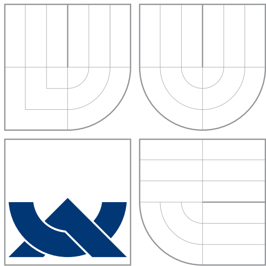
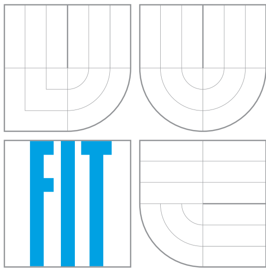


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÁ KONSTRUKCE HLÍDACÍCH OBVODŮ ZALOŽENÝCH NA KONEČNÝCH AUTOMATECH

AUTOMATIC CONSTRUCTION OF CHECKING CIRCUITS BASED ON FINITE AUTOMATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUCIE MATUŠOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této práce bylo studium aktivního učení automatů, navržení a implementace softwarové architektury pro automatickou konstrukci hlídacího obvodu dané jednotky implementované v FPGA a ověření funkčnosti hlídacího obvodu pomocí injekce poruch. Hlídací obvod, tzv. online checker, má za úkol zabezpečovat danou jednotku proti poruchám. Checker je konstruován z modelu odvozeného pomocí aktivního učení automatů, které probíhá na základě komunikace se simulátorem. Pro implementaci učícího prostředí byla použita knihovna LearnLib, která poskytuje algoritmy aktivního učení automatů a jejich optimalizace. Byla navržena a implementována experimentální platforma umožňující řízenou injekci poruch do designu v FPGA, která slouží k otestování checkeru. Výsledky experimentů ukazují, že při použití checkeru a rekonfigurace je možné snížit chybovost designu o více než 98%.

Abstract

The aim of this thesis was to study active automata learning, to design and implement a software architecture for the automatic construction of a checking circuit for a given unit implemented in FPGA, and to verify the functionality of the checking circuit by fault injection. The checking circuit, denoted as an online checker, introduces fault tolerance aspects to the unit. The checker is constructed from a model inferred by active automata learning, which is based on communication with a simulator. To implement the learning environment, LearnLib library has been employed. It provides active automata learning algorithms and their optimizations. An experimental platform enabling controlled fault injection into a design in FPGA was designed and implemented. The platform was used to test the capabilities of the obtained checker. The experimental results show that the error rate is reduced by more than 98% if the checker and reconfiguration is used.

Klíčová slova

hlídací obvod, online checker, aktivní učení automatů, Mealyho automat, SEU injekce

Keywords

checking circuit, online checker, active automata learning, Mealy Machine, SEU injection

Citace

Lucie Matušová: Automatic Construction of Checking Circuits Based on Finite Automata, diplomová práce, Brno, FIT VUT v Brně, 2014

Automatic Construction of Checking Circuits Based on Finite Automata

Declaration

I declare that I have worked on this thesis independently under the supervision of Ing. Jan Kaštil.

....
Lucie Matušová
May 27, 2014

Acknowledgments

I would like to thank my supervisor Ing. Jan Kaštil for his patient guidance and valuable advice. I would also like to express my deepest gratitude to my family and my boyfriend, Tomáš Korec, for their help, encouragement, and support during my studies. I could not have done it without you.

© Lucie Matušová, 2014.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction 4

2 Preliminaries 6

3 Fault Tolerance 8
- 3.1 System Dependability 8
- 3.2 Redundancy 9
  - 3.2.1 Duplex Systems 9
  - 3.2.2 N-modular Redundancy 9
- 3.3 Field-Programmable Gate Arrays 10
  - 3.3.1 FPGA Configuration 10
  - 3.3.2 Self-checking Functional Units 11
  - 3.3.3 Checking Circuits 13

4 Active Automata Learning 15
- 4.1 Learning DFAs 15
  - 4.1.1 MAT Model 16
  - 4.1.2 L^* Algorithm 16
- 4.2 Distinguishing Input and Output 17
  - 4.2.1 Learning Mealy Machines 18
  - 4.2.2 Register Automata and Register Mealy Machines 20
- 4.3 Learning in Practice 20
- 4.4 LearnLib 21
  - 4.4.1 Inferring Models 21
  - 4.4.2 Query Optimizations 22

5 Design and Implementation 24
- 5.1 Learning Platform 24
  - 5.1.1 Simulator 24
  - 5.1.2 Learning Environment 26
  - 5.1.3 FSM-VHDL Converter 28
- 5.2 Experimental Platform 29
  - 5.2.1 Design with Checker 29
  - 5.2.2 Connection with Board 31
  - 5.2.3 Experiment Controller 32
  - 5.2.4 SEU Generator 32

6 Experiments **34**
6.1 Inference of Models 34
6.2 Checker Implementation 37
6.3 SEU Injections 38

7 Conclusions and Future Work **41**

List of Figures

3.1 Possible propagation of the fault in the system 8
3.2 Duplex system 9
3.3 TMR, three identical functional units and voter 10
3.4 Frame address parts [40] 11
3.5 Row addresses [40] 12
3.6 Fault tolerant architectures based on PRMs [25] 13
3.7 Online checker 13

4.1 MAT model 16
4.2 An example Mealy machine 19
4.3 A hypothesized Mealy machine 20
4.4 Schematic view of active learning setup using a mapper 21
4.5 Extrapolating behavioral models with LearnLib [24] 22
4.6 Position of a filter in learning 22

5.1 The main components of the learning platform 24
5.2 Master and Slave Interfaces 25
5.3 The slave interface of PEU 26
5.4 The main components of the learning environment 26
5.5 Scheme of the experimental platform 29
5.6 The checker 29
5.7 Design with the checker 30
5.8 Constraining the groups of the design in PlanAhead 31
5.9 Structure of the SEU generator 32

6.1 Model A (4 states and 32 transitions) 37
6.2 Model B (18 states and 144 transitions) 38
6.3 Major spots where SEU can strike 39

Chapter 1

Introduction

Complex digital systems in critical applications such as health, space, or aircraft demand a high reliability. Today's trends of manufacturing VLSI systems show increasing logic density and reducing power consumption. Transistors are getting smaller and more of them can be packed onto a chip. However, smaller dimensions have effects on the systems reliability. Devices are more sensitive to noise and radiation, and thus vulnerable to faults.

High reliability can be achieved by employing fault tolerance. The system is fault tolerant, if it can operate correctly despite the presence of faults. This work is aimed at the method based on *online checkers*. The checker is connected to a functional unit and serves as a means of error detection. A method of automatic checker construction by employing *active automata learning* techniques is proposed.

Active learning is a subfield of machine learning. To explain what is active learning, best is to put it in contrast with passive learning. In passive learning, a learner learns through observing the environment which generates a training data. On the other hand, the learner is not only an observer in active learning. It interacts with the environment by executing actions and thus influences the generation of training data.

Active automata learning aims at inferring automaton model of a system. It is nowadays getting significant importance thanks to its applicability on software engineering problems. Active automata learning can be considered a key technology for dealing with systems where no or little knowledge about the internal structure is available, or when addressing real-world legacy systems. Then the system is approached as a *black box* given only information about the interface.

The aim of the thesis is to apply active automata learning techniques to a given VHDL design and use the inferred model as an online checker. For this purpose, a learning platform was developed. The platform connects an VHDL simulator, a learning environment, and a converter of the automaton. To infer the model of the given functional unit, Mealy Machine learning is employed. The Mealy Machine is an automata model which is able to capture input/output behaviour. The checker obtained by conversion of the inferred model focuses on checking the control signals of the functional unit whereas checking data and address signals is omitted.

To evaluate the function of the checker, it is instantiated in a design which is implemented to a Field-Programmable Gate Array on an experimental platform. The experimental platform enables fault injection to the design and provides numbers of occurred errors.

The thesis is organized as follows.

Chapter 2 covers definitions of terms from automata theory which are important for

comprehension of the active automata learning. Chapter 3 is designated to fault tolerance. Field-Programmable Gate Arrays and their configuration are described. At the end of the chapter, the focus is on the checking circuits and the related methodologies. Chapter 4 looks at the topic of active automata learning. It embraces major learning approaches, algorithms, and the description of LearnLib library. Chapter 5 describes the design and implementation of the learning platform and the experimental platform. The unit which the checker is constructed for is introduced and its learning setup is devised. Chapter 6 presents experiments with the learning and its results. It contains analysis of inferred automata and constructed checkers. The experiments with checkers under fault injections are described and evaluated. Finally, Chapter 7 concludes the thesis by summarizing the contributions and suggests the possible further direction of the work.

Chapter 2

Preliminaries

The aim in this chapter is to acquaint the reader with the definitions of the terms from automata theory and notions used in the thesis. The chapter does not attempt to cover the entire automata theory nor formal proofs. It rather serves as a concise base for Chapter 4. If already familiar with basic concepts, the reader may choose to skip to the next chapter and return when some notion is unclear.

Definition 1. A *finite automaton* is defined as a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet of input symbols,
- δ is a mapping $Q \times \Sigma \to 2^Q$ (transition function),
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is a set of final states.

Definition 2. If $\delta : Q \times \Sigma \to Q$, M is a *deterministic finite automaton* (DFA).

When we are talking about the computational model, we use the term *automata*. In hardware designs, it is *finite state machine* (FSM).

Definition 3. *Input word* is a string of symbols $a_1, a_2, ..., a_n$, where $a_i \in \Sigma$. The set of all words over alphabet Σ is denoted as Σ^*.

Definition 4. By $q \xrightarrow{a} q'$ we denote a *transition* $\delta(q, a) = q'$. We write $q \overset{w}{\Longrightarrow} q'$ if for $w = a_1 \cdots a_n$ there is a sequence of states $q = q^0, q^1, ..., q^n = q'$ such that $q^{i-1} \xrightarrow{a_i} q^1$ for $1 \leq i \leq n$. We say that w reaches q' from q. If there exists such a w and $q = q_0$, then q' is called *reachable*.

Definition 5. The *language* $L(M)$ *accepted* by automaton M is a set of input words accepted by automaton M: $L(M) = \{w | q_0 \overset{w}{\Longrightarrow} q, w \in \Sigma^*, q \in F\}$.

Definition 6. An *equivalence relation* $\equiv$ is a binary relation which is reflexive, symmetric, and transitive.

Theorem 1. For two equivalence relations R_1 and R_2, R_1 refines R_2 iff $R_1 \subseteq R_2$.

Definition 7. An equivalence relation $\equiv$ on Σ^* is *right invariant* iff $u \equiv v \Longrightarrow uw \equiv vw$ for every $u, v, w \in \Sigma^*$.

That means, equivalence relation has the right invariant property if two equivalent strings still are equivalent when the third string is appended to the right of both of them.

In [19], Nerode provided a necessary and sufficient conditions for a language to be regular, thereby established a link between regular languages and the deterministic finite automata. In the following, we consider $L \subseteq \Sigma^*$.

Definition 8. (Nerode equivalence) We say that two words $u, v \in \Sigma^*$ are *L-equivalent*, written $u \equiv_L v$, iff for all words $w \in \Sigma^*$, we have $uw \in L$ iff $vw \in L$.

In other words, the strings u and v are equivalent under $\equiv_L$ if, whenever we append the same string w to both of them, the resulting strings are both in L or both not in L.

Definition 9. $[u]$ is the *equivalence class* of $u \in \Sigma^*$ defined as: $[u] = \{v \in \Sigma^* | u \equiv_L v\}$. The number of equivalence classes of $\equiv_L$ is called *index* of the language L.

Theorem 2. (Myhill-Nerode theorem) The following three statements are equivalent:

1. The language L is accepted by DFA.
2. L is the union of some of the equivalence classes of a right invariant equivalence relation on Σ^* of finite index.
3. The equivalence relation $\equiv_L$ has finite index.

Therefore, a DFA M for regular language L, where $\equiv_L$ has finite index, can be constructed from $\equiv_L$ [13]. There is exactly one state $q_{[w]}$ for each equivalence class $[w]$ of $\equiv_L$. The initial state is set to $q_{[\epsilon]}$ and then transitions by one-letter extensions are formed, e.g. $q_{[u]} \xrightarrow{a} q_{[ua]}$. A state accepts if $[u] \subseteq L$. There is no other DFA that accepts L, yet has fewer states than M. M is unique up to isomorphism and we call it *canonical* DFA for language L [23].

Definition 10. A generalized *Mealy machine* is defined as a tuple $M = (Q, q_0, \Sigma, \Omega, \delta, \lambda)$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a finite input alphabet,
- Ω is a finite output alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function, and
- $\lambda : Q \times \Sigma \to \Omega^*$ is the output function.

