. Untestable faults that are safe to be removed are denoted as *redundant faults* [112]. In hardware systems that do not have a software part the number of redundant faults can be considered relatively low as they are typically highly optimised. Digital circuits like general-purpose processors are designed in such a way that they can execute a large variety of software programs and provide a large variety of functionalities. However, a single software program rarely uses all of the resources provided by the processor.

Particularly drivers and firmware are highly specialised software programs. This can create a large set of application-dependent uncontrollable and unobservable faults. For example when a software program never computes a multiplication, then faults in the multiplication unit become uncontrollable and unobservable. We discovered several such *application-dependent redundancies* in our experimental evaluation presented in Sections 6.2 and 7.1.

2.8.1 ATPG-based Testing in Practice

ATPG for combinational circuits can be considered a solved problem in practice. Commercial tools are available that can generate test vectors for designs of industrial complexity.

Generating test patterns for sequential circuits can be, in contrast to combinational circuits, very complex because they contain storage elements like registers which not only delaying the error propagation by storing values for a clock cycle but are also used in feedback loops where values are fed back into

the combinational part of the circuit which significantly increases the computational effort for every formal analysis and ATPG.

The complexity problem makes the application of ATPG algorithms infeasible for sequential circuits of large size.

A solution for the sequential ATPG problem is to use all registers in a sequential circuit as primary inputs and primary outputs, which reduces the problem to classical combinational ATPG [112].

This can be achieved by adjusting the circuit in such a way that all registers are connected to a long register chain, where the input of the first register can be configured to obtain its value from a primary input and the output of the last register can be configured to send a value directly to a primary output. In "system mode", the sequential circuit behaves like a circuit without this modification. In "scan mode", however, the input of each register is connected to the output of another register. This concept is called *scan chain* [112], because it is used to test, i.e., scan, the combinational part of the circuit.

The trade-off for this complexity reduction is that the added long wires to connect the registers create an area overhead and can have an negative impact on the circuit's performance when the capacitance of the extra wiring reduces the signal speed [49].

3 RELATED WORK

In recent years, significant progress in solver technology has lead to a substantial increase in the capacity of formal hardware verification tools. However, they still suffer from scalability problems that are inherent to conventional model checking when HW/SW systems of realistic complexity have to be analysed. With respect to formal analysis of fault effect propagation in industrial-scale HW/SW systems, across multiple abstraction levels, only little prior work exists.

Existing formal techniques [21, 43, 66, 96] which analyse the effects of hardware faults on the behaviour of a HW/SW system require manual effort to create a highly abstracted model from either the implemented HW/SW system or its specification.

Modelling a system on a high abstraction layer can provide important information about the reliability of an entire system, e.g., an autonomous vehicle [43] or power plant [66]. However, additional methods are required to close the gap between these high abstraction levels to which fault effects can propagate and the lower abstraction levels where the faults occur first. Providing such a method is the goal of this work.

Most work employing formal methods for fault analysis on lower abstraction levels either focus solely on the hardware [22, 71, 93, 99], or on the software [45, 68], and does not take the interaction between both into account. Identifying safe faults, as is done in [71], helps to certify the resilience of HW/SW systems for specific safety levels. Such methods can provide valuable insights in HW/SW systems where the software can undergo drastic changes during in-field product updates. For those systems, identified safe faults as used for the assessment of the safety level have to be software-independent. However, for embedded systems where the software changes only very little over the system's lifetime, the identification of application-specific safe faults can be necessary to provide a timely and cost-efficient certification of the required safety level.

Furthermore, recent work [80] demonstrates that reducing the vulnerability against certain faults by software-level hardening alone without consideration of the underlying hardware can actually increase the cross-layer vulnerability of a system.

In [35, 67, 82, 92] symbolic techniques are used to formally analyse the effects of faults. In contrast to our approach, the approach of [92] requires manual effort to create an abstract formal model and only considers hardware. Also [35] analyses only hardware and considers fault effects in a processor without taking into account the software. In [67] only faulty software behaviour is consid-

ered. A mapping to concrete hardware faults is not attempted. In [82], certain fault scenarios are examined using symbolic execution to formally analyse fault effects. The proposed method is able to find all faults that lead to a specific output scenario for a given input scenario. However, the approach is neither meant nor expected to scale for a comprehensive fault analysis, as is the objective of our work.

Most other work pursuing approaches with similar objectives as ours achieve scalability to large systems by modelling faults at high abstraction levels, or by using simulation and/or emulation [40, 41, 48, 85, 89].

It is in the nature of all simulative approaches that full confidence can never be gained on the absence of fault effects. Such simulative approaches can, however, nicely complement our work. For example, [48] has evaluated the effects of hardware faults on the architectural processor state and has elaborated on how intermittent hardware faults on the RT level affect the behaviour of processor components, including program-visible components like the register file. Knowledge about how the effects of physical defects propagate through the HW/SW layers can be used to develop realistic fault models on the architectural level and can provide a basis for methods like the one proposed in this work. The proposed formal approach can also be integrated into a fault analysis flow where a fast fault simulation discovers the most vulnerable parts and reduces the fault list for the subsequent formal phase. It is then the task of the formal approach to complete the picture with all other fault effects and to generate guarantees that certain faults can never propagate to certain locations. Such a general setup has already been pursued in [23, 94]. However, they only consider hardware [23], or suffer from scalability limitations [94] which can be mitigated in our work by a formally sound composition of models at different layers.

A key element in this work is the creation of a fault dictionary to identify both application-dependent safe faults and safety-critical faults in a fully formal way. In [9] the authors pursued a similar goal by applying a combination of simulation and formal technique to identify application-dependent safe faults. Scalability of the formal analysis is achieved by reducing the state space of the analysed sequential circuit. The method requires manual effort to constrain parts of the processor, e.g., those that are not used, and to define fault propagation barriers. It also relies on an automatic translation of simulation traces to formal properties which are the basis for the identification of safe faults. This exposes the method to the risk that incomplete simulation results, e.g., a not included corner-case scenario, cause non-safe faults to be classified as safe faults.

In contrast, our approach does not suffer from this potential inaccuracy by merit of the formal methods it is based on, which scale even for large HW/SW systems, as demonstrated in this dissertation. Furthermore, our approach can directly identify all application-dependent safety-critical faults. The application-dependent safe faults can, then, be determined as the complement set of the identified safety-critical faults w.r.t. the complete fault list.

4 PROGRAM NETLIST

The *program netlist* (PN) is the underlying model of the fault analysis methods proposed in this thesis. A PN formally models the behaviour of a processor hardware with respect to a specific software program. In this chapter, we start with a review of this model, originally published in [90]. We then, in Section 4.3 and Section 4.4, provide new contributions to increase the scalability of the PN generation and PN-based analyses like the methods presented later in this thesis.

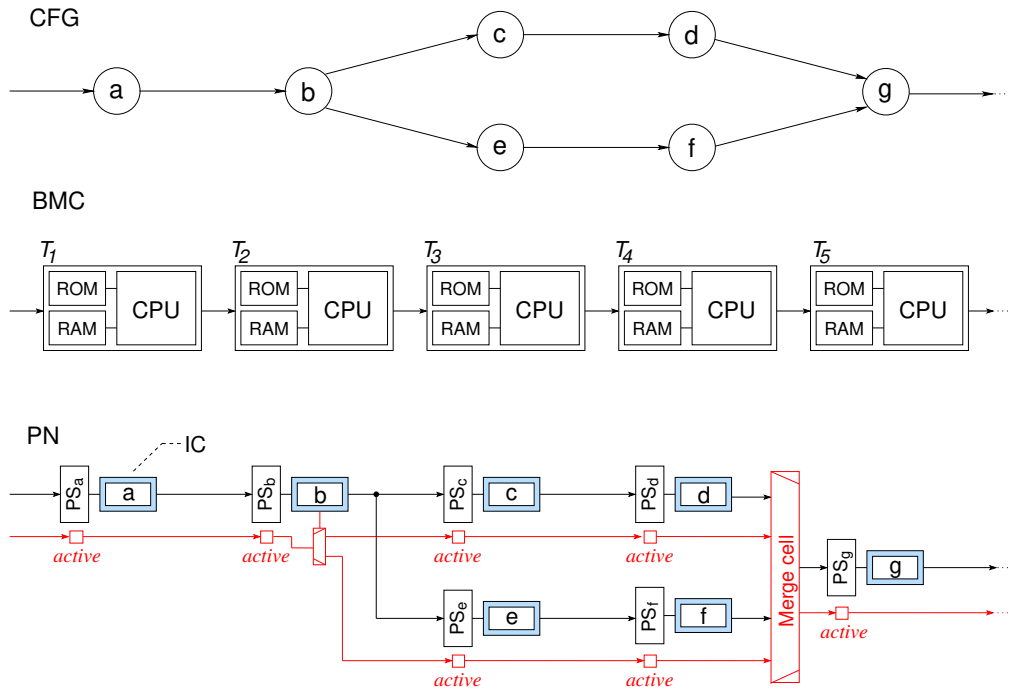


Figure 4.1: BMC unrolling of HW (middle) vs. program netlist (bottom) for a CFG (top)

A straightforward approach for the verification of hardware-dependent software could be to model the software as machine code stored in a ROM which is connected to the processor (including CPU and RAM). As a result, a hardware model for the entire system is obtained which is represented by its transition relation, T, in the usual way. Verification could be based on BMC by unrolling this transition relation for a finite number of time steps. For instance, the maximum number of clock cycles along the longest execution path of the program could be chosen for the unrolling. Figure 4.1 presents an example. In order

to keep the discussion simple, it is assumed that the CPU requires one clock cycle to execute each instruction.

Such a hardware-style BMC approach is attractive for performing a formal HW/SW cross-layer analysis since the behaviour of the software can be represented by hardware structures at the desired level of detail. However, the approach will yield a complex computational model, similarly as in sequential ATPG, representing the entire processor hardware multiple times, once in each i-th time step. Only very small designs and only short execution paths can be examined with such an approach.

Let us examine in detail what would happen if a SAT solver is used to reason on such a model when performing a given proof. Consider the piece of CFG and the BMC unrolling shown in the top and in the middle of Figure 4.1. The nodes in the CFG represent individual instructions of the machine code. Each T_i in the model describes all software behaviours that could occur in the i-th time step. In time step 1 instruction a is executed. No other instruction can be executed at this point in time. This means that the system can be modelled under the constraint that this particular instruction is performed. This fact can be exploited to drastically simplify the transition relation T_1. The same process can be followed to model the system at time point 2. (Only instruction b can be executed at that time point.) Now consider time point 3. At this time point, instruction c or instruction e can be executed. T_3 can still be simplified but it now needs to model both of these instructions. Hence, fewer simplifications to the transition logic can be performed.

We observe that in more complex CFGs with numerous branches and loops the simplification can benefit from such constraints only during a fairly small number of steps in the initial parts of a program. At later time points, there exist many possibilities as to what instructions can be performed. Therefore, when unrolling the transition relation, the individual T_i has to model (almost) the entire hardware system, since no (or only few) constraints can be identified. If a SAT solver has to enumerate the search space to prove some property on this model, it will obviously suffer from the sheer complexity of this representation.

Moreover, there is an additional problem for the SAT solver making the situation even worse. When backtracking through the search space the solver makes assignments to the variables of this model that mix situations occurring in different runs of the program. For example, if instruction c is performed at time 3, it is not possible that instruction f is performed at time 4. If the SAT solver makes assumptions in its branching decisions relating to instruction c at time 3 and instruction f at time 4, it will enter the non-solution area of the search space. It may take a large number of backtracks until this is discovered.

In conclusion, a SAT solver needs to deduce all information about the possible execution paths of the program via clause learning, backtracking, and similar concepts "from scratch". This is because the model lacks an explicit view on execution paths. Since the program's control flow is represented only implicitly by the unrolled hardware, reasoning on the program requires a high

computational effort. It is apparent that such a model, even if small, is computationally inefficient and requires excessive computational resources.

4.1 Program Netlist Generation

The PN approach is related to the BMC approach illustrated above, however, key obstacles to scalability are removed. The basic idea is the following. The unrolling of the processor with its instruction and data memory is not done by replicating the full transition relation at every time frame, but rather *instruction by instruction* (cf. the PN at the bottom part of Figure 4.1). At branching points in the software, the unrolled logic is duplicated, modelling each execution branch separately. This instruction-wise unrolling along execution paths allows for a significant reduction in the amount of logic that needs to be replicated: Since the actual instruction in every unrolled logic block is known and fixed, many constants exist that can be propagated in order to simplify the logic block so that all circuitry that is not needed for modelling the instruction behaviour is removed.

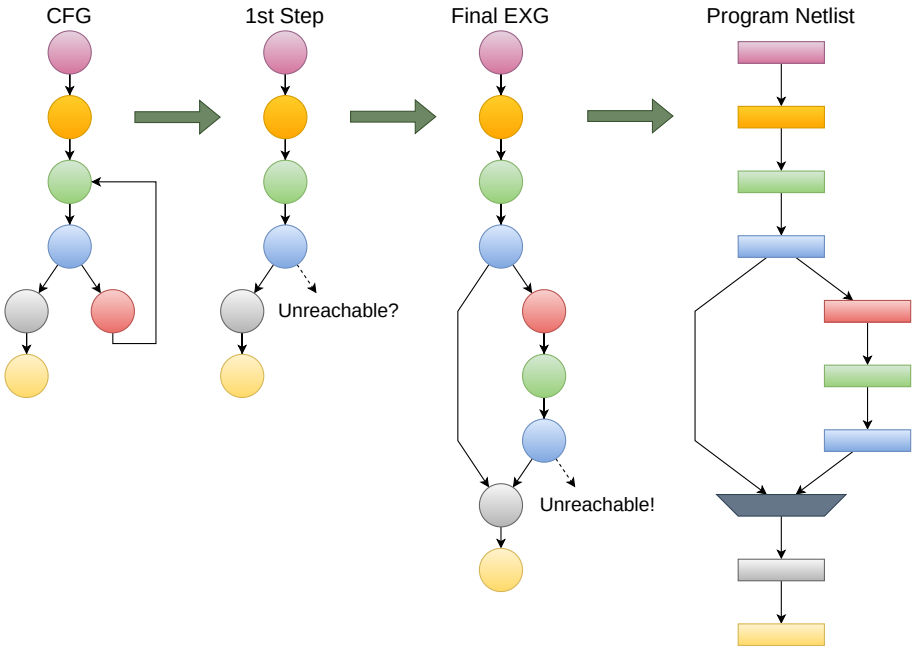


Figure 4.2: PN generation steps

This analysis is moved to the pre-processing phase, illustrated in Figure 4.2 where the CFG is unrolled into an *execution graph* (EXG) containing all feasible execution paths of the software. In each unrolling step, we unroll all program paths that are known to be reachable (breadth-first algorithm). Unrolling is performed until every program path either reaches one of the program end nodes, as defined by the verification engineer, or a control instruction where the reachability of its successors is unknown (cf. 1st step in Figure 4.2). We then perform a formal analysis to identify all reachable successor instructions based on an intermediate PN. We obtain an (intermediate) PN by replacing all nodes

of the (intermediate) EXG with combinational logic denoted as *instruction cell* that formally models the processor behaviour when executing the considered instruction. We describe instruction cells in detail in Section 4.2. In the final step of the unrolling process, when all program paths have reached program end nodes, we create the final PN. Compactness of the generated EXG is ensured by pruning, i.e., not unrolling, unreachable program paths and merging two or more program paths, unless doing so would create a loop [90]. For the industrial software programs used in our experiments, this effectively prevents path explosion as it is often experienced in symbolic simulation.

The resulting information about reachable execution paths of the software is encoded into the control logic of the program netlist (cf. the logic blocks and signals shown in red in the PN of Figure 4.1). Exploration of the program behaviour can be done now by activating or deactivating whole execution path segments by assigning the *active signals* associated to the control logic. These specific control structures make execution paths and the program's control flow explicit to the SAT solver. Previous work [90, 107, 108] have shown the efficiency of this approach. We also use the active signal to merge program paths. Only one of the program paths to be merged can possibly have a program state with an asserted active signal, because only one path is executed in any given program run. We use the active signal to decide which program state is forwarded to the next IC. This is illustrated in Figure 4.2 by the multiplexer in the PN.

The CFG used as the starting point for model generation can be incomplete, e.g., branch targets may be unknown because of indirect addressing, as is often the case when a CFG is generated from a real machine program. This incompleteness is acceptable because the missing information is generated during the model generation process. This is done by interleaving the unrolling process with a SAT-based analysis, as explained above, to fill in the missing information.

4.2 Instruction Cells

For a given ISA and program at machine level, the behaviour of the processor can be precisely modelled for each individual instruction of the program. A logic block that models atomically the effects of an individual instruction on a set of state variables is called an *instruction cell* (IC). The set of state variables that the cell modifies depends on the type of instruction and includes all registers of the hardware platform that are visible to the software as well as memory locations associated with program data variables and input/output registers. These state variables constitute the *program state* (PS) of the programmable HW/SW system.

A somewhat simplified example of an IC template in pseudo-code notation is shown in Figure 4.3. (Information about bit widths is abstracted in this and the following examples to make them better readable.) In our practical implementation instruction cells are specified in *SystemVerilog*. Figure 4.3 describes an ADD instruction for RISC-V [109]. As can be seen, the instruction

```
1 ADD(const Rd, const Rs1, const Rs2, in PS, out PS')
2 {
3   PS' = PS;
4   PS'.RegisterFile[Rd] =
5     PS.RegisterFile[Rs1] + PS.RegisterFile[Rs2];
6 }
```

Figure 4.3: Instruction cell - RISCV

cell takes as input the register names that the operation is performed on. This information is encoded in the specific assembler instruction of a program. The identifiers `Rd`, `Rs1` and `Rs2` in the template are replaced with the actual register addresses when the instruction cell is instantiated during PN generation. The instantiated instruction cell has only one input, the current program state, and one output, the next program state. The body of the instruction cell consists of a forwarding of the program state from the input to the output, with the exception of the one register that contains the result of the addition. Instruction cells are time-abstract. i.e., they do not provide cycle-accurate timing information. However, the ordering of a sequence of instructions in a PN provides an abstract model of time that we use for fault injection.

4.3 Scalable PN Generation

Improving the scalability of the PN generation process allows us to analyse larger software programs. This section presents a number of new techniques, developed as a part of this dissertation, for reducing the complexity of the PN generation. Each technique is independent of the others. The safety engineer is free to select any combination, as needed by the problem instance at hand.

4.3.1 Compositional PN Generation

Large or very complex software programs can cause unfeasible run times for PN generation. In the following, we present a compositional approach to reduce complexity by splitting a software program into segments and generating a PN for each segment. After PN generation, the segment PNs are combined into a composite PN that models the behaviour of the full software program.

Splitting a software program into segments is a manual procedure, and finding an optimal decomposition is not trivial. A simple approach would be to decompose the PN based on software *functions*. Such a decomposition, though conceptually simple, usually leads to suboptimal solutions because it does not take into account the context of the function call within a particular program execution. Such a general, context-agnostic, program decomposition based on function boundaries needs to represent and consider all possible argument values and usually leads to an unnecessarily complex PN. If, however, the calling context is available in the decomposition, then the actual arguments to the

function and other parameters like the stack pointer or the return address are known and can be used for model simplification.

We, therefore, propose to first extract the CFG from machine code, e.g., by using a static software analysis tool, in order to guide the safety engineer in selecting promising cut points for a decomposition. The CFG provides an abstract view on the machine program and allows the safety engineer to roughly assess the complexity of program paths in prospective software segments.

We model a given software program as a set of machine instructions, $U = \{\iota_1, \iota_2, \ldots\}$ and a set of program state variables. A machine instruction ι_k is a pair (a_k, c_k) of a program memory address a_k and an instruction word c_k. In a CFG, every instruction is assigned to a basic block consisting of a non-branching instruction sequence (cf. Definition 1). The ordering of a sequence of basic blocks is maintained by edges between two basic blocks. Every edge of the CFG can be used as a cut point for decomposition. A selected CFG edge indicates the last instruction of a $segmentSW_i$ and the first instruction of a new $segmentSW_j$.

Definition 2. *The over-approximation of the starting state of a $segmentSW_i$ modelled in a PN is called* abstract starting state $\psi_{i,0}$. *The set of program states which the modelled $segmentSW_i$ can end in after starting in any of the states in $\psi_{i,0}$ is modelled by the* abstract ending state $\psi_{i,end}$ *of the PN.*

The execution of a single program run is a sequential execution of segments, where each $segmentSW_i$ starts with program state $s_{i,0}$ and ends with program state $s_{i,\text{end}}$ which is the starting state of the next segment $segmentSW_j$. In our formal model, an abstract state (Definition 2) represents a set of concrete program states. When compared with the aforementioned, more naïve, decomposition based on functions, this approach allows us to define larger segments with benign complexity due to a less complex $\psi_{i,0}$.

After software decomposition by splitting into segments, we generate a PN for each $segmentSW_i$. For this task we have to constrain the abstract starting state $\psi_{i,0}$ of all $segmentSW_i$ which do not include the startup procedure by an over-approximation of the possible states at the given point of execution. Without such a constraint, PN generation would start from an *any-state* causing the generation process to add a large number of unreachable software behaviours. In the worst case it makes PN generation impossible. Furthermore, the increased complexity of a PN directly affects the complexity of any analysis using it.

We generate the over-approximation constraint for the abstract starting state $\psi_{i,0}$ by Abstract Interpretation. In our experiments, we use the static software analysis tool ValueAnalyzer from AbsInt for this purpose. In practice, the over-approximations obtained using Abstract Interpretations were so tight that we did not observe any unreachable software behaviours in the composite PN model.

We combine the segment PNs to a composite PN by assigning the ending state of one segment to the starting state of the next segment. The ordering of segments is determined by the chosen CFG decomposition.

4.3.2 Instruction Abstraction

Some instructions perform more complex computations than others, leading to an increased model complexity when generating the PN. This is especially true for multiplication or division. In many cases, it is possible to reduce the model generation complexity by abstraction.

We may replace an instruction by an abstract version of it such that the value space of the abstract computation result is an over-approximation of the concrete result. The simplest abstraction is obtained by removing all computation logic and replacing the output result with a symbolic variable representing any possible value.

