# Immunorepairing of Hardware Systems

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat*

by

**Norma Alicia Montealegre Agramont**

Paderborn, Germany
July 2013

*Supervisor:*

Prof. Dr. Franz Josef Rammig

*Reviewers:*

Prof. Dr. Franz Josef Rammig
Prof. Dr. Sybille Hellebrand

*Additional members of the oral examination committee:*

Prof. Dr. Marco Platzner
Prof. Dr. Hans Kleine Büning
Dr. Matthias Fischer

*Date of submission:*

31.01.2013

*Date of the oral examination:*

22.03.2013

# Summary

Repairing a hardware system that operates in an extreme and inaccessible environment, such as a hardware module of a spaceborne or underwater device on-site, is very costly or even sometimes impossible. That is the reason why, techniques for the design of a hardware system which is able to recover from a failure autonomously are necessary. Such hardware system able to recover from a failure autonomously is named a self-repairing hardware system and is composed basically of a fault recognition module, a recovery procedure module, and the circuit to be repaired. For the design of the fault recognition module, a low fault recognition latency is a requirement, so that it can operate online and concurrently to the operation of the circuit to be repaired. In the literature there are many approaches of self-repairing systems, however, most of them do not deliver details of the design of a fault recognition module which is able to work concurrently to the operation of the circuit to be repaired. Moreover, it has been noticed that many of the existing approaches propose new non-commercial hardware platforms or self-repairing procedures which are hard to reproduce. Hence, this thesis evaluates a set of fault recognition techniques that focus on reducing the fault recognition latency and hardware overhead and delivers a modular framework for the implementation of a self-repairing hardware system.

The human body recovers from illnesses and injury autonomously thanks to the immune system. The field of artificial immune systems transfers the biological principles of the immune system into models and algorithms for solving technical problems. This thesis makes use of some of those algorithms for the design of the fault recognition module, reason why in the title of this thesis immunorepairing is written instead of self-repairing. For the design of the fault recognition module, it is assumed initially the existence of a set of fault pattern vectors that address specific faults in the circuit. Each of those vectors has a recovery mechanism associated. A comparison of the vector containing the current state of the circuit with each of the fault pattern vectors in the available set is executed concurrently to the operation of the circuit using different vector distance metrics. Thereafter, the vector containing the current state of the circuit is classified based on the closest fault pattern vectors and a recovery mechanism is assigned to it if a fault is determined. Two kinds of hardware systems were recognized and taken into consideration: a hardware system with multiple line inputs and outputs and a hardware system with single line inputs and outputs. The inputs and outputs of the first one can be concatenated as vectors with real value elements expressed in fixed or floating point format. In the same way, the inputs and outputs of the second one can be concatenated as vectors with one-bit binary value elements. Hardware systems with multiple line inputs and outputs can be hardware cores that have n-bit floating point format input and output signals, or hardware systems that have input and output signals working with continuous values, such is the case of some sensor or actuator signals. Hardware systems with

single line inputs and outputs can be combinational or sequential digital systems.

In case of vectors with real value elements expressed in fixed or floating point format, the comparison of the vector containing the current state of the circuit with the stored set of fault pattern vectors is done using the Mahalanobis or the set of Minkowski distances. Thereafter, the assignment of a recovery mechanism is done using the k-nearest neighbor or the minimal distance classification methods. The performances of those methods depend on the size of the set of fault pattern vectors. That is, the number of fault pattern vectors and the number of elements in each vector, also called dimension. Therefore, in this thesis, methods for the reduction of the dimension of the fault pattern vectors are evaluated such us "Principal Component Analysis", "Singular Value Decomposition" and "Formal Immune Networks". Moreover, methods for the reduction of the number of fault pattern vectors are also evaluated such us "Death of immune cells with insufficient stimulation", "Elimination of auto-reactive immune cells" and "Apoptosis and auto-immunization". The methods that give the minimal number of incorrectly recognized faults, minimal dimension, and minimal number of fault pattern vectors, are the ones that are recommended to be employed in the design of the fault recognition module. Their use assures a low fault recognition latency and a low consumption of memory resources for storing the fault pattern vector set.

In case of vectors with one-bit binary value elements, the comparison of the vector containing the current state of the circuit with the stored set of fault pattern vectors can be done using the Hamming distance. Using that distance, the result is either zero or one, that is to say, either equal or unequal. When equal, the recovery mechanism associated to the fault pattern vector can be assigned to the vector containing the current state of the circuit in an straightforward manner, without using any classification method. The comparison of the vector containing the current state of the circuit with the stored set of fault pattern vectors is possible to be executed concurrently to the operation of the circuit, by monitoring firstly the current vector of inputs to the circuit. When the vector containing the current inputs matches one of the input vectors from the given fault pattern vectors, then, the vector of current outputs can be compared with the corresponding stored output pattern vector. Such a fault recognition module is comprised of an input vectors monitoring block, an output pattern vectors storage block, and an output vectors comparison block. The design of all those blocks can exploit the unspecified values in the elements of the fault pattern vectors in order to get a reduced hardware overhead. Furthermore, an output vector compactor block, at the output of the circuit, can be used for reducing the number of outputs to be compared, thereby reducing the hardware overhead produced by the output pattern vectors storage block.

Besides the fault recognition module, the self-repairing hardware system requires a recovery procedure module and other supporting modules such as: a controller, enablers of inputs and outputs, a reconfiguration controller for recovery, etc. A modular architecture of a self-repairing hardware system with all those modules is outlined and described in this thesis at the register transfer level using the hardware description language VHDL. The delivered description intends to provide ready to use modules for designing self-repairing hardware systems, and it is a further step towards the automatic design of self-repairing circuits described at the RTL level. The controller and enablers of inputs and outputs help on driving the outputs of the circuit to a safe value, while executing fault recognition and recovery and in case of determining an unrecoverable defect after unsuccessful recoveries. The description also includes a fault injector for debugging or testing the whole system injecting stuck-at faults. That architecture has been implemented and tested for the design of a self-repairing combinational circuit that uses as recovery mechanisms the switching to redundant modules

and the partial reconfiguration of the circuit. The self-repairing combinational circuit has been successfully synthesized and implemented into an FPGA hardware platform.

# Acknowledgments

I would like to thank my supervisor Prof. Franz J. Rammig for the opportunity that he gave me to work in his workgroup "Design of Distributed Embedded Systems" at the Heinz Nixdorf Institute in Paderborn with the objective of writing this dissertation, specially in the writing phase for the proofreading and his patience. I would also like to thank Prof. Sybille Hellebrand for her precise advice during the realization of this dissertation, I really appreciate the bibliographical sources and tools that she suggested me to use and refer to, and her patience with me. I want to thank also my colleague Sebastian Hagenkötter for providing data to test experimentally the algorithms and the work we did together. Finally, I want to acknowledge the members of the examination board Prof. Marco Platzner, Prof. Hans Kleine Büning and Dr. Matthias Fischer for evaluating my work.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Program Codes

# 1

# Introduction

A hardware system which is able to recover from a failure autonomously is named a self-repairing system. A self-repairing system avoids the time span from the manifestation of a failure in it to the presence of a repairing team on site. The logistics necessary for having a repairing team and replacement material on site costs time, effort and money. An extreme example is the on site repairing of the hardware module of a space telescope, which besides of having a very high cost, in some circumstances can even be impossible because of the extreme conditions. Thus, the interest in investigating how to design self-repairing hardware systems.

Biological systems show autonomous recovery mechanisms. In vertebrates the entity responsible of detecting injuries or illnesses and of triggering recovery mechanisms is the immune system. Taking the human immune system as inspiration, the field of artificial immune systems transfers those biological principles into models and algorithms which can serve to solve technical problems. This thesis searches in the field of artificial immune systems for useful methods in the design of self-repairing hardware systems. That is the reason why this thesis has the name of Immunorepairing of Hardware Systems.

This chapter is organized as follows. The first section presents the aim and main objectives of this thesis. The second section outlines the strategy for reaching those objectives, such as the preliminary architecture of a self-repairing system and the fault recognition and recovery mechanisms to be studied. Finally, the last section explains how this thesis is organized.

## 1.1 Objectives of this work

Regarding the duration of a fault in a hardware system, any fault can be classified into permanent, transient or intermittent, [Koren and Krishna, 2007]. A self-repairing hardware system which is implemented by using programmable logic, such as FPGAs, can use reconfiguration for repairing the hardware system from only transient faults. However different types of redundancy can be used for repairing a system, please refer to [Koren and Krishna,

2007]. Firstly, static redundancy is mostly implemented by triplicating the hardware module and determining its output using a voter. That kind of redundancy masks faults and can be used for repairing the hardware module from both transient and permanent faults. Secondly, dynamic redundancy activates an available spare hardware module under a detected fault. Such kind of redundancy can repair a hardware module from both transient and permanent faults. However its use makes more sense when a permanent fault is present. So, in [Bolchini et al., 2011] the type of fault, transient or permanent, is first determined for applying either reconfiguration or active redundancy in a hardware system implemented using FPGAs.

The presence of an error caused by a permanent or transient fault in a hardware system can be detected by using self-checking design techniques, such as the ones explained extensively in [Lala, 2000] and [Gössel et al., 2008]. Those methods serve for first detecting a fault, but the determination of which repairing mechanism should be applied is open.

Looking at concrete efforts of developing complete self-repairing hardware systems in the literature, which are briefly presented in chapter 2, it could be noticed that most of those approaches do not deliver details of the design of a fault recognition module which is able to work concurrently to the operation of the circuit to be repaired. Moreover, it has been noticed that many of the existing approaches propose new hardware platforms or self-repairing procedures which are hard to reproduce.

Hence, this thesis has the aim of providing a general architecture for the design of a self-repairing system. Thereby stressing on the design of a fault recognition module where the faults should not only be classified into transient and permanent, but permanent faults should be classified or distinguished according to the type of required repairing mechanism given at the design time. The objectives that support that aim are:

- The design of a versatile self-repairing architecture using a hardware description language
- The design of a multiclass fault recognition module with a low fault recognition latency

The design of the self-repairing architecture using a hardware description language would make that architecture independent of the implementation hardware platform to be used. A low fault recognition latency, in the second objective, would allow the fault recognition module to operate online and concurrently to the operation of the system.

## 1.2 Strategy

To achieve the proposed objectives, this section provides the methods and tools used in this thesis.

A self-repairing system is composed of the three modules drawn on figure 1.1: the circuit for self-repairing, the fault recognition module and the recovery procedure module. The fault recognition module executes fault recognition and delivers a recovery mechanism which is determined at the time of designing the system. The recovery procedure module executes the recovery mechanism provided by the fault recognition module.

Since the fault recognition module should operate online and concurrently to the operation of the circuit for self-repairing, the technique named Built-In Concurrent Self-Test, in short BICST, is applied, please see figure 1.2 and refer to [Sharma and Saluja, 1988]. So, the

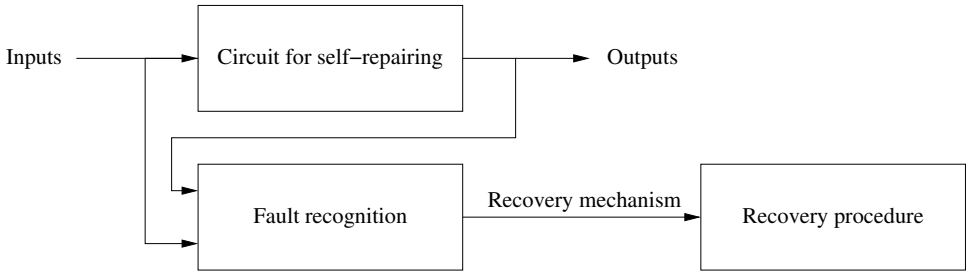



Figure 1.1: Self-repairing circuit

current inputs of the circuit for self-repairing are observed and compared with the fault pattern vectors stored in memory. When the current inputs match to the inputs of any stored fault pattern vector, it is verified if the current outputs of the circuit for self-repairing match with the desired outputs. If they do not match, the recovery mechanism associated to that fault pattern vector is delivered to the recovery procedure module.

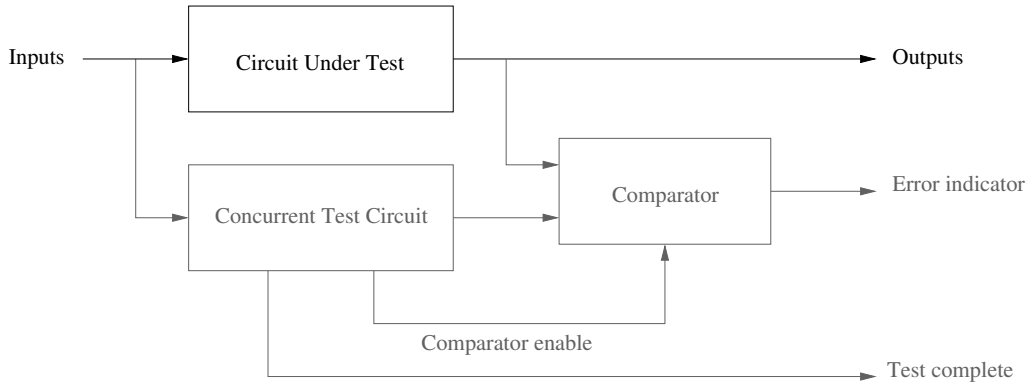



Figure 1.2: Built-In Concurrent Self-Test (abstracted from [Sharma and Saluja, 1988] ©1988 IEEE)

For getting a low fault recognition latency, the number of fault pattern vectors should be minimal. That can be complied considering only fault pattern vectors for critical faults, or by reducing the given set of fault pattern vectors. Therefore, methods for the reduction of the number of fault pattern vectors of the given set and for the reduction of the dimensions of those fault pattern vectors are studied.

A concurrent fault recognition module for combinational circuits can be implemented using a small set of fault pattern vectors, as stated in [Kochte et al., 2009]. In that publication, they use a small set of fault pattern vectors for the so called random pattern resistant faults, which are faults that can not be covered by a random generated testing set. For a self-repairing circuit, the set of fault pattern vectors to be used can be prepared to cover only critical faults. A review of techniques for the reduction of the hardware overhead produced by such a concurrent fault recognition module for combinational circuits by using unspecified values is done.

In the case of combinational or sequential circuits, the pattern vectors are vectors that contain one-bit binary value components. In case of hardware cores with, e.g., 32-bit floating point inputs and outputs, or analog hardware modules which have input and output signals

with continuous values, the fault pattern vectors are vectors that contain real value elements. In the last case, reduction of the given set of fault pattern vectors is necessary. Since fault-recognition can be inspired by methods that the biological immune system uses for finding illnesses, suitable methods were searched in the field of artificial immune systems. A special attention has been devoted to the cytokine Formal Immune Network method presented in [Tarakanov, 2008], because it performs reduction of fault pattern vectors and considers a class associated with each fault pattern vector, named cytokine, field which can be assumed to be the recovery mechanism. Besides, that method allows self-learning, which is understood as the addition of new fault pattern vectors online without requiring a recomputation of the transformation matrix used for fault pattern vector dimension reduction.

The recovery procedure is normally hardware platform dependent. However, techniques such as redundancy and partial reconfiguration can be used for many hardware platforms, reason why a hardware platform with an FPGA that supports partial reconfiguration has been used. Although, Xilinx FPGAs are SRAM-based and consequently prone to single event upsets, this thesis assumes that repairing should be executed when a permanent fault, rather than a transient fault, is detected. A permanent fault is modeled as a stuck-at fault and a fault injection module that inserts such faults for testing the self-repairing system in the simulation and in the final implementation is added to the self-repairing architecture.

The self-repairing architecture, shown in figure 1.3, applies some ideas of the self-repairing architecture taken from [Marinos, 1969] and shown in figure 1.4. That architecture considers an Input/Output Enabler which permits that only a verified result is present at the outputs. That feature is useful in the design of safety critical systems[1], which are systems that require fail-safe outputs, please refer to [Sutton, 2012]. A verified output is flagged by the output *Ready*, which can be seen in figure 1.4. After a determined number of failed recoveries, the system flags an unrecoverable defect in the system by means of the signal *Defect*, which can be seen on the same figure 1.4. In that case, the outputs maintain the verified value of the prior inputs, guaranteeing a fail-safe state of the system.

The hardware description language used for describing the self-repairing architecture is VHDL. The description proposes the delivery of modules, that is to say prepared template code for designing self-repairing hardware systems, inspired by the work in [Entrena et al., 2001] and briefly explained in section 2.4.1.

## 1.3 Organization of this work

This thesis is organized in more seven chapters with the following content. Chapter 2 presents briefly the work related to this thesis in the current literature and comments their differences and weaknesses. Chapter 3 provides a biological background, models and algorithms of artificial immune systems. That chapter explains biological concepts for non-biologists, provides the most known artificial immune algorithms with their pseudocodes, and gives a comparison mentioning the application of each algorithm. Chapter 4 deals with fault recognition concepts and thereafter concentrates on methods of fault recognition for systems whose fault pattern vectors contain real elements. That chapter explains mathematically the cytokine Formal Immune Network algorithm presented in [Tarakanov et al., 2005] and its similarities to the Principal Component Analysis algorithm for dimension reduction. Chapter 5 presents the

[1] A safety critical system is a system whose failure can produce injury to people or damage to the environment or equipment, reason why upon a failure the system should automatically become safe.