We write $q \xrightarrow{a/o} q'$ to express that on input symbol a the Mealy machine makes move from the state q to q' and produces output o. An extension of δ and λ dealing with words is defined as follows: $\delta^* : Q \times \Sigma^* \to Q$ and $\lambda^* : Q \times \Sigma^* \to \Omega$ respectively.

An intuitive interpretation of a generalized Mealy machine is following. It evolves through states $q \in Q$. It is possible to give inputs to the machine. By applying a symbol $a \in \Sigma$, the machine moves to a new state determined by $\delta(q, a)$ and puts a sequence of symbols $\lambda(q, a)$ on the output.

Chapter 3

Fault Tolerance

Over the last decade, the complexity of digital systems has increased dramatically. As the dimensions are getting smaller, the systems are becoming more vulnerable to transient faults caused by radiation or electromagnetic pulse. In critical applications like aerospace or automotive, high reliability is required. One of the ways how to achieve the reliability requirements is to design systems which can cope with faults.

In this chapter, we elucidate the *fault tolerance* from the point of the thesis. For better comprehension, we first present a basic set of definitions relating to dependability, its attributes, threats, and means for its achievement as explicated in [5]. Then, we present Field Programmable Gate Arrays (FPGAs), fault tolerant methodology of self-checking units, and the idea of checking circuits.

3.1 System Dependability

A *system* is an entity that interacts with other entities. The system implements its intended function, the *service*, which is described as a sequence of states.

When the system does not deliver the specified service, *failure* occurs. A system failure means that at least one external state of the system deviates from the correct one. The deviation is called an *error* and its reason is a *fault* (Figure 3.1). When the fault is *active*, it produces error. Otherwise it is *dormant*. Not every fault has to result in an error and not every error causes a system failure.

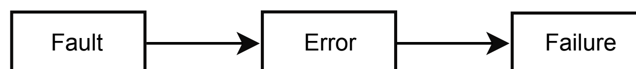


Figure 3.1: Possible propagation of the fault in the system

We classify the hardware faults by duration into three categories: a *permanent* fault which persists until the faulty component is repaired; a *transient* fault occurs for a short duration of time and goes away; and an *intermittent* fault oscillating between being active and dormant.

One of the fundamental properties of a system is a *dependability* defined as the ability to avoid service failures that are more frequent and severe than is acceptable. It comprises five attributes:

- **reliability**: continuity of correct service.

- **availability**: readiness for correct service.
- **integrity**: absence of improper system alterations.
- **safety**: absence of catastrophic consequences on the user and the environment.
- **maintainability**: ability to undergo modifications and repairs.

By *fault tolerance* we refer to a means of avoiding service failures in the presence of faults.

3.2 Redundancy

Fault tolerance in hardware is usually achieved by incorporating *hardware redundancy* which means the use of additional resources. There are three types of hardware redundancy: *static*, *dynamic*, and *hybrid* redundancy.

Static redundancy provides fault mitigation by *masking* them. The faults are hidden and there is no warning about them. This can be done by using the concept of N-modular redundancy. Dynamic redundancy usually comprises fault detection and recovery. A design consists of several functional units (FUs) from which only single one is operating at a time. In case of fault detection, the faulty FU is replaced by a spare one. Hybrid redundancy combines both previous approaches.

3.2.1 Duplex Systems

The simplest example of redundancy is a *duplex system* depicted in Figure 3.2. It consists of two exact FUs and a comparator. If both outputs agree, the output is assumed to be correct. Otherwise, a fault occurred and error handling takes place [14].

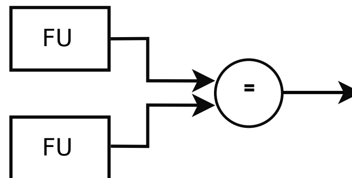


Figure 3.2: Duplex system

3.2.2 N-modular Redundancy

A system with N-modular redundancy (NMR), also referred to as an *M-of-N system*, is a system composed of N FUs from which at least M has to operate correctly. When fewer than M FUs are operational, the system fails [14].

The first such system was laid out by John von Neumann [29]. Today, the concept is known as a triple-modular redundancy (TMR). A Neumann's *majority organ* is what we now call *voter*. The output of the voter is the majority of the three inputs. Hence, TMR can tolerate single error. Figure 3.3 illustrates the concept.

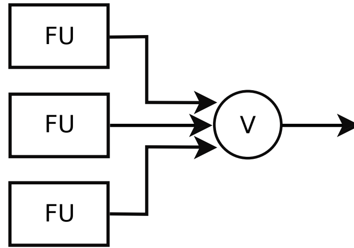


Figure 3.3: TMR, three identical functional units and voter

3.3 Field-Programmable Gate Arrays

Hardware redundancy is obviously not very cost effective as it brings area overhead. Therefore it is usually restricted to highly reliable and inaccessible systems. However, on a Field-Programmable Gate Array (FPGA) it is possible to implement it cheaply.

FPGAs are semiconductor devices equipped with SRAM configuration memory which can be reprogrammed and the implemented design can be changed. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs) which are custom manufactured for specific applications. Apart from flexibility, the main advantages of FPGAs are high computational power, relatively low cost and early time-to-market.

In this work, Xilinx products have been employed. Therefore, the following lines refer especially to Xilinx devices and tools.

FPGAs are based around a matrix of *Configurable Logic Blocks* (CLBs) connected via programmable interconnects. One can also find embedded Block RAM memory, Digital Signal Processing (DSP) tiles or even processors in today's modern FPGAs.

A CLB is the main logic resource for implementing circuits in FPGA. Each CLB is connected to a *switch matrix* to have access to the general routing matrix. A CLB contains a pair of *slices*. A slice holds four function-generators, four storage elements, three multiplexers, and carry logic [41]. For instance, the function generator in Virtex-5 is implemented as a Look-Up Table (LUT). There are six independent inputs and two outputs in each of the four LUTs in a slice. This means, a LUT can implement any six-input Boolean function.

Flexible interconnect routing routes the signals between logic resources and to and from I/Os. Routing comprises interconnects between CLBs, fast horizontal and vertical lines spanning the device, and routing for clocking and other global signals. The routing is hidden by the design software of an FPGA, thus the design complexity is reduced significantly.

3.3.1 FPGA Configuration

Each time an FPGA is powered up, a configuration file has to be loaded into its volatile memory. The configuration is created from the synthesis output, an *NGC* file. NGC is converted to a binary configuration data file called *bitstream*. The Virtex-5 bitstream contains commands to the FPGA configuration logic as well as configuration data. Its length for our FPGA is 20,019,328 configuration bits.

The basic configuration steps are *setup*, *bitstream loading* and *startup*. In setup phase, the memory is cleared sequentially. Then, the device gets synchronized and in order to prevent loading of a configuration formatted for a different device, the device ID check must pass. After that, the configuration data frames can be safely loaded. In startup, the bitstream instructs the device to enter the startup sequence.

Apart from power up, the configuration can be initiated during the operation on demand. In *full reconfiguration*, the user loads all the configuration data into the device. Some architectures have the ability to reconfigure a portion of an FPGA. This gives the user a possibility of *partial reconfiguration* where only certain areas of the design are reconfigured. When the device is active during the partial reconfiguration, it is denoted as *partial dynamic reconfiguration* (PDR). The advantage of PDR is that the remaining unaltered part of the design is still operational [36].

It is also possible to read loaded configuration from the device by sending a sequence of commands. The sequence initiates the *readback* procedure and the device dumps the contents of its configuration memory to the specified interface.

Placement Constraints

Implementation constraints are instructions given to the implementation tools to direct the mapping, placement, or timing while processing an FPGA design [34]. Constraints are generally placed in the *UCF* file. From the perspective of reconfiguration, it is desirable to have the power over positioning the particular parts of the design. This can be achieved by using placement constraints. A named group of logical blocks, which will be packed together by mapper, can be specified. Then, a range of device resources that are available to place logic contained in the group can be defined.

Frame addressing

The Virtex-5 FPGA configuration memory is arranged in *frames*. The frame is the smallest addressable segment that can be accessed. It can be thought of as a vertical stack of 1312 bits [40]. Each frame has a unique 32-bit address which is divided into five parts (Figure 3.4).

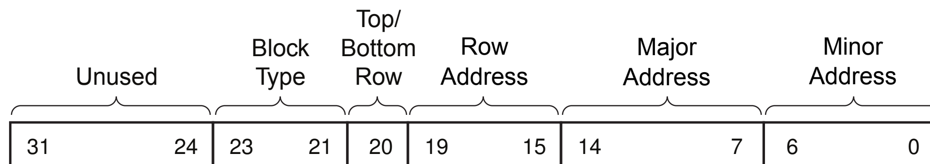


Figure 3.4: Frame address parts [40]

The Block Type part specifies the frame category by function or the way of access. There are four categories: interconnect and block configuration, Block RAM contents, interconnect and block special frames, and Block RAM non-configuration frame. The FPGA is divided into two halves, *Top* and *Bottom*. Each of them is then divided into rows which are numbered from zero starting from the centre as shown in Figure 3.5. Each row is divided into columns which correspond to a block in the array (CLB, DSP, Block RAM, etc.). Column number is called major address and starts from zero. A column then holds frames which are accessed using the minor address. The number of frames depends on the type of the block in the column (e.g. CLBs contain 36 frames).

3.3.2 Self-checking Functional Units

SRAM-based FPGAs with high density of memory cells are sensitive to transient faults, so-called *Single Event Upsets* (SEUs). An SEU occurs when a charged particle strikes a

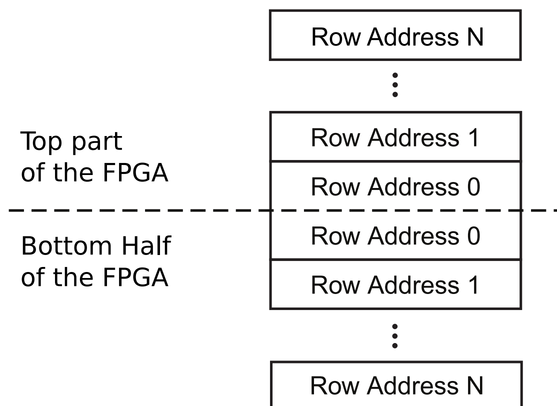


Figure 3.5: Row addresses [40]

memory cell and causes the state of a bit stored in the cell to flip.

The Xilinx experiment [37] demonstrates that in the vast majority of cases, an SEU only flips a single configuration bit. Multi-bit upsets almost never occur. There is also a high probability that the inverted bit will not have any effect on the design because only less than 20% of the configuration cells have a significance to a design implementation. The bit can be associated with the configuration of the device or the operational data of the design.

In many cases, a single-bit change in informational data can be tolerated (an incorrect pixel). However, when the definition of the design itself is impacted, the effect of the error can be significant and prolonged. A state machine entering an illegal state can serve as an example.

There are applications for which even the smaller risk is unacceptable. Fortunately, an SEU is a soft error and unlike hard errors (a broken wire), which need the replacement or a physical repair, its effect can be reversed without lasting damage. When a fault is detected in FPGA, a faulty unit can be repaired by reconfiguration.

The methodology for designing fault tolerant systems in FPGAs based on self-checking functional units has been proposed in [25]. The advantage of the approach is that any error detection technique can be used to construct the self-checking unit. The authors refer to the part of the FPGA which can be modified as *Partial Reconfigurable Region* (PRR) and the part of the design implemented into the PRR is called *Partially Reconfigurable Module* (PRM). The basic idea is to place each FU and its checker to one PRM. Then, it is possible to perform reconfiguration of the faulty unit while the rest of the design is still active.

Presented fault tolerant architectures are depicted in Figure 3.6. The blue regions designate the *dynamic* part of the design with fault tolerant components which can be reconfigured by means of PDR. The red dashed rectangles symbolize the *static part* holding units that are not fault tolerant and are not intended for reconfiguration.