In general, such an abstraction may add software behaviour to the formal model that does not exist in the actual, concrete, software. In principle, this may even increase the complexity of the PN instead of reducing it, e.g., when actually unreachable program paths become reachable in the abstract model. In the worst case, abstract instructions can make the PN generation infeasible, but they can never lead to an unsound model. While there may exist spurious executions in the PN, still, every execution in the actual software has a representation in the PN.

In practice, it is up to the verification engineer to select instructions for abstraction, and to choose appropriate replacements such that the overall model generation complexity is improved. Fortunately, most instructions causing complexity problems are used in data processing algorithms and often do not affect the control flow. In such cases, a simple ad-hoc approach can be to replace a complex instruction by a simpler one that uses the same operand and result registers. For example, a multiplication may be replaced by an addition. Such ad-hoc modifications may, in principle, compromise model soundness by ruling out execution paths that were feasible in the original program. However, we can easily cure this by running a formal property check that tests for newly missing branches and execution paths. These properties are generated automatically by our tool, and provide an effective aid to the verification engineer in the process of creating a sound and compact model of a software program.

4.3.3 Program Path Priorities

In order to keep the PN compact, its generation process tries to merge program paths as soon as possible, as long as doing so does not create a loop. Sometimes, a merge at location A can prevent a merge at a different location B, because the latter merge would only create a loop if the former was performed. However, there are cases where the latter merge would have been better in terms of model compactness and complexity. This issue can cause a significant increase in the complexity of the PN, impacting every subsequent analysis and the PN generation process. In the worst case, merge decisions make the PN generation infeasible due to overly long runtimes.

Finding the optimal locations for merges is difficult and they can change during the PN generation. Optimizing the PN size during its generation can, therefore, create an infeasibly large overhead.

A simpler solution is to orchestrate the PN generation by prioritizing certain functions in the CFG unrolling. This means that selected functions have a priority during the PN generation process, i.e., during the CFG unrolling. This prevents premature merges of non-prioritized PN parts with prioritized PN parts that have the potential to cause logic duplication of the corresponding program path due to unfavourable selection of merge locations.

4.3.4 Address Caching

In safety-critical embedded systems, static variables are commonly used to store non-temporary values. The compiler places them at a fixed location in the memory, i.e., the address range for such variables stays the same for the whole runtime of the program.

In addition, most functions in embedded system software access only a specific set of variables and the function arguments mainly specify the part, i.e., the memory addresses, of the variables they access during a single function execution. A similar behaviour can be observed for jumps, where, for a specific jump there is only a fixed set of target addresses and the input values only select an element from this set as jump target.

We exploit this observation by storing found addresses of memory accesses and jumps and mapping them to the corresponding instruction. At the next time, when addresses for the same instruction have to be found, we run a formal check whether these addresses include the stored set of addresses. If this check fails, we perform the regular address computation, beginning with an empty set of addresses. If, however, the check succeeds, we try to find more addresses by performing the address computation starting with the stored set of addresses.

Since, in the worst case, we always have to run one extra formal proof, we limit address caching to situations where the number of found addresses is larger than a given threshold.

4.3.5 Experiments

For our experimental evaluation of the techniques presented in this section we generated PNs for several software programs of realistic complexity, including:

- LIN: a driver for a slave node of a *Local Interconnect Bus* (LIN Bus), used as a gateway to external buses; obtained from Infineon, proprietary IP.
- FuelSys: a fuel rate controller for a combustion engine, taken from the fuel control system example of MATLAB [73].
- RSA loop: a loop-based implementation of the *Rivest–Shamir–Adleman* (RSA) encryption algorithm, obtained from [4].
- RSA recursive: a recursive implementation of the RSA algorithm, obtained from [5], providing a larger complexity than the implementation based on a loop.

- AES: an implementation of the *Advanced Encryption Standard* (AES) encryption algorithm, obtained from [64], providing a software program with a large number of memory accesses.

Table 4.1 shows the complexity of each software program in terms of lines of C code (LoC) and computational model size. We compiled each program with *GCC*[1] for a RISC-V architecture.

Software Program	Lines of C code	PN size (# ICs)
LIN	781	3,927
FuelSys	8,869	9,041
RSA loop	61	784
RSA recursive	68	3,063
AES	419	7,441

Table 4.1: Software programs

For these software programs we generated the PN and measured the time it took to complete the generation process. In order to limit the overall experimental effort, we set a limit of three days for a single PN generation, arguing that a longer PN generation for a single program is not feasible. PN generation processes that did not complete after three days are marked by the keyword *timeout*.

Program	PN generation runtime (hh:mm:ss)				
	No Technique	Decom-position	Unroll Priority	Instruction Abstraction	Address Cache
LIN	00:03:35	—	00:03:29	—	00:03:37
FuelSys	timeout	00:47:18	timeout	timeout	timeout
RSA loop	00:00:18	—	00:00:05	00:00:18	00:00:18
RSA recursive	timeout	—	timeout	35:11:26	timeout
AES	timeout	—	timeout	—	02:50:47

Table 4.2: Runtimes for PN generation

Table 4.2 shows the runtime of the PN generation for each selected software program. The value in the second column is our baseline, showing the time needed to generate the PN without employing any of the techniques of this section. The remaining columns show the PN generation times for each technique. A dash is printed if techniques are not feasible for the particular software program.

For example, LIN does not make use of instructions that could benefit from abstraction. Applying the compositional approach is, also, not meaningful since the time overhead needed for selecting a decomposition would already

1 **g**nu **c**ompiler **c**ollection

be larger than its short baseline PN generation time. The encryption algorithms only use one or two functions making a decomposition difficult and AES does not use complex instructions preventing the application of instruction abstraction.

Whether a particular technique reduces the time for PN generation depends on the nature of the individual software program. For example, prioritizing some functions for unrolling or activating address caching in some LIN instances neither improved nor degraded the run times for PN generation. In the loop implemented RSA algorithm, instructions can only access a single memory address and never use multiplication results for control decisions. Instruction abstraction and address caching, therefore, cannot reduce the PN generation time, but also don't increase it.

Due to the inherent complexity of FuelSys the PN generation for this program is infeasible without our compositional approach. We were also not able to generate a PN before timeout by using instruction abstraction or address caching. However, due to its code size FuelSys is a realistic example for a compositional PN generation. By analysing the CFG of FuelSys and the complexity of its functions, as provided by the source code, we manually decomposed FuelSys into nine segments. For each segment, we, then, computed the constraints for $\psi_{i,0}$ and generated the PN.

Segment	PN generation (time in s)		PN size
	Baseline	Optimized	(# ICs)
Segment1 compute	1,458	296	2,436
Segment2 TabIdxS49T1_a	8,947	14	488
Segment3 Tab2DIntp2I1T1_a	464	47	320
Segment4 TabIdxS49T1_a	93	13	569
Segment5 Tab2DIntp2I1T1_a	895	90	630
Segment6 Tab2DIntp2I1T1_a	659	86	618
Segment7 Tab2DS17I2T4169_a	727	350	1,930
Segment8 Tab2DS17I2T4169_a	52,557	1,922	1,564
Segment9 fuelratecontroller	50	5	486
Assembly	15	15	9,041
Overall	65,865	2,838	9,041

Table 4.3: FuelSys segments

Table 4.3 provides information for each segment. In the first column we mention the name of the function where the respective segment starts. Several segments start with the same function that is called at several program locations. The second column of the table reports the runtime needed for generating the PN without the use of other techniques. We then generated the segment PNs by using additional techniques, as applicable for the particular segment. The runtimes of the optimized PN generations are presented in the third column.

The last column shows the size of the generated PNs for the optimized run. As expected, the number of reachable program paths in a function depends on the set of possible function arguments and is reflected in the PN size.

A sequential generation of all segment PNs and the final assembly of the composite PN takes a total of about 45 minutes. When compared with the 18 hours and 17 minutes of the non-optimized PN generation, this is already a runtime reduction by 95%. Moreover, since the PN generation for one segment does not depend on the PN generation of other segments a full parallelization of the generating segment PNs is possible. With parallelization, the wall clock time for PN generation is dominated by the runtime for generating the PN of the most complex segment. In our example, this was segment 8 of FuelSys, which takes about half an hour to compute.

Our experiments show that the PN scalability techniques presented in this section enable PN generation for software programs that were previously too complex to process. With the exception of the recursive RSA algorithm the obtained PN generation runtimes are actually comparable to runtimes that were previously encountered only for much smaller and simpler programs. The new techniques boost the software complexity that can be handled in terms of memory accesses, arithmetic instructions or code size, by orders of magnitude.

4.4 Cone of Influence Computation

In some applications, resilience measures are desirable which do not protect the entire program but only specific functions, such as *safety functions*, or instruction sequences inside a function. For example, a loop counter may be considered more critical than some variable within the loop. In order to ensure the correct execution of these critical instructions, resilience measures only protecting these particular instructions might not be sufficient. Due to the nature of the program's computation a fault activated during the execution of instructions with low criticality might actually propagate to critical instructions. Therefore, we propose to perform an analysis to determine the data dependencies of critical instructions identifying the critical registers along which the relevant data or control information is propagated. This calculation can be done by analysing the PN and provides a precise description of all dependencies. Note that pure machine code would not be sufficient for this analysis since it yields only incomplete CFGs and therefore would lead to an over-approximation of possible dependencies.

Dependency Type	Meaning
Type 0	Data Dependency (Register)
Type 1	Data Dependency (Memory)
Type 2	Control Flow Dependency

Table 4.4: Types of dependencies

As an example, Figure 4.4 shows an excerpt of the results from a dependency analysis. The analysis was performed on the PN of the *Traffic Alert and Collision Avoidance System* (TCAS) developed by Siemens which is part of the Software-artefact Infrastructure Repository [102].

For demonstration of our analysis, an instruction was selected that delivers the value for a variable which is important for the result calculation of the overall algorithm. The instruction is shown as the tail node (with address 1346) at the bottom of Figure 4.4.

The figure shows a part of the dependencies existing for the considered instruction. These dependencies are extracted from all reachable program paths leading to this instruction and are represented by a graph as shown. Each node represents an instruction, its address and information on what registers or memory location the instruction reads from (R) or writes to (W). The annotation "`R: @R1 (0x0000)`", for example, indicates that the particular instruction reads from the memory address 0x0000 stored in register R1. Similarly, "`W: R1`" means that the particular instruction writes to R1.

As mentioned before, only an excerpt of the dependency analysis is shown. Parts which were removed are indicated by a dashed line. Solid lines indicate dependencies and are labelled with a type according to Table 4.4. "Type 0" indicates a direct data dependency where one instruction writes to a register which is used by another. "Type 1" also indicates a data dependency but in this case one that exists through a memory value rather than register content. The last type, "Type 2", indicates that the particular instruction depends on a correctly executed jump or branch instruction.

It is worthwhile noting that the uppermost node (with address 1432) represents a jump instruction which needs the value of a register to calculate the jump target address. Due to the characteristics of the used model all possible target addresses are known so that it is possible to trace in the PN both in forward and backward direction to extract the relevant dependencies.

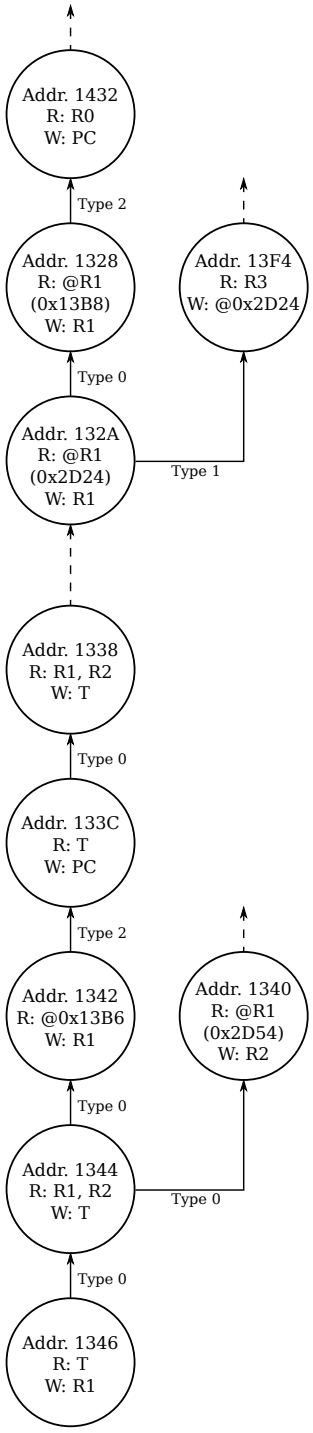


Figure 4.4: Dependency analysis

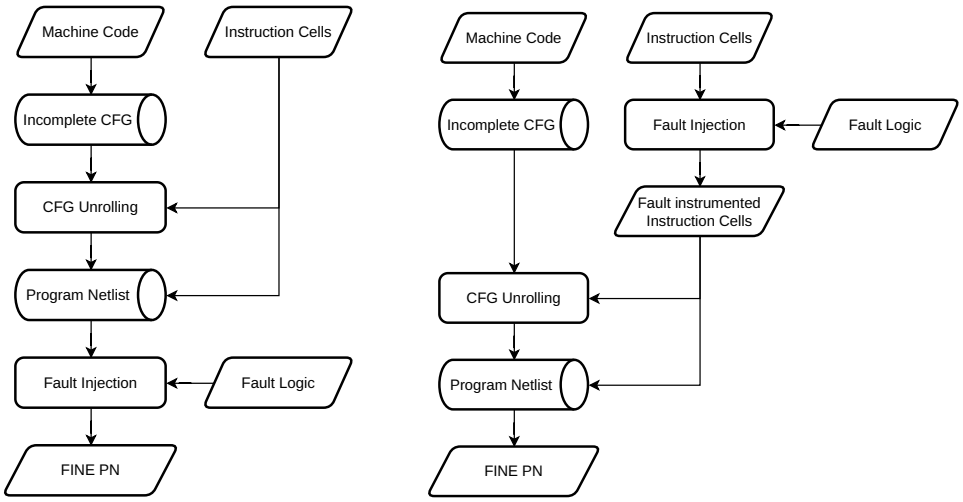
As can be noted, the paths in the dependency graph of Figure 4.4 are not numbered with consecutive instruction addresses. In fact, in our experiments it could be observed that the topology of the computed dependency graph is not identical and not even in a simple relationship with the topology of the program's execution graph. This demonstrates that indeed additional information is obtained from the proposed analysis which may be valuable when designing cost-efficient resilience solutions. Their effectiveness can be certified by proving equivalent behaviour of the protected code segment for a given fault list.

In our experiments we employ this dependency analysis to reduce the complexity of our PN-based analyses. The runtime of a PN-based analysis depends, to a large degree, on the number and the types of instructions modelled in the PN as well as the combinational depth, i.e., the longest path from a primary input to the analysed IC. Removing parts of the PN that are not relevant for a fault analysis proof target resulting from a given fault can significantly decrease the analysis runtime. For this purpose, we use the dependency analysis to perform a *cone-of-influence* (COI) reduction. The COI computation starts at the *head* instruction cell, e.g., a cell that accesses a safety-critical variable, and performs a structural backward trace to find the ICs in the transitive fan-in. The trace stops at the primary inputs of the PN that provide the starting state $s_0 \in \psi_{i,0}$ to the software program. Every IC not in the cone-of-influence, i.e., not in the dependency graph can be removed from the proof instance.

5 FAULT INJECTION IN PROGRAM NETLISTS

Analysing the behaviour of software with respect to hardware faults requires a model of the fault-affected HW/SW system. Every injected fault needs a description for its activation condition including the time at which the fault occurs and how long it lasts as well as a specification of its logical behaviour, i.e., how it affects the execution of an instruction. We model the logical behaviour of a hardware fault by describing its effects on the program state, i.e., how a faulty instruction execution deviates from the correct one. This can be accomplished by modifying the corresponding IC description in an appropriate way, as we explain in the next sections.

In order to model the temporal behaviour of a fault, we do not use clock cycle accuracy but a more abstract notion of time in order to handle larger processors with unpredictable execution times. Abstraction is performed in such a way that time is represented by the order of the instructions in the program. Every instruction cell inside a PN represents a unique abstract time point. An abstract time point in a PN represents one or more concrete time points that depend on the execution path taken by the program. A fault changing the PS that is read by an IC, therefore, models several faults that change the PS in the same way and appear at concrete time points before the execution of the corresponding instruction in the processor core.



(a) Type A(fter) (b) Type B(efore)

Figure 5.1: FINE PN generation flow

The proposed methods, however, are easily adaptable for time-accurate ICs that can be created for processors with predictable execution times. In [108] a

time-accurate version of an IC called *timed instruction cell* was introduced for this purpose.

We model faulty behaviour of a HW/SW system with *fault-injected PNs* (FINE PNs) by instrumenting fault-free ICs with fault logic. We distinguish between two FINE PN generation processes, depending on when the IC instrumentation is applied. If the fault injection is performed after the PN generation (cf. Figure 5.1a), we denote the resulting PN as *type A FINE PN* or *A-FINE PN*. If the fault injection is performed before the PN generation (cf. Figure 5.1b), the result is a *type B FINE PN* or *B-FINE PN*. Figure 5.1 show the FINE PN generation flow for both generation types.

The fully automated instrumentation and unrolling processes are the same for both types. The only difference is that, during the generation of a B-FINE PN, the fault logic inside the instrumented ICs allow the activation of program paths that are unreachable in the fault-free case. Every fault effect is, therefore, modelled inside the B-FINE PN, including not only modified data but also modified write and branch addresses. However, this incurs high complexity and runtime in the PN generation process and the subsequent analysis, especially if indirect branches (using an address register) are involved, in which case the control flow may take an arbitrary path and likely leads to a system failure. With a small modification, though, the PN generation becomes significantly less complex, at the price of a less accurate model of control flow errors. The idea is to generate the PN without injected faults at first and then add the fault injection logic to the fault-free PN in a separate step. As a consequence, the fault-injected PN, i.e., the A-FINE PN, does not include program paths which can only be reached as a result of a fault. In Sec. 8.2.1 it will be shown how A-FINE PN, in spite of this inaccuracy, can still be used for a conservative, i.e., formally sound, fault injection technique. It is the task of the safety engineer to select the appropriate FINE PN depending on the complexity of the software to be analysed. In our experimental evaluation we rarely observed cases where a fault changes the control flow but does not change the program's behaviour.

In general, a PN-based fault analysis provides a large degree of freedom in choosing different fault models. In this thesis we focus on stuck-at faults representing hard errors and SEUs representing soft errors. We discuss fault injection for both types of faults in Sections 5.1 and 5.2. Extensions to other fault models are possible.

5.1 Stuck-At Faults

The stuck-at fault is a widely used fault model to describe faults that originate from permanent defects in the physical structure of a circuit. A stuck-at fault inside the hardware of a processor core can have an effect on multiple processor instructions. The effects of faults belonging to this fault class can be modelled by creating a corresponding fault description for every affected instruction cell. Correct modelling of multiple faults and their effects on different instructions is more challenging. We achieve this, as elaborated below, by

adding auxiliary constants, registers and ports to the fault description of an IC.

```
1 fault_injection_1_preliminary(const Rn, in PS, out PS')
2 {
3    PS'.Activation_Condition += 1;
4    if(PS.Activation_Condition.bit(LSB) == 0)
5    {
6       PS'.RegisterFile[Rn].bit(MSB) =
7             PS.Fault_Register.bit(MSB);
8    }
9 }
```

Figure 5.2: Modelling stuck-at faults – example

The example shown in Figure 5.2 illustrates a very simple case of a stuck-at fault description that can be integrated into the description of an IC. The fault-injected IC could, then, be considered a mutation. However, if the injected fault logic prepends or appends the IC logic and modifies only the program state the fault logic can also be considered a saboteur that resides inside an IC. The stuck-at fault modelled in the example of Figure 5.2 is activated on every second execution of the injected instruction and affects the most significant bit of program-visible register *Rn*. This may be used to model, e.g., a situation where only one out of two adders in a superscalar pipeline is affected by the stuck-at fault. For this purpose the architectural state was augmented by two auxiliary registers: *Activation_Condition* and *Fault_Register*. The former has an initialization value of zero while the latter is left uninitialized. The *Activation_Condition* register is incremented every time the affected instruction is executed, and the fault becomes active whenever the least significant bit of the *Activation_Condition* register is zero, i.e., on every second incrementation of *Activation_Condition*. Then, the MSB of the target register is assigned the value of the MSB of the unspecified register *Fault_Register*. In effect, the MSB of the target register is treated like an open input in our formal analysis. This way, both faults, stuck-at-0 and stuck-at-1, can be considered at the same time.

Note that it is possible to describe more than one fault for a particular instruction. A fault description with more than one fault can serve two purposes. It can be used to model multiple faults and to examine their combined effect on the program. The second purpose is to model several faults (single or multiple) in the same program netlist, thus avoiding the effort of re-generating the program netlist for every fault to be examined.

In order to support such complex fault descriptions for fault lists with a large number of faults and to separate the activation and deactivation of faults from the computation of the internal processor state, the occurrence of a fault given in the fault list can be encoded into the data of an auxiliary memory at a specific location addressed through auxiliary ports. These auxiliary ports do

not correspond to variables of the original software but are only used in our computational model to gain better control on the activation conditions.

During fault analysis the values of these ports are set appropriately so that the activation of specific faults and combinations of faults can be enabled or disabled. In fact, using this construction, it is sufficient to generate a single FINE PN to analyse the effects of several single faults and/or several multiple faults together with the original fault-free behaviour.

```
1  fault_injection_1(const Rn, in PS, out PS', Fault_Port Port)
2  {
3     Port.Address = 0xABCD;
4     PS'.Activation_Condition += 1;
5     if((PS.Activation_Condition.bit(LSB) == 1) &&
6        (Port.Data == 1))
7     {
8        PS'.RegisterFile[Rn].bit(MSB) =
9                PS.Fault_Register.bit(MSB);
10    }
11 }
```

Figure 5.3: Modelling stuck-at faults – allowing for several faults in a single model

In Figure 5.3 the code of Figure 5.2 was modified such that a memory access to a configuration variable at address `0xABCD` was added to the fault description. The configuration variable is an auxiliary construct that allows to enable or disable the activation of a specific fault.

5.1.1 Insertion of Fault Injection Logic

Fault injection is provided by inserting a fault description at the start or at the end of the corresponding IC. Actual injection of a fault is later controlled by setting the corresponding permission bits in the auxiliary configuration variables for the fault (cf. Figure 5.3).