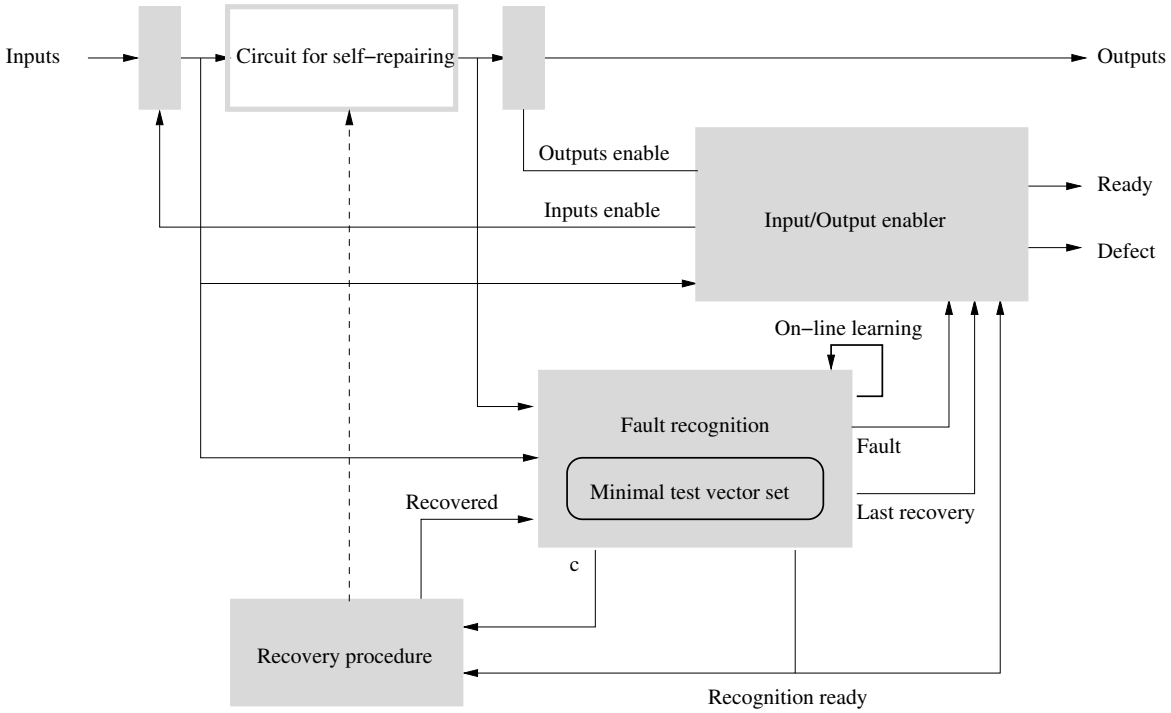



Figure 1.3: Self-repairing architecture

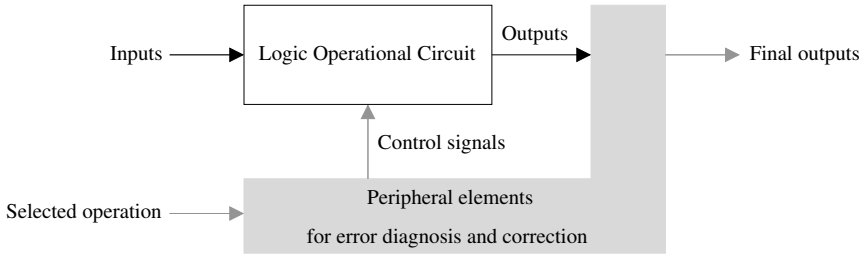



Figure 1.4: LOC and peripheral elements required for error diagnosis and correction (abstracted from [Marinos, 1969] ©1969 IEEE)

implementation and evaluation of the algorithms presented in chapter 4, which are adequate for fault pattern vectors with real value elements that proceed from complex digital systems such as hardware cores or analog systems. The second part of chapter 5 presents methods for reducing the hardware overhead of a concurrent fault recognition module for combinational circuits by using the unspecified values of its fault pattern vectors that have one-bit binary value elements. Chapter 6 presents the self-repairing architecture described in VHDL, its simulation and implementation. The last chapter exposes the major contributions of this thesis and lists possible further work. This thesis is organized in nearly independent chapters. That means that each chapter is understandable without reading prior chapters unless chapter 5, which requires the understanding of the algorithms presented in chapter 4.

## 1.4 Bibliography

Bolchini, C., Sandionigi, C., Fossati, L., and Codinachs, D. M. (2011). A reliable fault classifier for dependable systems on SRAM-based FPGAs. In *17th International On-Line Testing Symposium - IOLTS 2011*. IEEE.

Entrena, L., López, C., and Olías, E. (2001). Automatic Insertion of Fault-Tolerant Structures at the RT Level. In *7th International On-Line Testing Workshop*, pages 183–200. 48-50.

Gössel, M., Ocheretny, V., Sogomonyan, E., and Marienfeld, D. (2008). *New Methods of Concurrent Checking*. Frontiers in Electronic Testing. Springer.

Kochte, M. A., Zoellin, C. G., and Wunderlich, H.-J. (2009). Concurrent Self-Test with Partially Specified Patterns For Low Test Latency and Overhead. In *14th European Test Symposium*, pages 53–58. IEEE Computer Society.

Koren, I. and Krishna, C. M. (2007). *Fault Tolerant Systems*. Morgan Kaufmann.

Lala, P. K. (2000). *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann.

Marinos, P. N. (1969). The Organization of a Self-Repairing System from Multifunctional Units. *Proceedings of the IEEE*, 57(7):1320.

Sharma, R. and Saluja, K. K. (1988). An Implementation and Analysis of a Concurrent Built-In Self-Test Technique. In *18th International Symposium on Fault-Tolerant Computing - FTCS 18*, pages 164–169. IEEE.

Sutton, A. (2012). No Room for Error: Creating Highly Reliable, High-Availability FPGA Designs. Technical report, Synopsis.

Tarakanov, A., Goncharova, L., and Tarakanov, O. (2005). A Cytokine Formal Immune Network. In *8th European Conference on Advances in Artificial Life - ECAL 2005*, volume 3630 of *Lecture Notes in Computer Science*, pages 510–519. Springer.

Tarakanov, A. O. (2008). Formal Immune Networks: Self-Organization and Real-World Applications. In Prokopenko, M., editor, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 271–290. Springer.

# 2

# Related work

Research in the area of self-repairing hardware systems addresses: different types of faults, e.g., transient or permanent faults regarding its persistence; different types of hardware platforms, e.g., Field Programmable Gate Arrays, Programmable Systems on Chips, etc; and different levels of abstraction, e.g., system level, Register Transfer Level (RTL), gate level, transistor level, etc. Similar approaches usually differ on the employed error detection method, recovery mechanism, or system architecture, which in some cases are inspired by biological systems.

The revised work has been organized in the following sections: general work done on self-repairing hardware, approaches inspired by biological systems, work done on introducing self-repairing in the FPGA fabric, and finally work done on self-repairing introduced in the description of hardware systems. A subsection is devoted for each specific work, in which the principle is introduced with the help of a figure abstracted from the respective publication when necessary. At the end, a section is devoted to comments regarding the differences and the shortcomings of the different approaches. It is not strictly necessary to keep on reading sections 2.1, 2.2, 2.3 and 2.4.1 which introduce briefly the different approaches. It can be skipped to the section 2.5 devoted to comments and on demand or by interest look at the respective referenced subsection.

## 2.1 Self-repairing hardware

This section introduces work done in the field of self-repairing hardware systems in general, leaving approaches inspired by biological systems and work done on self-repairing of the FPGA fabric aside.

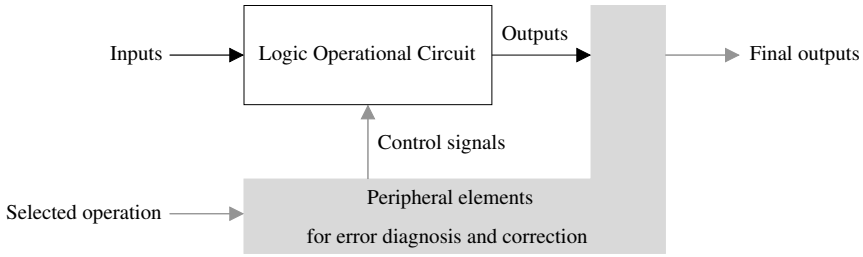



Figure 2.1: LOC and peripheral elements required for error diagnosis and correction (abstracted from [Marinos, 1969] ©1969 IEEE)

### 2.1.1 Multifunctional units

One of the most earlier publications on self-repairing systems and main reference in this thesis is the letter of Peter N. Marinos named "The Organization of a Self-Repairing System from Multifunctional Units", please refer [Marinos, 1969]. It proposes the design of a self-organizing system using multifunctional devices, which give to the system a high flexibility in the configuration of its functioning and a capability of self-repair. A multifunctional logic device contains a so called logic operational circuit, shown in figure 2.1, which besides inputs and outputs, has control signals for selecting the operation $f_i$ to be performed. In order that upon a failure, the logic operational circuit is able to perform the same operation but error free, each one of the operations $f_i$ has assigned additional values for the control signals. Thus, the values of the control signals can be partitioned into classes, a class corresponding to each function $f_i$. The peripheral elements required for the error diagnosis and correction of a logic operation unit are shown in figure 2.1. In this work the idea of having a multifunctional logic device for a self-organizing and self-repairing system, the arrangement of peripheral elements for error detection and correction, and the partitioning of the states of a self-repairing automaton in classes are valuable.

### 2.1.2 Dynamic partial reconfiguration for testing and repair

A Field Programmable Gate Array, in short FPGA, is an integrated circuit with programmable logic. The programmable logic allows to configure the functionality of the circuit after the manufacturing of the chip. Chip is a typical name given to an integrated circuit. By some FPGA chips it is possible to program only a part of its programmable logic while the remaining logic is operating, process named as dynamic partial reconfiguration. Dynamic partial reconfiguration can be used for implementing a large circuit into an FPGA that does not present enough hardware resources. The work shown in the publications [Paulsson et al., 2006b] and [Paulsson et al., 2006a] suggests using dynamic partial reconfiguration of an FPGA not only for implementing a given circuit, but also for detecting errors on it and for its recovery in the case of a detected error. For that, the FPGA is partitioned into slots which are partial reconfigurable regions able to contain a circuit partition each, as shown in figure 2.2. For error detection, a test bench module, created for testing a single slot or the circuit partition implemented in that slot, is configured into the FPGA when a slot is available. How the test bench module should be designed and executed is not mentioned, but it is recommended to restrict the testing time in order of not interfering the overall circuit operation. N-modular redundancy is a method of fault masking, where n copies of a circuit partition get the same

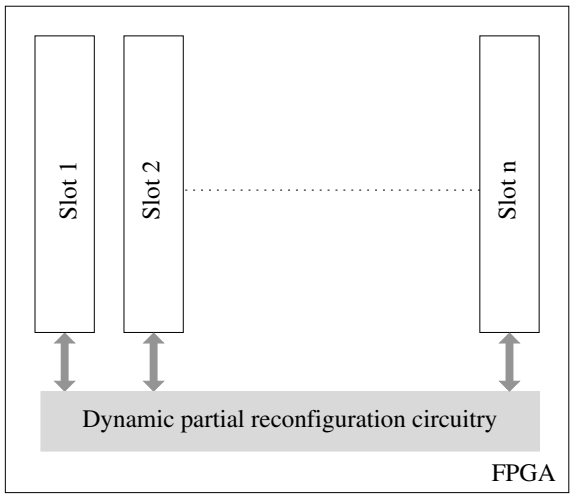



Figure 2.2: Partial reconfigurable regions in the FPGA as slots for dynamic partial reconfiguration (abstracted from [Paulsson et al., 2006a] ©2006 IEEE)

input data and deliver their outputs to a voter, which gives as final output the output of the majority. A subcase of N-modular redundancy with N = 3 is the so called Triple Modular Redundancy, in short TMR. Triple Modular Redundancy, where the copies of the circuit partition and the voter module are implemented in different slots using dynamic partial reconfiguration, named as dynamic TMR, is suggested as a second method for error detection. As recovery strategies are suggested the partial reconfiguration of a failed circuit partition in the same slot, the partial reconfiguration of a failed circuit partition in a different slot, and the blocking of a slot for future use when faulty. Although the presented testing strategies can be useful for large circuit that use dynamic partial reconfiguration for its implementation, for other circuits the complexity of the required controller looks too high.

### 2.1.3 Distributed self-repairing of a network of FPGA nodes

Self-repairing for a network of nodes composed each one of an FPGA board attached to a wireless networking module is presented in [Venishetti et al., 2007], [Sreeramareddy et al., 2008], [Akoglu et al., 2009] and [Sreeramareddy et al., 2010]. Thereby, self-repairing is executed at the node and at the network levels. For the determination if a node presents an error or not, the circuits implemented in the FPGA of each node are partitioned into components and subcomponents. Built-in self-testing hardware is added to each component. Thereby, a Linear Feedback Shift Register, in short LFSR, is used for generating pseudo-random input test patterns, and a Multiple Input Shift Register, in short MISR, is used for compressing the output vectors generating a signature. The generated signature is compared with the expected signature stored in extra memory. If a mismatch is detected, testing at the subcomponent level, only for that component, is triggered. Subcomponent testing is performed by applying test pattern vectors stored in memory at the inputs and by subsequently comparing the vectors at the outputs with the respective expected output vectors stored also in memory. Upon a mismatch, partial reconfiguration of the failed subcomponent is initiated. That process is able to counteract a temporary fault produced for example by a bit flip in an internal register of the FPGA. If the error remains after partial reconfiguration, it means that a permanent

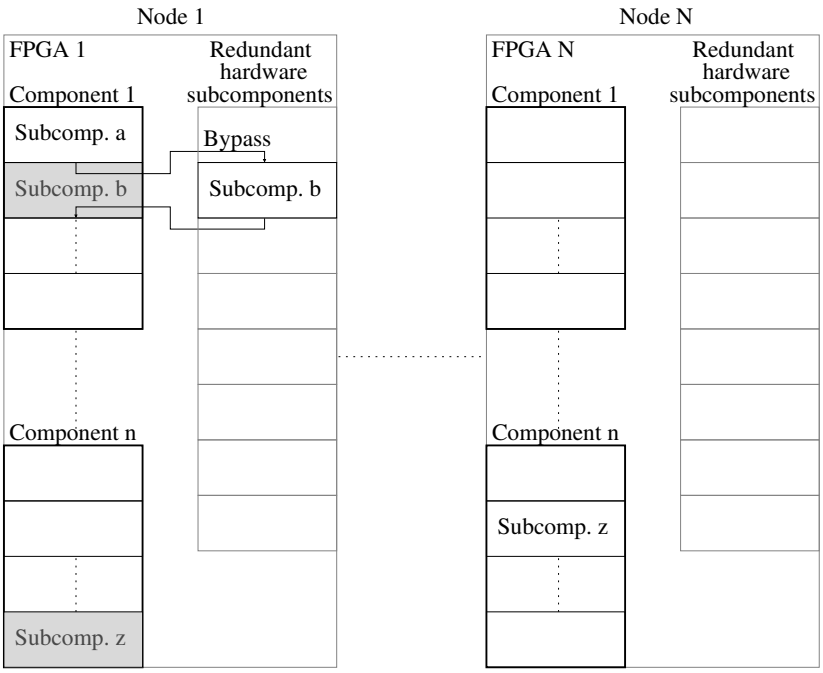



Figure 2.3: Subcomponent self-repairing (abstracted from [Akoglu et al., 2009] ©2009 Springer)

hardware fault in the FPGA area occupied by the subcomponent is present. Therefore, a redundant hardware module that implements the subcomponent functionality and placed in another area on the FPGA is enabled and connected diverting the data lines to and from the subcomponent, as shown in figure 2.3. If a further error in the same subcomponent is present and if all available redundant hardware modules that implement the failed subcomponent functionality are exhausted, fault information is provided to the wireless networking module of the node. That module sends information about the failed subcomponent to the master node, which reassigns the task of the failed subcomponent to another node in the network. Reassignment of tasks is also done by the master node when it perceives, by polling, that a node fell down. One observation about this work is that it does not consider as an issue the time that self-testing of components and subcomponents take and the frequency in which self-testing is triggered. Besides, it does not mention that a component should go off-line in order to execute self-testing using LFSR and MISR. Nevertheless, in [Sreeramareddy et al., 2010] it is proposed the use of partial bitstream relocation, in short PBR. The partial bitstream relocation method, taken from [Sudarsanam et al., 2009], allows firstly the reduction of the time that partial reconfiguration of a subcomponent takes and secondly the subcomponent relocation to an empty partial reconfigurable region in the FPGA on the fly. All that, by using only the frame data part of the partial bitstream of the subcomponent, saved in Block RAM memory in the FPGA instead of in external memory, for the partial reconfiguration.