The first architecture (TMRcmp) represents a TMR concept enriched by error localization. The outputs of functional units proceed to comparators and their outputs are processed in the error decoder. The second one (DUPLchck) is a duplex architecture. The outputs from checkers control a multiplexer which switches the data from the functional units. The last architecture (DUPLchckcmp) consists of a functional unit and its checker, and another functional unit and its corroborative checker. The output from the checker in

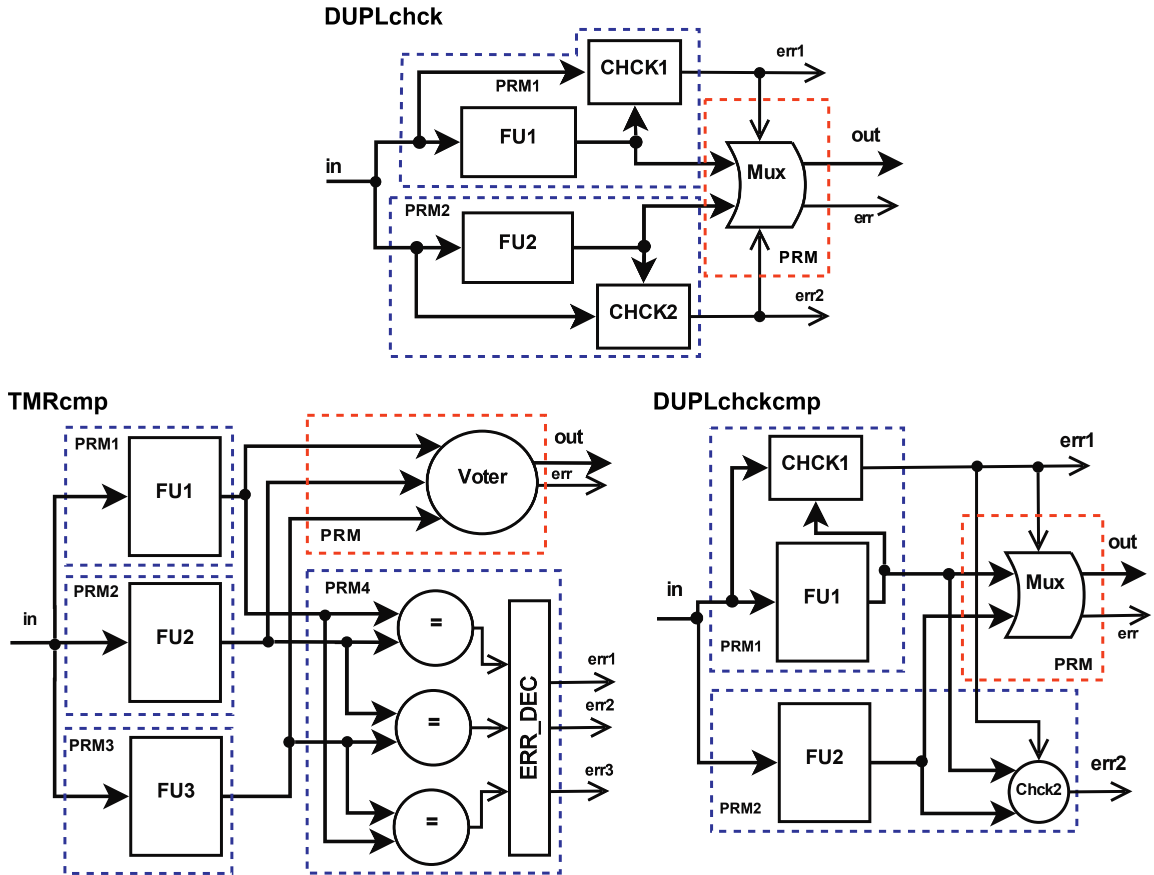


Figure 3.6: Fault tolerant architectures based on PRMs [25]

PRM1 reports an error in the unit. Corroborative checker reports a fault in PRM2 when PRM1 states a correct output and, at the same time, the outputs of PRM1 and PRM2 differ.

3.3.3 Checking Circuits

The checking circuit, which we also call *online checker* (Figure 3.7), is a means of error detection. Checker represents a supervisor who checks validity of the unit's output. When the output is evaluated as incorrect, error is reported.

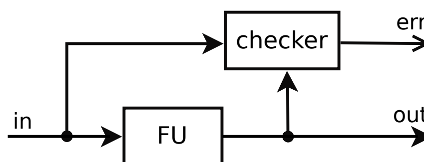


Figure 3.7: Online checker

The main advantage of online checkers comes from the fact that their implementation is different from the implementation of the FU. If the checking system is implemented exactly the same as the original FU, it can happen that the error causing faulty output will affect the checker as well. Such a situation can result in an undetected fault. Fluctuation of

the supply voltage outside of the safe range can serve as an example of such error. The different implementation of the checker minimizes probability of this type of undetected faults. Thus, even though it is possible to derive the checker directly from the original FU by comparing results of two copies of the FU, it is not recommended. By simply changing the FU to operate as a checker, the main advantage of the checker would be sacrificed.

The design of an online checker presents slightly different problem than the design of the FU. While the FU has to produce the desired output, the checker only needs to validate the output. It is important to note that often not all the function outputs need to be checked. For instance, when the checker is used to evaluate if the communication component behaves according to the bus specification, it is not needed to spend resources and time on validation of the bus transfer content. Therefore, the checker may require less resources than duplication of the FU. A qualified programmer with detailed understanding of the solved problem is required for the design of the checking circuit.

Straka et al. [26] presented the construction of online checkers based on finite automata. They used a formal language for the description of a system bus behaviour. The online checker was automatically constructed from the description. However, a human expert was required to construct the description of the system behaviour.

This work aims at automatic construction of the checkers which can be used in previously described self-checking architectures. We focus on units with deterministic behaviour, such as memory controllers or buses. The functional unit is treated like a black box. To create the checker of a black box, some knowledge about interface signals is needed. The checker is generated automatically by experimentation with the unit and by reasoning on the observed output behaviour. This is done by means of active automata learning which is covered in the next chapter.

Chapter 4

Active Automata Learning

This chapter is devoted to the field of active automata learning. The main goal of active automata learning is to build an automaton which matches the behaviour of a given target automaton. The construction is based on observations of target automaton using techniques from finite automata theory and machine learning.

In literature, the automata learning is sometimes designated as *regular extrapolation*. Indeed, constructing the best matching regular model consistent with observations is similar to polynomial extrapolation. Like there too, the quality of extrapolation depends on the structure of systems behaviour. The result of the learning is often reliant on employed learning optimizations.

Considerable development has been done in active automata learning in the last decade. The learning techniques can be used for inferring behaviour models of legacy or black box systems which then can be analysed, tested and validated using formal approaches. The first major success of the active automata learning application was in the area of regression testing in computer telephony integrated systems [9]. Now, application domains range from web-based services to embedded control software [30].

In practice, realistic systems are subject of learning. Those are *reactive* systems which interact with the environment. Their behaviour can be characterized by data inputs and outputs. Therefore, adequate treatment of data is crucial in active learning. It represents a major source of undecidability and, therefore, a key problem to achieve practicality [23].

In the following, we look at two approaches in automata learning under the perspective of treating of data. We introduce the first active automata learning approach based on construction of DFA where data are disregarded. Next, active learning of Mealy Machines as an automata model for systems with output is presented. At the end of the chapter, models which capture data-flow are mentioned.

4.1 Learning DFAs

If we assume that a system under learning (SUL) can be modelled by a DFA, we are looking for a regular language accepted by the DFA. Suppose that we are given data consisting of observations of the I/O behaviour of the black box. Gold [8] showed that the problem of finding the minimal DFA which agrees with given data is NP-complete.

However, in [4] Angluin proved that finite-state automata can be learned when it is possible to ask whether a string belongs to the language of the SUL. It has been also

proven that a regular language is learnable in polynomial time. A learning model for inference of regular languages was introduced and later adapted by many other active learning algorithms.

4.1.1 MAT Model

The fundamental assumption of Angluin's model is the existence of a *Minimally Adequate Teacher* (MAT). Omniscient teacher answers *membership* and *equivalence queries* asked by the learning algorithm, the *learner*. The learning proceeds in rounds alternating of a hypothesis construction phase (HC) and a hypothesis validation (HV) phase.

In the construction phase, a hypothetical DFA is gradually built by asking membership queries. The membership query tests whether the string is contained in the SUL's language or not. Thus, the teacher's answer can be either *yes* or *no*.

In the validation phase, equivalence query compares the obtained hypothesis model with the SUL. The teacher replies *yes* if the obtained hypothetical language is equal to the SUL's language and the learning successfully ends. Otherwise, it provides a string, so-called *counterexample*, which is in the symmetric difference of the hypothetical language and the SUL's language. Positive counterexample shows that the hypothesis is missing something and it is used to improve the hypothesis in the next round.

Figure 4.1 illustrates the model.

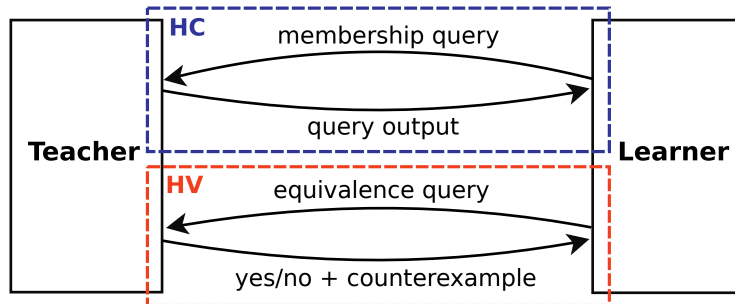


Figure 4.1: MAT model

In practice, however, we seldom have a complete model of a system to check against. Hence, the equivalence queries are approximated via membership queries. Also, there is not an absolutely reliable check for the whole black box system. We can not be sure whether the testing was extensive enough. This is the reason for the learning neither to be correct nor complete. It is guaranteed that the model is the most concise representation of the observed behaviour [24].

4.1.2 L^* Algorithm

The key idea of the algorithm is to approximate the Nerode equivalence $\equiv_L$ by an equivalence relation $\equiv_H$. In the process of finding the approximation, prefixes and suffixes are identified. The prefixes represent members of $\equiv_H$ and suffixes are used to reveal inequalities of the equivalence classes. When the equivalence query yields a counterexample, it means the approximation is too coarse and further refinement of $\equiv_H$ is needed. The learning algorithm eventually terminates and returns an acceptor isomorphic to the minimal DFA accepting the SUL's language [23]. The following explanation draws solely on [4].

To gather information about the strings, to classify strings as members and nonmembers, and to build a DFA, the learner maintains a central structure denoted as *observation table* (OT).

OT comprises:

- a non-empty finite set S of prefixes,
- a non-empty finite set E of suffixes, and
- a function T mapping $((S \cup S \cdot A) \cdot E)$ to $\{0, 1\}$, where A is a fixed known finite alphabet.

Initial setting is $S = E = \epsilon$. The result of $T(u)$ is 1 iff u is a member of the SUL's language.

We can imagine OT as an array which gets filled and augmented during the learning. The columns hold suffixes from E and rows are labelled by prefixes from S. $T(s{\cdot}e)$ represents the entry of row s and column e, i.e. the table cell containing the result of a corresponding membership query. A function row(s) is defined as $f(e) = T(s \cdot e)$.

In each round, OT is tested for *consistency* and *closeness*. We say the OT is *closed* when for each t in $S \cdot A$ there exists an s in S such that row(t) = row(s). OT is called *consistent* if for any $s_1, s_2 \in S$ such that row(s_1) = row(s_2), we have row($s_1 \cdot a$) = row ($s_2 \cdot a$).

When OT is not closed, the learner finds s_1 in S and a in A so that row($s_1 \cdot a$) $\neq$ row (s) for all s in S. Next, the string $s_1 \cdot a$ is added to S and T is extended to $(S \cup S \cdot A) \cdot E$ by asking membership queries for missing elements.

In case OT is not consistent, the learner finds e in E, s_1, s_2 in S, and a in A so that $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$ and row(s_1) = row(s_2). The string $a \cdot e$ is inserted to E and T is extended to $(S \cup S \cdot A) \cdot (a \cdot e)$ by asking membership queries for missing elements.

If OT is determined to be closed and consistent, the learner converts OT to DFA M as follows:

- set of states $Q = \{row(s) : s \in S\}$
- the alphabet A
- transition function $\delta(row(s), a) = row(s \cdot a)$
- $q_0 = row(\epsilon)$
- $F = \{row(s) : s \in S \text{ and } T(s) = 1\}$

The learner then asks an equivalence query with M. If the answer of the teacher is positive, L^* terminates with M as a result. If the teacher returns a counterexample t, the learner has to extend S by adding t and all its prefixes. The membership queries for the missing entries are asked and another iteration begins with the new OT.

4.2 Distinguishing Input and Output

In DFA learning, the learning procedure can distinguish between *accept* and *reject* only. An inferred model does not transcribe the output. Data are completely abstracted away

and not handled at all. However, real reactive systems produce output on given input and often do not terminate. Therefore, we need richer models which faithfully represent I/O behaviour of a reactive system.

In the following, we present the active automata learning based on the Mealy machine formalism including adopted learning algorithm. After that, formalisms suitable for applications where data-flow modelling is required are presented.

4.2.1 Learning Mealy Machines

Niese [15] adapted the DFA learning to a technique exploiting a structure of generalized Mealy machines (MM). The generalization of the former L^* algorithm, denoted as L^*_M, has been conceived. MM provide more direct modelling of the I/O based behaviour. The learned automata are more concise and the learning is faster.