An injected fault changes the behaviour of the original instruction cell by either performing additional changes of the program state or by overwriting changes of the fault-free part.

The example in Figure 5.4 shows an instruction cell providing several possible fault injections to a SuperH2 [86] ADD instruction. As described in [90], the PN model generation steps are interleaved with a SAT-based analysis to prune the control space of the program. When ICs are instrumented with fault logic before the PN is generated this analysis is extended to the instruction cells with their fault descriptions so that all possible fault scenarios are included in the FINE PN.

Note that in this case the FINE PN models all fault behaviours of the fault list. This makes it possible to perform a global reasoning over all faults or sets of faults. For example, one could compute the set of all faults that lead the program into a specific program state.

```
1  ADD(const Rm, const Rn, in PS, out PS', Fault_Port Port)
2  {
3    ProgramState PS_temp = PS;
4
5    PS_temp.RegisterFile[Rn] =
6      PS.RegisterFile[Rn] + PS.RegisterFile[Rm];
7
8    fault_injection_1(Rn,     PS_temp, PS_temp, Port);
9    fault_injection_2(Rm, Rn, PS_temp, PS_temp, Port);
10   fault_injection_3(Rm, Rn, PS_temp, PS_temp);
11   ...
12
13   PS' = PS_temp;
14 }
```

Figure 5.4: Instruction cell with fault injection

Obviously, a B-FINE PN modelling a large number of possible faults may turn out to be more complex than the corresponding PN for the fault-free case or the B-FINE PN for only a small subset of these faults. Depending on the complexity of the model it may therefore be advisable to partition the fault list and to analyse each partition in a separate B-FINE PN, or to perform the analysis based on an A-FINE PN.

5.2 SEU Faults

Large memory components like RAM blocks can be efficiently protected against SEUs by conventional fault resilience techniques such as *error correction codes*. Therefore, we do not consider faults in the main memory. Instead, we analyse the effects of SEUs that occur in or propagate to program-visible registers.

```
1  Inject_SEU(const Rs, const Time_ID, in PS, out PS_temp)
2  {
3    Activation_Condition =
4      (PS.Time_ID == Time_ID) && PS.Register_Selection[Rs];
5    Bitmask =
6      '32(signed(Activation_Condition)) & PS.Bit_Selection;
7
8    PS_temp.RegisterFile[Rs] =
9      PS.RegisterFile[Rs] ^ Bitmask;
10 }
```

Figure 5.5: Modelling SEU faults

The change in the program state, due to an SEU occurring in a register bit, is modelled by adding fault logic to the ICs as shown with stuck-at faults in

Section 5.1.1. The example in Figure 5.5 shows the logic for an SEU-injecting saboteur. In this example, the condition computed in *Activation_Condition* has to be fulfilled before changes are applied to the register *Rs*. In order to pass this information to the logic, we add the registers *Time_ID*, *Register_Selection* and *Bit_Selection* to the program state. *Time_ID* is a unique ID for each instruction in the PN modelling an abstract time point and used to enable fault activation only for a specific instruction cell inside the PN. Note, that the same machine instruction can appear in multiple instantiations as instruction cell in a PN. If an ID is selected that does not appear in a particular PN, then, no fault is allowed to become active and the PN models the fault-free behaviour. *Register_Selection* and *Bit_Selection* are used to select the register and bit position for fault injection. If the activation condition is checked, *Activation_Condition* is either 0 or 1 so that a sign extension will generate a bit vector where all bits are either 0 or 1, respectively. If the condition is not fulfilled, then all bits in *Bitmask* are 0 and, therefore, do not change the program state. If the condition is fulfilled, then *Bit_Selection* determines which bits in *Bitmask* are 0 and which are 1. Every bit position in register *Rs* is flipped when there is a 1 at the corresponding position in *Bitmask*, so that individual bits can be selected during fault analysis.

```
1  ADD_SEU(const Rd, const Rs1, const Rs2, const Time_ID,
2          in PS, out PS')
3  {
4    ProgramState PS_temp = PS;
5
6    Inject_SEU(Rs1, Time_ID, PS_temp, PS_temp);
7    Inject_SEU(Rs2, Time_ID, PS_temp, PS_temp);
8
9    PS' = PS_temp;
10   PS'.RegisterFile[Rd] =
11     PS_temp.RegisterFile[Rs1] + PS_temp.RegisterFile[Rs2];
12 }
```

Figure 5.6: Instruction cell with SEU fault injection

A comparison between the fault injection logic for a single register in Figure 5.5 and the IC logic in Figure 4.3 shows that every fault injection adds some amount of complexity to the PN comparable to that of an instruction. This may seem like a strong impact on the overall complexity of the fault analysis. Fortunately, this is not the case because fault injection logic only needs to be added to those registers that are read by the particular IC. Note that this optimization exploits the basic property of the PN model that under any program input only one path in the PN can be activated [90]. Therefore, a propagating fault can affect only one program path which prevents it from interacting with itself in case of path branches and path merges in the PN. The fault only becomes effective when it reaches the first IC in which the affected

register is read and processed. The injected and analysed fault represents a class of equivalent faults (cf. Section 2.6.2).

The IC in Figure 5.6 shows an example how faults can be injected in the ADD-IC from Figure 4.3. This instruction reads only from registers *Rs1* and *Rs2*. This is why only in these registers faults are injected. Here, *Time_ID* models the fault equivalence class.

According to the RISC-V ISA [109] description register 0 is special as it always contains the value 0. For the fault effect analysis we assume that this register is still a valid fault target as it could possibly be implemented as register that is initialised during the reset sequence.

6 ISA-LEVEL FAULT EFFECT ANALYSIS

We use FINE PNs to perform our *fault effect analysis* (FEA) where we formally analyse the effects of hardware faults on the software behaviour including program states, I/O sequences and the control flow. Note that all registers modelled in the PN are program-visible registers of the design. A FINE PN-based FEA, therefore, either finds the effects of ISA-level hardware faults on the ISA-level software behaviour or formally proves the absence of any effects.

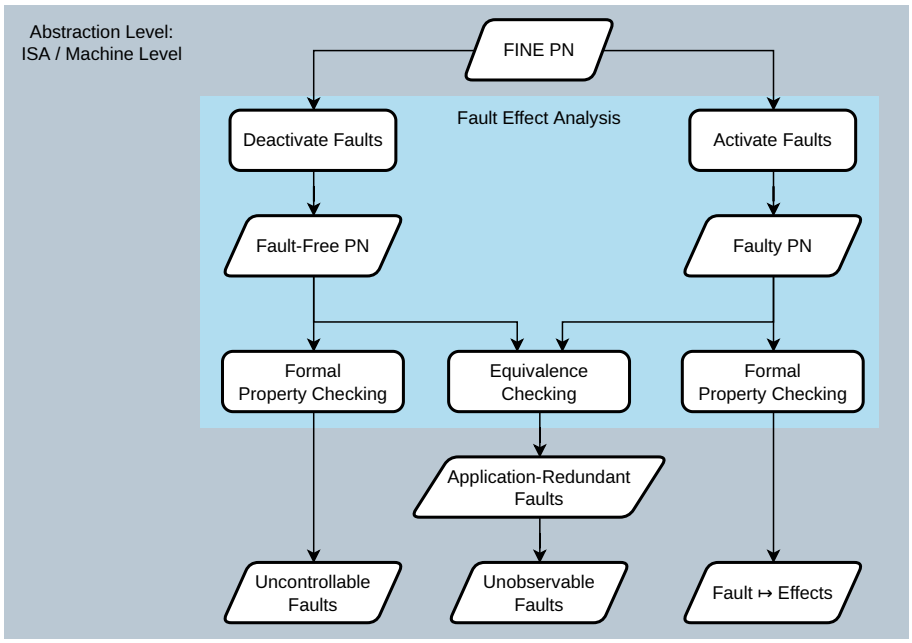


Figure 6.1: ISA fault effect analysis flow

We use FEA to achieve three different analysis goals:

1. Detect ISA-level register bits that cannot be controlled by the software.
2. Find ISA-level faults whose effects cannot be observed outside of the processor.
3. Identify the set of ISA-level faults that can have a specific effect on the software behaviour as specified by the safety engineer.

All three goals are shown on the bottom of Figure 6.1. The rest of Figure 6.1 shows the analysis flow of FEA for achieving these goals. Independent of the analysis goal FEA always starts with the generation of a FINE PN (cf. Chapter 5).

In the rest of this section we describe the flow depicted in Figure 6.1 for each analysis goal.

The preferred approach to analyse whether a fault has a certain effect (the right-most goal in Figure 6.1) is to formulate the effect that should be analysed, e.g, that even in the presence of faults store instructions never access certain memory areas, as a property of the FINE PN and prove it using formal property checking. The result of this analysis is a relation between the complete set of enabled faults and the checked effect; information about individual fault effects of the selected fault set is not preserved. The safety engineer can adjust the granularity of the analysis by performing multiple analysis runs and enabling only certain faults in each run, e.g., faults affecting a specific program-visible register. We follow this approach in Chapter 8 to identify all ISA-level faults that can corrupt a specific set of variables in the PS.

When the goal is to to find unobservable faults we propose to perform equivalence checking by comparing a fault-injected PN with its fault-free counterpart as depicted in the centre of Figure 6.1. The advantage of this approach is that it can benefit from sophisticated optimizations used in standard hardware equivalence checking. For this purpose, we duplicate the generated FINE PN, disable all faults in one of the instances (left-hand side in Figure 6.1) and enable some or all faults in the other (left-hand side in Figure 6.1). The FINE PN with all faults disabled, then, models the fault-free behaviour of the software program while the other FINE PN models the behaviour for the enabled fault set. Figure 6.2 shows the comparison between two example PNs by using "miters". A method for checking the equivalence of two structurally different PNs was already proposed in [107].

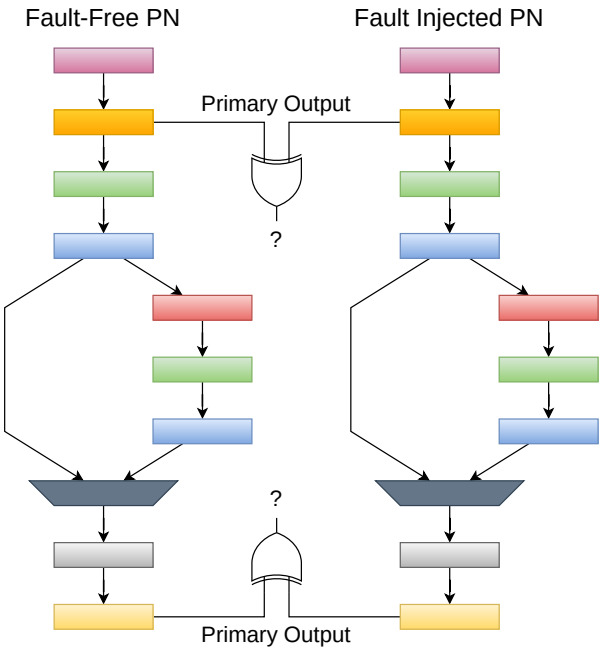


Figure 6.2: Comparing two PNs by using "miter"

We define two programs to be equivalent iff they produce the same output sequence for any applicable input sequence. Programs with enabled faults represent new programs different from the original. The safety engineer can adjust the granularity of the equivalence check by enabling only certain faults for an individual analysis run.

There are two possible outcomes of the equivalence check. The first one is that the PNs are equivalent, i.e., the corresponding programs produce the same I/O sequences. In this case, the considered fault has no effect on the program behaviour regardless of what values the inputs have. We denote such faults as *application-redundant*, or, like in [9], *application-dependent safe fault*. In the second case the PNs are not equivalent. This means that they differ in either data or address of one or more I/O accesses, in the number of I/O accesses, their order or any combination thereof. In such cases, a subsequent analysis may be used to categorize the error. For example, a simple structural analysis of the two PNs can yield the information on whether the considered fault can affect only data or may also modify the control flow of the program. If the software program modelled by the considered PN is merely a component of a larger software system a subsequent analysis, as presented in Chapter 8, can yield the information how fault effects can propagate further in the system.

Also formal property checking on the fault-free PN, as shown on the left-hand side in Figure 6.1, can be useful to gain valuable testability information about the system. In particular, it is often of relevance to determine what state bits of the system assume a constant value under all possible program runs. Similarly like in linting tools for hardware a set of assertions can be generated that checks for each architectural state variable in the PN whether or not it is constant. In the context of testing, the constant variables represent non-controllable fault locations. This can be useful in a subsequent analysis to explore gate-level testability of faults, as described in Chapter 7.

6.1 Uncontrollability & Unobservability in PNs

The signals in a combinational netlist can be uncontrollable or unobservable. Just like in the classical ATPG problem (cf. Section 2.8) the bits of a PS instance in a PN can be uncontrollable, i.e., their values are the same for every value assigned to the PN's primary inputs. If a bit is uncontrollable in all PS instances of a PN, then it is also uncontrollable in the corresponding program-visible registers of the processor. Likewise, faults injected into a PN can be found unobservable, i.e., there exists no value assignment for the PN's primary inputs such that the fault affects the PN's I/O behaviour. In Chapter 7 we exploit the presence of unobservable faults and uncontrollable bits in the architectural state to find more untestable faults at the gate level.

6.2 Experiments

A first set of experiments was conducted to evaluate the potential of FEA by analysing to what extent fault injection affects the runtime of the PN genera-

tion as well as the size of the resulting model. The experiments were conducted with two different programs:

- LIN: a driver for a slave node of a *Local Interconnect Bus*, used as a gateway to external buses; obtained from Infineon, proprietary IP.
- TCAS: a software-implemented traffic alert and collision avoidance system, developed by Siemens; obtained from the Software-artefact Infrastructure Repository [102].

For each considered software program two PNs were generated for the SuperH2 [86] ISA. The first PN, referred to as fault-free, was generated without fault injection logic, while in the second run a B-FINE PN was generated for a fault list consisting of stuck-at faults by using fault injection logic, as presented in Section 5.1. The list of faults that were provided for in the injection logic comprised all stuck-at-0 and stuck-at-1 faults at the individual bits of the program-visible data registers. In the SuperH2 ISA 552 such bits were identified. At the selected bits all single faults as well as all combinations of multiple faults are implicitly represented by the fault injection mechanism. In the generated FINE PN, all faults of the different fault types can be analysed either independently as single faults or in arbitrary combinations as multiple faults without repeating the model generation.

We used *gcc* to compile the software programs. All experiments were performed on an Intel i7-4790 CPU at 3.6 GHz with 16 GB RAM. The timing measurements were performed using the profiling tool *gprof*[1]. We also used timing reports on individual proof instances of the applied commercial tool [79].

Program	CPU time (in s)	
	Fault-Free	Faults Injected
TCAS	7.56	147.48
LIN	63.66	1,081.50

Table 6.1: CPU times for PN model generation (SuperH2)

Table 6.1 shows the time needed to generate the PNs. It can be observed that inclusion of fault injection logic has a significant effect on the runtime of the PN generation process. However, when taking into account that a large number of different single bit faults and an astronomical number of multiple faults are modelled in the FINE PN the increase in runtime seems acceptable. Also note that this model needs to be generated only once and can be re-used for all subsequent fault injection experiments. Moreover, in practice, it will usually not be desired to conduct a full formal analysis for all theoretically possible faults, as they are considered in the above experiments. Instead, simulation-based approaches may first be used to boil down the fault list to a smaller set of faults for which a detailed formal analysis is of interest, as outlined in Chapter 3.

1 GNU profiler

Alternatives to ensure scalability are the decomposition of the fault list into smaller parts and to generate a B-FINE PN individually for each part, the generation of an A-FINE PN if program paths that are unreachable in the fault-free case do not need to be considered, or the decomposition of the software program into smaller components and to perform a FEA individually on each component, as done in Section 8.

Program	# Instructions in PN	
	Fault-Free	Faults Injected
TCAS	656	771
LIN	1,862	3,313

Table 6.2: Number of instructions in PN models (SuperH2)

Table 6.2 shows how many instructions each generated PN contains. It can be observed that the fault-injected PN generated for the TCAS program is 115 instructions or 18% larger than the fault-free PN. Obviously, the faults that have been injected, in most cases, do not have an effect on the control flow of the program. For the LIN program the fault-injected PN is by 1451 instructions (78%) larger than the original version. As a result of modelling possible fault locations the program apparently was able to take program paths which were previously unreachable, adding many new instructions to the PN.

In the next experiment, we ignored scenarios with multiple faults and restricted our analysis to find out which of the injected single faults are application-redundant, i.e., do not have any effect on the possible program behaviours. The analysis was done by performing equivalence checks, as described in the first section of this chapter, using the commercial tool OneSpin 360 DV [79].

Taking into account the entire I/O behaviour of the system allows us to check application redundancy for all injected faults. CPU times for conducting the required equivalence checks between the fault-free PN and the FINE PN were nearly the same for all faults in both designs. In order to solve the SAT problem, the commercial tool [79] reported CPU time requirements of 1.73 seconds on average in case of TCAS and 5.13 seconds on average in case of LIN, per fault. For comparison, we conducted the same analysis with a A-FINE PN yielding in time requirements of 0.0 seconds on average for TCAS and 2.9 seconds on average for LIN. Note that these experiments can be easily parallelized in a multi-processor computing environment. We were able to prove application redundancy for 896 faults in the case of TCAS and 544 faults in the case of LIN.

As outlined at the start of this chapter, we proceeded to identify constant values in the program-visible registers. These constants identify non-controllable fault locations. Table 6.3 shows for each examined program the number of testable faults, the number of untestable faults in total, of those the number of non-controllable faults and the number of unobservable faults. Unobservable faults are identified by proving the untestability of both the single stuck-at-1 and stuck-at-0 fault at the respective fault location.

Program	ISA-level faults			
	testable	untestable	uncontrollable	unobservable
TCAS	208	896	490	896
LIN	560	544	443	512

Table 6.3: FEA: untestable faults (SuperH2)

Interestingly, not a single fault in the LIN driver and in the TCAS software could be identified that is untestable although it is separately controllable (when not observable) or observable (when not controllable). This strongly supports our argumentation in Chapter 7 that ISA-level observability and ISA-level controllability can be considered separately in the proposed gate-level testability analysis.

Obviously, a large amount of application-redundant faults could be identified in both case studies. This clearly shows that the effect of hardware faults on the software behaviour indeed varies widely and supports the original motivation of this thesis, as described in Section 1.3.

Beyond application redundancy also other fault scenarios of interest to the user can be explored. For example, for certain faults of the LIN bus we could observe that the LIN node was virtually disconnected from the bus.

6.2.1 Fault Effect Analysis - RISC-V Results

In the previous part of Section 6.2, we compiled two software programs for the SuperH2 ISA, created a B-FINE PN for each of them and applied FEA on each FINE PN to identify untestable ISA-level faults. In this section we present the results of a similar experimental setup where we compiled the software program TCAS for the RISC-V ISA [109], created an A-FINE PN for this program and applied FEA on the FINE PN. The other parameters of the experimental setup, like compiler and computer on which the analysis was run, are identical to the previous section. Like in Section 6.2, the fault list for the injection logic comprised all stuck-at-0 and stuck-at-1 faults at the individual bits of the program-visible data registers. For the RISC-V ISA we identified 1024 such bits.

Program	CPU time (in s)	# Instruction in PN
TCAS	2	237

Table 6.4: CPU times for PN generation and model size (RISC-V)

Table 6.4 shows the runtime for the model generation and the number of instruction cells in the generated A-FINE PN. These numbers are the same as for the fault-free PN, because in order to obtain an A-FINE PN a fault-free PN is generated first and then faults are injected into the fault-free PN. The time required to perform the fault injection, i.e., replacing the logic of the ICs, is insignificant compared to the time for PN generation. Furthermore, the fault

injection after PN generation does not increase the number of instructions in the PN.

Like in Section 6.2, in the first analysis we identified program-visible register values that cannot be controlled by TCAS in the fault-free case. The corresponding non-controllable faults are presented in column 3 of Table 6.5. In the second analysis, we allowed only single stuck-at-1 and single stuck-at-0 faults to be active and analysed whether the can affect the program output. The number of faults that do not have an effect are shown in column 4 of Table 6.5. The first two columns of this table show how many faults are testable and how many are untestable when both uncontrollable and unobservable faults are considered. Overall 65% of all ISA-level stuck-at faults are untestable. When compared with the results obtained from SuperH2 and B-FINE PN it shows that an A-FINE PN-based analysis on other ISAs can deliver similar results.

Program	ISA-level faults			
	testable	untestable	uncontrollable	unobservable
TCAS	708	1,340	1,285	1,154

Table 6.5: FEA: untestable faults (RISC-V)

An interesting result of FEA for the TCAS program is that for register 16 only a stuck-at fault in the most significant bit (MSB) is observable. A further analysis revealed that the value in the MSB decides which branch the program takes and that a single stuck-at fault anywhere else in the register cannot affect the branch decision. FEA also revealed that a stuck-at-0 in the least significant bit (LSB) of either register one or register five is not observable. The reason for this is that these registers are only used to compute jump target addresses which always have a zero LSB, because the start address of every instruction is memory-aligned, i.e., always a multiple of 2.

6.2.2 Dependency Analysis

We further demonstrate the use of the dependency analysis described in Section 4.4 by means of the LIN driver, compiled for the SuperH2 ISA. For a brief illustration of the dependency analysis (at the example of TCAS), please refer to Section 4.4.

We selected a specific code segment in the LIN driver that we assumed to be safety-critical depending on the LIN bus environment.

Table 6.6 shows the results. The first entry shows the number of nodes in the generated COI graph, as described in Section 4.4, for the selected code segment. By a structural trace in the COI graph we identified the state bits that potentially have an impact on the safety-critical code segment (second entry). Note that this number is significantly smaller than the total number of state bits (552) in the program-visible registers of the hardware platform.

After identifying these "syntactically critical" registers fault injection was performed at these registers in the same way as in the previous section and the testability of the faults was determined with respect to observability in

Nodes in cone of influence	259
Syntactically critical state bits	289
Semantically critical state bits	200
Total no. of safety-function-redundant state bits	352
Total no. program-visible state bits in SuperH2	552

Table 6.6: ISA-level dependency analysis – LIN driver

the safety-critical code segment. This was done by adapting the equivalence check discussed in the beginning of this chapter to compare only the variables of the selected code segment. It turned out that some faults are "redundant" w.r.t. the critical functionality. Only if at least one of the two stuck-at faults at a syntactically critical state bit is testable then the state bit is also "semantically critical" (third entry).