### 2.1.4 Small-scale reconfigurability for fault detection, diagnosis and recovery

The work presented in [Kumar and Lach, 2003] considers to decompose a circuit to the cone level. Where a cone is a combinational block with many inputs, but only one output. An

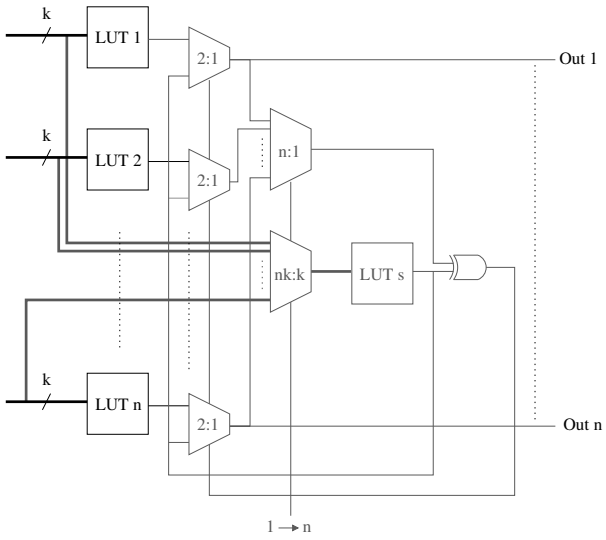



Figure 2.4: Cone-level fault detection, diagnosis and recovery (abstracted from [Kumar and Lach, 2003] ©2003 IEEE)

SRAM-based k-input lookup table, in short k-LUT, can implement the functionality of any cone that has as a maximum k inputs. For fault detection, one of the k-input lookup table executes redundant computation of all $n$ cones in a rotating manner. That is possible by using the multiplexer $nk : k$ shown in figure 2.4. Then, the output of the rotating k-LUT is compared with the output of the selected cone through a XOR gate. When a fault at the cone is flagged by that XOR gate, the k-LUT can be permanently configured to implement the functionality of the faulty cone. That can happen when the $nk : k$ multiplexer selects the inputs of the faulty cone and the $2 : 1$ multiplexer at the output of the faulty cone selects the LUT output, as can be seen also in figure 2.4. What this work does not mention is how the select lines of all multiplexers are controlled.

### 2.1.5 Logic self-repair

The work in [Koal et al., 2012], [Koal et al., 2011], [Koal et al., 2009], [Koal and Vierhaus, 2008] and [Kothe and Vierhaus, 2006] intends to adapt existing self-repairing approaches for memories and regular logic, like FPGAs, to logic designs. This work relies on the idea that there exist functional units in a logic design which are implemented multiple times, or seen from another side, the design is built out of regular components, e.g., ANDs, ORs XORs, blocks of gates, adders, ALUs, etc. Out of such identical functional units or regular components in a logic design, a reconfigurable logic block, in short RLB, is proposed. Such RLB is provided with one spare unit that can serve for replacing a faulty functional unit by controlling some MUXs in the required way, the same way as shown in subsection 2.1.4 and figure 2.4 where a functional unit is a Look Up Table. The functional units in an RLB can be tested at start-up for encountering production faults, or in idle times during operation for encountering wear-out defects. The test is proposed to be done by a built-in self-test module which executes a cyclic test of all functional units in the RLB. Thereafter, when a functional block is encountered faulty, a built-in self-repair module configures the multiplexers in the way that the spare unit replaces the faulty functional unit. Furthermore, for more flexibility,

in [Koal et al., 2009] it is proposed a regular switching scheme for the inputs and the outputs instead of multiplexers. With such a switching scheme and more spare units in the RLB, triple modular redundancy is proposed for safety critical functional units that have hard real time constraints. Regarding the granularity of a functional unit, it is recommended to apply redundancy through RLBs in functional units with more than 400 transistors due to the enormous overhead that redundancy can bring with in units with less granularity. It is important to note that this approach is dependent on the regularity of the logic and requires a method for finding such regularity.

### 2.1.6 Dual-FPGA architecture for autonomous self-repair

An architecture for autonomous self-repair with two FPGAs with a soft microcontroller each is presented in [Mitra et al., 2000] and [Mitra et al., 2004]. The function of the soft microcontroller in the first FPGA is to reconfigure the second FPGA and vice versa. For that, it must be assured that the communication between FPGAs, the bus, the memory that stores the alternative FPGA configurations, and the reconfiguration circuitry, are reliable. Three issues in autonomous self-repair are: concurrent error detection, fault location, and recovery. Since, the dual-FPGA architecture has been developed at the Stanford Center for Reliable Computing within the Reliability Obtained by Adaptive Reconfiguration project, in short ROAR, many techniques have been studied to cope with the issues of an autonomous self-repair. Those techniques are shortly listed with references below organized regarding the type of fault that they address by the detection, recovery and fault location.

Faults can be transient or permanent. Transient faults in an FPGA can be distinguished between: transient faults that affect the configuration of the FPGA; and transient faults that affect the user application data, when memory modules have been implemented using the Configurable Logic Blocks, in short CLBs, of the FPGA.

For concurrent error detection implemented in hardware, techniques for the detection of transient faults that affect the configuration of the FPGA are: the synthesis of a circuit with a parity check code towards a self-checking circuit, please see [Touba and McCluskey, 1997] and [Zeng et al., 1999]; and the use of design diversity for the design of a redundant circuit for error detection, please see [Mitra, 2000]. The last technique considers a model of a common failure mode, which is a failure present by using simple circuit duplication without considering diversity in the design.

For concurrent error detection implemented in software, in [Oh, 2000] the following techniques that address transient faults in the application memory are presented: Control Flow Checking by Software Signatures, where signatures are embedded into the program during compilation and compared with run-time signatures during execution; Error Detection by Duplicating Instructions, where instructions are duplicated at compile-time; and Error Detection by Diverse Data and Duplicated Instructions, where a program is compiled to a new program with diverse data.

A concurrent detection of permanent faults is a challenge. Some design attempts are the following: a built-in self-test circuitry in all subcircuits, using a reduced number of built-in seeds and encoded in the circuitry not in memory, please see [Al-Yamani, 2004]; the use of patterns stored in memory for testing the functionality of the circuit or subcircuits, please see an approach at [Li et al., 2008], [Mitra, 2008] and [Li et al., 2010]; the use of self-checking circuits designed using diverse duplication and organized in a pipeline fashion, where the outputs of the checkers are taken out through an scan chain towards a controller placed in

the second FPGA, please see [Mitra et al., 2004]; and finally, the reduction of the number of test configurations for an exhaustive test of defects on the hardware of the FPGA, please see [Chmelař, 2004b] and [Chmelař, 2004a].

For the recovery of the circuit from transient faults that affect the configuration of the FPGA, a fault free configuration can be loaded in the FPGA using partial reconfiguration. In case the transient faults affect the user application data, rollback and roll-forward techniques can be employed for recovery, see one approach at [Yu, 2001]. However, through partial reconfiguration, the content of the user application data can be altered. For that, the dirty bit memory coherence technique is presented in [Huang, 2001] and [Huang and McCluskey, 2001], which associates a bit for flagging a possible threat to a column of CLBs containing a CLB that has been configured as memory.

For the fault location and recovery of the circuit from a permanent fault, column based precompiled configurations having an intentionally unused column are used for blind reconfiguration, where all possible precompiled configuration are loaded one after another until a configuration that avoids the fault is found, please see [Mitra et al., 2004].

## 2.2 Self-repairing approaches inspired by biological systems

Some approaches for self-repairing hardware systems are inspired by biological processes. Some of the most important approaches are introduced below.

### 2.2.1 Immune system paradigm

It is an approach presented in [Avižienis, 2002], which proposes to insert a subsystem analogous to the immune system of the human body into a hardware system. That subsystem is called Fault Tolerance Infrastructure, in short FTI, and it follows a set of design principles which are based on key attributes of the human immune system. Taking those attributes, the FTI first of all must be software-independent, consist of hardware and firmware elements only, be distributed, have its own communication links, and have a fully fault-tolerant implementation. The proposed FTI in [Avižienis, 2006], which principles evolved from the Test-and-Repair processor STAR presented in [Avizienis et al., 1971], is a system composed of four types of controllers called nodes, which are implemented using read-only microcode.

The nodes in a FTI are responsible of protecting a computing node **C**. In particular, the so called monitor node **M** receives error messages from the computing node through an adapter node **A** and sends appropriate recovery commands. Those recovery commands are stored in read-only memory in the monitor node. For fault tolerance purposes there are more than one monitor nodes which are connected through an special bus. A monitor node is connected to the startup, shutdown and survival node **S3**. The functions of that node are to control the power-on, power-off, to provide a non-volatile and radiation hardened storage for system time and configuration, and to indicate the health status of the computing node. The **S3** node has been separated from the monitor node **M** to make the node that must survive catastrophic events as small as possible. The fourth type of node, the decision node **D**, provides decision services for N-version software executing on diverse computing nodes. Besides, that node serves as a communication link between the software implemented in a computing node and the monitor node.

The FTI can be placed in a hardware component, or it can be employed to protect a board with several components as shown in detail in [Avižienis, 2000]. Another option is to build a

hierarchy of FTIs, which requires dedicated communication links among them. Furthermore, considering that a self-organizing system is composed of agents, the immune system paradigm is proposed to be also employed implementing an FTI in each of the agents, please refer to [Avižienis, 2006].

### 2.2.2 POEtic tissue

In an attempt to classify hardware systems inspired by biology, three inspiring domains of biology: Phylogenesis, Ontogenesis and Epigenesis, were identified in [Sipper et al., 1997], resulting in the three axes of the POE model. Phylogenesis is defined as the development or evolution of a particular group of organisms [Dictionary.com, 2012]. Ontogenesis is defined as the development of an individual organism from embryo to adult [Dictionary.com, 2012]. Epigenesis is defined as the development of an organism considering the action of the environment [Dictionary.com, 2012]. In those three domains, remarkable approaches are: in the domain of Phylogenesis, *Evolvable hardware* which is motivated by the concept of evolution of a group of organisms; in the domain of Ontogenesis, *Replicating and regenerating hardware* which most important example is *Embryonics* that is motivated by the concept of the development of a multicellular organism from a mother cell; and in the domain of Epigenesis, *Learning hardware* which most important example is *Immunotronics* that is motivated by the concept of the development of an organism by learning.

In the project named "Reconfigurable POEtic Tissue" developed by a group of Universities, please see [École Polytechnique Fédérale de Lausanne et al., 2005], the concept of a programmable hardware device for assisting on the design of systems that apply the above mentioned three axes of biological inspiration has been developed. The hardware device consists of units having a three layer structure corresponding to each POE model axis, as seen in figure 2.5. The first layer, named genotype layer, disposes of a memory where the different functions that a unit in the device can adopt are stored. Such information is named the genome of the tissue and can be changed by evolution. The second layer, named mapping layer, is responsible of selecting which part of the information contained in memory will determine the operation of the unit. That information serves for configuring the unit in a similar way to the cellular differentiation and could be helpful at the time of implementing self-repairing of the hardware device. The third layer, named phenotype layer, contains an application dependent processing unit surrounded by the communication unit, which is there for getting stimulus from the environment for the learning process. For a detailed description of the POEtic tissue and its hardware implementation please see [Tyrell et al., 2003] and [Moreno et al., 2004]. Each unit in the POEtic tissue is understood as a molecule. However, for the implementation of self-repairing, several molecules are grouped together to form a cell. Faults are detected in a cell by duplicating its molecules and comparing their outputs. The system is recovered using spare cells, please see [Tyrell and Barker, 2006] and [Barker et al., 2007]. The implementation of a desired application into the proposed hardware requires that specialized hardware platform ready to use, reason why the evaluation of the performance of that hardware platform turns impossible.

### 2.2.3 Evolvable hardware

It is a new way of designing electronic circuits using algorithms inspired by the evolution of species. As described in [Torresen, 2004], at the very beginning a randomly generated

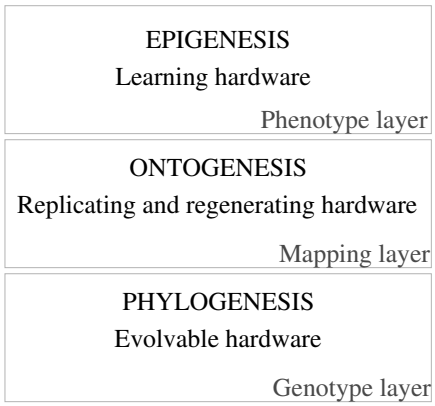



Figure 2.5: POEtic layers (abstracted from [Tyrell et al., 2003] ©2006 Springer)

population of circuits is created. All generated circuits are evaluated according to the provided specification which consists of input/output vectors only. For that evaluation, the circuits can be simulated or implemented in hardware. In case the circuits are implemented in hardware, FPGAs are very useful. That is a fact that allows to handle with the bitstream for configuring the FPGA as a circuit in the population. To each circuit a fitness number that reflects how well it performs according to the given specification is assigned. The population evolves in several iterations using crossover and mutation. Crossover combines parts of the two best circuits. Mutation changes some parts of the best circuits randomly. At the end the best circuit is given as a result. The limitation of this approach for designing circuits is the bitstream length.

The evolution of a circuit could also be executed online, but reconfiguration times are a limiting factor. An approach that uses evolution for self-repairing a part of a combinational circuit is shown in [Garvie and Thompson, 2004]. There, Triple Modular Redundancy, Scrubbing and Jiggling are combined together. Triple Modular Redundancy is used for avoiding shutting down the system during repairing. By using Scrubbing, a module, of the three available modules, is reconfigured when its output is different from the other two redundant modules. When that mechanism fails to repair the module, Jiggling is triggered. By using Jiggling, the two healthy modules maintain the circuit operating by driving the majority voter in the TMR, while a 1+1 evolution in the faulty module takes place. That evolutionary strategy considers one elite circuit and one mutated circuit, therefore the name. If the mutated circuit is superior to the other one, the mutated circuit replaces the elite circuit, otherwise another mutant circuit is created. TMR and Scrubbing are considered for repairing transient faults, and Jiggling is considered for repairing permanent faults. It is to note that Jiggling prevents the system from reacting against transient faults while repairing the failed module by evolution. In this approach, fault recognition is executed by using TMR, no redundancy of modules is added for repairing besides Triple Modular Redundancy, no saved bitstreams are necessary for reconfiguring a module, and the architecture of the TMR is enhanced at the voter by a repairer circuit which is aimed to be smaller than using redundant modules for repairing.

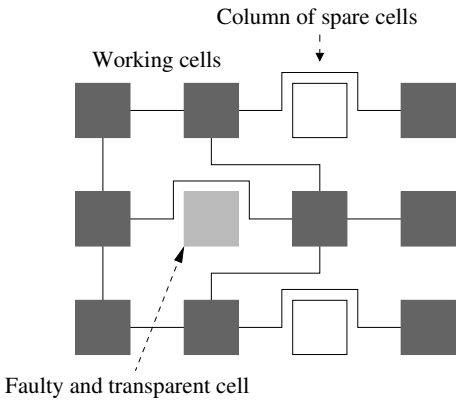



Figure 2.6: Self-repairing in an array with spare components (abtracted from [Tempesti et al., 1997] ©1997 Elsevier)

### 2.2.4 Embryonics

The term derives from the words embryology and electronics and has been coined by the Laboratoire de Systemès Logiques at the Ècole Polytechnique Fèdèrale de Lausanne, please see [Marchal et al., 1996]. Embryonics fusions concepts from biology, cellular automata and Field Programmable Gate Arrays and refers specifically to a two dimensional array of cells, where each cell contains a processing element and a memory. The processing element implements a particular logic function, which is determined by a configuration register in the memory. The configuration register is selected by the coordinates of the cell. Each cell contains the configuration registers of all cells in the same column only and not of all cells in the array for saving memory space. One extra register contains a so called transparent configuration which is loaded when the cell is reported faulty. With a faulty cell configured as transparent, the routing is done around that cell as shown in figure 2.6. Fault detection in a cell is performed by a test module which compares the outputs of the two identical modules that implement the function of the cell. An implementation of a cell using multiplexers is presented in [Tempesti et al., 1997].