MM are a variant of automata distinguishing between input and output alphabet. There is a very close relationship between MM and DFAs. We can look at MM as a deterministic finite automata over the union of the input alphabet and output alphabet with partially defined transition relation. MM distinguish *runs*. A run records an input and its final output as an abstraction from all intermediate outputs. A run is a semantic functional $[\![M]\!] : \Sigma^* \to \Omega$ defined by $[\![M]\!](w) = \lambda^*(s_0, w)$. Semantic equivalence of two Mealy machines $M \equiv M'$ is defined as $[\![M]\!] = [\![M']\!]$ [24].

To find out which functionals of $P : \Sigma^* \to \Omega$ are the semantics of an MM, we need to understand a notion of *equivalence of words wrt. P* which is similar to Nerode equivalence. It states that two inputs $u, v \in \Sigma^*$ are *P-equivalent*, written $u \equiv_P v$, iff for all continuations $w \in \Sigma^*$ concatenated words uw and vw are mapped to the same output by P. Then, the *characterization theorem*, an adoption of Myhill-Nerode theorem, says that a mapping $P : \Sigma^* \to \Omega$ is a semantic functional for an MM iff $\equiv_P$ has a finite index. The proof of the theorem can be found in [24].

Compared with DFA learning, the difference lies in the way how the states are characterized and distinguished. There are two characterizations of the state:

- by words entering and
- by its future behavior.

The first characterization maintains a spanning tree of words S_P reaching the state and its one letter continuations L_P.

The other one holds a an output vector of systems reactions to the input vector $D = \langle d_1, ..., d_k \rangle$ of strings from Σ^*, denoted as *distinguishing suffixes*. The future behaviour of the state $u \in S_P$ is represented by $\langle mq(u \cdot d_1, ..., mq(u \cdot d_k) \rangle \in \Omega^k$. This leads to an upper approximation of classes of $\equiv_{SUL}$. Active learning refines the approximation by extending the distinguishing suffixes.

Like in DFA learning, the learning algorithm proceeds in rounds by the alternation of membership and equivalence queries. Membership queries correspond to single test runs executed on the SUL. The inference starts with one state hypothesis automaton and refines it pursuant to the query results. During the learning, hypothesis construction and validation phases alternate.

During the construction, the learning algorithm can operate directly on the hypothesis model or, better, store the results of membership queries in an observation table. The

observation table has been introduced in Section 4.1.2. In Mealy machine learning, it is a mapping $O(U, D) : U \times D \rightarrow \Omega$, where $U = S_P \cup L_P$ represents a set of prefixes and D denotes a set of suffixes. The table cells contain the results of membership queries for words ud, where $u \in U, d \in D$.

If the hypothesis validation phase yields a counterexample, it is used to enlarge S_P or D. All its suffixes can be directly added into D and no analysis of the counterexample is needed. On the other hand, D grows tremendously which leads to unnecessary membership queries. In [24], optimized strategy has been proposed. The authors introduce an algorithm which adds exactly one suffix of each counterexample based on counterexample analysis.

When the hypothesis is considered equivalent with the SUL, the automaton is obtained from D and L_P as follows. Each output vector corresponds with an automaton state and L_P implies the transitions.

Example of Mealy Machine Inferrence

Let's look at an example of L^*_M inferring Mealy machine M given in Figure 4.2. The example has been taken from [1] and is using observation table to construct the hypothesis.

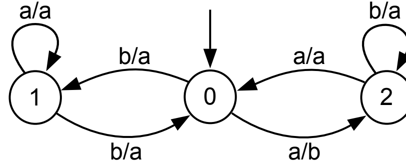


Figure 4.2: An example Mealy machine

The algorithm starts with $S = \{\varepsilon\}, E = \{a, b\}$ and asks membership queries for a, b, aa, ab, ba, and bb. The answers from the teacher are filled in the observation table OT_0 (Table 4.1a) which is consistent but not closed, since $row(\varepsilon) \neq row(a)$ and $row(\varepsilon) \neq row(b)$. Consequently, $row(a) = row(b)$, thus we can add one of them to S, let it be a, and the algorithm continues. Membership queries aaa, aab, aba, and abb construct OT_1 shown in Table 4.1b. It is both closed and consistent now, thus the hypothesis model shown in Figure 4.3 is presented to the teacher.

Table 4.1: Observation tables

(a)

OT_0	a	b
ε	b	a
a	a	a
b	a	a

(b)

OT_1	a	b
ε	b	a
a	a	a
b	a	a
aa	b	a
ab	a	a

(c)

OT_2	a	b
ε	b	a
a	a	a
b	a	a
bb	b	a
bba	a	a
aa	b	a
ab	a	a
ba	a	a
bbb	a	a
bbaa	b	a
bbab	a	a

(d)

OT_3	a	b	aa	ab
ε	b	a	a	a
a	a	a	b	a
b	a	a	a	a
bb	b	a	a	a
bba	a	a	b	a
aa	b	a	a	a
ab	a	a	b	a
ba	a	a	a	a
bbb	a	a	a	a
bbaa	b	a	a	a
bbab	a	a	b	a

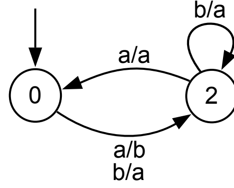


Figure 4.3: A hypothesized Mealy machine

The teacher selects a counterexample. We assume that the counterexample is *bba*. The algorithm adds *bba* and all its prefixes (b, bb) to S and queries $(S \cup S \cdot A) \cdot E$ are asked. The new OT_2 (Table 4.1c) is closed but not consistent as we have $row(a) = row(b)$ and $row(aa) \neq row(ba)$. aa and ab are added to E and the membership queries are asked again to fill the new columns. Resulting OT_3 shown in Table 4.1d is closed and consistent, so the algorithm asks a second equivalence query to the teacher. The teacher responds with *yes*. Before the L_M^* terminates, it merges equal rows because Q is defined as a set of distinct rows. The final Mealy machine contains three states and is equivalent with M.

4.2.2 Register Automata and Register Mealy Machines

The inputs and outputs in a real system often have parameters. In [7], a formalism based on a form of register automata (RA) was introduced. RAs are designed to deal with parametrized input and thus to describe the influence of data on control flow. They are suited to model systems which distribute data, e.g. communication protocols, which do not compute data and their behaviour does not depend on the data content they distribute. The active learning has been extended with this formalism in [11].

Recently [10], a generalized form of register automata which allows modelling data also in outputs has been developed. The inference of RA has been extended to also capture parametrized output. This extension is similar to a step from DFA learning to Mealy machine learning and the model is therefore denoted as *Register Mealy Machines* (RMMs). The authors claim that inference of RMMs is done fully automatically using systematic testing only and without any manual abstraction.

4.3 Learning in Practice

In practical active automata learning, the major challenge is to construct an application-specific *learning setup*. The learning setup comprises a suitable form of abstraction and managing the data influencing the system behaviour [18]. The adequacy of the abstraction is crucial to the success of the learning. It is usually done manually and tailored to the specific system. Even though approaches to automate abstraction have been devised, a certain level of manual effort or knowledge about the system is still needed.

When inferring models of a real system, the learning algorithm interacts with the SUL. As the learner's alphabet is an abstraction of the system alphabet, the means of bridging this gap are needed. *Mapper*, or a *test driver*, is a component which translates the abstract queries into the system language and, on the way back, the system outputs to the abstract language of the learner. Figure 4.4 illustrates mapper's position in active learning setup.

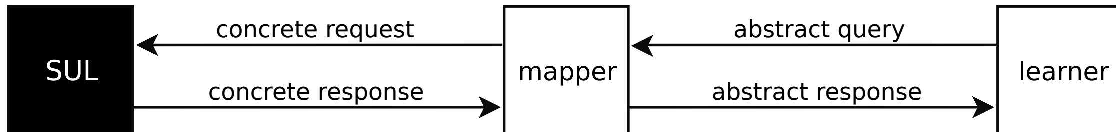


Figure 4.4: Schematic view of active learning setup using a mapper

4.4 LearnLib

LearnLib [33] is a Java framework comprising tools for automata learning and experimentation. It is subject to ongoing development at the University of Dortmund. LearnLib has been applied successfully to learn parts of the SIP and TCP protocol [1] and the biometric passport [2].

There are two versions of the library. The former closed-source version which is free of charge for academic uses [32] and the new version [33]. At the beginning, we have tried the former LearnLib but the fact it is closed source made us switch to the new version. Although the library is still in an early stage, it embraces all important implementations of learning algorithms, their variations, query optimizations, counterexample analysis, and several equivalence tests. Provided are statistics, logging, and export to *DOT*, a graph description language.

4.4.1 Inferring Models

LearnLib provides two interfaces for learning algorithm implementations. `DFALearner` is applicable for inference of finite automata and `MealyLearner` which aims at Mealy machines. Following two learning algorithms implementing MealyLearner interface have been used in our experiments.

The *DHC* (Direct Hypothesis Construction) algorithm [24] constructs the hypothesis automaton „on-the-fly“ using a queue of states to be explored. It uses a breadth-first search strategy. If a counterexample is found at the end of the round, all its suffixes are taken into account in the next iteration.

The second one, *Extensible* L_M^*, is the Mealy variant of L* [22]. It employs incremental construction of the hypothesis model. It does not work on the model directly but rather uses an observation table in similar way as described in Section 4.2.1. Instead of single symbol outputs, it stores rather words in the cells of the table.

Equivalence tests implement model-based testing methods and approximations based on membership queries. In our experiments, we have employed Random Walks, Random Words, Complete Exploration, W Method, and Wp Method. Interested reader may find more information about model-based testing in [6].

There are also many implementations of handling the counterexamples. Unfortunately, the LearnLib documentation is not very expansive in this regard. We are using `CLASSIC_LSTAR` which implements the standard procedure described by Angluin [4]. Several closing strategies are implemented. A closing strategy is a way of selecting the representative when multiple rows closing the observation table exist. In our work, Close First which selects the first for from each equivalence class has been used.

The overall algorithmic pattern underlying most active automata learning algorithms can be seen in Figure 4.5. On the left and right hand side, square boxes denote inputs and outputs, respectively.

First, the the learning setup is constructed. Then, the iterative procedure of the learning starts. The hypothesis is generated and refined by test-based exploration on the basis of membership and equivalence queries. The counterexamples obtained from equivalence queries are used for refining the hypothesis. When there is no counterexample found, the hypothetical finite state machine is finished. The output is written to a DOT file.

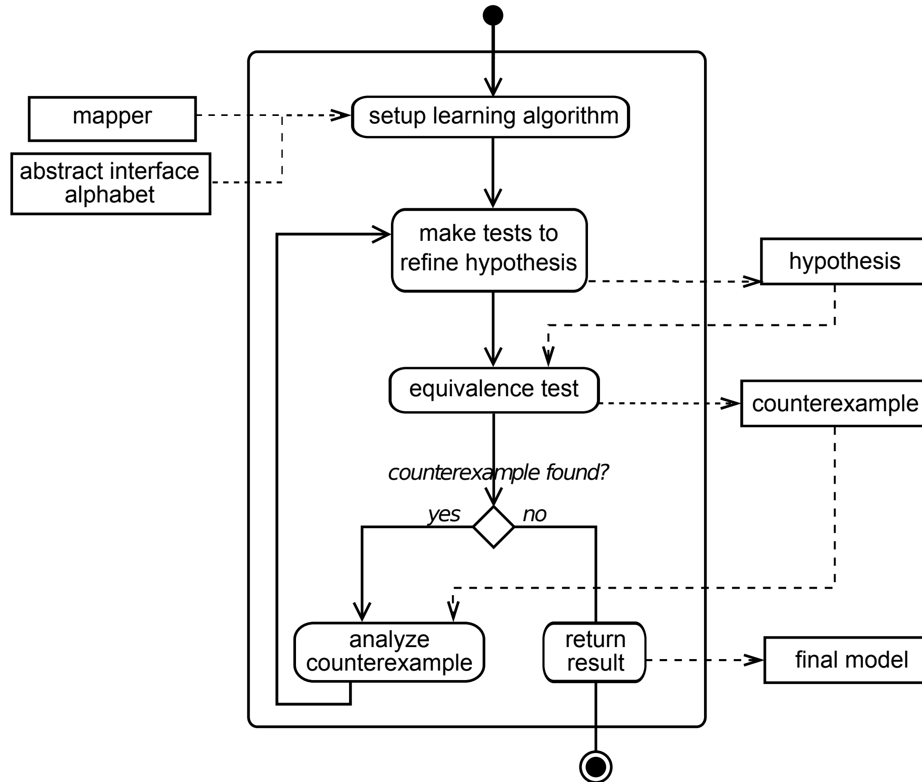


Figure 4.5: Extrapolating behavioral models with LearnLib [24]

4.4.2 Query Optimizations

On its own, automata learning algorithm generates a huge number of queries until it finds no more inconsistencies in the inferred system model. However, only some of the queries bring more knowledge to the system's behaviour. This makes the learning very expensive and impossible to fit real life scenarios.