In other words, if we assume that the critical code segment is the only safety function, as by ISO 26262, of the system, it is certified that the system is safe w.r.t. to all stuck-at faults (or bit-flips) occurring at register bits that are not semantically critical. The total number of these "safety-function-redundant" state bits, i.e., state bits which are not contained in the COI graph, or state bits which are only syntactically but not semantically critical, is shown as fourth entry in Table 6.6.

The fact that not all syntactically critical state bits are actually semantically critical can be used to prune the COI graph by removing all untestable registers and related dependencies. This can lead to a more compact model which is beneficial, for example, in a manual analysis of the COI graph for diagnosis purposes, or when taking design measures to improve the fault resilience of the critical code segment.

7 ISA/GATE CROSS-LEVEL FAULT ANALYSIS

In this chapter we answer the question how the results obtained from FEA at the ISA level can be used to evaluate faults in all other parts of the computing hardware w.r.t. their testability under the given software program. The enabling idea is to exploit that program-visible registers create a direct link to the gate-level, i.e., all faults modelled in the PN registers are also modelled in the corresponding gate-level registers.

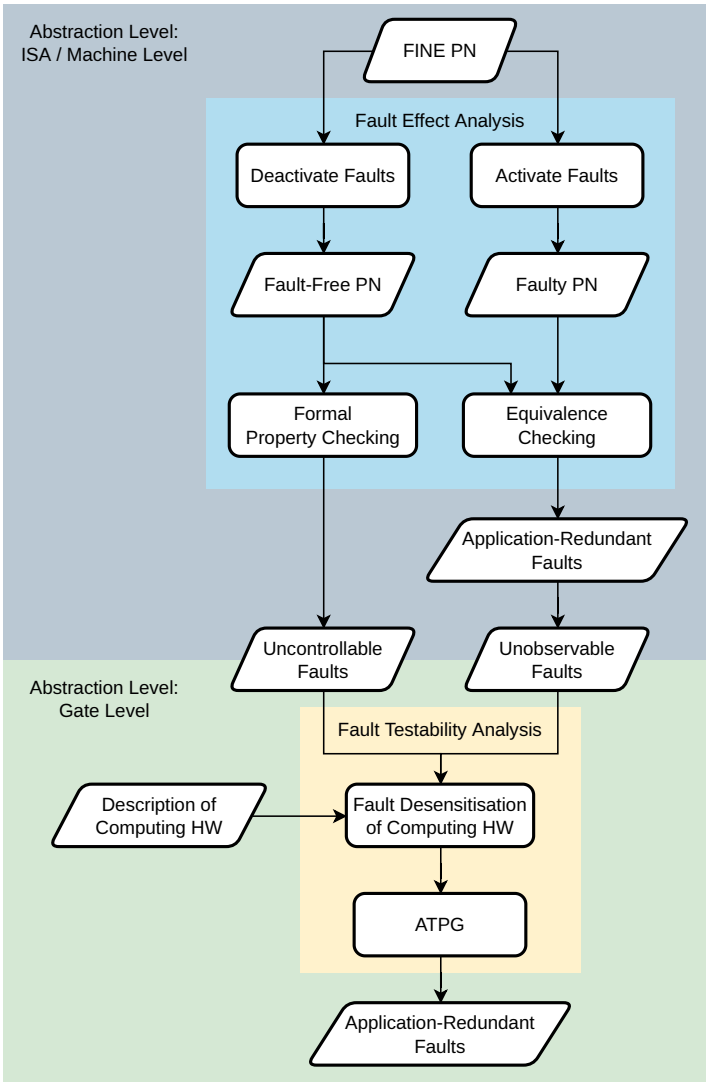


Figure 7.1: ISA/gate cross-level fault analysis flow

Figure 7.1 illustrates the steps of our ISA/gate cross-level fault analysis. In the first step, we model the effect of hardware faults on the software by injecting stuck-at faults into the bit-level variables of the architectural state of the PN, as explained in Chapter 5. In the second step we then analyse these faults w.r.t. their testability (i.e., controllability and observability) under all possible runs of the software. We described this in detail in Chapter 6.

In the third step, which is the topic of this chapter, we use the ISA-level testability information obtained from FEA to determine the testability of hardware faults anywhere in the combinational logic under the given software program.

We take the testability of faults in the architectural state bits determined by FEA, transfer this information into the hardware domain and perform a *fault testability analysis* (FTEA) at gate-level. FTEA determines a set of faults in the gate-level circuitry of the computing platform that are untestable because their testability depends on the testability of architectural state bits.

This is, in fact, a *sequential ATPG* problem. Before showing how our approach works, we first discuss how it can, in principle, be formulated as a classical gate-level sequential ATPG problem. The circuit under test consists of (at least) the processor, the main memory, the CPU bus and the bus interfaces of the I/O devices accessed by the program. The main memory holds instructions and data of the software. We define the primary inputs and the primary outputs of the circuit to be the contents of memory locations and I/O device registers at certain addresses. The evaluation of these inputs and outputs may be restricted to certain time points of execution. The problem we are considering is to find, for each considered fault in the logic circuitry of the computing platform, either a *test sequence* or the proof that no test sequence exists. A test sequence is a sequence of program inputs such that a fault effect propagates to one of the program outputs. If no test sequence exists, then the fault is application-redundant, i.e., untestable in the system running the considered software. In principle, a standard sequential ATPG tool could be applied to this problem.

Classical ATPG algorithms employ unrolling of the circuitry similar to the one shown in the middle part of Figure 4.1 to find test sequences reachable from the initial state. However, as already elaborated in Section 4, such an approach is clearly intractable for the problem instances considered in this thesis. The unrolling would have to span a potentially very large number of clock cycles representing the execution of many instructions of the program. (Out of curiosity, we actually conducted this experiment using a commercial ATPG tool. As expected, the tool was overwhelmed with the complexity of the problem and aborted with time-out.)

In contrast, the approach presented in this thesis is, in fact, computationally feasible for low-level software programs of realistic size. It exploits the higher abstraction level of program netlists to determine the sequential testability information for faults in program-visible state variables and then transfers this information to the gate level.

Figure 7.2 shows the general approach based on the canonical model of a sequential circuit as a *finite state machine* (FSM) with inputs, outputs and bit-level

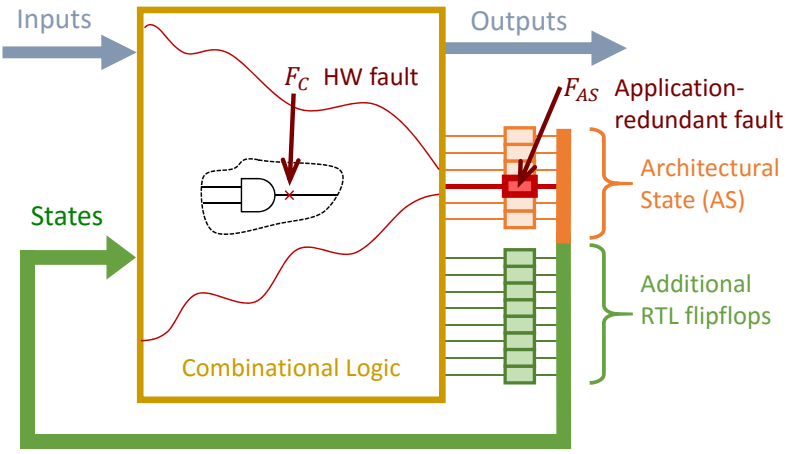


Figure 7.2: Hardware fault affecting software behaviour

state variables. A subset of the state variables belongs to the program-visible registers that are modelled also in the PN as architectural state (AS). They are marked orange in Figure 7.2. Other state variables, like internal buffers, pipeline registers, reservation stations, etc., exist only in the RTL. They do not have a direct mapping to state variables of the software model, however, they may influence these variables indirectly because the software-visible state variables are in their transitive fanout. They are marked green in Figure 7.2. Let us now assume that FEA found a stuck-at fault, F_{AS}, in some AS register bit to be application-redundant, i.e., it is not testable under all runs of the application. This means that there exists no sequence of inputs to the program such that the fault can be activated and observed at the same time. This has consequences on other fault locations in the hardware. Consider a stuck-at fault, F_{C}, in the combinational logic of the design (cf. Figure 7.2). While this fault may be testable in general, it may be redundant under the designated software application if its testability depends on the testability of the fault F_{AS}.

In the following we consider the combinational stuck-at test problem and treat flip-flops as pseudo-inputs and pseudo-outputs of the combinational logic. (Considering combinational rather than sequential ATPG greatly reduces the complexity of the analysis. Nevertheless, as will be shown below, we "import" the sequential testability information from FEA so that the computed results are sequential redundancies.) An input pattern is a set of binary value assignments to the inputs and pseudo-inputs. Let $C_{\mathrm{C}}(x)$ be the characteristic function of all input patterns x that activate the fault F_{C}, and let $O_{\mathrm{C}}(x)$ be the characteristic function of the input patterns that make the fault F_{C} observable at any one of the primary outputs or pseudo-outputs. $C_{\mathrm{C}}(x)$ represents the controllability of the fault F_{C}; it is equal to the logic function of the transitive fanin of F_{C} if the fault is stuck-at-0, and to its inverse if the fault is stuck-at-1. The conjunction $C_{\mathrm{C}}(x) \cdot O_{\mathrm{C}}(x)$ represents the characteristic function of all *test patterns* of F_{C}.

Likewise, $C_{\mathrm{AS}}(x)$ and $O_{\mathrm{AS}}(x)$ represent the controllability and observability patterns of the fault F_{AS}. Since we consider only the combinational circuitry

and the fault F_{AS} is located at a pseudo-output, the fault is observable at the gate-level under *all* input patterns and it is $O_{AS}(x) = 1$.

With respect to the sequential ATPG problem discussed above, the characteristic functions $C_{AS}(x)$ and $O_{AS}(x)$ represent conservative *over-approximations* of the respective sets of input patterns for controlling and observing the fault. This is because not all combinations of value assignments are possible at the pseudo-inputs to the combinational logic because, in the sequential circuit, the flip-flops hold the state of the processing hardware during software execution and the possible state vectors depend on the program state.

How can we now transfer testability information about architectural state bits into the gate-level ATPG problem? In FEA we inject the stuck-at fault F_{AS}, as described in Section 5.1, into all instruction cells that process the corresponding architectural state bit. Suppose the analysis determines that F_{AS} has no effect on the behaviour of the software. This means that for all input patterns x such that $C_{AS}(x) = 1$ the fault F_{AS} is not observable.

At the gate-level, this has consequences for the internal hardware fault F_C. It becomes untestable if it can only be observed through the location of the fault F_{AS} and if every test vector for F_C activates the fault F_{AS}, i.e., $C_C(x) \cdot O_C(x) \Rightarrow C_{AS}(x)$. This is a testability constraint that can be added to the ATPG problem for the faults in the combinational circuitry.

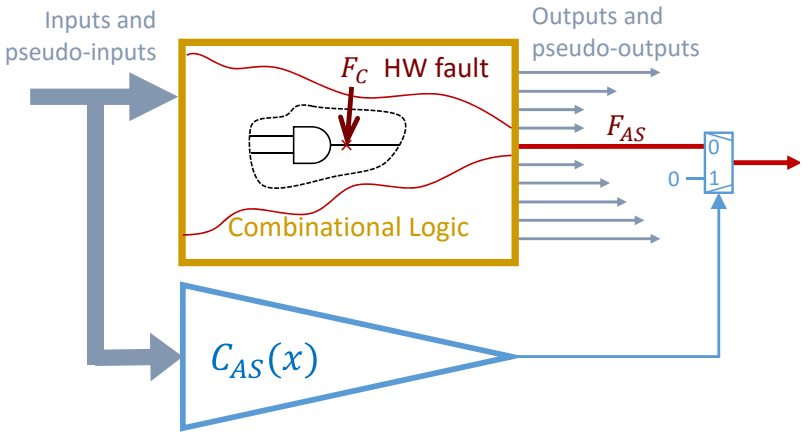


Figure 7.3: Auxiliary construction for ATPG

Figure 7.3 shows an auxiliary construction that allows us to easily add this constraint so that we can use a standard combinational ATPG tool for finding untestable faults anywhere in the combinational logic. We add a logic block computing the controllability condition, $C_{AS}(x)$, of the fault F_{AS}. This can be implemented by duplicating the transitive fanin logic of the fault site, and adding an inverter in case the fault is stuck-at-1. This logic block drives a multiplexer that routes the original signal to the output if $C_{AS}(x) = 0$, allowing any error signal to propagate and become visible at the output. In case $C_{AS}(x) = 1$, however, error propagation is blocked and a constant value appears at the output. This exactly models the testability constraint discussed above.

The auxiliary construction shown in Figure 7.3 may potentially lead to long ATPG runtimes because of the large reconvergent fanout introduced. Fortu-

nately, it can be simplified substantially. Note that a program netlist represents the possible program runs in a HW/SW system and therefore is of special nature. By construction of our model, always only one path in the generated execution graph can be activated at a time [90] by assertion of the corresponding *active signals* (cf. Section 4.1). The resulting program netlist, although containing a possibly high degree of reconvergent fanout, will only propagate signals along a single path that represents a program run. In a well-written software running on a reasonable hardware architecture, it will occur only in rare cases that a program input *necessary* to activate an architectural state bit destroys its observability along the same path. The intuition behind this argument is that if software computes a certain value it most likely also uses it (i.e., reads it) some time later. Therefore, untestable stuck-faults at the application level, as determined by FEA, are usually unobservable independently of controllability conditions or uncontrollable independently of observability conditions. This was confirmed by our experimental results. Consequently, when further exploring the testability of faults at the gate-level, it is sufficient in practice to only consider the following two special cases of ISA-level testability results:

Special case (1): The stuck-at-0 (stuck-at-1) fault F_{AS} is, by FEA, determined to be *uncontrollable* because in the program the AS state bit holds a constant logic value 0 (1) that is not modified by the software (cf. Chapter 6). Viewed in the gate-level circuit of Figure 7.2, this means that the AS state bit holds a constant value in all states visited by the FSM. This represents a constraint on the reachable state set of the FSM (in fact, an *invariant*). In the ATPG problem, such a constant value may be injected at the pseudo-*input* belonging to the AS state bit as a constraint. This constraint may cause other faults in the logic to become untestable. Note that nothing can be deduced for the pseudo-*output* corresponding to F_{AS} unless also unobservability of the fault has been proven by FEA at the ISA level.

Special case (2): Both faults, the stuck-at-0 and the stuck-at-1 fault, at the fault site F_{AS} are not testable. This means that the fault site is not observable, regardless of the logic value computed at that logic signal. Since controllability has no influence on the observability of the fault site, the auxiliary construction is simplified by setting the the multiplexer select input in Figure 7.3 to $C_{\mathrm{AS}}(x) + \overline{C_{\mathrm{AS}}(X)} = 1$. This is equivalent to completely removing the output marked "F_{AS}" from the combinational logic. As a result, all logic dominated by this output and possibly some faults also in other parts of the combinational logic become untestable.

Special case (2) can be made even more effective in identifying undetectable hardware faults on the gate level by the following observation. If a fault F_{AS} has been found to be unobservable by FEA, it is possible that it is unobservable independently of other unobservable faults. For example, some bit in a general-purpose register may never be read by the software, hence it does not influence the output of the program and it is unobservable, regardless of its contents. It is very likely that another bit in the same register has the same property. As is well known, however, faults may generally be dependent on each other, i.e.,

while a single fault may be unobservable, multiple fault signals may not. This must be taken into account by our algorithms.

The fault injection infrastructure presented in Section 5.1.1 allows us, in a single SAT instance, to analyse all multiple faults that result from all possible combinations of a selected set of single faults. Single faults that turn out to be untestable independently of each other can be used simultaneously for untestability detection on the gate-level.

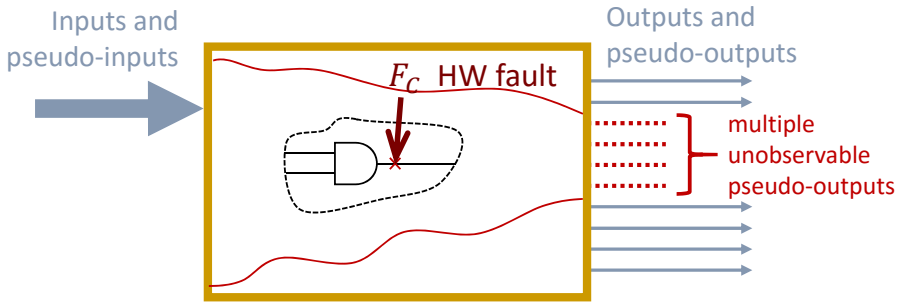


Figure 7.4: Multiple unobservable faults

Figure 7.4 illustrates how faults that are known to be unobservable independently of each other increase the likelihood that gate-level faults in the combinational logic become untestable. Given that FEA has proven that arbitrary combinations of fault signals do not propagate to any output of the program netlist, we may remove the corresponding pseudo-outputs as a group from the gate-level ATPG problem. As a result, all stuck-at faults in the combinational logic that propagate fault signals exclusively to any combination of these outputs will become redundant.

Algorithm 1 finds all combinations of faults that are unobservable and independent of each other. It takes as input a set of faults, F, that have been proven on the program netlist to be application-redundant by unobservability (i.e., special case (2) from above). Each fault location in F is both stuck-at-0 and stuck-at-1 application-redundant. The algorithm computes by an exhaustive depth-first search the maximum compatibility classes of unobservable fault locations. In other words, its output is the set of all largest sets of fault locations whose pseudo-outputs can be removed from the ATPG test problem simultaneously. It makes use of the recursive function `explore_fault_combination()`. This function has as input a current combination of faults, C, and a set of remaining faults to be tried, D. The function tries out all combinations of the set C with any one of the remaining faults (line 13). For each new combination FEA is invoked (line 16). If no test exists for the multiple-fault combination C' then at least the unobservable faults in C' are compatible with each other. We can try to add more faults to the combination by recursively calling `explore_fault_combination()` (line 18). If, however, the current combination C cannot be successfully combined with any element $d \in D$ then C needs to be stored as a maximal compatible class.

In principle, the runtime of this algorithm could become prohibitively long for large fault sets. In practice, however, it turns out that all or almost all unob-

```
1  L := ∅ ; /* global variable                                        */
2  /* input F: set of unobservable single AS fault locations        */
3  /* output L: set of maximal compatible fault sets                */
4  Function maximal_compatible_fault_classes(F) is
5  |  L := ∅;
6  |  explore_fault_combination(∅, F);
7  |  return L;
8  end
9  /* input C: current combination of faults                        */
10 /* input D: set of remaining faults to be tried                  */
11 Function explore_fault_combination(C, D) is
12 |  maximal := true;
13 |  foreach d ∈ D do
14 |  |  C' := C ∪ {d};
15 |  |  D' := D \ {d};
16 |  |  if multiple_faults_unobservable(C') then
17 |  |  |  maximal := false;
18 |  |  |  explore_fault_combination(C', D');
19 |  |  end
20 |  end
21 |  if maximal then
22 |  |  L := L ∪ {C}; /* keep current combination C                */
23 |  end
24 end
```

Algorithm 1: Algorithm for maximal compatible classes of unobservable faults

servabilities are independent of each other and can be put in a single compatibility class. This simplifies the algorithm to a (nearly) linear sweep through all involved bits. Runtimes for this algorithm, in our experience, therefore were negligible.

The results of the ISA-level testability analysis by FEA can now be used to identify application-redundant stuck-at faults at the gate level. The two special cases above are independent of each other and the resulting testability constraints can be applied simultaneously in the gate-level analysis.

Therefore, we proceed as shown in function `application_redundant_hw_faults()` in Algorithm 2. We use an ATPG tool to find untestable hardware faults in the combinational logic (line 10). We run ATPG once for each multiple unobservability class C found in algorithm `maximal_compatible_fault_classes()` of Algorithm 1. In every ATPG run we also inject the set of constant values determined by FEA (special case (1) above).

Before presenting experimental results, let us compare again the presented approach with the classical sequential ATPG problem discussed in the beginning of this chapter. The application redundancies computed by our approach for the gate-level circuit are sequential redundancies caused by the constraint of running a certain program. Due to the introduced simplifications it may be

```
1  /* input K: set of constant register bits                    */
2  /* input L: set of maximal compatible fault sets             */
3  /* output R: set of application-redundant hardware faults    */
4  Function application_redundant_hw_faults(K, L) is
5      R := ∅;
6      foreach C ∈ L do
7          /* prepare ATPG run                                  */
8          apply constant values K as pseudo-input constraints;
9          block error propagation through pseudo-outputs in C;
10         U := ATPG(); /* returns untestable hardware faults   */
11         R := R ∪ U;
12     end
13     return R;
14 end
```

Algorithm 2: Algorithm for computing application-redundant hardware faults

that our approach misses some application redundancies. For example, if a fault is observable for some inputs and controllable for others but never observable *and* controllable at the same time, then such a fault will be missed by our approach. However, as mentioned before, such faults are unlikely to exist in software programs, and we did not encounter any such instance in our experiments, as detailed in Section 7.1. It should be stressed that the proposed approach is conservative, i.e., it never deems a fault application-redundant that is not.

7.1 Experiments

We performed an experimental evaluation for a computing platform containing a 32-bit superscalar processor (Aquarius [2]). The platform is an open-source implementation of SuperH2 [86], an instruction set architecture developed by Hitachi and currently produced by Renesas. The experiments were conducted with two different programs:

- LIN: a driver for a slave node of a *Local Interconnect Bus*, used as a gateway to external buses; obtained from Infineon, proprietary IP.
- TCAS: a software-implemented traffic alert and collision avoidance system, developed by Siemens; obtained from the Software-artefact Infrastructure Repository [102].

For the following experiments, the computing platform Aquarius was synthesized to the gate level using Synopsys Design Compiler™ [100]. Synopsys TetraMAX™ [101] was used for running combinational ATPG on the synthesized design.

Table 7.1 shows some data for the gate-level computing platform used in our experiments before any ISA-level testability constraints from FEA have

primary inputs	51
primary outputs	73
state bits	1,217
target stuck-at faults	79,184
untestable stuck-at faults	3,367

Table 7.1: Gate-level analysis – Design statistics for Aquarius

been applied. This is done in our next experiment, following the procedures of Section 6.2 to identify application-redundant stuck-at faults at the gate level.