The embryonics project proposes also a model inspired by biology for organizing hardware which consists in four hierarchical levels. In that hierarchy the basic element in the first level is a molecule, which is the module defined as a cell in previous publications, e.g., [Ortega-Sanchez et al., 2000] and [Tempesti et al., 1997]. In the second level, a set of molecules forms a cell, which is basically a processor conceived as a set of smaller functional blocks. In the third level, a set of cells forms an organism, which is conceived as a multiprocessor system. The fourth level is seen as a set of organisms that gives rise to a population, which has not been clearly defined in hardware terms. In conclusion, the embryonics approach proposes a new FPGA composed of an array of configurable logic blocks which are conceived as molecules or basic elements of the four-level hierarchy.

### 2.2.5 Immunotronics

This term comes from immunological electronics and has been coined by the Intelligent Systems Research Group at the University of York. Immunotronics deals with the design of a fault recognition module for a hardware digital system using learning techniques inspired by

the immune system. Considering that many digital systems can be represented by a single or a set of finite state machines, in [Bradley, 2002] a single finite state machine with a fault detection module is presented. That fault detection module considers data vectors formed by concatenating inputs|previous-state|current-state. The recognition of faults is based on the so called nonself recognition. The nonself recognition consists on a partial matching of the current data vector inputs|previous-state|current-state obtained from the state machine with the so called tolerance condition vectors saved in memory. The use of partial matching allows to reduce the number of tolerance condition vectors saved in memory. A content addressable memory makes feasible to search in parallel all memory locations in a single clock cycle, accessing the device by using the data rather than by using the address. The tolerance condition vectors are generated offline by using a set of self vectors **S** which represent all possible vectors inputs|previous-state|current-state in the error-free state machine. Randomly generated tolerance condition vectors which partially match in more than $c$ contiguous bits are rejected and all other are saved as the vectors recognizers of faults. That process is named the negative selection algorithm which is inspired by the maturation of new immune cells in the human immune system. This approach supposes that the state machine is completely observable since the recognition is based in the input|previous-state|current-state vector. Other classical methods for fault detection of sequential circuits use an ordered sequence of input|output vectors considering that the state machine is not completely observable. A 100% fault recognition is guaranteed when a total matching is used and all tolerance condition vectors are stored in memory, which requires a huge memory for big circuits, reason why the fault recognition is traded off with the memory space.

An attempt to add self-repairing to Immunotronics has lead to a combination between Immunotronics and Embryonics as presented in [Bradley et al., 2000]. In figure 2.7 a second layer of immune cells can be observed. That network of immune cells replaces the fault detection mechanism that the embryonic cell had and employs the method of fault detection introduced by Immunotronics. Each immune cell is able to monitor all adjacent cells through the trans-layer connections. Besides, the immune cells are able to communicate among each other through the lymphatic connections. That scheme allows to have a distributed fault detection avoiding a single point of failure in the fault detection elements. A final hardware implementation of this approach has not been presented, however, the idea of this approach has been applied to the POEtic tissue which is a project executed by the same Universities that worked on Immunotronics and Embryonics.

### 2.2.6 e-DNA

e-DNA architecture is the name given at the Technical University of Denmark and presented in [Boesen et al., 2011] to a network of chip arranged as an array of homogeneous processing units named eCells. Each eCell can communicate through its network adapter with all its eight adjacent neighbors in the network. Each eCell has a RAM memory, a microprocessor and an arithmetic logic unit. The RAM memory in each eCell contains the whole eDNA program that is executed by the eCells and specified by the programmer. The eDNA program is written in the eDNA language, which has control structures like if-then-else and explicit control for parallelism. The microprocessor in the eCell configures the arithmetic logic unit to perform the function described in their corresponding gene, which is the part of the eDNA program corresponding to that eCell. The configuration program that runs in the microprocessor is named ribosomal DNA and is executed in the microprocessors of all eCells. The eDNA architecture

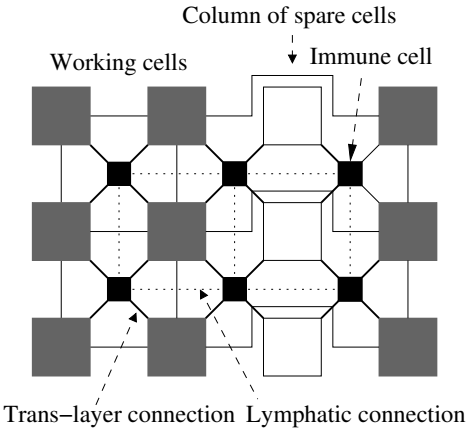



Figure 2.7: Immunotronics plus Embryonics concept (abstracted from [Bradley et al., 2000] ©2000 IEEE)

implements self-programming rather as self-organization by the autonomous determination of each eCell of which part of the eDNA to execute, please see [Boesen and Madsen, 2009]. Furthermore, self-repairing is also supported by the eDNA architecture, which is triggered in a faulty eCell after fault recognition. Fault recognition is implemented using time redundancy by a sort of double computation, as stated in [Boesen et al., 2011]. Self-repairing firstly consists on finding the nearest spare eCell, then writing the coordinate of the spare eCell as the coordinate of the faulty eCell in the routing table, and afterwards broadcasting a message to all eCells instructing to update the new coordinates of the faulty eCell in the eDNA program saved in RAM memory. This approach is being implemented in hardware, but results about the implementation of the self-repairing procedure in the hardware platform are not available yet.

### 2.2.7 Autonomic System on Chip

It is a project that tries to implement the aspects of Autonomic Computing: self-configuration, self-optimization and self-healing, given in [Kephard and Chess, 2003], which are present in a self-governing system, to a System on Chip. This project has been developed in the context of the priority program 1183 Organic Computing funded by the German Research Foundation (DFG) at the Technical University of Munich. That priority program aimed to develop technical systems with lifelike properties including all self-X properties which they coined as organic properties. The proposed autonomic SoC architecture in [Lipsa et al., 2005] consists of two logical layers: the functional layer which contains Intellectual Property components like CPUs, memories, on-chip buses, etc, named functional elements, in short FEs; and the autonomous layer which consists of autonomic elements, in short AEs, together with its interconnecting structure, please see figure 2.8. An autonomic element observes the state of its corresponding functional element through a monitor. An evaluator in the autonomic element assesses the state of the functional element. And an actuator in the autonomic element initiates, when necessary, a corrective action. The evaluator, as given in [Zeppenfeld et al., 2010], applies a reward-based reinforcement learning technique using a Learning Classifier Table that helps on judging the effectiveness of an specific corrective action comparing the state of the system before and after the action has been applied. The monitors are intended to collect

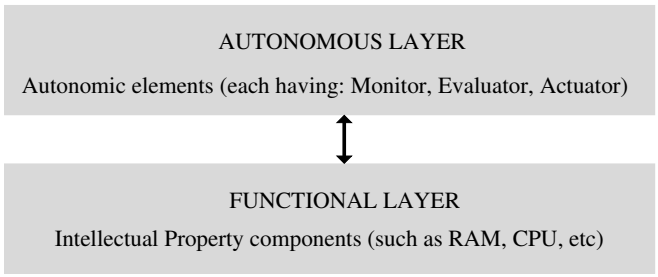



Figure 2.8: Layers of the Autonomic System on Chip (abstracted from [Bouajila et al., 2006] ©2006 IFIP)

data for computing the error rate of the function elements. That information should serve for adjusting frequency, voltage, etc in order to correct the malfunction of the functional elements due to environmental variations, please see [Bouajila et al., 2006]. Self-repairing is introduced at the processor level in a processor pipeline architecture using a micro-rollback strategy with an error detection mechanism able to detect multiple transient and timing errors. The processor pipeline architecture uses shadow registers and comparators at the inter-stage registers of the datapath and a history register for saving the data of the inter-stage registers enabling recomputation for error correction. Error detection and correction can be possible in two clock cycles, please see [Bouajila et al., 2011], [Zeppenfeld et al., 2010] and [Bouajila et al., 2006]. This approach addresses a specific architecture of processor considering transient and timing errors. The autonomic layer seems to help on long term actions which do not contribute directly on the self-repairing of the functional units. The hardware implementation of the integration of the autonomic layer and the functional layer has not been presented and evaluated yet.

### 2.2.8 Immunocomputing

It is a term coined by Alexander Tarakanov for a computational approach that is inspired by the principles of information processing of proteins and immune networks. A protein is considered as the basic component in an immune network because cells in general communicate each other through the binding of ligands and a receptors, which are proteins that are placed on their surfaces. In his book *Immunocomputing: Principles and Applications* [Tarakanov et al., 2003], Alexander Tarakanov presents: a mathematical model of a protein in three dimensions using quaternions; the computation of the binding energy between two proteins using a bilinear form; and the concept of a chip with a memory organized in layers P, M, R, w and C, and processing units. The memory stores real values in a floating point format. The processing units compute the bilinear form $w_{ij} = P_{ij}^T M R_{ij}$ using the values of memory elements of layers P, R and M and put the result on elements of layer w. The layer C serves for changing the values of the elements in the layer R, which triggers recomputation of the values of element on layer w. The mathematical description of this approach is hard to understand and is confusing in some places. Nevertheless, the so coined formal immune network, in short FIN, has been better explained in further publications such as [Tarakanov, 2008] and applied to pattern recognition. A formal immune network is based on the mathematical modeling of the binding energy among proteins and is basically a transformation of multidimensional pattern vectors to vectors with reduced dimensions, which allows to per-

form a faster pattern recognition at the same time saving memory for storing the transformed pattern vectors.

## 2.3 Self-repairing in FPGAs

There exist some approaches of self-repairing which are specific for FPGAs. Some of the most relevant are described briefly below.

### 2.3.1 Roving STAR

In [Abramovici et al., 1999], [Emmert et al., 2000] and [Abramovici et al., 2001] a method for testing an FPGA is presented, where the configurable logic blocks and interconnects of the entire chip are tested by roving one horizontal and one vertical self-testing area, as shown in figure 2.9. This method uses partial reconfiguration for moving the operating areas and leaving the self-testing area, in short STAR, available for being tested. The testing, diagnosis, partial reconfiguration, and repairing using partially defective and spare logic and routing resources for bypassing faults, is executed by a controller implemented in an external microprocessor which accesses the FPGA using the boundary-scan interface. This method uses off-line testing methods and does not need to stop the operation of the system for fault recovery, since it works on an inoperative area. But, it is to note that for reordering the operating area in order to make place for the STAR requires partial reconfiguration, which indeed stops parts of the system running on the FPGA. The implementation of the controller is not presented, but looks quite complex for being implemented. Furthermore, the constant reconfiguration of the FPGA for testing purposes can reduce the life time of the chip.

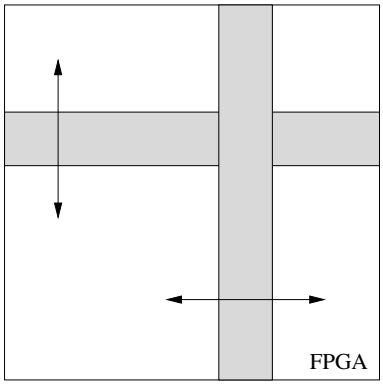



Figure 2.9: FPGA with roving STAR (abstracted from [Abramovici et al., 2001] ©2001 IEEE)

### 2.3.2 TMR + RoRA

A single event upset, in short SEU, is a fault caused by radiation that produces a bit flip in a bit register. Since a single event upset in the configuration memory of non radiation hardened SRAM-based FPGA can produce more than one errors in a circuit, triple modular redundancy can not guarantee masking a SEU fault when the errors caused by a single SEU affect two redundant modules at the same time. That problem has been identified and reported in [Bellato et al., 2004] stressing that the FPGA interconnection resources are more sensitive

to SEUs in comparison to the logic blocks. Therefore, in [Reorda et al., 2005b] a tool for analyzing the SEU effects on placed and routed FPGA design is presented. That tool verifies that the voter and the three replicated modules are placed in dedicated FPGA areas, if that is not the case, a report of the resources and the configuration memory bits that violate that rule is created. That report is used as input by the RoRA tool, presented in [Reorda et al., 2005a], which executes placing and routing of the reported critical areas again avoiding that a single configuration bit controls two or more routing segments. Since the RoRA tool, which name comes from Reliability-oriented place and Route Algorithm, is computationally expensive and less efficient than commercial tools, it can not be used for placing and routing the whole design. Therefore, in [Sterpone and Violante, 2005] it is proposed to integrate the analyzer and RoRA tools to the FPGA design flow of commercial tools like the ISE from Xilinx, as shown in figure 2.10. In that flow, the TMR tool replicates the circuit three times and adds a voter according to the TMR architecture. The analyzer and RoRA tools work on an already placed and routed design. Although triple modular redundancy can be inserted automatically with the Xilinx TMRTool in designs to be implemented in Xilinx FPGAs, the Analyzer and RoRA tools are not commercial standard tools yet.


Figure 2.10: New design flow using TMR and RoRA (abstracted from [Sterpone and Violante, 2005] ©2005 IEEE)

## 2.4 Self-repairing introduced at the hardware description

One approach has been found, that intends to automatically insert structures in the hardware description at the register transfer level to attain fault tolerance. That approach is described briefly below.

### 2.4.1 Automatic insertion of fault tolerant structures in the RTL description

In [Entrena et al., 2001] and [López-Ongil et al., 2007] ideas regarding a library-based tool for inserting hardware and information redundancy in designs described at the register transfer level is given. As the resulting fault tolerant design is also described at the RTL level, commercial tools can be used for synthesizing it. This work gives also some initial ideas for inserting pre-prepared template code lines for fault-repairing in the hardware design description.

## 2.5 Comments

This section intends to make a comparison of the approaches described in the prior subsections considering aspects such as hardware level of abstraction, hardware platform for the implementation, type of addressed fault, error detection technique, and recovery mechanism.

### 2.5.1 Hardware level of abstraction

Regarding the hardware level of abstraction, self-repairing can be implemented at different levels. Seeing from top to bottom, it can be implemented in a network by assigning the tasks of

the failed node to other nodes, such as the approach explained in the subsection *"Distributed self-repairing of a network of FPGA nodes"* of subsection 2.1. It can also be implemented at the board level with the help of an infrastructure for self-repairing as the approach explained in the subsection *"Immune system paradigm"* of subsection 2.2. In a single chip, in which a system on chip[1] is implemented, as proposed in [Moreno et al., 1998], which unfortunately uses a non-commercial available reconfigurable system on chip, reason why it has not been introduced in last subsections. In a network on chip[2], as the approach introduced in subsection *"e-DNA"* of subsection 2.2. A processing unit is composed of a datapath and a control unit. An approach for self-repairing a pipelined datapath of a processing unit using shadow registers has been introduced in subsection *"Autonomic System on Chip"* of subsection 2.2. A control unit can be implemented as a finite state machine like any sequential digital circuit is implemented. An approach of error recognition in a state machine using a method inspired by the immune system has been introduced in the subsection *"Immunotronics"* of subsection 2.2. Processing units, memories, input/output modules or specialized modules are usually described using hardware description languages, such as VHDL, which describe a digital circuit at the register transfer level[3]. An approach for inserting error recognition and recovery RTL structures on the description of a digital circuit has been shown in subsection 2.4.1. Two approaches of self-repairing of combinational circuits using spare modules and multiplexers or switches have been introduced in subsections *"Small-scale reconfigurability for fault detection, diagnosis and recovery"* and *"Logic self-repair"* of subsection 2.1. Finally, self-repairing at the transistor level has been proposed in [Kothe and Vierhaus, 2006] however not introduced in last subsections.

### 2.5.2 Hardware platform for the implementation

Systems on chip or digital circuits described at the register transfer level can be synthesized, placed, routed and downloaded into a field programmable gate array[4]. Most FPGAs are reconfigurable and some of them are able to be reconfigured partially. Total and partial reconfiguration can be used for fault recovery, therefore FPGAs are attractive in the design and implementation of self-repairing hardware systems as can be seen in subsections *"Dynamic partial reconfiguration for testing and repair"*, *"Distributed self-repairing of a network of FPGA nodes"*, *"Dual-FPGA architecture for autonomous self-repair"* and *"Evolvable hardware"*. Other approaches such as the ones shown in subsections *"POEtic tissue"* and *"Embryonics"* propose the manufacturing of a new kind of field programmable gate array, which makes those self-repairing approaches hard to be implemented or be used in the standard circuit design flow.

[1] A system on chip, in short SoC, integrates many modules that are usually seen as single chips on a board, such as processing units, memory modules, input/output modules, etc, in a single chip.

[2] A network on chip, in short NoC, is a paradigm for implementing the intermodule communication in a system on chip using switches that relay messages from a source module to the destination module.

[3] The register transfer level, in short RTL, is the level at which a circuit is described considering the flow and the logical operations performed on signals between hardware registers. Hardware registers are modules that store a determined number of bits of information, which can be read or overwritten.

[4] A field programmable gate array, in short FPGA, is an integrated circuit designed with programmable logic in order to be configured to implement any logic function after the integrated circuit has been manufactured.

### 2.5.3 Type of addressed fault

An FPGA can store its configuration in SRAM[5], flash[6] or antifuse[7] memory, which all are fabricated with complementary metal-oxide-semiconductor technology, in short CMOS.