A strategy to filter irrelevant queries and reduce the number of required experiments drastically has been introduced in [16]. Optimizations exploiting four kinds of *expert knowledge* are employed. In LearnLib, implementations of these optimizations are called *filters*. The position of a filter in the learning can be seen in Figure 4.6.

Figure 4.6: Position of a filter in learning

Redundancy

In classical implementation of the Angluin's algorithm, redundant membership queries are generated and stored in the observation table. The redundancy filter is implemented as a hash table $T : \Sigma^* \rightarrow \{unknown, true, false\}$. The output for given query $MQ(q)$ express the knowledge about q. The rules are:

- $T(q) = true \Rightarrow MQ(q) = true$,
- $T(q) = false \Rightarrow MQ(q) = false$,
- $T(q) = unknown \Rightarrow$ it is necessary to execute the membership query.

Prefix Closure

The language of a real system is *prefix closed*. That means, in a run, its every prefix also belongs to the language. This leads to a powerful optimization. The principle can be used e.g. when a positive counterexample is obtained from the equivalence query. All of its prefixes are directly added to the model.

The prefix closure filter is also implemented by means of cache. The filter rules are as follows ($prefix(q)$ gives us the set of all prefixes of q):

- $\exists q' \in \Sigma^* : T(q.q') = true \Rightarrow MQ(q) = true$
- $\exists q' \in prefix(q) : T(q') = false \Rightarrow MQ(q) = false$

Independence of Actions

Some actions can be executed in any order and end up in the same system state. The filter needs to be specified by a domain expert in form of an *independence relation*. It specifies which events can be shuffled in any order.

Symmetry

In hardware systems, there are often components which are identical. The symmetry filter brings optimization by finding such components. Again, an expert needs to be involved to determine which devices or units fulfil this condition.

Chapter 5

Design and Implementation

Let's assume we have an SUL described in VHDL and we want to generate its checker. We need a means of triggering the queries on the SUL and reading the answers. This can be achieved by running a simulation of the SUL in a simulator. Then, we shall possess the environment which will be responsible for generating the queries and construction of the model. The result of the learning will be automaton, presumably described in a simple text format. Thus, we will make use of a converter to a hardware language. After the checker is created, we shall perform a set of SEU injections to test the checker capabilities.

For effective construction of checkers in the manner described above, a comprehensive learning platform and the learning setup has been designed. This section covers the design and implementation of the learning platform, description of the major parts and their interconnection. The SUL and its learning setup is introduced.

An experimental platform for SEU injections has been designed and implemented. The description of its main components together with a means of connection is given in the second half of the chapter.

5.1 Learning Platform

The learning platform compounds a simulator holding the SUL, a learning environment, and a converter of the obtained model (Figure 5.1). The design and implementation of the components is described in more detail in the following sections.

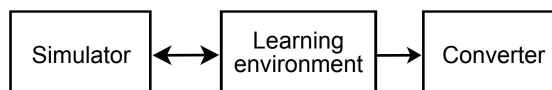


Figure 5.1: The main components of the learning platform

5.1.1 Simulator

For the purpose of the thesis, the *ModelSim* simulator has been chosen. ModelSim is a vendor independent tool from Mentor Graphics for simulation, verification and debugging of FPGA designs. Simulation of a design runs the VHDL *testbench* which instantiates the unit under test, provides a set of stimuli and captures the output.

The learning environment needs an access to the simulation. In this regard, the advantage of running a simulation in ModelSim is that it can be set up and controlled remotely

via sockets. Thus, the interaction between the learning environment and the simulator is realized over TCP/IP. The Tcl script [12] is used to run a server in the simulator. The server is listening on a certain port. After our modification, it now invokes received commands and sends back the simulation output.

During the learning, we apply stimulus to a given VHDL signals, run the simulation by the specified time range and examine the obtained results. For these purposes, commands *force*, *run*, and *examine* are employed.

SUL and Learning Setup

In this work, a part of Wishbone bus [20] has been chosen as the SUL and is simulated. Wishbone is an open source hardware bus developed by OpenCores hardware community. It represents an System-On-Chip (SoC) interconnect architecture for connection of IP cores. Wishbone is a logic bus, i.e. it is specified in terms of signals and clock cycles. The reason why we have chosen Wishbone comes from the fact that it is a part of an exemplary system for testing fault tolerant methodologies developed at our faculty department [21]. As it implements the communication of the components, it represents a central element and a critical part of the system.

Wishbone defines two types of interfaces: master and slave (Figure 5.2). The wishbone signals are located at one side of the master and slave interface unit. On the other side, there are application-specific signals. The master device specifies requests by setting the address, data and control (bus cycle) signals. Slave accomplishes read or write actions on masters requests. It responds to input control signals by setting its output control signals.

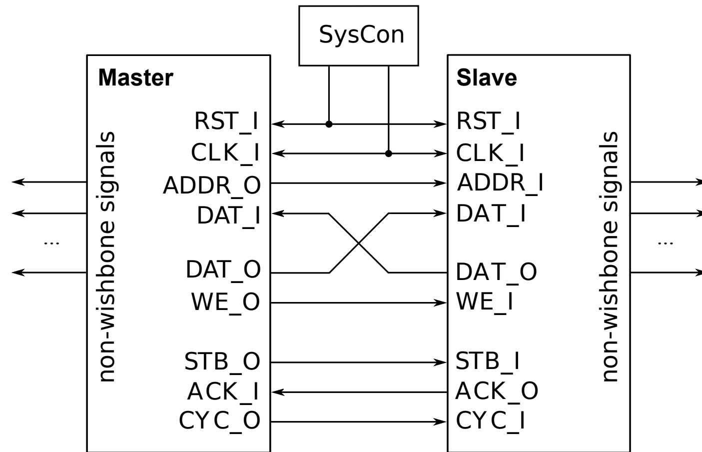


Figure 5.2: Master and Slave Interfaces

We suppose that if we focus on control signals omitting the address and data, the slave interface unit can be described by a Mealy machine model and it can be effectively inferred by means of active automata learning. Thus, it has been chosen as our SUL of which the checker is constructed.

To create the learning setup, we have to look closely on our implementation of the slave interface unit. It is used with the position evaluation unit (PEU) which computes X, Y coordinates on given distances from control points A, B, and C. Thus, it performs a read operation of three values or a write of two.

Wishbone signals can be observed on the left side of the block in Figure 5.3. On the right side, we can see signals specific to operation of the PEU. We omit address (`ADDR_I`)

and data signals (DAT_I, DAT_O, A, B, C, X, Y) and analyse the control signals only. Three input and three output one-bit signals remain: WE_I, CYC_I, STB_I, ACK_O, NOT_CYC_WRITE, and NOT_CYC_READ.

The write enable input WE_I gives information about the type of the bus cycle. It is asserted during write cycles and negated during read cycles. The cycle input CYC_I, indicates a valid bus cycle is in progress. It is asserted during the whole bus cycle. The strobe input STB_I, when asserted, indicates that the slave is selected. A slave responds to other Wishbone signals only when this signal is asserted. The acknowledge output ACK_O announces the termination of a bus cycle. NOT_CYC_WRITE and NOT_CYC_READ signals are used to control the read and write operations inside the PEU.

There are three control inputs (marked red in Figure 5.3) which are of our interest at the slave interface. Therefore, the input alphabet consists of all possible combinations of three binary signals, that is 2^3 symbols. The outputs which we will consider are highlighted in blue.

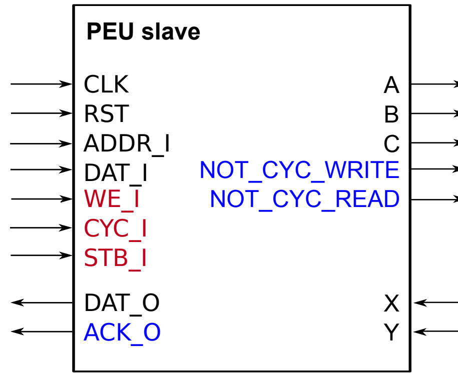


Figure 5.3: The slave interface of PEU

In our case, the input domain is very small. However, if the unit had more inputs, the learning would take longer. It would be necessary to reduce the learning complexity by minimizing the input alphabet. A human specialist who has a knowledge of the interface signals would need to select only some combinations of signals or create a suitable abstraction mapping.

5.1.2 Learning Environment

The learning environment consists of five main components: a telnet, a mapper, an oracle, a learner, and an equivalence oracle (Figure 5.4).

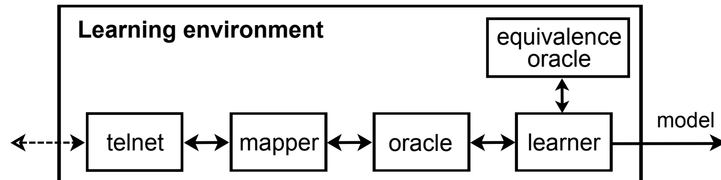


Figure 5.4: The main components of the learning environment

The learner uses an abstract alphabet to generate membership queries. A membership query is a word of symbols from the abstract alphabet. There are eight possible symbols

as each of them represents a setting of three inputs (WE_I, CYC_I, STB_I), i.e. the abstract alphabet contains eight symbols. The learner uses SUL responses to build a model. The output alphabet is represented by all the responses. It is filled during the learning and thus not created explicitly.

The queries go through the oracle which serves as an abstraction layer and facilitates unified access to the mapper. It also implements basic filter functionality to avoid duplicate queries.

To connect the oracle with the SUL, a mapper is needed. The mapper implements the learning setup tailored to the SUL as described in Section 5.1.1. It provides the translation from the abstract alphabet into the real SUL alphabet. Each symbol in a membership query represents simulation stimuli. The stimuli are applied by specifying signal values, enabling clock and running the simulation for a designated amount of time. The mapper decomposes the stimuli and creates Tcl commands for corresponding signals. Then, commands which enable clock and advance the simulation by a period are generated. After the commands are invoked in the simulator, the mapper examines the SUL output and hands it over to the oracle.

To get access to a command-line interface of the simulator, the telnet component is employed. The component is connected to the server running in the simulator. It allows us to transmit commands created by mapper, invoke them in the simulator and capture the results.

At the end of a learning round, the learner consults his hypothesis with the equivalence oracle and, eventually, outputs a model.

To evaluate the learning and its result, a means of time profiling and logging of models are used. After the learning process finishes, learning time, the number of membership queries and executed symbols are available apart from the inferred model.

The main method resides in `WBSlaveLearner` class. The class holds a static instance of the `AutoTelnetClient` class. To implement telnet component, Apache Commons Net library [27] which implements many basic Internet protocols is used. The `AutoTelnetClient` class has been inspired by [31]. The `AutoTelnetClient` constructor tries to connect to the server and the port given by its parameters. After the connection is established, buffers of types `InputStream` and `PrintStream` are involved in transmitting the data. To send the data to the server, a `write` method is used. A `readResult` method is responsible for getting the simulation output. It reads the data from the input buffer until the end character is found and returns the output consisting of three signal values.

The remaining components implement the learning functionality using LearnLib framework presented in Section 4.4.

At the beginning, the input alphabet is filled with symbols. The alphabet is implemented by `WBSlaveAlphabet` class. Each symbol is an instance of `WBSlaveInput` class which contains two instance variables: action and data. In our implementation, we have one action called `execute_symbol`. It is a mapper method residing in `WBSlave` class. Data variable holds a string which represents simulation stimuli. The mapper ensures their translation to corresponding input signals of the PEU slave interface and enables clock for one period. For example, the symbol [`execute_symbol, 011`] shall be mapped to

```
force WE_I 0
force CYC_I 1
force STB_I 1
force CLK 1
run 20 ns
```

```
force CLK 0
run 20 ns.
```

The output of the unit is read by mapper by invoking following commands:

```
examine -value ACK_O
examine -value NOT_CYC_WRITE
examine -value NOT_CYC_READ.
```

After that, we create an instance of `WBSlaveAdapter` class. It implements the `SUL` interface that provides three methods for executing actions on the SUL. As we do not perform any adjustments before (`pre()`) the learning starts, we implement only execution of an input on the SUL (`step()`) and reset of the SUL (`post()`).

The membership query oracle is an instance of `SULOracle` class created by passing a previously created oracle containing cache. The cache is an implementation of filtering and is provided by static method `createSULcache` of `Cache` class.