Program	Testability of Gate-Level Faults			
	application-redundant	uncontrollable	multi-signal unobservable	single-signal unobservable
TCAS	30,949	26,439	24,858	12,234
LIN	28,529	25,728	22,689	8,584

Table 7.2: Gate-level fault analysis for Aquarius

Table 7.2 shows the results of our gate-level fault analysis. For both programs a large number of application-redundant stuck-faults could be identified at the gate level (first data column). In fact, more than a third of all modelled stuck-at faults at the gate level turned out to be application-redundant for these programs. The second data column shows the number of untestable faults at the gate level that result when only the uncontrollabilities identified by FEA of Table 6.3 are applied, i.e., when removing line 9 in the algorithm of Algorithm 2. The third column shows the results when only unobservability is taken into account, i.e., when line 8 in Algorithm 2 is removed. Finally, the last column shows the results when the experiment of the third column is repeated but only Algorithm 1 is omitted, i.e., only single-signal unobservabilities are used for the gate-level analysis. A comparison between column 3 and 4 shows that examining and grouping multiple unobservabilities indeed increases the number of application redundancies that can be identified at the gate level.

CPU times for the different runs of ATPG on the gate-level implementation of Aquarius varied between 0.81 and 1.41 seconds for all experiments described in the above tables.

7.1.1 Fault Testability Analysis - RISC-V Results

We performed the same experimental evaluation as in Section 7.1 for the FWRISC-S [11] computing platform containing a 32-bit non-pipelined processor. The platform implements the RV32IMC variant of the RISC-V instruction set architecture which includes instructions for multiplication and support for a compressed instruction format. The RISC-V ISA was developed by the University of California.

We started with an ATPG run on the gate-level FWRISC-S computing platform before any ISA-level testability constraints from FEA were applied. Table 7.3 shows some data obtained from this ATPG run.

primary inputs	68
primary outputs	103
state bits	2,596
target stuck-at faults	91,146
untestable stuck-at faults	14,552

Table 7.3: Gate-level analysis – Design statistics for FWRISC-S

We then applied constraints to the gate-level processor obtained from FEA on the ISA level and performed FTEA. For this purpose we used FEA to find the maximum compatibility class of multiple unobservable fault locations as discussed earlier in this chapter. Table 7.4 shows the results of this analysis. Like in FTEA for Aquarius (cf. Section 7.1) our analysis identified a third of all gate-level stuck-faults as application-redundant (first column). The second column shows the number of uncontrollable gate-level faults when only uncontrollable faults found at the ISA level are applied. The last column shows the number of unobservable gate-level faults when only observability, i.e., the computed compatibility class, is taken into account.

Program	Testability of Gate-Level Faults		
	application-redundant	uncontrollable	multi-signal unobservable
TCAS	33,308	26,288	24,314

Table 7.4: Gate-level fault analysis for FWRISC-S

For the experiments conducted in this section, an individual ATPG run on the gate-level implementation of FWRISC-S took between 1.52 and 1.68 seconds. Overall, we conducted four ATPG runs on the FWRISC-S computing platform.

8

ISA/C CROSS-LEVEL FAULT ANALYSIS

In this chapter we answer the question how ISA-level effects of hardware faults can be mapped to the C-level and how the propagation of the abstracted fault effects can be analysed at C level for a complete software system. This is particularly useful for software systems that are too large to be analysed by a single ISA-level FEA. The basic idea of this cross-level analysis is that the software interface specified by the ISA description serves as a foundation for higher software abstraction levels.

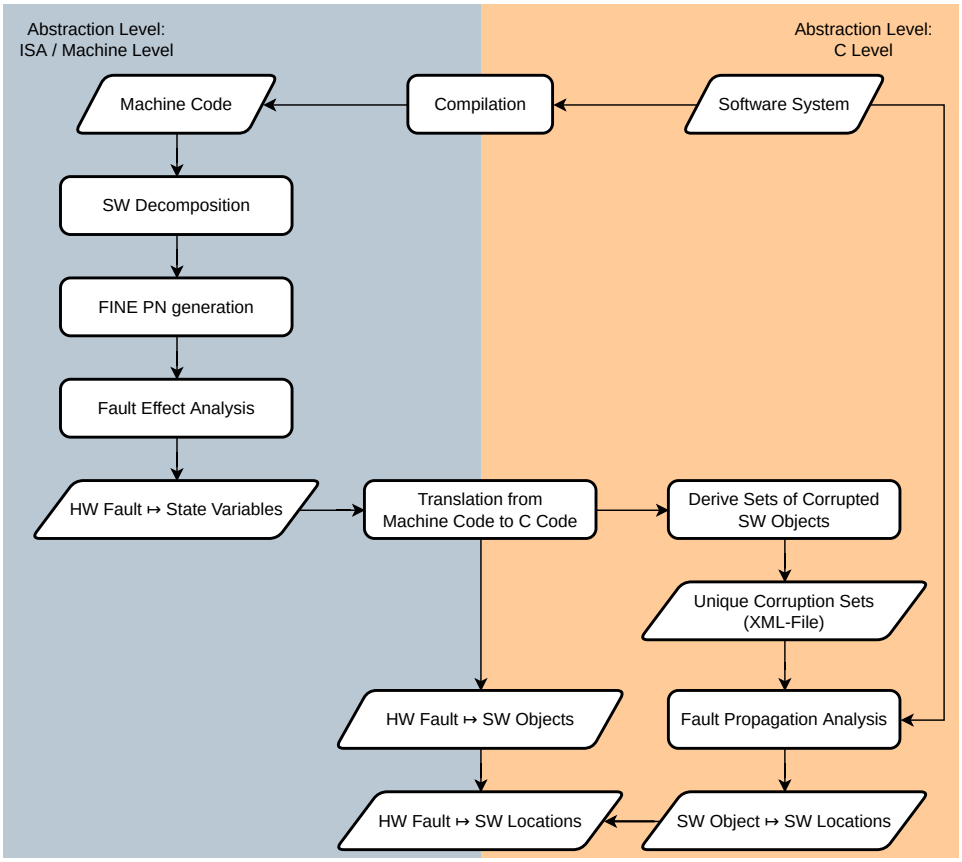


Figure 8.1: ISA/C cross-level fault analysis flow

The flow chart in Figure 8.1 shows the steps of our method that combines the fault effect analysis on ISA level with a *fault propagation analysis* (FPA) on C level to formally analyse the cross-layer effects of faults. FEA employs formal property checking and equivalence checking while FPA is based on Abstract Interpretation. The only input of the combined approach is the source code

of a software program on which it performs two fault analyses at different abstraction levels.

The first one is the FEA (cf. Chapter 6), operating at the ISA and machine level. In order to obtain accurate results the software program is compiled for the target system. This ensures that the machine code analysed by FEA is the same that runs on the embedded system. We achieve scalability to large software programs by using two orthogonal methods. The first method decomposes the software and generates a FINE PN for each software component. The second method consists of a set of techniques for accelerating PN generation. Section 4.3 describes how this helps to scale the formal software model used by FEA to complex programs composed from several modules. Our approach is largely automated. Only the software decomposition and the corresponding configuration of the PN generation are manual work. PN generation as well as FEA are fully automated.

FEA computes, for every hardware fault in a program-visible register, the set of potentially affected state variables at the interface of a software component. These state variables are associated with corresponding software objects at the C level using information from software compilation. The output generated by FEA is a mapping between hardware faults in program-visible registers and all potentially affected high-level software objects, thereby creating a formally sound abstract representation of all faults at the C level.

Based on this mapping we compute sets of corrupted C-level software objects which are passed to the second analysis, FPA. FPA operates on source code written at C level. It determines whether the high-level software image of the hardware fault can affect user-defined locations in the software, such as certain safety functions that were defined by a safety engineer. In the final step, the outputs of both methods are combined, so that all possible locations on C level to which a fault at the machine level can propagate are known.

8.1 Software Decomposition

For scalability of our approach, we split the software program into components and perform a separate run of FEA for each component. It is not possible to combine the results of the individual FEAs by a simple aggregation to obtain a formal description of the global fault effects. This is because faults can propagate and spread out through multiple components where they can affect more or other parts of the software.

We address this problem by using the local FEA results as inputs to the proposed FPA conducted globally at a higher abstraction level. The interface between the software components is, therefore, also the interface between the two abstraction levels, specifically the machine level for FEA and the C level for FPA. The challenge is to define this interface in such a way that fault effects identified at the low abstraction level can be modelled at the high abstraction level without any loss of formal precision.

The given software program, *systemSW*, consists of a set of machine instructions, $U = \{\iota_1, \iota_2, \ldots\}$ and of program state variables. A machine instruc-

tion ι_k is a pair (a_k, c_k) of a program memory address a_k and an instruction word c_k. The software is decomposed into m components $componentSW_i$ with $U = \cup_{i=1}^{m} U_i$, where U_i is the set of machine instructions of component i.

Each software component communicates with other components and, across individual executions, also with itself, through a set of n state variables, $V = \{v_1, \ldots, v_n\}$. A state variable, v_j, is a processor register or a memory location, including ones mapped to peripheral device registers. The content (valuation) of a state variable, $s(v_j)$, contributes to the program state. The program state space is given by the product of the individual variable state spaces. We define the program state s_k as the combined contents of all program state variables *immediately before* instruction $\iota_k$is executed: $s_k = (s(v_1), s(v_2), \ldots, s(v_n))$.

Definition 3. *The* interface *of a $componentSW_i$ is defined by partitioning the state variables V_i into the following classes, $V_i = \pi_i \cup \kappa_i \cup \varphi_i \cup \gamma_i \cup \vartheta_i$:*

1. Persistent variables π_i*: accessed only by $componentSW_i$, e.g., static module variables;*
2. Communication variables κ_i*: used by $componentSW_i$ to communicate with $componentSW_l$, where $i \neq l$, e.g., shared-memory variables*
3. Foreign variables φ_i*: only read by $componentSW_l$, with $i \neq l$, e.g., static variables of other components*
4. Volatile variables γ_i*: e.g., device registers*
5. HW-protected variables μ_i
6. Other variables ϑ_i*: e.g., temporary variables, unused or inaccessible memory locations*

In normal, fault-free operation, only a subset of the program state variables are visible to the software. Faults may, however, affect addressing and thereby change the set of state variables accessed by the software.

Note that the classification of state variables μ_i as hardware-protected is possible for hardware platforms providing memory protection units. If a fault modifies a write operation such that the access becomes illegal then the fault can be considered as detected and can be ignored in FEA.

The partitioning according to Definition 3 is a straightforward process and is similar to already existing practices of structuring the memory space in safety-critical systems [50].

The disadvantage of decomposing the software and analysing the components individually is that we lose information about the program context when a component begins execution. However, we can compute an over-approximation of the program starting state of the component using Abstract Interpretation [32]. We use the commercial tool *ValueAnalyzer* [1] from AbsInt for this purpose.

Definition 4. *The over-approximation of the program starting state of componentSW$_i$ is called the* abstract starting state $\psi_{i,0}$. *The set of program states which componentSW$_i$ can end in after starting in any of the states in $\psi_{i,0}$ is called its* abstract ending state $\psi_{i,end}$.

As will be described below, the abstract states $\psi_{i,0}$ and $\psi_{i,\text{end}}$ (cf. Definition 4) are used in FEA to take into account the connection of *componentSW$_i$* with other components.

FPA requires that the affected parts of the interface at the machine level can be translated to the corresponding parts at the C level. Note that some register and memory locations, e.g., temporary variables allocated on the stack, do not have a representation at higher abstraction levels. This is especially true after compiler optimizations have been applied. In practical systems, when using a "natural" decomposition, e.g., by function calls, the variables of classes π, κ and γ can be mapped to C-level objects. Only some parts of φ and ϑ, typically located on the stack or in registers, cannot be mapped. In the few cases where a fault redirects a write operation to such an variable in memory, an additional FEA analysis is run targeting this very variable.

FEA also requires that the abstract time point when the communication happens is uniquely identified. It is given by some machine instruction (a, c) that writes to an element of κ_i. The corresponding program locations at the C level can be determined using DWARF debug information.

8.2 Fault Effect Analysis for Software Components

For each software component FEA is performed on the corresponding FINE PN. As discussed in Section 6, we build a "miter" structure similar to hardware equivalence checking, by combining a fault-free PN with a fault-injected PN. We use a SAT solver to find a program execution in which the fault-injected PN produces a behaviour in the considered program state variable that differs from the fault-free PN for the same inputs.

FEA computes which program state variables written by a software component can be affected by a given fault. It creates a 1-to-n mapping between each fault and a set of potentially affected variables of the component's interface, as defined in Section 8.1. A fault effect analysis for a particular fault or software component does not depend on other fault effect analyses, which allows for full parallelization of the computation. Runtime can be traded for fault site resolution by adapting the granularity level of the analysis. For example, if we do not care about the exact bit position of an SEU fault in a program-visible register, we may opt to analyse all single bit flips in the same register in a single run of FEA.

8.2.1 Fault Injection

As presented in Chapter 5, we can insert fault logic into the PN after (type A) or before (type B) the PN generation. Generating a B-FINE PN leads to a model where all fault effects are formally modelled in the generated PN.

However, since this incurs high complexity and runtime in the PN generation process and the subsequent analysis, especially if indirect branches (using an address register) and indirectly addressed memory locations are involved, we employ A-FINE PNs in our experimental evaluation. The inaccuracies, i.e., not modelled fault induced program paths and memory accesses can be compensated as discussed in the following.

Not all variables in a component's interface are relevant for the fault analysis — only those that affect other components. Also variables affecting future invocations of the same component are relevant as they could be affected by latent faults (cf. Section 2.3). Modifications of temporary and other variables that go out of scope when a software component terminates can be ignored.

Definition 5. *The fault-relevant variables of componentSW$_i$ are the persistent, communication and foreign variables, $\tilde{V}_i = \pi_i \cup \kappa_i \cup \varphi_i$. The fault-relevant program state $\tilde{s}_{i,k} = (s(\tilde{v}_1), s(\tilde{v}_2), \ldots, s(\tilde{v}_r))$ is the sub-state given by the combined valuations of the fault-relevant variables, $\tilde{v}_i \in \tilde{V}_i$, before executing the instruction ι_k. By $\tilde{s}_{i,end}$ we denote the fault-relevant program state after termination of componentSW$_i$.*

Definition 6. *For a given componentSW$_i$, the transition function, $\tilde{\Delta}_i(s_0)$, computes the fault-relevant program ending state, $\tilde{s}_{i,end}$, as a function of the program starting state s_0.*

Definition 6 defines the transition function of the program netlist describing *componentSW$_i$*. (In our compositional approach it is constrained by $\psi_{i,0}$ describing the abstract starting state, as derived by Abstract Interpretation using ValueAnalyzer [1].)

If a B-FINE PN was generated, we could determine all relevant fault effects simply by checking whether there exists a starting state for which the execution of the software component computes different ending states in the faulty and the fault-free case, i.e., $\exists s_0 \in \psi_{i,0} : \tilde{\Delta}_i(s_0) \neq \tilde{\Delta}'_i(s_0)$. In practice, however, this would mean that all variables at all possibly modified addresses in all modified control flows have to be checked. This is feasible only for small problem instances. For the simplified model, i.e., the A-FINE PN, obtained by injecting faults only after fully generating the fault-free PN for the component, we have to take a modified approach.

We separate the possible fault effects into two separate categories. The first category is given by fault effects that propagate to a component's interface (cf. Definition 3) along execution paths explicitly modelled in the A-FINE PN. The second category consists of fault effects that lead to severe control flow alterations not modelled by the A-FINE PN. They are considered separately, as described in Section 8.2.2.

For the first category of fault effects, it is a key observation that a fault can affect the state variables of a component's interface only in one of three ways:

1. It can change the data when writing to a state variable of the interface.
2. It can change the address of a write operation to a program state variable.

3. It can activate or deactivate a write operation to an interface variable by changing the control flow, e.g., by modifying execution predicates or branch conditions.

A software component modelled by a PN produces a fixed number of write transactions, modelled by a vector of "write ports" in the PN model. For a given starting state s_0, a certain set of writes are active while the others are inactive, depending on the execution path taken as a result of s_0.

Definition 7. *A write transaction $\tau(s_0)$ is a triple $\tau(s_0) = (\alpha, \beta, \eta)$ consisting of an* active flag *$\alpha(s_0)$, a target address $\beta(s_0)$ and write data $\eta(s_0)$. All three elements are functions of the starting state s_0 of the $componentSW_i$. A* fault-relevant *write transaction $\tilde{\tau}$ is one where the target address belongs to a fault-relevant state variable.*

The *active flag* of a transaction τ corresponds directly to the *active-signal* of store instructions in the PN. (As illustrated in Figure 4.1, the *active-signal* indicates for every instruction whether or not it lies on the execution path triggered by s_0.) The target address, $\beta(s_0)$, points to a program state variable. A fault may modify the target address, and, thus, the variable written to in the transaction. The PN model contains address resolution logic that computes what fault-relevant or fault-irrelevant program state variables are affected by a write transaction. Finally, a fault may modify the data, $\eta(s_0)$ for a given starting state, s_0.

T_i is the vector of all write transactions of $componentSW_i$ for a given $s_0 \in \psi_{i,0}$. Note that T_i is a vector, not a set, because there may be several accesses to the same variable and their order is relevant. $\tilde{T}_i$ is the sub-vector of all *fault-relevant* write transactions. A fault has an effect on the fault-relevant state variables of the program ending state $s_{i,\text{end}}$, if there exists a starting state s_0 for which the fault-free execution and the fault-affected execution produce different write transaction(s):

$$\exists s_0 \in \psi_{i,0} : \tilde{T}_i \neq \tilde{T}_i'$$

In our tool, this check is implemented as an assertion checked on the computational model which consists of the PN modelling the software component and the set of constraints, $\psi_{i,0}$, modelling the abstract starting state of program execution. The check is much simpler than the one for the B-FINE PN because the simplified model excludes severe control flow alterations leading to arbitrary modifications of program state. Instead, the control flow is confined to the software segments existing in the original software. While wrong branch decisions are modelled, wrong branch target addresses are not.

8.2.2 Modelling Severe Control Flow Errors

Faults may, nevertheless, affect the flow of control in such a way that it completely deviates from the execution paths of the original program. This can happen whenever the fault effect modifies the contents of a register that is used in a branch target address computation. A branch that depends on the contents of a register is called an *indirect branch*. Indirect branches are often

used in jump or call tables. Faults affecting indirect branches lead to *control flow errors* (CFEs) that are likely to result in a program crash.

In the PN, control flow is modelled using the *active-signal* that indicates, for every instruction, whether it is on the execution path under the current assignment of $s_0 \in \psi_{i,0}$. Instruction cells modelling branches have two or more successors, each continuing a different execution path, depending on the branch condition. Whenever the IC modelling the branch instruction is *active*, at most one of its successors has an asserted *active-signal*. For indirect branches, the PN compares the computed branch address against a list of possible successor addresses and asserts the active-signal of the one successor that matches. If none of the addresses match, the active-signal is lost.

We detect severe control flow errors that alter the execution such that it is no longer modelled by the paths in the PN *by detecting the loss of the active-signal*. The branch instruction where this happens is identified and reported to the high-level FPA at C level. Additionally, we conservatively mark all program variables of all $componentSW_i$ as corrupted.

Note that control flow errors in indirect branches that do not lead to a loss of the active-signal are still modelled by the approach described in the previous sections.

8.2.3 Composing the Fault Dictionary

FEA determines, for a given $componentSW_i$ and a given fault λ of the fault list:

- whether the fault leads to a severe control flow error corrupting all program state variables, or, if not,
- the set of fault-relevant interface variables affected by the fault.

The result of FEA is a database mapping faults to program state variables, back-annotated to the C statements writing the variables, by specifying source file and line number.

Fault propagation analysis at the C level, to be described in the following section, takes this database as input. For a given fault, it iterates over all $componentSW_i$ to determine the fault effects. FPA takes as starting point the set of program variables in a $componentSW_i$ affected by a selected fault. It enumerates all situations in which the affected C statements are executed, i.e., if the $componentSW_i$ is called several times, then an analysis is performed for every call.

The result of FPA is a set of fault effects for a given fault related to high-level software objects, such as selected variables or safety functions, as can be specified by a safety engineer.

For a given fault, the results of all FPA runs are aggregated and inserted into the fault dictionary to document the high-level consequences of this fault.

In the following section, we describe the technology underlying a single run of FPA.

8.3 Fault Propagation Analysis

The fault propagation analysis was developed by AbsInt [1] during a joint research project. It leverages the taint analysis framework available in the static analyser Astrée [44, 58] whose main purpose is to perform runtime error analysis in C/C++ programs. Data corruptions are translated into taint hues, whose propagation is analysed as a part of Astrée's data and control flow analysis. With its combined runtime error and taint analysis, Astrée can track the flow of data corruptions through the program to determine all affected control decisions and induced runtime errors. Since Astrée is a sound analyser based on Abstract Interpretation, the absence of alarms about runtime errors or corrupted control decision constitutes a proof of absence of such defects. In this section we will give a brief overview of Abstract Interpretation, the design of Astrée and the foundations of taint analysis, and then focus on the taint analysis-based FPA.

8.3.1 Astrée

Astrée's main purpose is to report program defects caused by unspecified and undefined behaviours in C/C++ programs. The reported code defects include integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (buffer overflows, null pointer dereferencing, dangling pointers, etc.), data races, lock/unlock problems, deadlocks, etc.

Astrée uses abstractions to efficiently represent and manipulate over-approximations of program states. One simple example of abstraction used pervasively in Astrée is to consider only the bounds of a numeric variable, forgetting the exact set of possible values within these bounds. However, more complex abstractions can also be necessary, such as tracking linear relationships between numeric variables. As no single abstraction is enough to obtain sufficiently precise results, Astrée is actually built by combining a large set of efficient abstractions. Some of them, such as abstractions of digital filters or finite state machines, have been developed specifically to analyse control-command software as these constitute an important share of safety-critical embedded software. In addition to numeric properties, Astrée contains abstractions to reason about pointers, pointer arithmetic, structures and arrays (in a field-sensitive or field-insensitive way). Finally, to ensure precision, Astrée keeps a precise representation of the control flow, by performing a fully context-sensitive, flow-sensitive (and even partially path-sensitive) inter-procedural analysis. Combined, the available abstract domains enable a highly precise analysis with low false alarm rates.