SRAM-based FPGAs, such as the FPGAs from Xilinx, are prone to Singular Event Upsets, which are bit flips caused by radiation. A bit flip in a data register in the FPGA can be perceived as a transient fault, but a bit flip in a configuration register is perceived as a permanent fault, since it modifies the logic function implemented in the FPGA. An approach against SEUs affecting the configuration memory is the so called scrubbing, which consists on the total or partial reloading of the configuration memory with the correct bitstream, as in the approaches presented in subsections *"Dynamic partial reconfiguration for testing and repair"*, *"Distributed self-repairing of a network of FPGA nodes"*, *"Dual-FPGA architecture for autonomous self-repair"* and *"TMR + RoRA"*. As can be seen, many approaches have appeared to go against SEU effects, although there are commercial radiation hardened FPGA chips. The problem is that those chips are very expensive and they do not have the memory capacity and characteristics that new in mass produced FPGA chips have.

While flash-based FPGAs, such as the FPGAs from Actel, are not prone to radiation, they are fabricated with CMOS technology. Chips fabricated with CMOS technology can fail due to out of range temperatures, pressure, aging, or electromagnetic interference. Furthermore, the constant decreasing of die size in a silicon wafer is making chips prone to power fluctuations, or to such low levels of radiation as the presented on the surface of the earth. Such permanent faults in FPGAs are addressed by the approaches presented in subsections *"Evolvable hardware"*, *"Roving STAR"*, *"Dual-FPGA architecture for autonomous self-repair"* and *"Distributed self-repairing of a network of FPGA nodes"*. Permanent faults in hardware in general has been also addressed by the approaches in subsections *"Multifunctional units"*, *"Small-scale reconfigurability for fault detection, diagnosis and recovery"*, *"Logic self-repair"*, *"Immune system paradigm"*, *"POEtic tissue"*, *"Embryonics"* and *"e-DNA"*.

Faults which has been introduced as transient or permanent according to their persistence, can be modeled as stuck-at, stuck-open, bridge, etc. Most approaches consider stuck-at faults, however the approach presented in the subsection *"TMR + RoRA"* addresses bridge faults. According to [Avižienis et al., 2004], timing failures can also be present in a system and be modeled as a delay, problem which has been addressed in the subsection *"Autonomic System on Chip"* and in [Wong et al., 2007] which has not been introduced in last subsections.

A fault is the cause of a failure and a failure can be recognized through the detection of an error into the system, please see [Avižienis et al., 2004]. A self-repairing system requires error detection and recovery mechanisms.

### 2.5.4 Error detection technique

Errors can be detected offline when the circuit is not operating, or online at the start of the operation of the circuit, concurrently to the operation of the circuit, when the circuit is idle, or at scheduled slots of time such as the approach in [Al-Asaad and Shringi, 2000]. In the

[5] SRAM means static random-access memory. It is a volatile memory that does not require refreshing during operation, in comparison with dynamic random-access memory, but it still requires to be powered for remembering data.

[6] Flash memory is a nonvolatile memory which can be electrically erased or reprogrammed.

[7] An antifuse is a device that has initially a high resistance and after programmed the contrary, it becomes into an electrically conductive path. An antifuse is one time programmable.

last case, the preemption of the circuit can happen triggered by time or by an event. Offline error detection is executed in the approach presented in subsection *"Roving STAR"* for detecting errors in an FPGA, however only a part of the FPGA is not operating and being tested. Online error detection methods require some kind of redundancy in the design such as: space redundancy, e.g., duplication with comparison or self-checking circuits which are also named concurrent checking hardware; time redundancy, e.g., double execution; or information redundancy, e.g., error detecting and correcting codes. Duplication with comparison has been proposed on the approaches of subsections *"Small-scale reconfigurability for fault detection, diagnosis and recovery"* and *"Embryonics"*. Self-checking state machines or self-checking circuits using diverse duplication and organized in a pipelined fashion are proposed and introduced in subsection *"Dual-FPGA architecture for autonomous self-repair"*, however they serve for detecting transient errors. Double execution is proposed in the approach of subsection *"Dynamic partial reconfiguration for testing and repair"* using a so called dynamic TMR. Error detecting and correcting codes such as the Hamming encoding can be used for coding the states of a state machine for detecting and correcting the effect of a SEU in a state register, as proposed in [Burke and Taft, 2004]. That technique is offered by some synthesis tools for FPGA designs such as the one of Synopsis, please see [Sutton, 2012].

Error detection should be executed concurrently, or in other case fast enough in order to not affect the operation of the circuit. Most of the approaches do not go in detail in the error detection mechanism, although it is a key part of a self-repairing system. In some of the approaches such as the ones in subsections *"Dynamic partial reconfiguration for testing and repair"* and *"Distributed self-repairing of a network of FPGA nodes"*, a Built-In Self-Test technique is proposed for error detection. However, BIST requires to stop the circuit for executing error recognition, fact which is not realized in those approaches. Test patterns or seeds for the decompression of test patterns in case of having a LFSR, for being used by the BIST circuitry can be applied to the system when it is not operating, or they can be stored in memory, which is not usual because of the huge amount of data which should be stored. In [Li et al., 2008] - subsection *"Dual-FPGA architecture for autonomous self-repair"* and in [Avižienis, 2000] - subsection *"Immune system paradigm"* it is mentioned the use of a set of test patterns stored in memory for testing the functionality of cores or nodes, respectively. But, no details about the error detection, consequently no information about the size of the stored data set, which determines how long error detection can take and the memory needed, is given. In the approach introduced in subsection *"Immunotronics"*, a set of the so called tolerant condition vectors is stored in memory for detecting errors on a state machine concurrently to its operation, however it demands that the states of the state machine are observable. The so called formal immune network introduced in subsection *"Immunocomputing"* reduces the dimensions of multidimensional vectors, however those are vectors with real elements represented in floating point format. Testing vectors of digital circuits have usually elements with binary values 0 or 1. Nevertheless, testing vectors of hardware cores or systems with continuous signal inputs and outputs, can profit of that reduction method in order to save memory.

### 2.5.5 Recovery mechanism

System recovery can be achieved by error handling and fault handling. Error handling consists on bringing the system to an error free state by looking back for a saved prior error-free state, process named rollback, or by copying data from a redundant system and recomputing

the output again, process named roll-forward. Those methods are useful for recovering from transient faults as applied in the approaches presented in subsections *"Dual-FPGA architecture for autonomous self-repair"* and *"Autonomic System on Chip"*. Fault handling tries to avoid the error to happen again going against the cause of a failure, that is removing the fault by the following subsequent processes: fault diagnosis, fault isolation, system reconfiguration and system reinitialization. Fault diagnosis is very hard to achieve concurrently because is time demanding and difficult, nevertheless, the approach presented in subsection *"Dual-FPGA architecture for autonomous self-repair"* proposes blind reconfiguration using column based precompiled configurations for finding out which column partition of the FPGA is faulty. The approach presented in subsection *"Roving STAR"* executes offline fault diagnosis with the help of an external controller circuit. Fault isolation is executed by the approaches presented in subsections *"Embryonics"* and *"e-DNA"* at the cell level of partitioning. System reconfiguration is applied in most approaches that use as hardware platform an FPGA, such as those in subsections *"Dynamic partial reconfiguration for testing and repair"*, *"Distributed self-repairing of a network of FPGA nodes"*, *"Dual-FPGA architecture for autonomous self-repair"*, *"Evolvable hardware"* and *"Roving STAR"*. Triple modular redundancy, in short TMR, is a technique that uses redundancy for masking faults, and has been used in the approaches *"TMR + RoRA"* and *"Evolvable hardware"*. However, the last approach considers the repairing of a faulty module of the TMR by using a technique inspired by evolution. System reconfiguration can also be applied configuring a redundant module to execute the task of the faulty module by means of multiplexers or switches, such is the case of the approaches presented in subsections *"Multifunctional units"*, *"Small-scale reconfigurability for fault detection, diagnosis and recovery"*, *"Distributed self-repairing of a network of FPGA nodes"*, *"Logic self-repair"*, *"Immune system paradigm"*, *"Embryonics"* and *"e-DNA"*. Thereby, the number of redundant modules determine how many faults can be repaired.

## 2.6 Bibliography

Abramovici, M., Emmert, J. M., and Stroud, C. E. (2001). Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis and Fault Tolerance for FPGAs in Adaptive Computing Systems. In *3rd NASA/DoD Workshop on Evolvable Hardware*, pages 73–92. IEEE Computer Society.

Abramovici, M., Stroud, C., Hamilton, C., Wijesuriya, S., and Verma, V. (1999). Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications. In *International Test Conference*, pages 973–982.

Akoglu, A., Sreeramareddy, A., and Josiah, J. G. (2009). FPGA based distributed self healing architecture for reusable systems. *Cluster Computing*, 12(3):269–284. Springer.

Al-Asaad, H. and Shringi, M. (2000). On-line built-in self-test for operational faults. In *AUTOTESTCON 2000*, pages 168–174.

Al-Yamani, A. A. (2004). *Deterministic Built-In Self Test for Digital Circuits*. PhD thesis, Stanford University.

Avižienis, A. (2000). A Fault Tolerance Infrastructure for Dependable Computing with High-Performance COTS Components. In *International Conference on Dependable Systems and Networks - DSN 2000*, pages 496–500. IEEE.

Avižienis, A. (2002). An Immune System Paradigm for the Design of Fault Tolerant Systems. In *4th European Dependable Computing Conference on Dependable Computing - EDCC 4*, Lecture Notes in Computer Science, pages 81–83. Springer.

Avižienis, A. (2006). An Immune System Paradigm for the Assurance of Dependability of Collaborative Self-organizing Systems. In *19th World Computer Congress, TC 10: 1st International Conference on Biologically Inspired Computing*, volume 216 of *IFIP International Federation for Information Processing*, pages 1–6. Springer.

Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *Transactions on Dependable and Secure Computing*, 1(1):11–33. IEEE.

Avizienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., and Rubin, D. K. (1971). The STAR (Self-Testing and Self-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. *Transactions on Computers*, 20(11):1312–1321. IEEE.

Barker, W., Halliday, D. M., Thoma, Y., Sanchez, E., Tempesti, G., and Tyrell, A. M. (2007). Fault Tolerance Using Dynamic Reconfiguration on the POEtic Tissue. *Transactions on Evolutionary Computation*, 11(5):666–684. IEEE.

Bellato, M., Bernardi, P., Bortolato, D., Candelori, A., Ceschia, M., Paccagnella, A., Rebaudengo, M., Reorda, M. S., Violante, M., and Zambolin, P. (2004). Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA. In *Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 584–589.

Boesen, M. R. and Madsen, J. (2009). eDNA: A Bio-Inspired Reconfigurable Hardware Cell Architecture Supporting Self-organisation and Self-healing. In *NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2009*, pages 147–154.

Boesen, M. R., Madsen, J., and Keymeulen, D. (2011). Autonomous Dynamically Self-organizing and Self-healing Distributed Hardware Architecture the eDNA Concept. In *Aerospace Conference*, pages 1–13. IEEE.

Bouajila, A., Zeppenfeld, J., Stechele, W., and Herkersdorf, A. (2011). An Architecture and an FPGA Prototype of a Reliable Processor Pipeline Towards Multiple Soft- and Timing Errors. In *14th International Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2011*, pages 225 – 230. IEEE.

Bouajila, A., Zeppenfeld, J., Stechele, W., Herkersdorf, A., Bernauer, A., Bringmann, O., and Rosenstiel, W. (2006). Organic Computing at the System on Chip Level. In *International Conference on Very Large Scale Integration*, pages 338–341. IFIP.

Bradley, D., Ortega-Sanchez, C., and Tyrell, A. (2000). Embryonics + Immunotronics: A Bio-Inspired Approach to Fault Tolerance. In *2nd NASA/DoD Workshop on Evolvable Hardware*, pages 215–223. IEEE Computer Society.

Bradley, D. W. (2002). Immunotronics - Novel Finite-State-Machine architectures with built-in self-test using self-nonself differentiation. *Transactions on Evolutionary Computation*, 6(3):227–238. IEEE.

Bradley, D. W. and Tyrell, A. M. (2001). The Architecture for a Hardware Immune System. In *3rd NASA/DoD Workshop on Evolvable Hardware*, pages 193–200. IEEE Computer Society.

Burke, G. and Taft, S. (2004). Fault Tolerant State Machines. Technical Report D160/MALPD 2004, Jet Propulsion Laboratory, California Institute of Technology.

Chmelař, E. (2004a). Minimizing the Number of Test Configurations for FPGAs. In *International Conference on Computer Aided Design - ICCAD 2004*, pages 899–902. IEEE/ACM.

Chmelař, E. (2004b). *The Test and Diagnosis of FPGAs*. PhD thesis, Stanford University.

Dictionary.com, L. (2012). English dictionary. dictionary.reference.com.

École Polytechnique Fédérale de Lausanne, University of York, The University of Glasgow, Université de Lausanne, and Universitat Politécnica de Catalunya (1.09.2001 - 31.1.2005). Reconfigurable POEtic Tissue (POETIC) Project. http://cordis.europa.eu.

Emmert, J. M., Stroud, C. E., Skaggs, B., and Abramovici, M. (2000). Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration. In *Symposium on Field-Programmable Custom Computing Machines*, pages 165–174. IEEE.

Entrena, L., López, C., and Olías, E. (2001). Automatic Insertion of Fault-Tolerant Structures at the RT Level. In *7th International On-Line Testing Workshop*, pages 183–200. 48-50.

Garvie, M. and Thompson, A. (2004). Scrubbing away transients and Jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In *10th International On-Line Testing Symposium - IOLTS 2004*, pages 155–160. IEEE.

Huang, W.-J. and McCluskey, E. J. (2001). A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations. In *International Symposium on Field-Programmable Gate Arrays - FPGA 2001*. ACM/SIGDA.

Huang, W.-J. R. (2001). *Dependable Computing Techniques for Reconfigurable Hardware*. PhD thesis, Stanford University.

Kephard, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36(1):41–50. IEEE Computer Society.

Koal, T., Scheit, D., Schölzel, M., and Vierhaus, H. T. (2011). On the Feasibility of Built-In Self Repair for Logic Circuits. In *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems - DFT 2011*, pages 316–324. IEEE.

Koal, T., Scheit, D., and Vierhaus, H. T. (2009). A Concept for Logic Self Repair. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools - DSD 2009*, pages 621–624.

Koal, T., Ulbricht, M., and Vierhaus, H. T. (2012). Combining On-Line Fault Detection and Logic Self Repair. In *15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2012*, pages 288–293. IEEE.

Koal, T. and Vierhaus, H. T. (2008). Basic Architecture for Logic Self Repair. In *14th International On-Line Testing Symposium - IOLTS 2008*, pages 177–178. IEEE.

Kochte, M. A., Zoellin, C. G., and Wunderlich, H.-J. (2009). Concurrent Self-Test with Partially Specified Patterns For Low Test Latency and Overhead. In *14th European Test Symposium*, pages 53–58. IEEE Computer Society.

Kothe, R. and Vierhaus, H. T. (2006). Embedded Self Repair by Transistor and Gate Level Reconfiguration. In *Conference on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2006*, pages 208–213. IEEE.

Kumar, V. V. and Lach, J. (2003). Fine-Grained Self-Healing Hardware for Large-Scale Autonomic Systems. In *14th International Workshop on Database and Expert Systems Applications*, pages 707–712. IEEE Computer Society.

Li, Y., Makar, S., and Mitra, S. (2008). CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns. In *Design, Automation and Test in Europe - DATE 2008*, pages 885–890.

Li, Y., Mutlu, O., Gardner, D. S., and Mitra, S. (2010). Concurrent Autonomous Self-Test for Uncore Components in System-on-Chips. In *28th VLSI Test Symposium - VTS 2010*, pages 232–237. IEEE.

Lipsa, G., Herkersdorf, A., Rosenstiel, W., Bringmann, O., and Stechele, W. (2005). Towards a Framework and a Design Methodology for Autonomic SoC. In *2nd International Conference on Autonomic Computing - ICAC 2005*, pages 391–392.

López-Ongil, C., Entrena, L., García-Valderas, M., and Portela-García, M. (2007). Automatic Tools for Design Hardening. In Velazco, R., Fouillat, P., and Reis, R., editors, *Radiation Effects on Embedded Systems*, pages 183–200. Springer.

Marchal, P., Nussbaum, P., Piguet, C., Durand, S., Mange, D., Sanchez, E., Stauffer, A., and Tempesti, G. (1996). Embryonics: The Birth of Synthetic Life. In Sanchez, E. and Tomassini, M., editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 616–196. Springer.

Marinos, P. N. (1969). The Organization of a Self-Repairing System from Multifunctional Units. *Proceedings of the IEEE*, 57(7):1320.

Mitra, S. (2000). *Diversity Techniques For Concurrent Error Detection*. PhD thesis, Stanford University.