As Mealy machines are learned, the learner component is either instance of `MealyDHC` or `ExtensibleLStarMealy` which are both implementations of `MealyLearner` interface. When creating `MealyDHC` instance, input alphabet and oracle is specified. `ExtensibleLStarMealy` requires initial set of suffixes (can be also empty), the specification of counterexample handling and the closing strategy.

Next, an equivalence test is build using the `MealyEquivalenceOracle` class. Several extending classes such as `RandomWalkEQOracle` or `MealyWMethodEQOracle` are available. Implementation details concerning the chosen equivalence oracles are given in Section 6.1.

Now, everything is set up for the learning to begin. A learning experiment instantiating `MealyExperiment` class is constructed by specifying the learning algorithm, the equivalence oracle and the input alphabet. The experiment is started by invoking the `run` method. The result is of type `MealyMachine` and can be passed to a static `write` method of `GraphDOT` class which renders a DOT file.

5.1.3 FSM-VHDL Converter

When the learning is finished, the model is saved in a DOT file. It contains a textual representation of the FSM which can be processed or visualized by many editors. However, our checker is not represented by the FSM only. To get the error-reporting feature, it would not be enough just to convert the inferred model to VHDL description. A mechanism which compares the output of the FSM and output of the SUL has to be included. Only then, the unit may be called *the checker* and it is ready for synthesis and implementation to FPGA.

A part of the converter developed at the Concordia University [3] has been used. The converter is composed of two modules. DOT-to-KISS[1] module accepts a DOT file, analyses it for possible syntax errors and performs conversion to KISS formatted FSM. The other part, KISS-to-VHDL module, generates the VHDL description from the KISS file.

The authors have been able to provide only the KISS-to-VHDL module. Thus, we have implemented the other one by ourselves using Python. Designs generated by the original KISS-to-VHDL module do not implement the reset function. However, the reset is an integral capability in a checker. Therefore, the generation process of KISS-to-VHDL module has been expanded by adding the reset and the generic comparing mechanism which announces the error has been added.

[1]Keep It Simple Stupid (KISS) FSM format

5.2 Experimental Platform

When the checker is constructed, we would like to perform experiments comprising SEU injections and evaluate how good the checker is.

A unit instantiating the checker and the original functional unit was designed and implemented to FPGA. The function of the unit is directed by a controller in a computer. The controller is responsible for starting the experiments and reading the results. The SEU injection employs the generator which is connected with the board by Xilinx JTAG programmer. The last part which has to be devised is a means of connection of the controller and the board. The experimental platform is depicted in Figure 5.5.

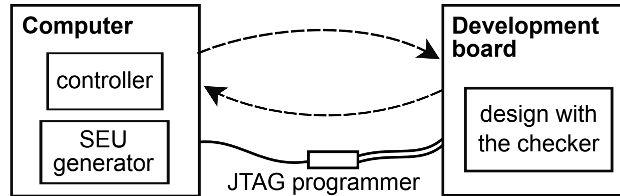


Figure 5.5: Scheme of the experimental platform

To capture the number of errors caused by SEU injection, a counting mechanism is included. The numbers of three types of errors are stored. The first value represents the number of real errors. The second one gives us the number of true positives, i.e. errors which the checker caught successfully. And, in the last one, there is the number of false positives, i.e. errors which were reported by the checker but have not really occurred.

An *execution* of the design represents processing of finite number of values stored in the memory and counting the errors. When the execution itself is performed, we get no errors and thus zeros on the output. An *experiment* designates the SEU injection followed by the execution. When the SEU hits an important part of the design, we should observe non-zero values on the output.

5.2.1 Design with Checker

The design is divided into two groups. The first one contains the functional unit together with the checker (the blue rectangle in Figure 5.7). This part will undergo the fault injections. The rest of the design represents the other part. That is a generator of inputs driven by start-stop unit, three counters of errors and another instance of the functional unit which serves as a reference for counting real errors.

To implement the checker, the automaton inferred by learning and converted to VHDL is used (Figure 5.6). The output from the resulting FSM and the original FU is compared. When a mismatch occurs, the error is reported.

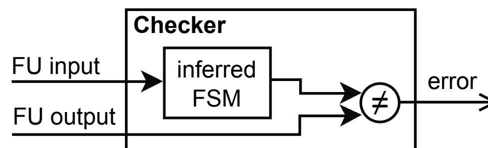


Figure 5.6: The checker

The execution of the whole design is controlled by the start signal. The input generator produces inputs for two functional units and the checker. The generator is implemented

as a Block RAM and a counter for generating the read address. The Block RAM module was created using Xilinx CORE Generator System [35]. It stores 11608 values which were gathered during the learning process from the membership queries.

The outputs of the two functional units are tested for conformity. If they are not equal, the SEU has caused an error and the respective counter increments its value. On the left, the blocks `cnt FP`, `cnt TP` and `cnt` represent counters which count the number of false positives (FP), true positives (TP) and real errors, respectively. The counters keep the maximum achieved value and do not overflow back to zero. In this way, it is possible to determine from the results that the maximum value has been reached or exceeded. When the last value from the input generator is processed, the execution is over and the result together with `read` enable signal is outputted.

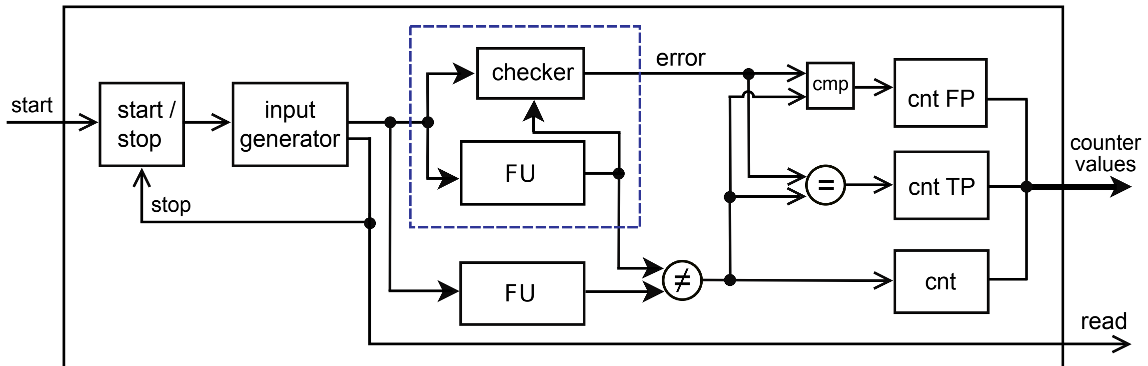


Figure 5.7: Design with the checker

Thus, within an execution, 11608 test vectors are processed. This means that up to 11608 errors could occur when an SEU is injected. To capture such a number of errors, a 14-bit counter is needed. In our design, there are three such counters, the start signal and the read signal, i.e. 44 pins to transfer them. Such number of pins is not available on our experimental platform and we do not like to decrease the reliability of the test set by reducing the number of test vectors. Therefore, 6-bit counters are used to count the errors.

Having two groups of the design allows placing both of them to the pre-determined regions in the FPGA. Then, injection of the faults into the selected one can be performed. To create placement constraints, we used the PlanAhead tool [39] which provides a comprehensive design and implementation environment with GUI. To constrain the blocks, one needs to enable the *keep hierarchy* option before the synthesis. This approach is called *Hierarchical Design*. It ensures the architecture not to be flattened during the synthesis and keeps the design components in the netlist. Then, the components can be easily grouped and placed into specified areas in the FPGA.

Figure 5.8 illustrates positioning the first group with the checker vertically (block A) and placing the other group (block B) to a different area. Block A occupies one CLB (horizontally), i.e. 36 frames.

The output of the PlanAhead placing process is a UCF file with generated placement constraints. For example, the constraints for the block A follow:

```
INST "fu\_checker" AREA\_GROUP = "A";
AREA\_GROUP "A" RANGE=SLICE\_X12Y60:SLICE\_X13Y79;
```

Whereas the placer does not locate any logic elements of another group in a region, the router can route static connections through all the regions. We are not aware of any

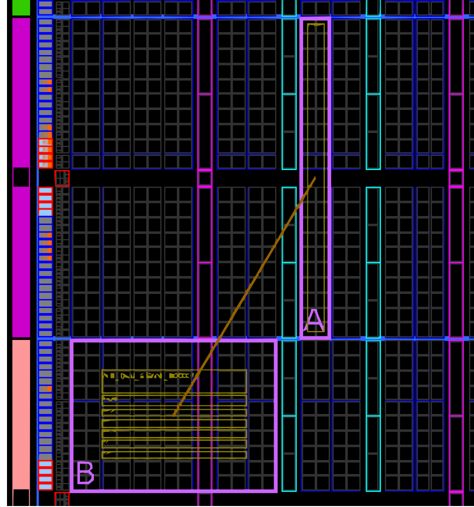


Figure 5.8: Constraining the groups of the design in PlanAhead

simple method how to automatically keep the routes out of the specified region. One should be able to change the routing manually using *FPGA Editor* or *RapidSmith*. To do this, detailed FPGA knowledge and more experiments would be required. The solution could be to put the evaluation mechanism aside of the injected design which is, however, beyond the scope of this work.

The final design was implemented and tested on Xilinx ML506 board with Virtex-5 XC5VSX50T-1FF1136 chip. To test and debug the design, ChipScope Pro [38] providing integrated controller (ICON) and logic analyzer (ILA) was used.

5.2.2 Connection with Board

The design in the development board produces results which are to be processed in a computer. One of the possibilities how to connect the board and the computer is a universal asynchronous receiver/transmitter (UART). Using UART port requires working with Xilinx USB to UART drivers which could take some time before having the peripheral up and running. Thus, we have chosen a different approach using the FITkit platform.

FITkit [28] is a standalone hardware board which contains the Spartan3 FPGA, a microcontroller and various peripherals. The advantage of employing FITkit in our experimental design is that we can implement a handshake mechanisms for synchronization using the microcontroller.

The FITkit represents a communication layer and can also serve as a debugging point for communication between the computer and the ML506 board. The connection between the FITkit and the computer is realised by a serial bus (USB). The development board and the FITkit are interconnected via pinheaders. The assignment and connection of pins on the board and on the FITkit is given in 5.1.

Table 5.1: Connection of pinheaders

FITkit	11 13 15 17 19 21	23 25 27 29 31 33	12 14 16 18 20 22	24	26
ML506	2 4 6 8 10 12	14 16 18 20 22 24	26 28 30 32 34 36	38	40
signal	cnt	cnt FP	cnt TP	read	start

5.2.3 Experiment Controller

An experiment controller is a C/C++ application designed with the intention of use in the batch mode. The controller generates the start signal which activates the execution of the design on the development board. Then it waits for the results and outputs them on stdout. In such a way, the controller can be invoked repeatedly and the obtained results can be appended to a file.

At the beginning, USB discovery is performed. The controller looks for the FITkit which is dedicated to the connection with the development board. A simple handshake mechanism has been implemented. The controller opens a serial connection and sends `FITKIT` command. If the response `YES` comes, the connection with the board is established and the `START` signal is sent. After that, the controller waits for the results. Another handshake process was implemented to catch the values sent by FITkit. The results are being sent until the `THANKS` command from the controller is received. Results coming from the FITkit are in hexadecimal base and are converted to decimal before printing out on stdout.

5.2.4 SEU Generator

An SEU injection is the process of changing one bit in the configuration memory or in the memory of the FPGA design. To perform SEU injections on the design with the checker, we need a tool for SEU simulation.

The SEU generator chosen in this work was developed at the Department of Computer Systems of The Faculty of Information Technology, BUT [25]. It meets four main criteria. *Universality* is achieved by generator's ability to place the SEU at any place on the FPGA. *Locality* is ensured by placing the SEU into a pre-determined area of the FPGA. There is a guarantee that other areas remain unmodified. The generator is *separate* and independent on the function in the FPGA. Thus, it cannot damage itself and there is no need to rebuild the attacked design. The SEU injection is an *atomic* process. The applied SEU is not overwritten or replaced in any way.

The generator is a standalone application implemented in Tcl language with the use of ChipScope libraries. It combines readback and dynamic reconfiguration to achieve the SEU injection. The structure of the generator is depicted in Figure 5.9.