To deal with concurrency defects, Astrée implements a sound low-level concurrent semantics [75] which provides a scalable sound abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of runtime errors found in sequential programs, to report data races, and lock/unlock problems, i.e., inconsistent synchronization. The set of shared variables does not need to be specified by the user: Astrée

assumes that every global variable can be shared, and discovers which ones are effectively shared, and on which ones there is a data race. After a data race, the analysis continues by considering the values stemming from all interleavings. Since Astrée is aware of all locks held for every program point in each concurrent thread, Astrée can also report all potential deadlocks.

Practical experience on avionics and automotive industry applications are given in [55, 76]. They show that industry-sized programs of millions of lines of code can be analysed in acceptable time with high precision for runtime errors and data races.

8.3.2 Taint Analysis

Taint analysis is widely used in security analysis for the verification of secure information flows. It aims at tracking the origin of values computed by a program, by assigning imaginary taint hues to variables at the program locations of interest, and propagating them during analysis. Astrée has been equipped with a generic abstract domain for taint analysis. It allows Astrée to perform normal code analysis, with its usual process-interleaving, interprocedural and memory layout precision, while carrying and computing taint information at the byte level. Any number of taint hues can be tracked by Astrée, and their combinations will be soundly abstracted.

Taint propagation can be formalized using a non-standard semantics of programs, where an imaginary taint is associated to some input values. Considering a standard semantics using a successor relation between program states, and considering that, at C-level, a program state is a map from memory locations (variables, program counter, etc.) to values in $\mathbb{V}$, the *tainted* semantics relates tainted states, which are maps from the same memory locations to $\mathbb{V} \times \{\text{taint}, \text{notaint}\}$, and such that if we project on $\mathbb{V}$ we get the same relation as with the standard semantics.

To define what happens to the *taint* part of the tainted value, one must define a *taint policy*. The taint policy specifies:

- *Taint sources* which are a subset of input values or variables such that in any state, the values associated with that input values or variables are always tainted.
- *Taint propagation* describes how the tainting gets propagated. Typical propagation is through assignment, but more complex propagation can take control flow into account, and may not propagate the taint through all arithmetic or pointer operations.
- *Taint cleaning* is an alternative to taint propagation, describing all the operations that do not propagate the taint. In this case, all assignments not containing the taint cleaning will propagate the taint.
- *Taint sinks* is an optional set of memory locations. This has no semantical effect, except to specify conditions when an alarm should be emitted when verifying a program (an alarm must be emitted if a taint sink may become tainted for a given execution of the program).

Tainted input is specified through directives (`__ASTREE_taint((var;hues))`) attached to program locations. Such directives can precisely describe which variables, and which part of those variables, are to be tainted, with the given taint hues, each time this program location is reached.

Taint sink directives (`__ASTREE_taint_sink((var))`) may be used to declare that some parts of some variables must be considered as taint sinks for a given set of taint hues. When a tainted value is assigned to a taint sink, then Astrée will emit a dedicated alarm, and remove the sinked hues, so that only the first occurrence has to be examined to fix potential issues with the security data flow.

The main intended use of taint analysis in Astrée is to expose potential vulnerabilities with respect to security policies or resilience mechanisms. Thanks to the intrinsic soundness of the approach, no tainting can be forgotten, and that without any bound on the number of iterations of loops, size of data or length of the call stack. This makes it well suited as a basis for sound FPA.

8.3.3 Taint-based Fault Propagation Analysis

The FPA of Figure 8.1 is implemented by making extensions to the taint analysis in Astrée. As described above, taint sources and taint sinks are defined by Astrée directives that can be conceptually applied to arbitrary program locations. Those directives can be directly inserted in the code, but they can also be specified externally, without source code changes, by a formal language specifying locations of the abstract syntax tree [60].

In order to model FPA, each data corruption is translated into an Astrée taint directive that taints the affected set of variables determined by FEA, as described in Section 8.2, with a hue that identifies this particular corruption. The destructive effect on the values of the affected variables is expressed by an additional Astrée directive ((`__ASTREE_modify((var;full_range))`)) that models an arbitrary change of its values. In the sequel of the program, such value changes might cause runtime errors, or changes in its data and control flow.

With its combined runtime error and taint analysis, Astrée can track the flow of data corruptions through the program. Whenever a control decision is reached by a tainted variable, a taint alarm is raised. In order to track back runtime errors or data and control flow changes to faults, the results of the Astrée analysis can be compared to the results of the analysis before fault modelling.

Since Astrée is a sound analyser that operates on over-approximations, the absence of an Astrée alarm for a control statement is proof that the statement is not affected by data corruptions. In the same way, statements that are not reported to be affected by runtime errors, are guaranteed to be free from runtime errors.

The following program fragment shows the modelling of a data corruption on the variable `j` in line 1. The first directive expresses the arbitrary destruction of values, the second taints the affected variable with hue 1. The corruption has two critical effects on the code that are reported by Astrée:

- an array index out-of-bounds error in `arr[j]`
- an influence on the execution paths controlled by the statement `if (x > b)`

```
1 j = a;
2 __ASTREE_modify((j));
3 __ASTREE_taint((j; 1));
4 x = arr[j];    /* ALARM: array index out-of-bounds */
5 if (x > b) {   /* ALARM: hue 1 reaches taint sink */
```

Figure 8.2: Astrée Directives in C Code

To automate this analysis, Astrée provides an XML interface for specifying byte-level memory locations that are corrupted at specific program locations. This set of corruptions is automatically transformed into Astrée directives that are conceptually at the affected program locations, as displayed above. The listing below shows a simplified example, expressing corruption of the first two bytes of the variable `j`.

```
1 <memory_location mlid="0" ...>(char*)&j+0</memory_location>
2 <memory_location mlid="0" ...>(char*)&j+1</memory_location>
3 <program_location ... line="1" column="1"/>
4 <data_corruption mlid="0" plid="0">1</data_corruption>
```

Figure 8.3: FEA-to-FPA Interface

8.4 Experimental results

For our experimental evaluation we implemented a demonstrator system consisting of several software modules representative in size and complexity of typical embedded-system software. The software modules interact with each other through shared memory.

The demonstrator includes four software modules:

- LIN: a driver for a slave node of a *Local Interconnect Bus*, used as a gateway to external buses; obtained from Infineon, proprietary IP.
- FuelSys: a fuel rate controller for a combustion engine, taken from the fuel control system example of MATLAB [73].
- TCAS: a software-implemented traffic alert and collision avoidance system, Developed by Siemens; obtained from the Software-artefact Infrastructure Repository [102].
- RSA: An encryption algorithm based on a loop implementation, obtained from [4].

- Scheduler: controls the execution of the above modules, own development.

Since firmware examples, such as the above, are hard to find as open-source, and since the system composed by these modules is only of medium complexity, we decided to demonstrate the scalability of our approach by creating a larger system using numerous instances of these modules. Although this system may not be technically meaningful, it creates a similar computational complexity for our analysis as would be the case for a system with the same number of different modules of comparable size. Table 8.1 shows the lines of C code for each module of the system. The table also shows the number of instances for each module that we integrated into the system. The complete software system subject to our compositional FPA comprises a total of 137,900 LoC. It has been compiled with *gcc* for a RISC-V architecture.

Module	Lines of C-code (per instance)	# instances	PN generation (time in s)	PN size (# ICs)
Scheduler	2,829	1	590	9,246
LIN	781	1	215	3,927
TCAS	125	5	2	237
FuelSys	8,869	15	2,838	9,041
RSA	63	10	5	784
System	137,900	-	43,435	-

Table 8.1: Software System

The scheduler orchestrates the executions of the different modules by calling them in a pre-defined order: At first, the LIN slave node receives a set of data through the bus LIN communication variables (modelled by κ_{LIN}), which are then distributed to other software modules by the scheduler using the shared memory variables of the receiving modules ($\kappa_{\text{FuelSys}}, \kappa_{\text{TCAS}}, \kappa_{\text{RSA}}$). After the execution of any of the software modules its computation result is moved to the input variables of the LIN node, κ_{LIN}. Finally, the LIN node sends this data out on the bus.

In this experimental setup we selected SEUs appearing program-visible register bits as our fault model in order to provide an application of FEA for soft errors.

8.4.1 Fault Effect Analysis (FEA)

The experiments for FEA were run on a computer with two Intel®Xeon®Gold 6234 CPUs and 252 GB of main memory. As back-end property checker for FEA we used OneSpin 360 DV [79]. With this tool a property checking run consists of several phases. Parsing, elaboration and compilation cannot be parallelized but run as single threads. The actual property check on the computational model can be parallelized and was configured to use 30 threads.

We decompose the system based on the "natural" boundaries of its modules, i.e., we consider one $componentSW_i$ for each module instance. A PN modelling the fault-free behaviour is then generated for each component. The last two columns in Table 8.1 show the runtime of PN generation and the size of the generated PNs. The total PN generation time aggregated over all modules and their different instances amounts to about 12 hours.

For each module, we classify the interface variables according to Definition 3. We use debug information to identify the binary versions of the C-level variables and analyse the PN to determine stack and heap areas. With this information we obtain and partition a total of 940 state variables into interface classes for each $componentSW_i$. This was done manually and took less than 30 minutes for each component. We also used the PN to identify all abstract time points where a $componentSW_i$ performs a write operation, i.e., a write transaction (cf. Definition 7), to the program state.

Module	# of Variables						# write transactions
	π_i	κ_i	φ_i	γ_i	ϑ_i	μ_i	to memory
Scheduler	15	93	1	2	1	828	1,079
LIN	14	2	2	2	1	919	534
TCAS	13	2	2	2	1	920	24
FuelSys	51	2	2	2	1	882	359
RSA	0	6	1	2	1	930	70

Table 8.2: Interface Classes & Write Transaction

Table 8.2 presents for each module the number of interface variables and the number of potential write transactions. Note that the complexity of FEA is sensitive to the number of interface variables and the number of write transactions because a fault may possibly affect every variable in combination with every write transaction. Table 8.2 demonstrates the efficacy of the compositional approach since the number of interface variables and transactions to be considered is drastically smaller for a single component than for the system as a whole.

For the subsequent Fault Effect Analysis, we generate a COI-reduced A-FINE PN for every instruction performing a write transaction. In the context of our COI computation (cf. Section 4.4) this instruction is the *head* at which the COI reduction starts. FEA considers all single SEUs over all program-visible registers. Note that fault injection in our formal approach (cf. Chapter 5) allows us to consider several faults implicitly in a single run of FEA, leaving it to the SAT solver to enumerate all single fault injections. This increases the performance of our analysis at the price of losing the information which individual fault of the selected fault set is responsible for the fault effect possibly observed by FEA. In our experiments, we configured our tool to consider all single-bit SEUs of a selected register over all abstract time points in a single run. This is motivated by the assumption that hardening only a single bit of a register and only for specific time points is not practical.

The runtime of FEA for the *componentSW* LIN is delineated in Figure 8.4. The blue curve plots the runtime for each COI (left y-axis), where the COIs are sorted by increasing PN size (# of instruction cells) on the x-axis. The red curve shows the ratio of runtime and PN size.

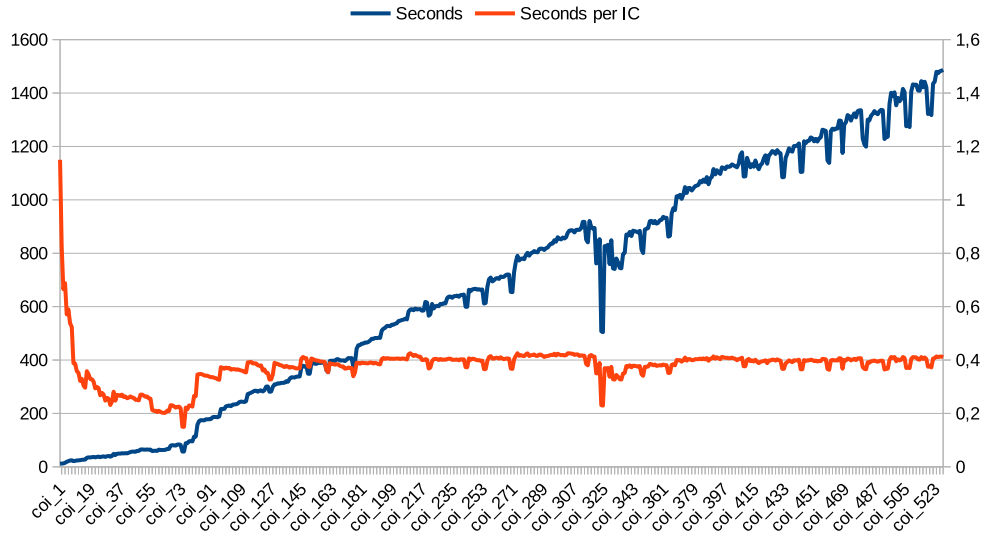


Figure 8.4: LIN SEU Fault Analysis Runtime

We note that the runtime of FEA increases nearly linearly with the number of instruction cells in the PN. The long runtime per node for the first COIs is due to preprocessing overhead when a small PN has to be analysed.

Module	Injected SEUs	Runtime (in h)	% of propagating SEUs w.r.t. Variables $\tilde{V}$	Control Flow
Scheduler	9,467,904	460.00	75	69
LIN	4,021,248	7.00	38	34
TCAS	242,688	0.02	28	6
FuelSys	9,257,984	55.00	69	69
RSA	802,816	0.50	69	63
System	161,600,512	1,297.10	75	75

Table 8.3: FEA Results

Table 8.3 shows the results of FEA applied to the complete SEU fault list for the different $componentSW_i$. The second column shows the number of faults that are injected at the different abstract time points of the $componentSW_i$. This number is #SEU-affected register bits $\times$ #ICs. The computation time for analysing these faults is given in column 3. Thousands of individual properties, each with a runtime of only a couple of seconds, are processed sequentially in our experiments. In the case of FuelSys, for example, FEA checks 670,000 properties. Each property is independent of all other properties and checks

whether an SEU which strikes at an arbitrary abstract time point and an arbitrary bit in a specific register can have an effect on a specific write operation such that a specific interface variable is affected by the fault. It is straightforward to highly parallelize these numerous calls to the property checker which would drastically reduce the elapsed real time for FEA. Moreover the runtime is dominated by the front-end phases of parsing, elaboration and compilation for the numerous calls of the commercial property checker. We expect that a more direct coupling of its back-end solvers with FEA can reduce computation times by an order of magnitude. (This, however, is beyond the scope of our research environment.)

Column 4 presents the percentage of fault-relevant interface variables (cf. Definition 5) to which a fault was propagated. The right-most column gives the percentage of faults that cause a severe control flow alteration (Section 8.2.2). The bottom row of the table shows cumulative results for the system as a whole. In particular, we observe that only SEUs in 24 out of 32 registers (i.e., 75 %) can affect some fault-relevant variable in any of the $componentSW_i$, or lead to severe control flow alterations somewhere in the system.

Note that, by merit of the formal nature of our approach, we can provide guarantees on which faults never cause a certain fault effect.

Based on this analysis we create a mapping between all SEUs and the potentially affected interface variables. Overall 618 taints for the FPA on the C level were generated.

8.4.2 Fault Propagation Analysis (FPA)

The taints generated by FEA are used as input to FPA. The communication between both analyses is fully automated through the XML interface (cf. Section 8.3.3). Table 8.4 displays some general information about the input data that is supplied by FEA to FPA. The second column shows the number of C-level addresses (at byte granularity) belonging to C variables affected by some fault. Similarly, column 3 provides the number of source code locations in which a fault manifests itself.

Module	Memory locations	Program locations	Taint hues	Directives	Runtime (in s)
Scheduler	631	221	19	21,764	7,200
LIN	16	79	12	1,179	5,300
TCAS	110	21	9	1,813	1,400
Fuelsys	41	123	22	3,466	1,500
RSA	1	2	20	21	1,400
Total runtime (system w. multiple $componentSW_i$ instantiations)					56,000

Table 8.4: Taint Analysis Inputs and Runtime

When analysing the fault database generated by FEA we observe that often a number of different machine-level faults lead to an identical fault effect in

the high-level software, i.e., they have the same abstract image in terms of C-code objects. The fourth column in Table 8.4 shows the number of different abstract fault images for each $componentSW_i$. Each such fault image creates a unique taint hue (cf. Section 8.3.2) for FPA. Every taint hue is applied to the affected memory and program locations, each leading a corresponding set of Astrée directives (cf. Section 8.3.2). Column 5 reports the number of generated Astrée directives for each module. The right-most column shows the runtime of Astrée for injecting all faults to a $componentSW_i$ and evaluating their effect in the composed system of 137,900 LoC. Without exploiting its inherent options for parallelization, FPA completes the analysis for all faults and all $componentSW_i$ after a total of 56,000 seconds, clearly demonstrating the scalability of FPA to large systems.

We used FPA to analyse the fault effects w.r.t. runtime errors, control flow errors and corrupted variables.

Runtime errors

As a first experiment, we used Astrée to identify all runtime errors (cf. Section 8.3.1) existing in our system without any fault injection. Astrée identified 65 software weaknesses in different modules, possibly causing such runtime errors. We then used FPA to examine what fault injections could possibly lead to additional runtime errors. For one taint hue, i.e., for one set of machine-level hardware faults, Astrée identified a catastrophic runtime error induced by a pointer corruption, and stopped all further analysis. After excluding this case from consideration, an additional 20 runtime errors were identified and related to the hardware faults causing them. Conversely, our compositional approach formally proves for all other faults that no additional runtime errors can be caused.

Control Flow Errors

Astrée identified a total of 3,724 control decisions in the software. The static analysis for the fault-free software identified none of these decisions as potentially flawed. For the fault-injected system, FPA reveals that 1,157 control decisions can be affected by one or several faults. For more than 2/3 of the control decisions it is formally proven that they cannot be affected by any of the faults.

Corrupted Program Variables

Our compositional fault propagation analysis yields a database, as described in Section 8.2.3, that links fault effects at the C level to SEUs at the hardware level. Besides runtime errors and control flow errors, the collected fault effects also include data corruptions in the individual program variables of the software system.

The generated database allows for a detailed analysis, e.g., identifying a set of faults which affects a set of program variables belonging to a selected safety

function. For example, we chose an apparently safety-critical variable tracking transmission errors in the LIN bus and determined that SEUs in only 3 registers can propagate to this safety function. This provides valuable information when implementing hardware level resilience measures. Our database also supports implementing resilience measures at the software level. For the example of the safety-critical LIN variable, we can determine that all SEUs occurring during execution of any $componentSW_i$ other than LIN do not propagate to this variable.

Similarly, our database allows for error containment analysis. For example, it yields that SEUs occurring during execution of LIN / TCAS / FuelSys / RSA can propagate to at most 2.77 % / 2.76 % / 2.82 % / 2.72 % of program variables anywhere else in the software system, respectively. In combination with our fault dictionary, such information provides a strong basis for HW/SW cross-layer resilience measures.

The database can be extended for even more fine-grained risk assessments by additionally taking into account fault probabilities as well as use case statistics for the software.

9 CONCLUSION & OUTLOOK

The main contributions of the concepts and methods presented in this thesis are the following.

Scalable Program Netlist Generation By exploiting properties that are characteristic for embedded systems software, like firmware and drivers, we developed several techniques that can significantly reduce the complexity of the PN generation process and any subsequent analyses, e.g., fault analyses, on the generated PN.

We have shown the efficacy of the techniques in our experimental evaluation, where they allowed the PN generation for software programs that was previously infeasible.

Each technique is independent from the others allowing the safety engineer to freely select any combination, as needed by the problem instance at hand.

Fault Injection in Program Netlists We presented techniques to describe and inject fault logic in a formal model by employing saboteurs that allow to enable or disable the activation of specific faults for individual analysis runs. The instruction cell-based fault injection provides a large degree of freedom in describing faults at ISA level.

By example we showed how widely used single-event upset and stuck-at fault models can be injected into program netlists. The inserted fault logic allows to formally analyse individual single-faults, any combination of single-faults and any combination of multiple-faults.

Fault Effect Analysis on ISA level We demonstrated how formal techniques can be employed to analyse the effects of hardware faults in program-visible registers on the software behaviour. Our formal method has the advantage that it can actually *certify* the absence of error effects for a given application.

The conducted experiments showed that for embedded software a large fraction of faults at the ISA level can never have any effect on the software behaviour at all. In analogy to redundant faults in testing, we call such faults *application-redundant*. We ensured realistic results in our experiments by analysing industry-oriented software programs that are publicly available.

An interesting observation is that the nature of untestable stuck-at faults at the application level is different from the nature of stuck-at redundancies in combinational circuits. In realistic circuits typically only a small number of stuck-at faults turns out to be combinationally redundant. Combinational redundancies are usually caused by complex logic dependencies in the circuit

making it impossible to fulfil the requirements for fault controllability and observability simultaneously. They indicate sub-optimal circuitry and are usually removed in a well-optimized circuit. Stuck-at redundancies at the application level, on the other hand, can occur in fairly large numbers. Experimental data obtained from fault injection campaigns with simulation like in [36, 69] support this finding.

At the application level, it is rare that redundancies result from contradictory controllability and observability requirements along all possible program runs. If it occurs this might indicate some optimization potential in the computing platform, similarly like in the case of combinational stuck-at redundancies. We did not encounter any such case in our experiments. Instead, an application-level redundancy most often results from a completely unused hardware resource leading to a fault site where, independently of each other, the requirement of controllability and/or the requirement of observability are violated.

Fault Testability Analysis on the Gate Level We used the results obtained from ISA-level fault effect analysis for a gate-level testability analysis by exploiting the connection between ISA and gate level, as provided by program-visible registers.

The presented deductive method uses knowledge about faults that are uncontrollable and unobservable at application level to reduce the controllability and observability of faults at the gate level. We applied this method at gate level on an Aquarius computing platform that implements the SuperH2 ISA and on an FWRISC-S computing platform implementing the RISC-V ISA. The Aquarius contains a superscalar processor with five pipeline stages, while the processor in FWRISC-S is non-pipelined. Despite the differences we were able to identify a significant number of application-redundant faults that become untestable for a certain software program for both computing platforms.