Mitra, S. (2008). Globally Optimized Robust Systems to Overcome Scaled CMOS Reliability Challenges. In *Design, Automation and Test in Europe - DATE 2008*, pages 941–946.

Mitra, S., Huang, W.-J., Saxena, N. R., Yu, S.-Y., and McCluskey, E. J. (2000). Dependable Adaptive Computing Systems - The Stanford CRC ROAR Project. In *Pacific RIM International Symposium on Dependable Computing - Fast Abstracts*.

Mitra, S., Huang, W.-J., Saxena, N. R., Yu, S.-Y., and McCluskey, E. J. (2004). Reconfigurable Architecture for Autonomous Self-Repair. *Design and Test of Computers*, 21(4):228–240. IEEE.

Moreno, J., Thoma, Y., Sanchez, E., Torrez, O., and Tempesti, G. (2004). Hardware Realization of a Bio-Inspired POEtic Tissue. In *NASA/DoD Conference on Evolvable Hardware*, pages 237–244. IEEE.

Moreno, J. M., Madrenas, J., Faura, J., Cantó, E., Cabestany, J., and Insenser, J. M. (1998). Feasible Evolutionary and Self-Repairing Hardware by Means of the Dynamic Reconfiguration Capabilities of the FIPSOC Devices. In *2nd International Conference on Evolvable Systems: From Biology to Hardware - ICES 1998*, Lecture Notes in Computer Science, pages 345–355. Springer.

Oh, N. (2000). *Software Implemented Hardware Fault Tolerance*. PhD thesis, Stanford University.

Ortega-Sanchez, C., Mange, D., Smith, S., and Tyrell, A. (2000). Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties. *Genetic Programming and Evolvable Machines*, 1(3):187–215.

Paulsson, K., Hübner, M., and Becker, J. (2006a). Methods for Run-time Failure Recognition and Recovery in Dynamic and Partial Reconfigurable Systems Based on Xilinx Virtex-II Pro FPGAs. In *Symposium on Emerging VLSI Technologies and Architectures*. IEEE Computer Society.

Paulsson, K., Hübner, M., and Becker, J. (2006b). Strategies to On- Line Failure Recovery in Self- Adaptive Systems based on Dynamic and Partial Reconfiguration. In *1st NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2006*, pages 288–291. IEEE Computer Society.

Reorda, M. S., Sterpone, L., and Violante, M. (2005a). Efficient Estimation of SEU effects in SRAM-based FPGAs. In *11th International On-Line Testing Symposium - IOLTS 2005*, pages 54–59. IEEE.

Reorda, M. S., Sterpone, L., and Violante, M. (2005b). Multiple errors produced by single upsets in FPGA configuration memory - a possible solution. In *European Test Symposium*, pages 136–141.

Sharma, R. and Saluja, K. K. (1988). An Implementation and Analysis of a Concurrent Built-In Self-Test Technique. In *18th International Symposium on Fault-Tolerant Computing - FTCS 18*, pages 164–169.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Pérez-Uribe, A., and Stauffer, A. (1997). A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *Transactions on Evolutionary Computation*, 1(1):83–97. IEEE.

Sreeramareddy, A., Josiah, J. G., Akoglu, A., and Stoica, A. (2008). SCARS: Scalable Self-Configurable Architecture for Reusable Space Systems. In *NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2008*, pages 204–210.

Sreeramareddy, A., Kallam, R., Dasu, A. R., and Akoglu, A. (2010). Self-configurable architecture for reusable systems with Accelerated Relocation Circuit (SCARS-ARC). In *International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum - IPDPSW 2010*, pages 1–4. IEEE.

Sterpone, L. and Violante, M. (2005). A Design Flow for Protecting FPGA-Based Systems Against Single Event Upsets. In *20th International Symposium on Defect and Fault Tolerance in VLSI Systems - DFT 2005*, pages 436–444. IEEE.

Sudarsanam, A., Kallam, R., and Dasu, A. (2009). PRR-PRR Dynamic Relocation. *Computer Architecture Letters*, 8(2):44–47. IEEE.

Sutton, A. (2012). No Room for Error: Creating Highly Reliable, High-Availability FPGA Designs. Technical report, Synopsis.

Tarakanov, A., Goncharova, L., and Tarakanov, O. (2005). A Cytokine Formal Immune Network. In *8th European Conference on Advances in Artificial Life - ECAL 2005*, volume 3630 of *Lecture Notes in Computer Science*, pages 510–519. Springer.

Tarakanov, A. O. (2008). Formal Immune Networks: Self-Organization and Real-World Applications. In Prokopenko, M., editor, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 271–290. Springer.

Tarakanov, A. O., Skormin, V. A., and Sokolova, S. P. (2003). *Immunocomputing: Principles and Applications*. Springer.

Tempesti, G., Mange, D., Mudry, P.-A., Rossier, J., and Stauffer, A. (2007). Self-Replicating Hardware for Reliability: The Embryonics Project. *Journal on Emerging Technologies in Computing Systems - JETC*, 3(2):Article No. 9. ACM.

Tempesti, G., Mange, D., and Stauffer, A. (1997). A Robust Multiplexer-Based FPGA Inspired By Biological Systems. *Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems*, 43(10):719–733. Elsevier.

Torresen, J. (2004). An Evolvable Hardware Tutorial. In *14th International Conference on Field Programmable Logic and Applications - FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 821–830. Springer.

Touba, N. A. and McCluskey, E. J. (1997). Logic Synthesis of Multilevel Circuits with Concurrent Error Detection. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):783–789. IEEE.

Tyrell, A. M. and Barker, W. (2006). The Poetic Hardware Device: Assistance for Evolution, Development and Learning. In Higuchi, T., Liu, Y., and Yao, X., editors, *Evolvable Hardware*, Genetic and Evolutionary Computation, pages 99–119. Springer.

Tyrell, A. M., Sanchez, E., Floreano, D., Tempesti, G., mange, D., Moreno, J.-M., Rosenberg, J., and Villa, A. E. (2003). POEtic Tissue: An Integrated Architecture for Bio-inspired Hardware. In *Evolvable Systems: From Biology to Hardware*, volume 2606 of *Lecture Notes in Computer Science*, pages 129–140. Springer.

Venishetti, S. K., Akoglu, A., and Kalra, R. (2007). Hierarchical Built-in Self-testing and FPGA Based Healing Methodology for System-on-a-Chip. In *2nd NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2007*, pages 717–724.

Wikipedia (2012). Searched words: fault-tolerant system, fault-tolerant design, single point of failure, integrated circuit, chip, partial reconfiguration, system on chip, network on chip, hardware register, CMOS, fault model, SRAM, flash memory.

Wong, J. S. J., Sedcole, P., and Cheung, P. Y. K. (2007). Self-characterization of Combinatorial Circuit Delays in FPGAs. In *International Conference on Field-Programmable Technology - ICFPT 2007*, pages 17–23.

Yu, S.-Y. (2001). *Fault Tolerance in Adaptive Real-Time Computing Sytems*. PhD thesis, Stanford University.

Zeng, C., Saxena, N., and McCluskey, E. J. (1999). Finite State Machine Synthesis with Concurrent Error Detection. In *International Test Conference*, pages 672–678.

Zeppenfeld, J., Bouajila, A., Herkersdorf, A., and Stechele, W. (2010). Towards Scalability and Reliability of Autonomic Systems on Chip. In *3rd International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing - ISORC 2010 - Workshop on Self-Organizing Real-Time Systems*, pages 73–80. IEEE.

# 3

# Artificial immune systems

Artificial immune systems is a subfield of artificial intelligence based on principles of the biological immune system. Artificial intelligence is the science and engineering of making intelligent machines, please see [Wikipedia, 2010]. The term artificial intelligence has been coined by John McCarthy, the inventor of the LISP programming language, in 1956. Intelligent machines are based on automated inference engines. Inference engines infer certain consequences based on certain conditions. They follow the construct *if-then*. When the construct have the form *if adjective then noun* the machine is a classifier. When the construct have the form *if adjective then verb* the machine is a controller. However, controllers do classify the condition before inferring actions. Classifier and controller machines make use of machine learning techniques to train the machine and pattern recognition techniques for condition matching. In many cases condition matching does not imply absolute, but rather the closest match. Methods of artificial intelligence can make use of a symbolically structured knowledge base and statistical analysis, such is the case of expert systems, case based reasoning and Bayesian networks, or can make use of the learning of empirical data such is the case of neural networks, fuzzy systems, evolutionary computation, cellular automata and artificial immune systems. The first group of methods constitute the subbranch named conventional artificial intelligence and the second group constitute the subbranch named computational intelligence. Computational intelligence takes also some other names such as soft computing or non-symbolical artificial intelligence.

Artificial immune systems at the moment consists of a set of representative algorithms. Those algorithms are based on biological immune system models taken from the field of immunology. Immunology is a subfield of biology that studies the immune system of plants, animals and the human. Immunology uses models for understanding the structure and function of the immune system. Those models are sometime quite complex due to the biological immune system is indeed a complex system and many processes are not well understood till date. Some of those models seem to be potentially useful for solving some computational

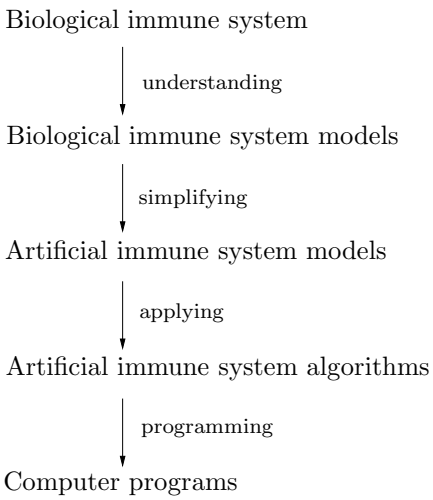



Figure 3.1: Artificial immune systems flow

problems. The simplification of such biological immune system models can produce artificial immune system models, that applied to determined problems, could be the basis of artificial immune system algorithms and consequently computer programs. Figure 3.1 shows the addressed flow.

In order to have a funded idea of the structure and the function of the biological immune system for understanding how artificial immune systems are conceived and work, the next section gives a short but complete summary of the most important concepts of the biological immune system, focusing on the human. The section afterwards presents the most known artificial immune system models and algorithms. At the end a brief comparison of the presented algorithms is given showing its biological analogy, input and outputs and its main application.

## 3.1 Biological immune system

The biological immune system is a system that protects a biological entity, i.e. the human, against internal or external agents that cause disease on it and thereafter death. As causes of disease can be listed: external or internal malicious agents, extreme temperature, extreme pressure, radiation, nutrient deficiency, aging, inherited disorders, etc. A disease in any living biological entity can be defined as the malfunctioning, the excessive reproduction or the unexpected death of cells. In this context, next subsections deal with cells inside a biological entity as internal agents, pathogens that endanger the biological entity as external agents, communication among all those agents, the immune system infrastructure and the immune system agents. The information presented in the sections about communication among agents and immune system agents have been extracted from the book [Kimball, 1994] and its on-line actualization. Immunology is a dynamic field thus, the on-line actualization of that book has been a very helpful resource.

### 3.1.1 Internal agents

A cell is considered as the most basic functional unit inside a biological entity. A cell executes tasks autonomously and is able to react with its environment. Thus, a cell is considered as an internal agent.

Biological entities, also called organisms, can consist of only one cell or more than one cell. When an organism has more than one cell it is named a multicellular organism. Animals, plants and the human are multicellular organisms. The cells in a multicellular organism are specialized and collaborate each other to fulfill joint objectives in the context of the organism as a whole. For that, free intercellular material on the environment or cellular material on the surface of the cells serve as information that helps for a coordinated work, that is to say, a harmonious functioning of all individual cells for the most effective results.

The term cell comes from the latin word *cellula* that means small room. A cell is composed of a cell membrane that separates the interior of the cell from its environment, please see figure 3.2. Such cell membrane is permeable for letting pass through substances inside or outside the cell. The cell membrane has on its surface receptors that the cell uses for communicating with other cells. Several specialized components are found inside the cell in a liquid named cytoplasm. Such components are responsible of synthesis of substances; processing of food, intern worn-out components or engulfed cells; energy generation and storage; food and waste storage and intern substances transportation. Some cells have a nucleus and some others not. The nucleus contains inherited information about the growth, specific function over its life span, reproduction and death of the cell.

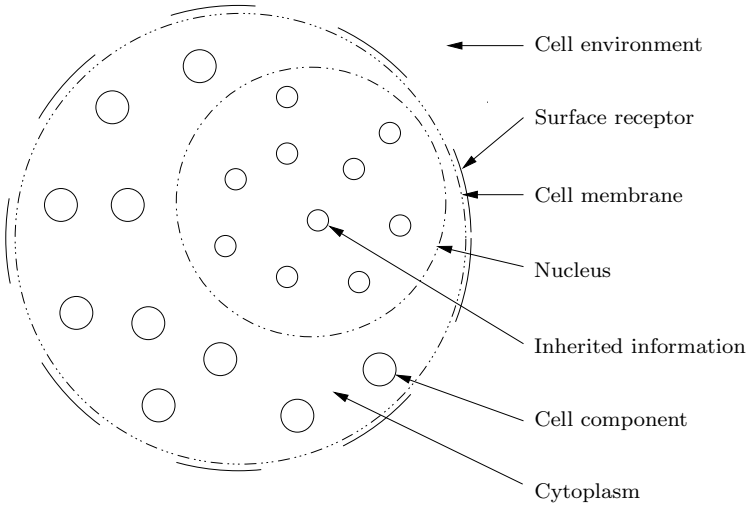



Figure 3.2: Cell parts

**Cell growth** Cell growth is given by metabolism. Metabolism consists in the synthesis and decomposition of substances. Decomposition of substances generates energy, raw material and waste.

**Cell reproduction** Cell reproduction is given by cell division and controlled by intercellular communication. Cell reproduction is useful for organism growth, replenishing of cells of tissues of damaged organs or any other biological process. Organs are functional units

constituted of several kinds of tissues, a tissue being a structural material of equal cells grouped together. Excessive and uncontrolled cell reproduction is a disorder which can cause an abnormal mass of tissue. The cells in such abnormal mass of tissue are called tumor cells and can be considered as internal malicious agents inside the organism.

**Cell death** There are two kinds of cell death, programmed cell death and cell death caused by external malicious agents. When cell death is programmed, died cells are easily recognized and removed by other cells. Programmed cell death is named apoptosis. The name apoptosis has been inspired in the falling off of leaves in plants. The term apoptosis derives from the Greek words *apo* that means off and *ptõsis* that means falling. Apoptosis is useful for maintaining internal stability under varying conditions inside the organism, i.e. removal of damaged or infected cells. Apoptosis is also useful in tissue development, i.e. elimination of transitory cells, tissues or organs. The programmed death of a cell can be triggered or inhibited by the cell itself or by other cells using intercellular communication. A cell triggers its own death when it is stressed by heat, radiation, nutrient privation, insufficient oxygen, etc.

Uncontrolled apoptosis during tissue development can derive in atrophy, which is an excessive death of cells, or in the formation of abnormal mass of tissues, which is an insufficient death of cells.

When cell death is caused by external malicious agents, died cells are not easily recognized by other cells for removal. Consequently, dead tissue is built in the surroundings of cell death. Such form of unexpected cell death is named necrosis, term derived from the Greek word *necro* that means dead body.

### 3.1.2 External agents

Every unicellular or multicellular organism outside a specific biological entity is considered an external agent. Some external agents are harmless and live in harmony with the biological entity but other external agents are malicious since they produce disease on the biological entity. Pathogens are external agents that produce disease in the organism where they intrude. The term pathogen is derived form the Greek words *pathos* that means suffering and *gen* that means give birth to, giving the meaning of a suffering creator. There are three types of pathogens: viruses, bacteria and fungi, please see figure 3.3. The definition of each class of pathogen is given below.

**Virus** A virus, latin word that means poison, is a biological particle that replicates only inside the cells of organisms.

**Bacterium** A bacterium, term derived form the Greek word *bakterion* that means small staff, is an small organism constituted of a single cell without a nucleus. Small organisms are called in the biological jargon as microorganisms. Bacteria live in any habitat such as in inorganic matter, in organic matter, in plants or in animals.

**Fungus** A fungus, term derived from the Greek word *sphongos* that means sponge, exists as a microorganism constituted of a single cell with a nucleus, i.e. yeast; as a multicellular microorganism, i.e. mould; and as an organism, i.e. mushrooms. Fungi eat organic matter.

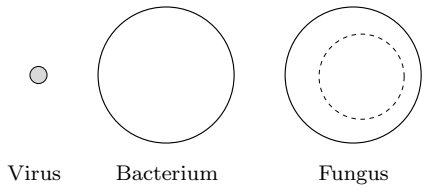



Figure 3.3: Types of pathogens

The biological immune system in the human body has a layered protection to act against external malicious agents like the pathogens described before. The layers are the following: the pathogen barriers, the innate immune system and the adaptive immune system. Those layers are described below and illustrated in figure 3.4.

| | |
|---|---|
| Pathogen barriers | Skin Mucous membrane<br>Tears Earwax Saliva Gastric acid<br>Flora |
| Innate immune system | Innate immune response<br>( Leukocytes ) |
| Adaptive immune system | Adaptive immune response<br>( Adapted leukocytes ) |

Figure 3.4: Layers of protection in the human immune system

**Pathogen barriers** The layer of pathogen barriers consists of barriers that can be mechanical, chemical and biological.