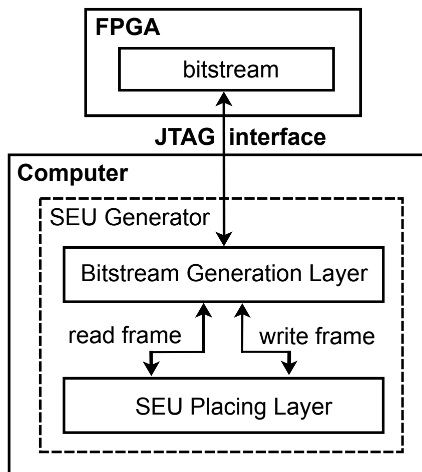


Figure 5.9: Structure of the SEU generator

The Bitstream Generation Layer communicates with the FPGA through the JTAG interface. It accepts the frame address and frame data from the SEU Placing Layer and generates a bitstream which performs a readback or a write of data for the given frame. The SEU Placing Layer creates read and write frame requests according to the given placing policy. One of the placing policies that can be taken advantage of in our work is changing a given bit in a frame of bitstream.

It should be noted that the configuration memory is usually not documented. Due to this, it is not possible to predict whether the bit is required for the design function or not.

The SEU injection involves the following steps:

1. Frame selecting – the frame is addressed by combination of four values (top/bottom part of the FPGA, row address, major and minor address);

2. Readback – the frame is readbacked without interrupting the computation in the FPGA;

3. SEU generation – the given bit in the configuration data is flipped;

4. Write frame – the changed frame is written back into the configuration memory, again without interrupting the computation.

Chapter 6

Experiments

At first, we tried to use former LearnLib for the model inference. However, we experienced problems during the runtime and thus we switched to the new LearnLib version which is open source. In this section, we describe the settings of the learning experiments. The statistics of the learning processes are discussed and the analysis of the inferred models is given. Implementation of the obtained models into FPGA is shown. The experiments using SEU injection are demonstrated and their results are given.

6.1 Inference of Models

A wide set of experiments executing the inference of the model of the wishbone slave was performed. 22 representatives were selected and are reported in this section. We employed DHC and Extensible L* learning algorithms. There are 11 learning setups for each algorithm which represent different equivalence tests for generating counterexamples. The experiments were carried out on Intel Core i7 with 1.5 GB RAM. To make the experiments reproducible, the implementation details on the learning setups are included in Table 6.1.

We recorded how long it takes for learning to finish, the final dimensions of the inferred automaton, and also other properties denoted as *learning characteristics*. The learning characteristics represent the number of learning rounds, membership queries, and the total number of executed symbols. Some of the experiments finished within minutes and others were running for several hours or even days. One of the factors in this regard is a handshake between the simulator and the learning environment conducted via telnet.

Results

The results of the experiments with DHC and Extensible L* are presented in tables 6.2 and 6.3, respectively.

In 14 experiments, the model of 4 states and 32 transitions was inferred. When compared, the models are identical. Note that all the experiments returning the 4-state model ended after one round, even though many of them have been running for a few hours (e.g. Wp Method II). The attentive reader may notice that the learning characteristics of these experiments are the same for both learning algorithms and only the learning times differ. No counterexample was found and the learning stops at the very beginning in all cases.

The Complete Exploration setup consumes similar amount of time as Random Walks I in respective algorithms while generating twice as much membership queries. Further increasing of the depth to 4 for Complete Exploration, W Method, and Wp Method setups

Table 6.1: Equivalence test settings

Random Walks I	RandomWalkEQOracle: probability of reset = 0.05, maximum of steps = 20, reset the step counter after counterexample = false, random = new Random(46346293)
Random Walks II	RandomWalkEQOracle: probability of reset = 0.05, maximum of steps = 100, reset the step counter after counterexample = false, random = new Random(46346293)
Random Walks III	RandomWalkEQOracle: probability of reset = 0.05, maximum of steps = 100, reset the step counter after counterexample = true, random = new Random(46346293)
Random Words I	MealyRandomWordsEQOracle: minimum length = 1, maximum length = 20, maximum of tests = 20, random = new Random(46346293)
Random Words II	MealyRandomWordsEQOracle: minimum length = 1, maximum length = 100, maximum of tests = 100, random = new Random(46346293)
Random Words III	MealyRandomWordsEQOracle: minimum length = 100, maximum length = 300, maximum of tests = 300, random = new Random(46346293)
Complete Exploration	CompleteExplorationEQOracle: maximum depth = 3
W Method I	MealyWMethodEQOracle: maximum depth = 1
W Method II	MealyWMethodEQOracle: maximum depth = 3
Wp Method I	MealyWpMethodEQOracle: maximum depth = 1
Wp Method II	MealyWpMethodEQOracle: maximum depth = 3

Table 6.2: Learning setups of DHC algorithm

DHC	Random Walks I	Random Walks II	Random Walks III	Random Words I	Random Words II	Random Words III
learning time [s]	345	1382	1618	269	2326	8071
rounds	1	3	10	1	2	2
membership queries	264	1294	1294	281	1395	1590
executed symbols	914	11660	12017	1099	19832	72739
symbols per query	3.46	9.01	9.29	3.91	14.22	45.75
states	4	18	18	4	18	18
transitions	32	144	144	32	144	144

	Complete Exploration	W Method I	W Method II	Wp Method I	Wp Method II
learning time [s]	312	924	24415	651	139169
rounds	1	1	1	1	1
membership queries	712	728	34136	512	20312
executed symbols	2248	3016	218184	2024	127048
symbols per query	3.16	4.14	6.39	3.95	6.25
states	4	4	4	4	4
transitions	32	32	32	32	32

Table 6.3: Learning setups of Extensible L* algorithm

Extensible L*	Random Walks I	Random Walks II	Random Walks III	Random Words I	Random Words II	Random Words III
learning time [s]	447	6392	6421	340	11689	101820
rounds	1	3	10	1	2	2
membership queries	264	3562	3850	281	4335	14041
executed symbols	924	53885	55436	1099	97569	1006347
symbols per query	3.50	15.13	14.40	3.91	22.51	71.67
states	4	18	18	4	18	18
transitions	32	144	144	32	144	144

	Complete Exploration	W Method I	W Method II	Wp Method I	Wp Method II
learning time [s]	394	807	28116	496	14215
rounds	1	1	1	1	1
membership queries	712	728	34136	512	20312
executed symbols	2248	3016	218184	202	127048
symbols per query	3.16	4.14	6.39	3.9	6.25
states	4	4	4	4	4
transitions	32	32	32	32	32

prolongs the learning time to a matter of days and does not bring any new model attributes. For our SUL, these setups turned out to be inconvenient from the time perspective.

The most comprehensive results were achieved using Random Walks and Random Words equivalence oracle. Three models with 18 states and 144 transitions were obtained in 8 experiments. From the analysis of the models, it is clear that there exists an isomorphism between them, i.e. the models are the same when the states are renamed accordingly.

On average, DHC algorithm was more than 6 times faster than the Extensible L* in delivering the 18-state model. The Extensible L* could not find inconsistency in its 18-state model behaviour even when each of 14041 membership queries executed more than seventy inputs on the SUL. We believe such model should serve well as a basis for the checker.

Now, we get to the examination of the functionality covered by the generated models. The smaller model is depicted in Figure 6.1 and denoted as *model A*. The other model referred to as *checker B* is shown in Figure 6.2. Transitions were simplified by removing the labels describing input and output and by merging parallel edges pointing in the same direction. As we have learned in Section 5.1.1, the original SUL either writes three consecutive values to registers `A`, `B`, and `C` or reads from registers `X` and `Y`.

Model A captures behaviour of the SUL only partially, whereas the read operation is covered. In the state 0, the automaton awaits initialization of the transfer cycle. If the inputs are set correctly, we can go to state 1 by 2 transitions which correspond to read and write. Otherwise (6 transitions), we would remain in the same state. If the write was requested, we get from the state 1 back to the initial state (4 transitions). Provided the read, transfer of value in the register X takes place and we go to the state 2. The state 2 is similar to the initial state where initialization of the transfer cycle is expected. The situation differs in that we have to read the value of the second register. If the transfer cycle was not set correctly, we remain in the state. Otherwise, we go to the state 3. If write

was set, we go back to the state 2. In the case of read, the value of Y is read and we appear in the initial state again.

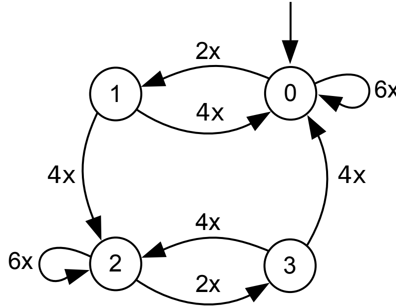


Figure 6.1: Model A (4 states and 32 transitions)

By better generation of counterexamples, the learning algorithm found a case which extends the model A and inferred model B. The case introduces the write operation to the model. As there are three write operations in a row, the model has grown. Therefore, we do not describe the whole model in detail.

In the simplified model B, we can observe that some of the transitions from the state 1 now lead to a new state 4 where the write to registers A, B, C starts. Another difference can be found in the transitions from the state 3. Some of them lead to the state 16 which represents the situation when read has been performed and is followed by the write. This never happens in the real execution of the SUL because the master device is implemented to generate precisely two consecutive read requests.

6.2 Checker Implementation

Model A and model B were converted to VHDL. Then, *checker A* and *checker B* which represent respective models enriched by error-reporting mechanism were generated. The four designs were synthesised and implemented to FPGA. Optimizations were turned off during the implementation phase. From the place and route report, we got the device utilization information summarized in Table 6.4.

Table 6.4: FPGA utilization

	Model A	Model B	Checker A	Checker B	Original
slice registers	4	21	4	21	90
LUTs	3	24	4	25	111
min clock period [ns]	0.987	1.559	0.917	1.562	2.285

In the last column, we observe the implementation of the original PEU slave unit. At first glance, it is apparent that the inferred models consume less space than the original unit. Model A which captures the read operation only is very small using only 4 slice registers and 3 LUTs. Model B occupies 4.3 times less slice registers and 4.6 times less LUTs in comparison with the original unit.

The error-reporting mechanism uses one LUT in both checker implementations. The small size of the checkers is caused also by the fact that checkers validate control signals only and data and address is omitted. The table shows that the speed of both checkers is high enough to work on the maximal speed of the checked unit.

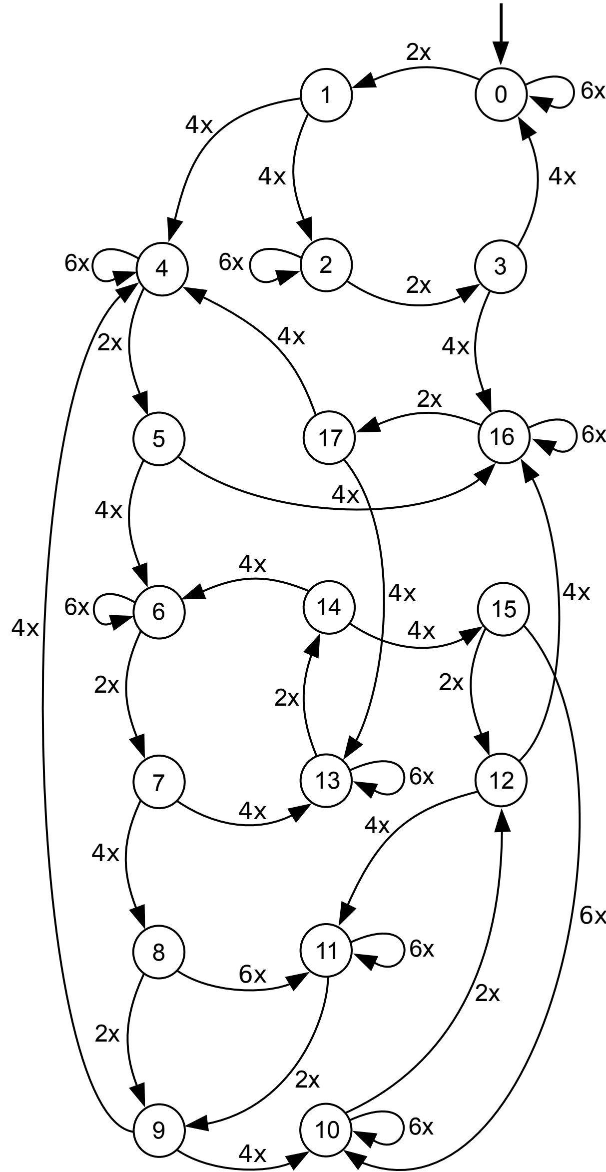


Figure 6.2: Model B (18 states and 144 transitions)

6.3 SEU Injections

We performed SEU injections to the specified region where the design with the checker is placed. A bash script goes through all the bits in all the frames within the region. The region occupies 36 minor columns each containing 1312 bits which gives us 47232 bits altogether. At first, the SEU is injected which causes negation of the specified bit. Then, the experiment controller starts the execution of the design function. When the execution is finished, values of three counters which count real errors, true positives (TP), and false positives (FP) are collected. These values are later used to evaluate the experiment. After that, the design is returned to its original state by another SEU injection. The bit is flipped back and the experiment continues. Experiments involving injections into the entire region are quite a time-consuming task. It takes around 3 days and 18 hours to finish.