The work of other research groups support our findings also here. A masking of over 30% of gate-level faults was found in [69]. In [36] an even larger fraction of 72% of the injected gate-level faults were masked. In such cases, the proposed approach can provide full confidence by a formal proof of redundancy. The high number of redundancies that were identified reflects the fact that the given software does not fully exploit all available resources and "features" the hardware provides. Unlike a combinational circuit that is aggressively optimized for a fixed Boolean function, a standard hardware platform is built to run a wide variety of software. It is therefore never perfectly optimized w.r.t. the specific software that is running on it in some application.

Fault Propagation Analysis on C level We concluded this thesis by presenting a scalable and formal approach to analyse fault effects for large hardware/-software systems. One important contribution is that safe faults can be determined with formal rigour, avoiding overly pessimistic estimations and providing more precise classifications.

The basic ingredients of this approach are i) our fault effect analysis at ISA-level, ii) a smart decomposition of the software system and the partitioning

of its state variables and iii) a C-level static analysis based on Abstract Interpretation. We have shown how the problem of fault propagation analysis can be decomposed such that a formally sound compositional approach based on these ingredients can be developed. In particular, two formal methods at different abstraction levels, SAT-based Interval Property Checking and Abstract Interpretation, are combined. At the low level, we conduct a local analysis of machine code and model the occurrence of faults in program-visible registers of the gate-level hardware. Data corruptions are automatically mapped to the C level and their propagation is tracked by a sound taint analysis. Experimental results show that our approach is feasible for embedded systems of realistic size.

9.1 Outlook

The focus of this thesis was to develop a formal and scalable method to analyse the effects of hardware faults across abstraction levels and the hardware/software boundary.

The methods and findings of this thesis can provide the basis for subsequent research projects.

9.1.1 Bespoke Processor Design

The observation from our experiments is that a significant number of faults at the ISA and gate levels are application-redundant, i.e., they do not contribute to the functionality provided by the hardware/software system.

As mentioned in the conclusion on our Fault Testability Analysis, we found that low-level software employed in embedded systems does not fully exploit the resources provided by a general-purpose processor. This indicates a large hardware optimisation potential for embedded systems. Strict requirements for area and power provide motivation to develop a systematic and sound *application-specific* optimisation method for general purpose processors. A fully automated customisation of commercial-off-the-shelf processors can be a faster and less expensive alternative, compared to a fully customised application specific circuit design.

In a recently published work of our research group [95] we developed a formal analysis that can identify application-dependent uncontrollable signals in gate-level processors. As presented in [95], this information can be used to optimise the hardware by removing the identified application-redundant signals.

The next step would be to extend this analysis to identify unobservable faults at RT or gate level to further optimise the hardware design.

9.1.2 Online Error Detection

Most measures to detect errors during the operational lifetime of a hardware/-software system rely on knowledge obtained from CFGs. However, CFGs have the disadvantage that they do not explicitly model individual program paths,

because every instruction is represented only once inside a CFG. CFG-based error detection techniques have, therefore, the intrinsic disadvantage that they are based on a model that over-approximates the space of reachable program paths. This could allow several faults to remain undetected.

Furthermore, recent findings in [87] show that CFG-based online control flow checking methods can actually increase the vulnerability of a hardware/-software system against soft errors.

This may motivate the development of a program netlist-based approach for online control flow checking. We expect that the explicitly exposed program paths in program netlists enable the design of stricter control flow checkers.

A limitation of a program netlist-based approach for online detection might be that data flow is only modelled implicitly in program netlists. Therefore, faults that cause only silent data corruptions and never affect control decisions cannot be detected. A program netlist-based approach can still provide an improvement for existing error detection methods.

The methods presented in this thesis could, then, be used to formally certify the efficacy of the developed online detection methods.

9.1.3 Abstract Interpretation for Program Netlists

The experiments in this thesis demonstrated the scalability of techniques based on Abstract Interpretation. As discussed in Section 8.3, the key concept behind Abstract Interpretation is to replace complex concrete semantics of operations with simpler abstract semantics.

The trade-off is that the abstract semantics lead to some degree of over-approximation that can result in false negatives when applied to property checking. In terms of fault analysis this means that a hardware fault could be identified as having some effect on the software behaviour while it actually does not.

If successfully applied to program netlist-based analyses, e.g., by implementing interval arithmetic [33], Abstract Interpretation can further improve scalability. Note that the program netlist generation itself would benefit from such scalability improvements.

10 KURZFASSUNG

Die Erfolge bei der Entwicklung neuer Fertigungsprozesse in der Halbleiterindustrie erlauben die Produktion von digitalen Schaltungen mit immer kleineren Abmessungen. Dieser Fortschritt ermöglicht es nicht nur, große analoge Komponenten durch kleine digitale Komponenten zu ersetzen, sondern auch völlig neue Funktionen in einem Produkt zu realisieren.

Die Kehrseite dieser Miniaturisierung ist, dass die kleineren und komplexeren Strukturen von Bauelementen, wie z. B. Transistoren, deutlich fehleranfälliger sind. Die Konsequenzen daraus können nicht nur unerwünscht sein, wie zum Beispiel fehlerhaft funktionierendes Infotainment, sondern auch lebensgefährlich, wenn zum Beispiel Industriemaschinen nicht mehr erkennen, ob sich ein Arm oder Bein zwischen beweglichen Teilen befindet.

Die Robustheit gegenüber Hardwarefehlern in digitalen Systemen ist deshalb ein Hauptanliegen bei der Entwicklung von eingebetteten Systeme, in deren Anwendungsgebiet Personen zu Schaden kommen könnten. Eine Reihe von Standards tragen diesem Anliegen Rechnung und fordern für die Gewährleistung der Sicherheit von Personen (im Folgenden kurz „Sicherheit“ genannt) , insbesondere für die höchsten Sicherheitsgrade, eine systematische und gründliche Untersuchung der Systeme.

In der Theorie ließe sich eine Schaltung vielleicht gegenüber jedem möglichen Fehler absichern, in der Praxis wäre das allerdings, falls überhaupt, nur mit gewaltigen Kosten möglich. Unserer Meinung nach sollte sich der Schutz gegen Hardwarefehler stattdessen auf solche konzentrieren die das Systemverhalten in einer Art verändern, dass es unsicher wird.

Bei eingebetteten Systemen, in denen Hardware und Software miteinander interagieren, kann die Berücksichtigung der ausgeführten Softwareprogramme helfen, die Anzahl der zu betrachtenden Fehler deutlich zu reduzieren. Der Grund dafür ist, dass die auf einem Prozessor ausgeführten hardwarenahen Programme, wie z. B. Treiber oder Firmware, meist nicht den gesamten Funktionsumfang des Prozessors verwenden. Hinzu kommt, dass solche Programme in eingebetteten Systemen sehr spezifische Funktionen bereitstellen, welche, falls überhaupt, meist nur geringfügige Änderungen während der Produktlebenszeit des Systems erfahren. Diese Eigenschaft erlaubt es, programmspezifische Schutzmaßnahmen sicherheitskonform einzusetzen.

Im Unterschied zu simulationsbasierten Methoden, wie sie üblicherweise in der Industrie eingesetzt werden, können formale Methoden das gesamte Verhalten eines Hardware/Software-Systems betrachten und sind deshalb in der Lage, die Sicherheit eines eingebetteten Systems zu zertifizieren. Eine auf formalen Methoden basierende Fehleranalyse kann deshalb die zu untersuchen-

den Fehler in zwei Kategorien aufteilen: in der einen Kategorie sind die Fehler, welche die Sicherheit beeinträchtigen können und gegen die deshalb Maßnahmen ergriffen werden müssen, und in der anderen Kategorie sind die Fehler, welche, unter Betrachtung des gesamten Hardware/Software-Verhaltens, niemals die Sicherheit gefährden können und deswegen ignoriert werden können.

Der Forschungsbeitrag dieser Arbeit besteht in der Entwicklung formaler Methoden, mit denen sich:

1. die Fehlerauswirkungen auf das Programmverhalten mittels Fehlerinjektion auf der Ebene der Instruktionssatzarchitektur (ISA) formal analysieren lassen,

2. die auf der ISA-Ebene ermittelte Abwesenheit von Fehlereffekten auf die Gatterebene übertragen lässt und es dann gestattet, deduktiv mittels Testmustergenerierung „applikationsredundante“ Fehler zu identifizieren,

3. die auf der ISA-Ebene ermittelten Fehlerauswirkungen auf Objekte der höheren Softwareebene übertragen und dann induktiv mittels abstrakter Interpretation weitere Softwareobjekte, wie z. B. Sicherheitsfunktionen identifizieren lassen. Das Ergebnis dieser Analyse ist eine 1-zu-n-Relation von Fehlern auf der ISA-Ebene und ihren Auswirkungen auf der C-Ebene.

10.1 Programmnetzlisten

Für die Analyse von Fehlerauswirkungen auf der ISA-Ebene verwenden wir Programmnetzlisten. Diese wurden bereits in [90] vorgestellt, weswegen wir hier nur deren grundlegende Konzepte erklären, bevor wir in den nachfolgenden Abschnitten den Forschungsbeitrag dieser Arbeit vorstellen.

Eine Programmnetzliste modelliert das Verhalten eines Prozessors, welcher ein bestimmtes Softwareprogramm ausführt. Für die Generierung von Programmnetzlisten wird ein Softwareprogramm auf Maschinenebene und eine Beschreibung der Maschinenbefehle auf Architekturebene verwendet. Schleifen im Kontrollfluss des Maschinenprogramms werden abgerollt, um ein azyklisches Modell zu erhalten. Der Kontrollfluss kann zu diesem Zeitpunkt noch unvollständig sein, sodass formale Analysen des bereits abgerollten Teils notwendig sind. Die Ausführung eines Maschinenbefehls wird mit Hilfe von Schaltungsbeschreibungssprachen, wie z. B. SystemVerilog, modelliert. Diese Modelle beschreiben die Veränderungen am Programmzustand, d. h. die Werteänderungen in programmsichtbaren Registern und im Speicher.

10.1.1 Skalierbarkeit von Programmnetzlisten

Um die Erzeugung von Programmnetzlisten auch für größere und komplexere Programme zu ermöglichen, haben wir in dieser Arbeit eine Reihe von Verbesserungen entwickelt, die wir in den nachfolgenden Abschnitten beschreiben. In unseren Experimenten konnten wir zeigen, dass die nachfolgend beschriebenen Techniken die Programmnetzlistenerzeugung signifikant beschleunigen

können. Die Techniken nutzen jeweils bestimmte Eigenschaften von Software, welche in eingebetteten Systemen eingesetzt wird, aus. Die Programmnetzlistenerzeugung für Programme, welche die jeweilige Eigenschaft nicht erfüllen, kann daher nicht von diesen Techniken profitieren, sie wird aber auch nicht davon beeinträchtigt.

Kompositionelle Programmnetzlistenerzeugung Anstatt die Programmnetzlisten monolithisch für das gesamte Softwareprogramm aufzubauen, analysieren wir zunächst den Kontrollflussgraphen eines Programms, um für eine kompositionelle Programmnetzlistenerzeugung geeignete Stellen im Programmfluss zu finden. Funktionsaufrufe sind zum Beispiel geeignete Stellen, an denen sich die Programmnetzlistenerzeugung mit geringem Aufwand auftrennen lässt.

Zunächst wird für jede gewählte Trennstelle, eine eigene Teil-Programmnetzliste erzeugt. Anschließend werden alle Teile zu einer Gesamt-Programmnetzliste zusammengefügt. Um bei dieser Prozedur Überabschätzungen im modellierten Programmvehalten der Teil-Programmnetzlisten zu minimieren, berechnen wir mit Hilfe von *Abstrakter Interpretation* erreichbare Wertebereiche der Variablen im Startzustand der Teil-Programmnetzliste.

Überabschätzungen können im schlimmsten Fall dazu führen, dass die erzeugte Programmnetzliste auch nichterreichbares Programmverhalten modelliert. Für Fehleranalysen bedeutet das, dass einige Fehler, welche eigentlich keine Auswirkungen haben, in der Analyse als Fehler mit Auswirkungen ausgegeben werden (falschpositiv). Falschnegative Ergebnisse sind nicht möglich.

Befehlsabstraktion Komplexe Maschinenbefehle können die während der Programmnetzlistenerzeugung durchgeführten Analysen erschweren. Diese können jedoch durch einfachere Maschinenbefehle ersetzt werden, wenn der oder die Ersatzbefehle das Ausgabeverhalten des Originalbefehls überabschätzen. Auch dadurch kann zusätzliches Programmverhalten hinzukommen, was jedoch, genau wie bei der kompositionellen Programmnetzlistenerzeugung, nur zu falschpositiven und nicht zu falschnegativen Ergebnissen führen kann.

Programmpfadprioritäten Zur Vermeidung von Programmpfadexplosionen, können mehrere Programmpfade zu einem einzigen zusammengefügt werden, sofern dabei keine Schleife entsteht. Es kann jedoch passieren, dass zum Beispiel, wenn an zwei Stellen Programmpfade zusammengefügt werden könnten, es nur an einer der beiden Stellen tatsächlich möglich ist, da ansonsten beim Zusammenfügen an der jeweils anderen Stelle eine Schleife erzeugt werden würde. Durch ungeschickte Wahl der Stellen, an denen Programmpfade zusammengefügt werden, kann es vorkommen, dass andere Pfade einzeln ausgerollt werden müssen. Eine größere Anzahl von Pfaden, welche nicht mit anderen zusammengelegt werden können, kann die Komplexität von Programmnetzlisten stark ansteigen lassen. In einigen Situationen kann suboptimales Zusammenfügen durch Priorisierung einzelner Programmabschnitte vermieden

werden. Ein priorisierter Programmabschnitt muss vollständig abgerollt sein, bevor ein Zusammenlegen mit anderen Programmpfaden erlaubt ist.

Adressspeicher In hardwarenahen Softwareprogrammen greifen Befehle im Maschinencode häufig auf die gleichen Speicheradressen zu. Hierbei sind vor allem diejenigen Befele von Interesse, welche für die Registerwerte zur Berechnung der Speicheradresse verwenden. In der Theorie kann sich der Registerwert bei jeder Befehlsausführung unterscheiden, in der Praxis ändern sich diese Werte bei hardwarenaher Software jedoch häufig nicht. Diese Besonderheit lässt sich für die Erzeugung von Programmnetzlisten ausnutzen, indem gefundene Adressen, auf die ein Befehle zugreift, zwischengespeichert werden. Wenn für den gleichen Befehl wieder die Adressen, auf welche dieser zugreift, ermittelt werden sollen, wird überprüft, ob es die zwischengespeicherten sind, wodurch nicht jede Adresse erneut einzeln ermittelt werden muss.

10.2 Fehlerinjektion

Um mittels Programmnetzlisten Fehlerverhalten modellieren zu können, fügen wir Fehlerinjektionslogik in die Befehle der Programmnetzliste ein, welche die Werte im Programmzustand verändern kann. Die Fehlerinjektionslogik erlaubt es, je nach gewählter Konfiguration, die Auswirkungen von einem Einzelfehler, einer beliebigen Menge an Einzelfehlern oder einer beliebigen Menge an Mehrfachfehlern in einer Programmnetzliste zu modellieren und zu analysieren. Es ist möglich, einzelne oder alle Fehler zu deaktivieren. Sind alle Fehler deaktiviert, modelliert die fehlerinjizierte Programmnetzliste das fehlerfreie Verhalten.

Die in dieser Arbeit verwendeten Fehlermodelle sind stuck-at-Fehler, welche zu der Klasse der permanenten Fehler gehören, und single-event-upsets, die zu der Klasse der temporären Fehler gehören.

10.3 Fehlereffektanalyse

Für die Analyse von Fehlereffekten auf der Architekturebene erzeugen wir zunächst eine Programmnetzliste für ein gegebenes Programm und eine bestimmte Prozessorarchitektur (hier RISC-V und SuperH2). Daran anschließend fügen wir Fehlerinjektionslogik in die Befehle der Programmnetzliste ein und duplizieren die fehlerinjizierte Programmnetzliste. In der einen Programmnetzliste deaktivieren wir alle Fehler, während wir in der anderen nur die Fehler aktivieren, deren Auswirkungen während eines einzelnen Analysedurchlaufs betrachtet werden sollen. Mit der Aktivierung von nur bestimmten Fehlern lässt sich die Granularität der Analyse bezüglich der Relation von Fehlermenge zu Fehlereffekten einstellen.

Unsere Experimente zeigen, dass nicht nur etliche stuck-at-Einzelfehler, sondern auch eine große Menge an stuck-at-Mehrfachfehlern keine Auswirkungen auf das Ausgabeverhalten der analysierten Programme haben. Die ent-

sprechenden Fehler sind also an den Programmausgängen, und damit auch außerhalb des eingebetteten Systems, nicht beobachtbar.

Außerdem konnten wir in unseren Experimenten nachweisen, dass die Werte in etlichen programmsichtbaren Registern vom Softwareprogramm nicht kontrolliert werden können. Das bedeutet, dass es nicht möglich ist, dem betrachteten Programm eine Wertesequenz zu übergeben mit der sich die entsprechenden Programmzustandswerte ändern lassen.

10.4 Fehlertestbarkeitsanalyse

Weil die Fehlereffektanalyse nur Aussagen auf der Architekturebene liefert, ist das Ziel der Fehlertestbarkeitsanalyse herauszufinden, ob es auch auf der Gatterebene Fehler gibt, welche nicht mittels sogenannter Testpattern testbar sind, weil das ausgeführte Programm das Prozessorverhalten zu sehr einschränkt. Testbar bedeutet hier, dass es eine Wertebelegung für die Schaltungseingänge gibt, wodurch ein bestimmter Fehler fehlerhafte Werte an den Schaltungsausgängen erzeugt.

Zu diesem Zweck haben wir eine Methode entwickelt, mit der sich maximale Verträglichkeitsklassen an nicht beobachtbaren stuck-at-Mehrfachfehlern auf der Architekturebene berechnen lassen. Wir verwenden diese Verträglichkeitsklassen, um mittels hinzugefügter Logik die Propagation der entsprechenden Fehler auf der Gatterebene zu unterbinden. Damit lässt sich die Fehlerbeobachtbarkeit für den Prozessor auf Gatterebene einschränken.

Analog dazu lässt sich die Fehlertestbarkeit auch dank der entdeckten konstanten Werte im Programmzustand durch Hinzufügen von Logik weiter einschränken.

Zum Ermitteln der softwareabhängigen testbaren und nicht-testbaren Fehler verwenden wir automatische Testpatterngenerierung (ATPG).

Für die experimentelle Evaluation haben wir die Methode auf den beiden unterschiedlichen Rechnerplattformen Aquarius und FWRISC-S angewandt. Die Aquarius Rechnerplattform implementiert die SuperH2 ISA und besitzt eine fünfstufige Instruktionspipeline, während die FWRISC-S Rechnerplattform die RISC-V ISA implementiert und keine Instruktionspipeline besitzt. Außerdem haben wir sowohl einen Vertreter für datenflussdominierte als auch einen Vertreter für kontrollflussdominierte Softwareprogramme untersucht. Für beide Rechnerplattformen und Softwareprogramme zeigen unsere Experimente, dass auch auf der Gatterebene ein erheblicher Anteil an Fehlern nicht testbar wird, wenn konkrete Programme das Prozessorverhalten einschränken.

10.5 Fehlerpropagationsanalyse

Bei der Fehlerpropagationsanalyse untersuchen wir, wie Fehlereffekte von der Architekturebene in die höheren Softwareebenen, wie z. B. die C-Ebene, und dort in Funktionen, welche für die Sicherheit besonders wichtig sind, hinein propagieren können.

Um dies zu erreichen haben wir eine Fehlereffektanalyse mit single-event-upsets durchgeführt, in der wir untersuchten, welche Fehler sich auf relevante Variablen des Programmzustandes auswirken können. Wir konnten zeigen, dass zahlreiche Programmzustandsvariablen für die Fehlerpropagation irrelevant sind, weil zum Beispiel deren Werte nicht mehr verwendet werden oder weil sie geschützt sind und daher jedes fehlerinduzierte, illegale Überschreiben entdeckt wird.

Die entdeckten Fehlereffekte bezüglich der relevanten Programmzustandsvariablen haben wir an die Fehlerpropagationsanalyse auf der C-Ebene weitergereicht, welche von Forschungspartnern der AbsInt GmbH entwickelt und durchgeführt wurde. Anhand der Fehlerpropagationsanalyse konnten eine Reihe von Laufzeitfehlern, wie zum Beispiel Veränderungen des Kontrollflusses oder Speicherzugriffsverletzungen, identifiziert werden.

In einem abschließenden Experiment konnten wir dank der Kombination von Fehlereffektanalyse und Fehlerpropagationsanalyse diejenigen Fehler identifizieren, deren Effekte eine von uns ausgewählte kritische Funktion beeinträchtigen können. Die Differenzmenge von Fehlerliste und der identifizierten Fehler liefert, dank Einsatz formaler Methoden, die Menge an Fehlern, die sich garantiert nicht auf die ausgewählte kritische Funktion auswirken können.

10.6 Fazit

In dieser Arbeit wurden Methoden entwickelt, mit denen sich die Effekte von Hardwarefehlern auf das Verhalten hardwarenaher Software formal analysieren lässt. Die entwickelten Methoden erlauben es, die Möglichkeit von Fehlereffekten, sowie deren Abwesenheit, über Abstraktionsebenen hinweg zu bestimmen. Der Verifikationsingenieur kann hierbei bestimmen, ob die Effekte eines einzelnen Fehlers oder eine beliebige Kombination mehrerer Fehler betrachtet werden sollen. In unseren Experimenten konnten wir zeigen, dass die Methoden auch für Softwaresysteme mit industrienaher Komplexität skalieren, und, dass ein großer Teil von Hardwarefehlern sich tatsächlich niemals auf das Softwareverhalten auswirken kann.

BIBLIOGRAPHY

[1] AbsInt Angewandte Informatik GmbH, Germany. *AbsInt*. URL: https://www.absint.com.

[2] T. Aitch. *Aquarius: a pipelined RISC CPU*. 2003. URL: http://opencores.org/project,aquarius.

[3] F. E. Allen. "Control flow analysis". In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. DOI: 10.1145/390013.808479.

[4] Amruth Pillai. *RSA*. Accessed: 2021-04-22. URL: https://gist.github.com/AmruthPillai.