The skin and the mucous membrane block the intrusion of pathogens into the body, therefore they are considered as mechanical pathogen barriers. The mucous membrane is the tissue that covers cavities in the human body that connect the external environment to internal organs of the respiratory, gastrointestinal, visual, auditory and urogenital systems, among others. The prolongation of such cavities inside internal organs such as the stomach or the lungs, are also covered by a mucous membrane.

Substances secreted by the mucous membrane like nasal mucus, saliva, gastric acid, tears, earwax, breast milk, etc, trap pathogens which intend to enter the body, and inhibit their proliferation. Consequently those substances are considered as chemical pathogen barriers.

The flora is harmless bacteria that lives on the internal walls of the gastrointestinal tract, specifically in the internal walls of the stomach and the intestines. The flora is an example of a biological pathogen barrier since it competes with pathogenic bacteria for food and space. That fact reduces the probability that any foreign pathogenic bacteria which enters in the gastrointestinal tract proliferate sufficiently for causing disease.

**Innate immune system** Pathogens that passed the pathogen barriers successfully, meet the second layer of protection, the innate immune system. The innate immune system executes the innate immune response. The innate immune response consists of recognizing pathogens and taking a set of actions in order to get rid of them. Such tasks are performed by immune system agents, which are cells in the human body called leukocytes.

**Adaptive immune system** If the existing leukocytes can not stop the proliferation of pathogens, they initiate the adaptive immune response. The adaptive immune response tries to produce adapted leukocytes which are able to recognize exactly a determined pathogen and react accordingly in order to combat it.

### 3.1.3 Communication among agents

Agents communicate among each other for getting information over their environment and for reacting accordingly following a defined goal. Communication is the process of information interchanging. Cells are internal agents inside an organism that interchange information in order to execute their defined tasks. Information is represented by biological material that a cell exposes on its surface or releases to the environment for communicating with distant cells, please see figure 3.5. Communication among cells makes a joint work inside an organism be coordinated and not chaotic. Furthermore, interaction of an organism with its environment is only possible when its peripheral cells communicate with external agents which can also be cells of other organisms, please see again figure 3.5. In summary, cells of the same class inside an organism can communicate, cells of different classes inside an organism can communicate or cells of an organism with the cells of other organism can also communicate. In this subsection a model of cell communication organized in three parts is presented. That model uncovers the issue how cells of the immune system recognize pathogen. Furthermore the cell communication model is extended to the communication of a group of cells where a sort of network is built.

#### Cell communication model - Part 1

Claude Shannon and Warren Weaver presented in 1949 a general component based model of communication *Sender*→*Message*→*Receiver*, statement taken from [Wikipedia, 2010]. In the context of cell communication, the sender and the receiver are cells and the message is cellular material that contains information called ligand. A ligand can be present as surface ligand or as free ligand, please see figures 3.6 and 3.7. A target cell receives only messages when it has a specific receptor that interacts with the present ligand. Therefore, the component based model of cell communication is *Ligand*→*Receptor* and the binding of a ligand to a receptor in a cell is called ligand-receptor interaction, please see figure 3.8. Ligands and receptors are mostly proteins. In consequence, a ligand-receptor interaction can be considered as a protein-protein binding. In order to understand how that binding takes place, the components of a protein and its structure, the different classes of ligands and classes of receptors are explained below.

**Protein** A protein is a chemical substance composed of the following chemical elements: carbon, hydrogen, nitrogen and oxygen expressed with the symbols C, H, N and O respectively. Carbon, hydrogen, oxygen and nitrogen atoms are arranged into molecules, where every molecule has the same number of atoms of each chemical element. The

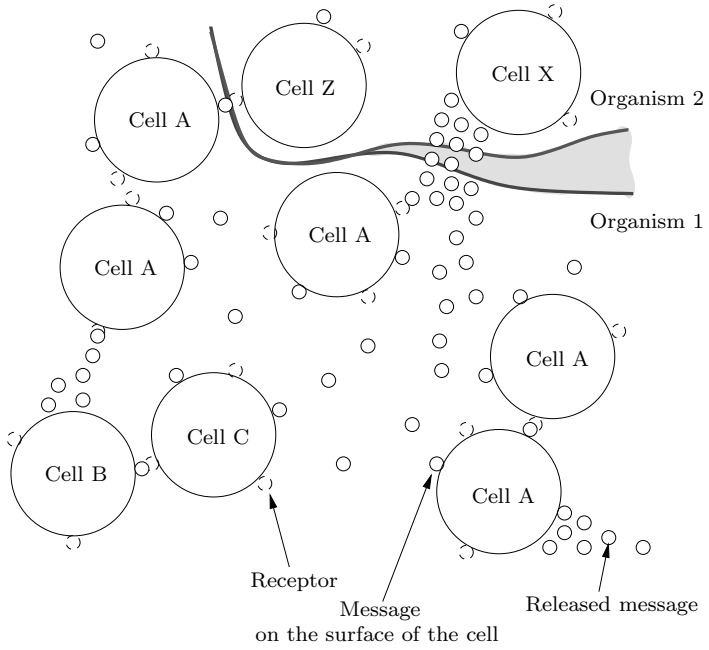



Figure 3.5: Cell communication

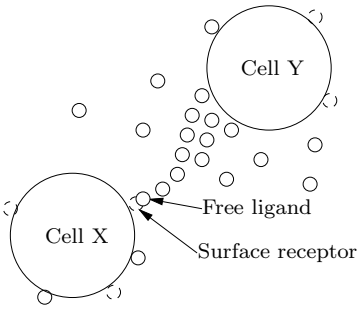
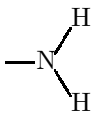
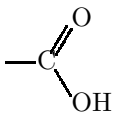



Figure 3.6: Cell surface receptor and free ligand interaction

number of atoms of each of the different chemical elements in a molecule give the exact formula of the chemical substance, please see figure 3.9. A protein molecule is very large and has its carbon atoms arranged into chains to which the atoms of the other chemical elements hang up. Molecules with carbon chains are called in chemistry organic compounds, thus, a protein molecule is an organic compound.

A protein molecule is the result of bonding amino acids of a set of 20 different ones by means of peptide bonds. Amino acids are simpler chemical compounds organized in an amine group  and a carboxylic acid group  both attached to the same carbon in a carbon chain – R – called residue. Residues have different sizes among different amino acids. A peptide bond is the result of the reaction of the carboxil group of one amino acid with the amino group of the adjacent amino acid with loss of

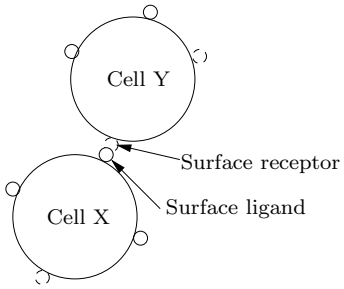



Figure 3.7: Cell surface receptor and cell surface ligand interaction

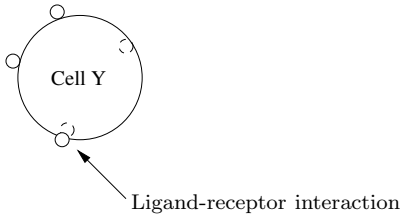
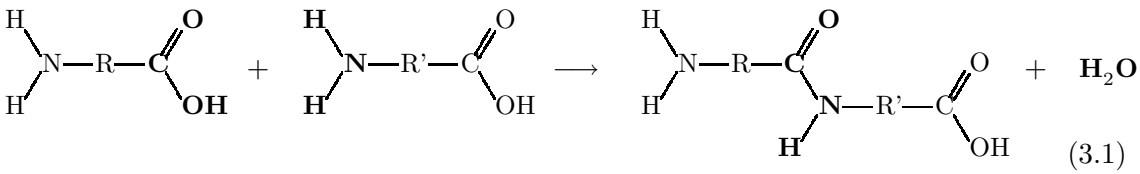



Figure 3.8: Ligand-receptor interaction

a water molecule $H_2O$ as shown in the following chemical reaction. Note that – R – represents the carbon chain of the first amino acid and – R' – the carbon chain of the second amino acid.




(3.1)

The complex structure of a protein molecule can be determined analyzing its primary, secondary, tertiary and quaternary structures. The primary structure of a protein constitutes the order in the sequence and the number of amino acids tied by peptide bonds, see figure 3.10. The secondary structure of a protein constitutes the arrangement of the sequence of amino acids into regularly repeating structures like the alpha-helix, with the form of an spiral, and beta-sheet, with the form a twisted and pleated sheet, see figure 3.11. Those structures are stabilized by bonds of hydrogen atoms. The tertiary structure is formed when last repeating structures, seen as units in a three dimensional space, arrange spatially giving an specific three dimensional position to every contained atom in the protein molecule, see figure 3.12. That structure is stabilized mainly with the help of non covalent bonds. Non covalent bonds are weaker bonds in comparison with covalent bonds such as the peptide bonds. A covalent bond results from sharing a pair of electrons between two atoms. A noncovalent bond results from the attraction of two opposite electrically charged sides. If the up to now compound with its tertiary structure is considered as a subunit with the name of polypeptide, we can say that some proteins are composed of only one of such subunits or more than one subunits. The spacial arrangement and position of those subunits constitutes the quaternary structure

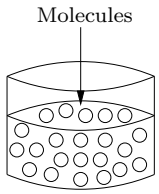



Figure 3.9: Molecules in a chemical substance

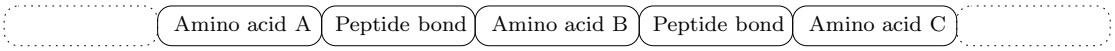



Figure 3.10: Primary structure of a protein

of a multisubunit protein, see figure 3.13. The quaternary structure stabilizes in the same way as the tertiary structure.

The structure of a protein determines its function. For example, solubility in water is dependent on the structure of a protein. Some proteins are soluble in water and some others not. Therefore, proteins can be classified regarding its solubility in water in fibrous proteins and globular proteins. Fibrous proteins build long chains of proteins in the form of filaments, they are not soluble in water and therefore have an structural function that maintain the shape of the conforming biological material even against mechanical movement or external forces. Examples of fibrous proteins are the keratin in hair, the collagen in tendons, the actin and myosin in muscles. Globular proteins have an spherical structure, are soluble in water and are present in liquid biological matter. Globular proteins have a variety of functions such as enzymes, transporters, messengers, containers and regulators among others. Enzymes speed up very specific chemical reactions in normal conditions without being consumed by the reaction. One example is the enzyme lactase that speed up the break down of the lactose, sugar found in milk, into small digestible sugars such as the glucose and galactose. Transporter proteins carry elements from one site to another site. One example is the hemoglobin that transports oxygen and carbon dioxide. Messenger proteins are proteins that participate in cell communication, such as ligands and receptors. Examples of ligands in cell communication are hormones, neurotransmitters and cytokines. Examples of receptors in cell communication are the immunoglobulins. Some immunoglobulins are used to neutralize external pathogenic agents. One example of a protein used as element container is the

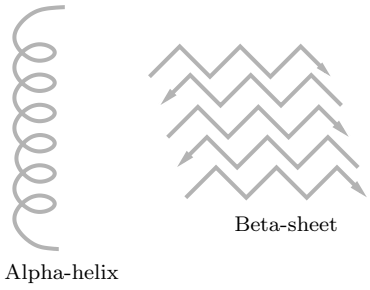



Figure 3.11: Secondary structure of a protein

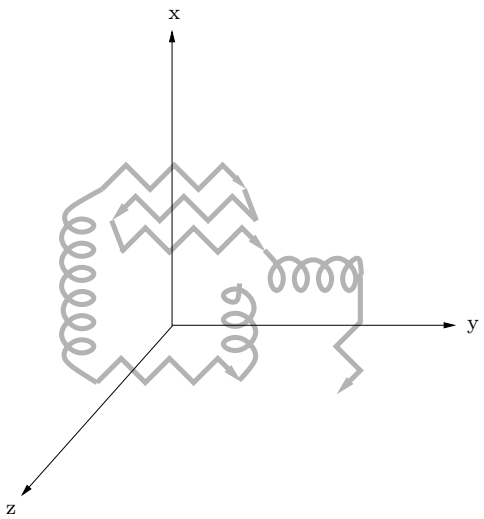



Figure 3.12: Tertiary structure of a protein

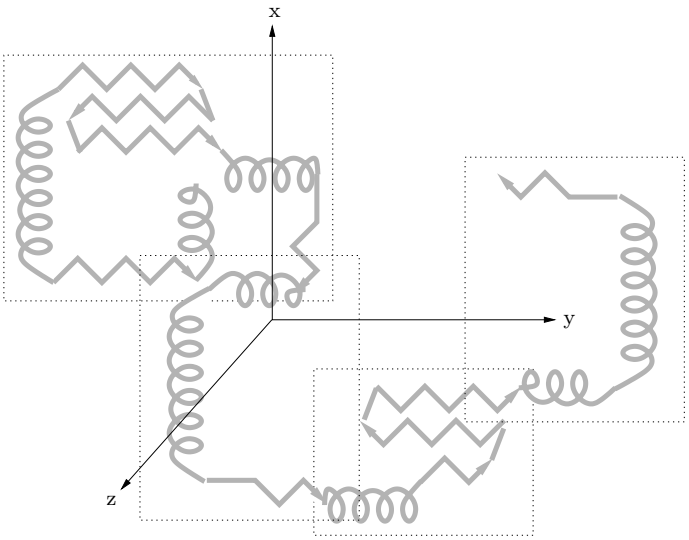



Figure 3.13: Quaternary structure of a protein

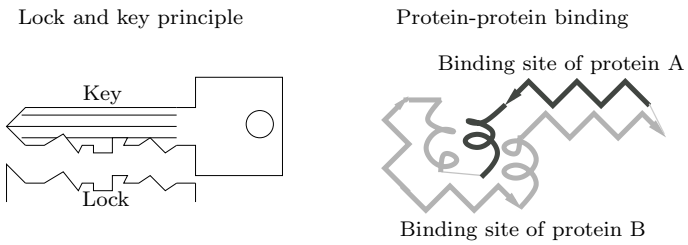



Figure 3.14: Lock and key principle and protein-protein binding analogy

ferritin that stores iron in the spleen. Regulator proteins regulate biological processes, such as the albumin in blood that regulate the osmotic pressure in blood. The osmotic pressure is the pressure necessary to prevent the flow of solvent, often water, through a semipermeable membrane that separates the solvent from a solution containing a solute to which the membrane is impermeable. In this way the flow of water through the membrane cell into the cell is regulated. Toxins are proteins synthesized by biological entities which are used to kill own or foreign cells. Some of such toxins help to increase the flow of water into a cells until it explodes. Such process is called cytolysis.

**Protein-Protein binding** Proteins are molecules that build a sort of molecular machines that execute molecular processes. All that is possible because protein molecules interact by binding each other. Sometimes a protein binds to another protein to form a protein complex, for transporting the binding protein, to modify the binding protein, etc. A protein has a region in its surface named binding site which binds with the binding site of another protein. The form of the binding site is determined by the three dimensional structure of the protein. Protein-protein binding can be explained with the "Lock and Key" model proposed by Emil Fischer in 1894 to explain the binding between an enzyme protein and a substrate protein. The "Lock and Key" model consists in the binding of two specific complementary geometric shapes, because one shape fits exactly into the other shape, see figure 3.14. Then two complementary shapes bind with a non-covalent interaction. However shape is not everything at the time of binding. Chemical properties of the amino acid chains in the surroundings of the binding site like hydrophilic or hydrophobic characteristics or electrical charge influences the binding process and the binding specificity, i.e. to which protein the protein binds. An hydrophilic protein is a protein that tends to interact with water. An hydrophobic protein is a protein that is repelled from water.