Types of Errors

There are several types of errors which can be caused by an SEU depending on where it strikes. The scheme in Figure 6.3 shows a simplified view on possible points of an SEU hit.

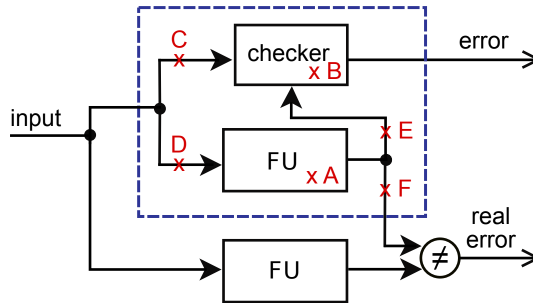


Figure 6.3: Major spots where SEU can strike

Point A represents the situation when the SEU hits the FU. In such case, the checker reports an error. The counter of real errors and the counter of TPs are increased. When the checker is hit (point B), the FSM or the error-reporting mechanism can get broken. As a result, there can be more real errors than the ones reported by checker or some FPs.

Moreover, the input of the checker can be damaged (point C) and thus different from the input of FU, and vice versa (point D). As a result, the outputs of the checker and the FU differ which can result in FPs. Also, the output of FU going to checker can be struck (point E). The checker uses an incorrect value for the comparison, most probably reports an error and the false positive is counted. Similar problem appears when the output of FU on its way outside the region is hit.

We have restricted the blue part of the design to a region and the rest of the design to a different region. However, the router is not bound by the placement constraints and the routings can go through the whole FPGA. This means that SEU placed in a region can hit routings from the rest of the design and thus can have an impact on any part of the design. Furthermore, the clock can be damaged which has fatal consequences on the whole design.

Results

Two major experiments were performed. In the first one, SEUs were injected into the design with the checker. After that, we made another experiment without the checker being implemented. This gave us a better idea about the difference between the number of real errors in different implementations. The total number of real errors was not the same in these cases. In the experiment with the checker, 6881 out of 47232 executions reported real errors. 7228 out of 47232 executions with real errors occurred in the experiment without the checker. This is caused by the router which routes the interconnections differently each time the design is implemented. The implementation without the checker probably contained large amount of routing going through the region which is being injected.

The rest of the result analysis is focused on the experiment with the checker. A part of the output of the experiment can be seen in Table 6.5. We can observe that at the end of the column 30, an important part of the design was placed. After the SEU injection, the counters of real errors, TP, and FP are at their maximum. Placing the SEU to the first bit of column 31 did not cause any error. Recall that due to routing, the values can be influenced by fault injections.

Table 6.5: Results of a few executions after SEU injection

Minor column	Bit	Real errors	True positives	False positives
30	1309	63	63	35
30	1310	63	63	63
30	1311	63	63	63
31	0	0	0	0
31	1	63	63	63

If we had used larger error counters, we would have been able to analyse every test vector in the execution individually. Since we use 6-bit counters, the evaluation of the results was simplified. The analysis was done on the level of executions, i.e. rows in Table 6.5. We investigated, how many executions with real errors and TPs occurred and how often was our checker wrong. The results are given in Table 6.6.

Table 6.6: Analysis of the experiment with the checker

Total number of executions	Cases with real errors	Cases with TP	Real errors not reported	Cases with FP	Cases with FP only
47232	6881	6799	82	7244	887

The experiment performed the total number of 47232 SEU injections and consecutive executions. In 6881 cases, some real errors occurred. In 6799 cases, the checker reported errors correctly. Thus, there were 82 cases when the checker failed to report errors. These errors are fatal to the system as they are not caught and handled.

If we perform reconfiguration when the error is detected by the checker, e.g. the design is a part of an NMR, 0.174% of SEU injections will end up with an undetected error.

The cases when the checker misbehaved were counted. Misbehaviour is represented by occurrence of FP (and no TP) or undetected errors. There were 7244 situations with FP. However, only 887 of them occurred separately, without TP. Then, there were already mentioned 82 cases with undetected errors. This gives us 969 cases with wrong results.

Therefore, if we do not reconfigure and rely only on error-reporting capabilities of the checker, 2.052% of SEUs cause erroneous results.

If there was no checker in the design, the number of undetected faults would be 6881. That means, 14.569% injected SEUs would cause erroneous result. When the checker and reconfiguration is used, the error rate of the unit can be reduced from 14.569% to 0.174%, i.e. by 98.808%.

Chapter 7

Conclusions and Future Work

The reader was acquainted with the discipline of fault tolerance. We focused on FPGAs and described the FPGA configuration process. The methodology of self-checking units and online checkers as a means of error detection were highlighted. The field of active automata learning and the approaches of active automata learning from the perspective of the treating of data were presented. The prime learning algorithm and its latter generalization were explained. The principle of Mealy machine learning was demonstrated on the example of inferring a simple automaton. Learning in practice and the importance of application-specific learning setup were discussed. LearnLib library, its algorithms, and options for automata inference were presented.

The work proposes to use active automata learning for automatic construction of online checkers. The only human expertise required for the construction is the selection of the interface signals protected by the checker.

One of the outcomes of the thesis is the design and implementation of the learning platform which connects the simulator, the learning environment and the finite state machine converter. The learning platform was used to infer the model of the part of the Wishbone bus, the slave unit. The learning setup of the slave unit was devised. It focuses on the control signals while the data and address signals are disregarded.

A number of learning strategies through experimenting LearnLib algorithm settings was developed. Two major models came up from the experiments which lasted from a few minutes up to several hours. The examination of the models shows that the basic behaviour patterns of the original unit is described. The checkers constructed from the inferred models utilise less FPGA resources and achieve higher speed than the original functional unit.

The part of the work which deals with the construction of the checkers was submitted for publication in an international conference [17].

The platform for probing the quality of checkers was designed and implemented. The platform consists of the experiment controller, the SEU generator, and a design implemented in an FPGA. The design was divided into two parts which were implemented in two separate regions in the FPGA. Two experiments involving SEU injections into the specified region of the FPGA were performed. Each of them took almost four days to complete.

The experimental results show that without the checker incorporated in the design, 14.569% of the injected faults cause errors. Provided that we rely on error-reporting capabilities of the checker and do not reconfigure, the checker fails to report the error in 2.052% of the cases. When using the checker together with reconfiguration, 0.174% of SEUs end up with an undetected error. As a result, the checker reduces the error rate of the unit by 98.808%.

The future work should focus on inferring models for different and more complex systems. The evaluation mechanism in the experimental platform could be separated from the injected design, e.g. implemented in FPGA on FITkit. This would prevent the unrelated routing to interfere with the main design. In addition, larger error counters could be used which would enable an examination of each test vector in the execution. Another direction would be the use of Register Mealy machines which support the inferring of models with data parameters. Such an approach would require contribution to LearnLib framework and its further extension.

Bibliography

[1] Aarts, F. Inference and Abstraction of Communication Protocols. Master's thesis, Radboud University Nijmegen, Uppsala University, November 2009.

[2] Aarts, F., Schmaltz, J., and Vaandrager, F. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, vol. 6415 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 673–686.

[3] Abdel-Hamid, A., Zaki, M., and Tahar, S. A tool converting finite state machine to VHDL. In *Electrical and Computer Engineering, 2004. Canadian Conference on* (2004), vol. 4, pp. 1907–1910.

[4] Angluin, D. Learning Regular Sets from Queries and Counterexamples. vol. 75. 1987, pp. 87–106.

[5] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing 1*, 1 (2004), 11–33.

[6] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*, vol. 3472. Springer Berlin Heidelberg, 2005.

[7] Cassel, S., Howar, F., Jonsson, B., Merten, M., and Steffen, B. A succinct canonical register automaton model. In *Automated Technology for Verification and Analysis*, vol. 6996 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 366–380.

[8] Gold, E. M. Complexity of Automaton Identification from Given Data. *Information and Control 37*, 3 (1978), 302–320.

[9] Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., and Ide, H.-D. Efficient regression testing of cti-systems: Testing a complex call-center solution. *Annual Review of Communication, Int. Engineering Consortium (IEC) 55* (2001), 1033–1040.

[10] Howar, F., Isberner, M., Steffen, B., Bauer, O., and Jonsson, B. Inferring Semantic Interfaces of Data Structures. In *Leveraging Applications of Formal Methods, Verification, and Validation*, vol. 7609 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 554–571.

[11] Howar, F., Steffen, B., Jonsson, B., and Cassel, S. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation*, vol. 7148 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 251–266.

[12] HT-LAB. Modelsim Remote Control Demo using Tcl. http://www.ht-labcom/howto/remotemti/remote_modelsim.html, Jan. 2009.

[13] Khoussainov, B., and Nerode, A. *Automata Theory and its Applications*. Progress in Computer Science and Applied Logic. Birkhäuser Boston, 2001.

[14] Koren, I., and Krishna, C. M. *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[15] Margaria, T., Niese, O., Raffelt, H., and Steffen, B. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop. Ninth IEEE International* (2004), pp. 95–100.

[16] Margaria, T., Raffelt, H., and Steffen, B. Analyzing second-order effects between optimizations for system-level test-based model generation. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International* (2005), pp. 461–467.

[17] Matušová, L., Kaštil, J., and Kotásek, Z. Automatic Construction of On-line Checking Circuits Based on Finite Automata. In *Euromicro Conference on Digital System Design* (2014). Submitted.

[18] Merten, M., Isberner, M., Howar, F., Steffen, B., and Margaria, T. Automated Learning Setups in Automata Learning. In *Leveraging Applications of Formal Methods, Verification and Validation*, vol. 7609 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 591–607.

[19] Nerode, A. Linear automaton transformations. *Proceedings of the American Mathematical Society 9*, 4 (1958), 541–544.

[20] OpenCores. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. http://opencores.org/opencores,wishbone.

[21] Podivinsky, J., Simkova, M., and Kotasek, Z. Complex Control System for Testing Fault-Tolerance Methodologies. In *Proceedings of The Third Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN 2014)* (2014), COST, European Cooperation in Science and Technology, pp. 24–27.

[22] Shahbaz, M., and Groz, R. Inferring Mealy Machines. In *Proceedings of the 2Nd World Congress on Formal Methods* (2009), FM '09, Springer-Verlag, pp. 207–222.

[23] Steffen, B., Howar, F., Isberner, M., et al. Active Automata Learning: From DFAs to Interface Programs and Beyond. *Journal of Machine Learning Research 21* (2012), 195–209.

[24] Steffen, B., Howar, F., and Merten, M. Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems*, vol. 6659 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 256–296.

[25] Straka, M., Kastil, J., Kotasek, Z., and Miculka, L. Fault tolerant system design and SEU injection based testing. *Microprocessors and Microsystems 37*, 2 (2013), 155 – 173.

[26] Straka, M., Tobola, J., and Kotasek, Z. Checker Design for On-line Testing of Xilinx FPGA Communication Protocols. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems* (2007), pp. 152–160.

[27] The Apache Software Foundation. Apache Commons Net Library. http://commons.apache.org/proper/commons-net/apidocs/org/apache/commons/net/telnet/TelnetClient.html.

[28] Vasicek, Z. FITkit. www.fit.vutbr.cz/FITkit, Apr. 2011.

[29] Von Neumann, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies 34* (1956), 43–98.

[30] W. Smeenk, D. J., and Vaandrager, F. Applying Automata Learning to Embedded Control Software. http://www.mbsd.cs.ru.nl/publications/papers/fvaan/ESM/, Jan. 2013.

[31] WWW. Automated Telnet Client. http://twit88.com/blog/2007/12/22/java-writing-an-automated-telnet-client/.

[32] WWW. LearnLib - closed-source library. http://ls5-www.cs.tu-dortmund.de/projects/learnlib.

[33] WWW. LearnLib - new version of the library. http://www.learnlib.de.

[34] XILINX. Virtex-5 FPGA Constraints Guide. v10.1.

[35] XILINX. ChipScope Pro Software and Cores, 2000. UG029.

[36] XILINX. Partial Reconfiguration User Guide, 2010. UG702.

[37] XILINX. SEU Strategies for Virtex-5 Devices, 2010. XAPP864 v2.0.

[38] XILINX. CORE generator Guide, 2012.

[39] XILINX. PlanAhead User Guide, 2012. UG632.

[40] XILINX. Virtex-5 FPGA Configuration User Guide, 2012. UG191.

[41] XILINX. Virtex-5 FPGA User Guide, 2012. UG190.