[5] Andrew Kiluk. *RSA*. Accessed: 2021-04-13. URL: https://github.com/andrewkiluk/RSA-Library.

[6] T. Arons, E. Elster, et al. "Efficient symbolic simulation of low level software". In: *Proceedings of the conference on Design, automation and test in Europe - DATE*. ACM Press, 2008. DOI: 10.1145/1403375.1403577.

[7] AUTOSAR. *AUTOSAR*. URL: https://www.autosar.org/.

[8] A. Avizienis, J.-C. Laprie, et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: 10.1109/tdsc.2004.2.

[9] A. C. Bagbaba, F. A. da Silva, et al. "Automated Identification of Application-Dependent Safe Faults in Automotive Systems-on-a-Chips". In: *Electronics* 11.3 (Jan. 2022), p. 319. DOI: 10.3390/electronics11030319.

[10] R. Bagnara, A. Bagnara, et al. "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software". In: *Static Analysis*. Springer International Publishing, 2018, pp. 5–23. DOI: 10.1007/978-3-319-99725-4_2.

[11] M. Ballance. *Featherweight RISC-V*. 2021. URL: https://github.com/Featherweight-IP/fwrisc.

[12] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *International Workshop on Resiliency in Embedded Electronic Systems*. 2015.

[13] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*. Ed. by R. Wimmer. Albert-Ludwigs-Universität Freiburg, 2016, pp. 10–21. DOI: 10.6094/UNIFR/10634.

[14] C. Bartsch, N. Rödel, et al. "A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems". In: *2016 17th Latin-American Test Symposium (LATS)*. 2016, pp. 153–158. DOI: 10.1109/LATW.2016.7483356.

[15] C. Bartsch, C. Villarraga, et al. "Efficient SAT/Simulation-based model generation for low-level embedded software". In: *17th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2014, Böblingen, Germany*. Ed. by J. Ruf, D. Allmendinger, et al. Cuvillier, 2014, pp. 147–157.

[16] C. Bartsch, C. Villarraga, et al. "Safety across the HW/SW interface - Can formal methods meet the challenge?" In: *International Symposium on Integrated Circuits, ISIC 2016, Singapore, December 12-14, 2016*. IEEE, 2016, pp. 1–3. DOI: 10.1109/ISICIR.2016.7829707.

[17] C. Bartsch, C. Villarraga, et al. "A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems". In: *Journal of Electronic Testing* 33.1 (2017), pp. 77–92. DOI: 10.1007/s10836-017-5643-3.

[18] C. Bartsch, S. Wilhelm, et al. "Compositional Fault Propagation Analysis in Embedded Systems using Abstract Interpretation". In: *2021 IEEE International Test Conference (ITC)*. 2021.

[19] C. Bartsch, S. Wilhelm, et al. "Combining Fault Effect Analysis and Fault Propagation Analysis to Determine Source Code Level Effects of Hardware Faults". In: *Embedded World*. WEKA FACHMEDIEN GmbH, 2022.

[20] M. A. Ben Khadra. "Techniques For Efficient Binary-Level Coverage Analysis". en. PhD thesis. 2021. DOI: 10.26204/KLUEDO/6410.

[21] C. Bernardeschi, A. Fantechi, et al. "Model checking fault tolerant systems". In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 251–275.

[22] A. Bernardini, W. Ecker, et al. "Efficient handling of the fault space in functional safety analysis utilizing formal methods". In: *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, Sept. 2016. DOI: 10.1109/vlsi-soc.2016.7753546.

[23] A. Bernardini, W. Ecker, et al. "Where formal verification can help in functional safety analysis". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–8. DOI: 10.1145/2966986.2980087.

[24] A. Biere, A. Cimatti, et al. "Symbolic Model Checking Without BDDs". In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207. ISBN: 3-540-65703-7. DOI: 10.1007/3-540-49059-0_14.

[25] R. E. Bryant. "Symbolic Simulation—Techniques and Applications". In: *Conference proceedings on 27th ACM/IEEE design automation conference - DAC*. ACM Press, 1990. DOI: 10.1145/123186.128296.

[26] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer US, 2002. DOI: 10.1007/b117406.

[27] C. Cadar and K. Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782.

[28] W. Carter, W. Joyner, et al. "Symbolic Simulation for Correct Machine Design". In: *16th Design Automation Conference*. IEEE, 1979. DOI: 10.1109/dac.1979.1600119.

[29] C.-Y. Cher, K. P. Muller, et al. "Soft error resiliency characterization and improvement on IBM BlueGene/Q processor using accelerated proton irradiation". In: *International Test Conference*. 2014, pp. 1–6. DOI: 10.1109/TEST.2014.7035317.

[30] N. Cohen. *Samsung at foundry event talks about 3nm, MBCFET developments*. May 10, 2022. URL: https://techxplore.com/news/2019-05-samsung-foundry-event-3nm-mbcfet.html.

[31] C. Constantinescu, M. Butler, et al. "Error injection-based study of soft error propagation in AMD Bulldozer microprocessor module". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 2012, pp. 1–6. DOI: 10.1109/DSN.2012.6263922.

[32] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proc. of POPL'77*. Los Angeles, CA: ACM Press, 1977, pp. 238–252.

[33] P. Cousot. "Abstract Interpretation". In: *ACM computing surveys* 28.2 (June 1996), pp. 324–328. ISSN: 0360-0300. DOI: 10.1145/234528.234740.

[34] V. D'Silva, D. Kroening, et al. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. DOI: 10.1109/tcad.2008.923410.

[35] A. Darbari, B. A. Hashimi, et al. "A New Approach for Transient Fault Injection Using Symbolic Simulation". In: *14th IEEE International On-Line Testing Symposium*. 2008, pp. 93–98.

[36] J. M. Daveau, A. Blampey, et al. "An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip". In: *IEEE International Reliability Physics Symposium*. Apr. 2009, pp. 212–220.

[37] U. D. of Defense. *Procedures for Performing a Failure Mode Effects and Criticality Analysis*. 1949.

[38] S. E. Diehl, J. E. Vinson, et al. "Considerations for Single Event Immune VLSI Logic". In: *IEEE Transactions on Nuclear Science* 30.6 (1983), pp. 4501–4507. DOI: 10.1109/tns.1983.4333161.

[39] P. Dodd, M. Shaneyfelt, et al. "Production and propagation of single-event transients in high-speed digital logic ICs". In: *IEEE Transactions on Nuclear Science* 51.6 (Dec. 2004), pp. 3278–3284. DOI: 10.1109/tns.2004.839172.

[40] M. Ebrahimi, L. Chen, et al. "CLASS: Combined logic and architectural soft error sensitivity analysis". In: *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. 2013, pp. 601–607.

[41] L. Entrena, M. Garcia-Valderas, et al. "Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection". In: *IEEE Transactions on Computers* 61.3 (2012), pp. 313–322. ISSN: 0018-9340.

[42] V. Ferlet-Cavrois, L. W. Massengill, et al. "Single Event Transients in Digital CMOS—A Review". In: *IEEE Transactions on Nuclear Science* 60.3 (June 2013), pp. 1767–1790. DOI: 10.1109/tns.2013.2255624.

[43] J. Fritzsch, T. Schmid, et al. "Experiences from Large-Scale Model Checking: Verifying a Vehicle Control System with NuSMV". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2021. DOI: 10.1109/icst49551.2021.00049.

[44] J. Giet, L. Mauborgne, et al. "Towards Zero Alarms in Sound Static Analysis of Finite State Machines". In: *Computer Safety, Reliability, and Security (SafeComp)*. Springer, 2019, pp. 3–18. ISBN: 978-3-030-26601-1.

[45] T. Given-Wilson, N. Jafri, et al. "An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, Aug. 2017. DOI: 10.1109/trustcom/bigdatase/icess.2017.250.

[46] L. Goldstein. "Controllability/observability analysis of digital circuits". In: *IEEE Transactions on Circuits and Systems* 26.9 (Sept. 1979), pp. 685–693. DOI: 10.1109/tcs.1979.1084687.

[47] J. Gracia, J. Baraza, et al. "Comparison and application of different VHDL-based fault injection techniques". In: *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. IEEE Comput. Soc, 2001. DOI: 10.1109/dftvs.2001.966775.

[48] J. Gracia-Moran, J. Baraza-Calvo, et al. "Effects of Intermittent Faults on the Reliability of a Reduced Instruction Set Computing (RISC) Microprocessor". In: *IEEE Transactions on Reliability* 63.1 (2014), pp. 144–153. ISSN: 0018-9529.

[49] P. Gupta, A. Kahng, et al. "Routing-aware scan chain ordering". In: *Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference*. IEEE, 2003. DOI: 10.1109/aspdac.2003.1195137.

[50] J. Guyomarc'h and J.-B. Hervé. "Static and Verifiable Memory Partitioning for Safety-Critical Systems". In: *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 79–84. DOI: 10.1109/ISSREW51248.2020.00041.

[51] ISO 26262. *Road vehicles – Functional safety – Part 5: Product development at the hardware level*. 2018.

[52] ISO 26262. *Road vehicles – Functional safety – Part 6: Product development at the software level*. 2018.

[53] R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Springer US, 2002. DOI: 10.1007/978-1-4615-1101-4.

[54] D. Kästner and C. Ferdinand. "Efficient verification of non-functional safety properties by abstract interpretation: Timing, stack consumption, and absence of runtime errors". In: *Proceedings of the 29th International System Safety Conference ISSC2011, Las Vegas*. 2011.

[55] D. Kästner, B. Schmidt, et al. "Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software". In: *Proceedings of the SAE World Congress 2019*. SAE International, 2019.

[56] D. Kästner. "Applying Abstract Interpretation to Demonstrate Functional Safety". In: *Formal Methods Applied to Industrial Complex Systems*. John Wiley & Sons, Inc., July 2014, pp. 191–234. ISBN: 9781119004707. DOI: 10.1002/9781119004707.ch8.

[57] D. Kästner and C. Ferdinand. "Proving the Absence of Stack Overflows". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 202–213. DOI: 10.1007/978-3-319-10506-2_14.

[58] D. Kästner, L. Mauborgne, et al. "High-Precision Sound Analysis to Find Safety and Cybersecurity Defects". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Toulouse, France, 2020. URL: https://hal.archives-ouvertes.fr/hal-02479217.

[59] D. Kästner, M. Pister, et al. "Confidence in Timing". In: *SAFECOMP 2013 - Workshop SASSUR (Next Generation of System Assurance Approaches for Safety-Critical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*. Toulouse, France: HAL, 2013. URL: http://hal.archives-ouvertes.fr/SAFECOMP2013-SASSUR/hal-00848489.

[60] D. Kästner and J. Pohland. "Program Analysis on Evolving Software". In: *CARS 2015 - Critical Automotive applications: Robustness & Safety*. Ed. by M. Roy. Paris, France, 2015. URL: https://hal.archives-ouvertes.fr/hal-01192985.

[61] D. Kästner and S. Wilhelm. "Generic Control Flow Reconstruction from Assembly Code". In: *Proceedings of the joint conference on Languages, compilers and tools for embedded systems software and compilers for embedded systems - LCTES/SCOPES '02*. ACM Press, 2002, pp. 46–55. DOI: 10.1145/513829.513839.

[62] J. Kloos, T. Hussain, et al. "Risk-Based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis". In: *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Mar. 2011. DOI: 10.1109/icstw.2011.90.

[63] P. Kocher, J. Horn, et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019. DOI: 10.1109/sp.2019.00002.

[64] kokke. *AES*. Accessed: 2021-04-30. URL: https://github.com/kokke/tiny-AES-c.

[65] W. Kunz, J. Marques-Silva, et al. "Sat and ATPG: Algorithms for Boolean Decision Problems". In: *Logic Synthesis and Verification*. Springer US, 2002, pp. 309–341. DOI: 10.1007/978-1-4615-0817-5_12.

[66] J. Lahtinen. "Verification of Fault-Tolerant System Architectures Using Model Checking". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 195–206. DOI: 10.1007/978-3-319-10557-4_23.

[67] D. Larsson and R. Haehnle. "Symbolic Fault Injection". In: *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21*. Vol. 259. 2007, pp. 85–103.

[68] H. M. Le, V. Herdt, et al. "Resilience evaluation via symbolic fault injection on intermediate code". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2018. DOI: 10.23919/date.2018.8342123.

[69] M. L. Li, P. Ramachandran, et al. "Accurate microarchitecture-level fault modeling for studying hardware faults". In: *IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 105–116.

[70] M. Lipp, M. Schwarz, et al. "Meltdown: Reading Kernel Memory from User Space". In: (2018), pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[71] S. Marchese and J. Grosse. "Formal Fault Propagation Analysis that Scales to Modern Automotive SoCs". In: *DVCON Europe*. 2017.

[72] R. Mariani, G. Boschi, et al. "Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508". In: *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, Apr. 2007. DOI: 10.1109/date.2007.364641.

[73] MathWorks. *Fixed-Point Fuel Rate Control System*. Accessed: 2021-04-02. URL: https://www.mathworks.com/help/fixedpoint/ug/fixed-point-fuel-rate-control-system.html.

[74] D. Mavis and P. Eaton. "Soft error rate mitigation techniques for modern microcircuits". In: *IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No.02CH37320)*. IEEE, 2002. DOI: 10.1109/relphy.2002.996639.

[75] A. Miné. "Static analysis of run-time errors in embedded real-time parallel C programs". In: *Logical Methods in Computer Science (LMCS)* 8.26 (1 2012), p. 63.

[76] A. Miné and D. Delmas. "Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software". In: *2015 International Conference on Embedded Software (EMSOFT)*. IEEE, Oct. 2015. DOI: 10.1109/emsoft.2015.7318261.

[77] MISRA. *MISRA-C*. URL: https://www.misra.org.uk/misra-c/.

[78] M. D. Nguyen, M. Thalmaier, et al. "Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants". In: *IEEE Transactions on Computer-Aided Design* 27.11 (Nov. 2008), pp. 2068–2082.

[79] OneSpin Solutions GmbH a Siemens Business. *OneSpin 360 DV-Verify*. URL: https://www.onespin.com/products/360-dv-verify/.

[80] G. Papadimitriou and D. Gizopoulos. "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, June 2021. DOI: 10.1109/isca52012.2021.00075.

[81] C. S. Păsăreanu and W. Visser. "A survey of new trends in symbolic execution for software testing and analysis". In: *Int. J. Softw. Tools Technol. Transf.* 11.4 (Oct. 2009), pp. 339–353. ISSN: 1433-2779.

[82] K. Pattabiraman, N. M. Nakka, et al. "SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework". In: *IEEE Transactions on Computers* 62.11 (Nov. 2013), pp. 2292–2307. DOI: 10.1109/tc.2012.219.

[83] Radio Technical Commission for Aeronautics. *RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[84] Radio Technical Commission for Aeronautics. *RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A*. 2011.

[85] L. Rashid, K. Pattabiraman, et al. "Characterizing the Impact of Intermittent Hardware Faults on Programs". In: *IEEE Transactions on Reliability* 64.1 (2015), pp. 297–310. ISSN: 0018-9529.

[86] Renesas Electronics Corporation TYO. *SH-1/SH-2/SH-DSP Software Manual, Rev. 5.0*. 2005. URL: http://www.renesas.com/.

[87] A. Rhisheekesan, R. Jeyapaul, et al. "Control Flow Checking or Not? (For Soft Errors)". In: 18.1 (Feb. 2019). ISSN: 1539-9087. DOI: 10.1145/3301311.

[88] X. Rival and K. Yi. *Introduction to Static Analysis*. MIT Press Ltd, Feb. 11, 2020. 320 pp. ISBN: 0262043416.

[89] H. Schirmeier, M. Hoffmann, et al. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *11th European Dependable Computing Conference (EDCC)*. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.

[90] B. Schmidt, C. Villarraga, et al. "A New Formal Verification Approach for Hardware-dependent Embedded System Software". In: *IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013)* 6 (2013), pp. 135–145.

[91] C.-J. H. Seger and R. E. Bryant. "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories". In: *Formal Methods in System Design* 6.2 (Mar. 1995), pp. 147–189. ISSN: 0925-9856. DOI: 10.1007/bf01383966.

[92] S. A. Seshia, W. Li, et al. "Verification-Guided Soft Error Resilience". In: *Design Automation and Test in Europe*. 2007, pp. 1–6.

[93] S. Shazli and M. Tahoori. "Using Boolean satisfiability for computing soft error rates in early design stages". In: *Microelectronics Reliability* 50.1 (Jan. 2010), pp. 149–159. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2009.08.006.

[94] F. A. da Silva, A. C. Bagbaba, et al. "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs". In: *IEEE European Test Symposium (ETS)*. 2020, pp. 1–6. DOI: 10.1109/ETS48528.2020.9131568.

[95] S. G. Sørensen, C. Bartsch, et al. "Generation of Formal CPU Profiles for Embedded Systems". In: *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, Oct. 2022.

[96] W. Steiner, J. Rushby, et al. "Model checking a fault-tolerant startup algorithm: from design exploration to exhaustive fault simulation". In: *International Conference on Dependable Systems and Networks*. 2004, pp. 189–198.

[97] C. E. Stroud. *A Designer's Guide to Built-In Self-Test*. Kluwer Academic Publishers, 2002. DOI: 10.1007/b117480.

[98] P. Subramanyan, S. Malik, et al. "Verifying information flow properties of firmware using symbolic execution". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 337–342.

[99] M. Syal and M. S. Hsiao. "New techniques for untestable fault identification in sequential circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.6 (June 2006), pp. 1117–1131. DOI: 10.1109/tcad.2005.855967.

[100] Synopsys Inc. *Design Compiler User Guide*. 2010.

[101] Synopsys Inc. *TetraMAX ATPG User Guide*. 2013.

[102] The SIR Project. *Software-Artifact Infrastructure Repository*. Accessed: 2015-09-01. URL: http://sir.unl.edu.

[103] H. Theiling. "Extracting Safe and Precise Control Flow from Binaries". In: *Proc. International Conference on Real-Time Systems and Applications (RTCSA)*. 2000, pp. 23–30. DOI: 10.1109/rtcsa.2000.896367.

[104] Tom's Hardware. *842 Chips Per Second: 6.7 Billion Arm-Based Chips Produced in Q4 2020*. Accessed: 2022-10-22. URL: https://www.tomshardware.com/news/arm-6-7-billion-chips-per-quarter.

[105] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

[106] C. Urban, S. Ueltschi, et al. "Abstract Interpretation of CTL Properties". In: *Static Analysis*. Springer International Publishing, 2018, pp. 402–422. DOI: 10.1007/978-3-319-99725-4_24.

[107] C. Villarraga, B. Schmidt, et al. "An Equivalence Checker for Hardware-Dependent Software". In: *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2013, pp. 119–128.

[108] C. Villarraga, B. Schmidt, et al. "Software in a Hardware View: New Models for HW-dependent Software in SoC Verification and Test (Invited Paper)". In: *Proc. International Test Conference (ITC'14)*. 2014.

[109] A. Waterman and K. Asanović. *The RISC-V Instruction Set Manual, Version 20191213*. 2019. URL: https://riscv.org.

[110] H. Watson and M. Mearns. *Fault Tree Analysis*. 1962.

[111] M. Wedler, D. Stoffel, et al. "Normalization at the arithmetic bit level". In: *Proceedings of the 42nd annual conference on Design automation - (DAC-05)*. ACM Press, 2005. DOI: 10.1145/1065579.1065699.

[112] M. Williams and J. Angell. "Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic". In: *IEEE Transactions on Computers* C-22.1 (Jan. 1973), pp. 46–60. DOI: 10.1109/t-c.1973.223600.

[113] H.-J. Wunderlich, ed. *Models in Hardware Testing*. Springer Netherlands, 2010. DOI: 10.1007/978-90-481-3282-9.

ACRONYMS

AES Advanced Encryption Standard. 47

AS Architectural State. 18

ASIL Automotive Safety Integrity Level. 30

ATPG Automated Test Pattern Generation. 34

AUTOSAR Automotive Open System Architecture. 29

BMC Bounded Model Checking. 21

CFE Control Flow Error. 85

CFG Control Flow Graph. 31

CMOS Complementary Metal-Oxide-Semiconductor. 16

COI Cone-of-Influence. 51

CPU Central Processing Unit. 2

DSET Digital Single-Event Transient. 26

EXG Execution Graph. 41

FEA Fault Effect Analysis. 61

FINE PN Fault-Injected Program Netlist. 54

FMEA Failure Mode Effect Analysis. 31

FPA Fault Propagation Analysis. 79

FSM Finite State Machine. 70

FTA Fault Tree Analysis. 31

FTEA Fault Testability Analysis. 70

GCC Gnu Compiler Collection. 47

HDL Hardware Description Language. 16

HW/SW Hardware/Software. 2

I/O Input/Output. 2

IC Instruction Cell. 42

IP Intellectual Property. 32

IPC Interval Property Checking. 21

ISA Instruction Set Architecture. 17

LIN Local Interconnect Bus. 46

LoC Lines of C Code. 47

LSB Least Significant Bit. 12

MISRA Motor Industry Software Reliability Association. 29

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor. 16

MSB Most Significant Bit. 12

PN Program Netlist. 39

PS Program State. 42

RAM Random Access Memory. 12

ROM Read-Only Memory. 12

RSA Rivest–Shamir–Adleman. 46

RTL Register Transfer Level. 16

SAT Satisfiability. 21

SET Single-Event Transient. 26

SEU Single-Event Upset. 26

TCAS Traffic Alert and Collision Avoidance System. 50

LIST OF FIGURES

LIST OF TABLES

LEBENSLAUF

Name: Christian Bartsch
Geburtsort: Moers, Deutschland

Werdegang

2013 - 2022	:	Doktorand im Fachbereich Elektro- & Informationstechnik Technische Universität Kaiserslautern, Deutschland
2007 - 2013	:	Abschluss: Diplom-Ingenieur Studium Elektro- & Informationstechnik Technische Universität Kaiserslautern, Deutschland
2004 - 2006	:	Studium Mathematik (Diplom) Westfälische Wilhelms-Universität, Münster, Deutschland
2001 - 2004	:	Abschluss: Allgemeine Hochschulreife Mercator Berufskolleg, Moers, Deutschland
1995 - 2001	:	Abschluss: Fachoberschulreife Heinrich-Pattberg Realschule, Moers, Deutschland