The binding site of a protein takes different names, a binding site in a protein that works as ligand is named epitope and the binding site in a protein that works as receptor is named paratope. Some proteins, like the receptors of some immune cells have besides a binding site working as a paratope, another binding site working as epitope named idiotope. Cells expose many identical receptors on their surfaces, but each receptor has only one paratope and, in some cases, one idiotope. The same holds for free receptors. Cells also expose many identical ligands on their surfaces, but each ligand has only one epitope. The same holds for a free ligand. A pathogen can be biological particle, like in the case of a virus, a cell, like in the case of bacteria or monocellular fungi, or multicellular organism, like in the case of multicellular fungi. Then a pathogen is composed of a set of molecules, mostly proteins with binding sites named generally

| Agent | | Molecule mostly protein | | Binding site |
|---|---|---|---|---|
| Immune cell | many → | Receptor | one → | Paratope and one idiotope |
| | | Free receptor | one → | Paratope and one idiotope |
| Immune cell | many → | Ligand | one → | Epitope |
| | | Free ligand | one → | Epitope |
| Pathogen(Antigen) | many → | Pathogenic molecule(Antigen) | one → | Epitope (Antigenic determinant)(Antigen) |

Figure 3.15: Name conventions for binding sites

as epitopes. Such molecules are pathogenic molecules that are potential producers of an immune response. In the immune response, antibodies are secreted by some immune cells. Antibodies are receptors able to bind to epitopes of pathogenic molecules neutralizing them. Therefore any producer of antibodies is named antigen and the binding sites of that antigen are named antigenic determinants. Please do not get confused when in this chapter or in the literature you find the term antigen referring a pathogen, a pathogenic molecule or an antigenic determinant, since all of them produce an immune response that can lead to the production of antibodies. See figure 3.15 for more clarity.

**Ligands** Ligands are molecules mostly protein molecules, amino acids or small inorganic compounds which act as messages during cell communication. A ligand is synthesized by a cell, stored and then placed inside the cell, on the external surface of the membrane of the cell or released into the extracellular liquid, see figure 3.16. In case a ligand is placed inside the cell, it can interact only with a receptor produced by the same cell lying also inside the cell. When a ligand is present on the surface of the membrane of the cell, the ligand travels together with the cell searching for a matching receptor on the surface of other cell. In case the ligand is released into the extracellular liquid, it travels freely and crosses the gap towards a target cell that has a matching receptor. Some free ligands are able to interact with a matching receptor on the surface of other cells. Other free ligands are able to cross the membrane of cells for interacting with receptors which lie inside the cell.

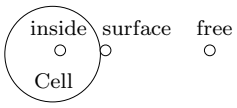



Figure 3.16: Ligand placement

According to their functionality, ligands are classified and get some special names. The most known groups of ligands are described below.

**Cytokines** The term cytokine comes from the words *cyto*, prefix derived from the Greek word *kutos* that means cell or container, and *kinos*, that means movement. Cytokines are protein molecules secreted by cells that move towards target cells for cell communication. They circulate in concentrations of $10^{-12}$ mol/liter. One mol has $6{,}02\times10^{23}$ elemental units, therefore we can say that around $6{,}02\times10^{11}$ cytokines

circulate in each liter of the extracellular liquid. Some examples of cytokines are: interferon, histamine, tumor necrosis factor, etc. Interferons are protein molecules released by virus infected cells for warning other cells about the presence of that virus. Histamine is a compound which is characterized for containing nitrogen. Histamine serves for informing other immune system cells over the place where an infection takes place. Tumor necrosis factor is a molecule that induces programmed cell death in its target cells.

**Hormones** Hormones also serve for cell communication, however, they are produced by endocrine glands. Endocrine glands are organs in the body that secrete hormones into the blood. Hormones usually target cells that are located far away from the ligand releasing cell or gland. Around $6{,}02\times10^{14}$ hormone protein molecules circulate in each liter of the extracellular liquid.

**Neurotransmitters** These are molecules that transmit information from neurons to other cells across a synapse. A synapse is a junction between two cells which permit passing chemical signals through. Neurons communicate only with one cell through that synapse.

**Toxins** It is a poisonous substance which molecules are produced by living cells. The term comes from the latin word *toxicum* that means poison. One effect of toxins in cells is the trigger of the process named cytolysis. Cytolysis, term that comes form *cyte*, that means cell, and the Greek word *luein*, that means to break down, is the dissolution of a cell caused by excessive movement of water inwards so that its cell membrane cannot withstand the pressure of the water inside and consequently it explodes. Toxins destroy specifically or non-specifically other cells. Examples of toxins that target specific cells are neurotoxins, which affect cells of the nervous system, or hemotoxins, which destroy red blood cells. Examples of toxins that target all cells they encounter are necrotoxins, which cause necrosis. Toxins are produced due to two reasons, on one side for predation, such is the case of the toxins produced by scorpions or for defense, such is the case of the toxins produced by ants or bees.

**Chemokines** These are ligands that attract the cells they bind to a determined site in the body. They can also be named chemoattractants and the process of attracting cells or other entities to a determined site is called chemotaxis. Chemotaxis, is a term derived from the word *chemical* and the Greek word *táxis* that means to arrange. Chemotaxis is the process that determines how single cells or multicellular organisms direct their movements according to certain chemicals in their environment. Some other examples of chemotaxis are: the movement of bacteria towards the highest concentration of food or towards the lowest concentration of poison, the movement of sperm towards the egg during fertilization, etc. Chemotaxis is positive if the movement is in the direction of a higher concentration of chemoattractant and negative if the movement is in the direction of a lower concentration of chemoatracttant.

**Damage-associated molecular patterns** In short DAMPs, are molecules produced by cellular stress, i.e. uric acid. They are present on the surface of cells which died by apoptosis. Cells died by necrosis do not show such marks on their surfaces, passing by unnoticed to other cells. However some cell that die by necrosis release molecules of their genetic material which are considered by other cells as DAMPs.

**Pathogen-associated molecular patterns** In short PAMPs, are molecules present in pathogen or produced by them, i.e. bacterial carbohydrates or nucleic acids of viruses. Their structure is found only in pathogen but not in the organism where the pathogen intrudes. An specific PAMP molecule is present in many related pathogens, in consequence the ligand-receptor interaction is not pathogen specific and trigger immediately the innate immune response, which is a generic response. In contrast, the adaptive immune response is a pathogen specific response that is triggered with a time delay only when a serious level of danger for the whole organism is perceived or when the innate immune response has not been sufficiently effective to combat with pathogen.

**Histocompatibility molecules** The term histocompatibility comes from the Greek word *histos* that means tissue. Histocompatibility molecules are molecules present in the surface of the membrane of all cells, except the red blood cells and the nervous system cells. Each person have a unique set of histocompatibility molecules. The histocompatibility molecules are encoded in the major histocompatibility complex, which is a set of genes found in the hereditary information inside cells. A histocompatibility molecule is a cell surface protein whose extracellular region forms a sort of groove, or container with the opening place directed outside the cell, able to host a molecule attached by means of non covalent forces. There are two classes of histocompatibility molecules MHC-I and MHC-II, meaning MHC major histocompatibility molecule. They differ in its structure, its function, the cell where they reside, the molecule that they host and the type of cell that they bind to.

The major histocompatibility complex I molecule, in brief MHC-I, is a molecule present in almost all cells of the body. The molecules that they host are fragments of all the proteins within the cell where they reside. That includes synthesized proteins within the cells. There are three cases where the MHC-I molecules can be helpful for the immune system: by virus infected cells, cancer cells and transplanted tissues. A virus that enters a cell, uses that cell for hiding, for reproducing and for producing pathogenic molecules using its genetic material but the resources of the cell. Each cell in the body uses MHC-I molecules, as mechanism of defense, for displaying such pathogenic molecules in its surface informing external cells that itself is infected by a virus. Cancer cells, which genetic material has been damaged by mutation, start producing suspicious molecules. Those suspicious molecules are displayed on the cell surface by means of MHC-I molecules, warning external cells about the cell damage. Finally, the cells of a transplanted tissue display MHC-I molecules that are different to the MHC-I molecules of the host body. MHC-I molecules of the cells of the transplanted tissue appear suspicious to immune system cells, reason why tissue rejection is produced.

The groove side of the MHC-I molecule and the hosted molecule bind to a matching T-cell receptor of a CD8+ T-cells, please see figure 3.32. Besides, the MHC-I molecule presents in its extracellular region an epitope which binds to the CD8 receptor of the CD8+ T-cell. That second signal is necessary for a successful interaction among both cells that leads to the activation of the CD8+ T-cell.

The major histocompatibility complex II molecule, in brief MHC-II, is a molecule present in antigen presenting cells, or APCs. B-cells and dendritic cells, cells of the immune system are considered as antigen presenting cells. The molecules that an

antigen presenting cell presents in the groove of MHC-II molecules, are fragments of ingested pathogenic molecules taken from its surroundings, that means from outside the cell. APCs ingest extracellular material enclosing it forming a sort of compartment named endosome. The endosome, once inside the cell, fuses with a lysosome. That lysosome, which is a component in the cytoplasm of a cell, digests the extracellular material producing fragments which are, by means of MCH-II molecules, shown in the surface of the APC.

The groove side of the MHC-II molecule and the hosted molecule bind to a matching T-cell receptor of a CD4+ T-cells, please see figure 3.33. Besides, the MHC-II molecule presents in its extracellular region an epitope which binds to the CD4 receptor of the CD4+ T-cell. The second signal is necessary for a successful interaction among both cells that leads to the activation of the CD4+ T-cell.

**Receptors** Receptors are molecules, mostly proteins or amino acids, which interact with matching ligands during cell communication. A receptor has a binding site named usually paratope. There is a great variety of receptors in the cells of the body. The receptors differ in their structure. There are specialized receptors in the immune cells which repertoire is immense, like the B-cell and T-cell receptors. The structure of those receptors is encoded by a gene assembled combining information taken randomly from a pool of gene segments of the genetic material in the cells. The assembled gene is then transcribed into messenger ribonucleic acid (mRNA), process known as transcription, and then synthesized into the corresponding protein or sequence of amino acids, process named as translation.

A receptor is synthesized by a cell, stored and then placed on the external surface of the membrane of the cell, inside the cell or released into the extracellular liquid, please see figure 3.17. Receptors placed on the external surface of the membrane of the cells are called cell-surface receptors, receptors placed inside the cell are named intracellular receptors and receptors released to the extracellular liquid are named free receptors. Cell-surface receptors are used by the cells for sensing the external environment. For instance, immune system cells use cell-surface receptors for recognizing: external agents, i.e pathogenic molecules; internal malicious agents, i.e. infected cells, tumor cells; and cellular debris, i.e. dead cells, worn-out cells. Receptors placed inside the cell interact with ligands which crossed the membrane of the cell, i.e. steroids, or with matching ligands produced by the same cell. Receptors released into the extracellular liquid serve to mark external agents or cellular debris for being processed by immune system cells.

According to their functionality, receptors are classified and get different names. The most important groups of receptors are presented below.

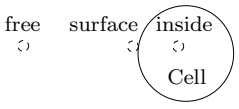



Figure 3.17: Receptor placement

**Cytokine receptor** Are receptors that bind to cytokines. This term is in a way not much descriptive, however can be found at the time of talking in terms of cytokines.

**Pathogen Recognition Receptors** In short, PRRs, are receptors that bind with pathogen-associated molecular patterns found in pathogens and with damage associated molecular patterns produced by stressed or death cells. Please see figure 3.18.

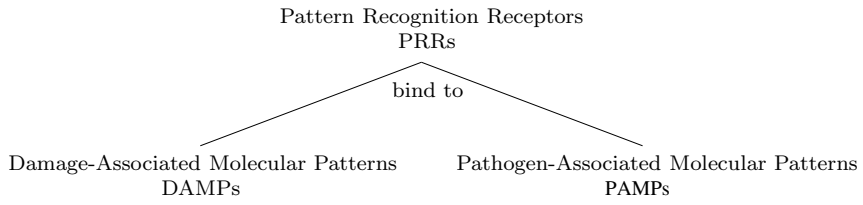



Figure 3.18: PRRs, DAMPs and PAMPs

PRRs exist free in the blood and lymph or as surface receptors in cells. Free PRRs bind to pathogenic molecules marking them for phagocytosis, process called opsonization. Phagocytosis is the the process of ingestion of solid extracellular matter by a cell, while pinocytosis is the process of ingestion of liquid matter. Both processes are named endocytosis since the ingestion consists in the invagination of a portion of the membrane of the cell forming a vesicle named endosome. Surface PRRs receptors can also help in the phagocytosis process binding a PAMP of a pathogenic molecule and promoting the formation of an endosome around it, process named as receptor-mediated endocytosis. Reason why such PRRs are called endocytic PRRs. Macrophage cells, cells of the immune system, have lots of those receptors for executing its main task of phagocytosis.

Other surface PRR receptors, named signaling PRRs, after binding with PAMPs initiate an appropriate response for combating the pathogen, i.e. production of specific cytokines, inflammation in the site of infection, etc. Such receptors are named toll-like receptors and recognize roughly the nature of the pathogen. In mammals 12 kinds of TLRs are known. TLR-1 and TLR-2 bind to bacteria like the streptococci and staphylococci. TLR-3 binds to the double-stranded RNA of viruses. RNA stands for ribonucleic acid, acid that contains genetic material. TLR-4 binds to bacteria like the salmonella and escherichia coli. TLR-5 binds to the motile part of the bacteria listeria. TLR-6 binds to mycoplasma bacteria like the one that causes the atypical pneumonia. TLR-7 and TLR-8 bind to single-stranded RNA of viruses like influenza, measles and mumps. TLR-9 binds unmethylated CpG oligodeoxynucleotides which are single-stranded synthetic DNA molecules present in microbial genetic material. TLR-11 bind to parasitic protozoa like the plasmodium that causes malaria or the toxoplasma has a serious effect in the fetus of an infected mother. The exact functions of TLR-10 and TLR-12 are not well understood.

**B-cell receptor** A B-cell receptor is a molecule produced by a B-cell. It is able to interact with a very specific pathogenic molecule. A B-cell shows thousands of copies of the same B-cell receptor on its surface. The interaction of a B-cell receptor of a B-cell with a particular pathogenic molecule initiates the activation process of the B-cell.

A secreted B-cell receptor that moves freely in the body is called antibody. Pathogenic molecules able to induce the generation of antibodies are also called antigens,

name that comes from antibody generator. An antibody binds to a pathogenic molecule marking it for being removed from the body. The removal of an opsonized antigen is executed by special cells through phagocytosis. Paghocytosis is not possible without the opsonization of the antigen. That is because the membrane of the phagocytic cell and the antigen, both have negative charge, making difficult that both come close together. The antibody neutralizes the antigen by interacting with it.

A B-cell receptor is a so called glycoprotein. The prefix glyco is given because it has attached carbohydrate molecules to them. That protein molecule is composed of four subunits, please see figure 3.19. Two identical subunits called light chains, with two hundred amino acids, and other two identical subunits, twice bigger as the light chains, named heavy chains. The first one hundred amino acids of all four chains vary from B-cell receptor to B-cell receptor. That is why are called all together the variable region an the rest the constant region. In the variable region there are regions where the variability is the highest, that regions are named hypervariable regions. Since the hypervariable regions form the three dimensional structure of the binding site, or paratope, of the B-cell receptor, are also named complementarity determining regions. So, the epitope of the antigen binds by complementarity to the paratope of the B-cell receptor. Then, the constant region determines which response will be triggered by the cell after interaction.

The light chains can present two different kinds of constant regions named kappa ($\kappa$) and lambda ($\lambda$). The heavy chains can present five different kinds of constant regions named mu ($\mu$), gamma ($\gamma$), alpha ($\alpha$), delta ($\delta$), and epsilon ($\epsilon$). The B-cell receptors get a combination of one of the two kinds of constant region in the light chain with another one of the five kinds of constant region in the heavy chains. According to which kind of constant region in its heavy chains the B-cell receptor has, it gets different names: IgM, IgG, IgA, IgD and IgE. The prefix Ig comes from another name given to the B-cell receptor that is immunoglobulin.

The specificity of a B-cell receptor to a particular pathogenic molecule is given before the B-cell encounters that pathogenic molecule. The specificity is encoded in gene segments which by random combination give birth to a big repertoire of B-cell receptors in the B-cells of the body. For that there are four kinds of gene segments: V for variable, D for diverse, J for joining and C for constant. For the variable region of the heavy chains there are 51 V gene segments, 27 D gene segments and 6 J gene segments in the chromosome 14 of the human genetic material. For the constant region of the heavy chain there are 9 gene segments also in the chromosome 14 with the following distribution: 1 gene segment for a $\mu$ constant region, 1 gene segment for a $\delta$ constant region, 4 gene segments for the four types of $\gamma$ constant regions, 1 gene segment for an $\epsilon$ constant region and 2 gene segments for the two types of $\alpha$ constant regions. For the variable region of the light chains of B-cell receptors with a kappa constant region there are 40 V gene segments and 5 J gene segments in the chromosome 2. For the variable region of the light chains of B-cell receptors with a lambda constant region there are 31 V gene segments and 4 J gene segments in the chromosome 22. Thus, there is the possibility of having $51 \times 27 \times 6 \times 40 \times 5 \times 31 \times 4 = 2.5 \times 10^8$ different B-cell receptors, without considering the constant region which is responsible for the reaction not for

the binding specificity. Moreover, there are some more complicated processes in the segment combination that makes the B-cell receptor binding specificity even more diverse. The 9 gene segments for the constant region allows the B-cell to produce B-cell receptors with different effector function, that is to say, B-cell receptors or antibodies that react in different way against the antigen they bind.

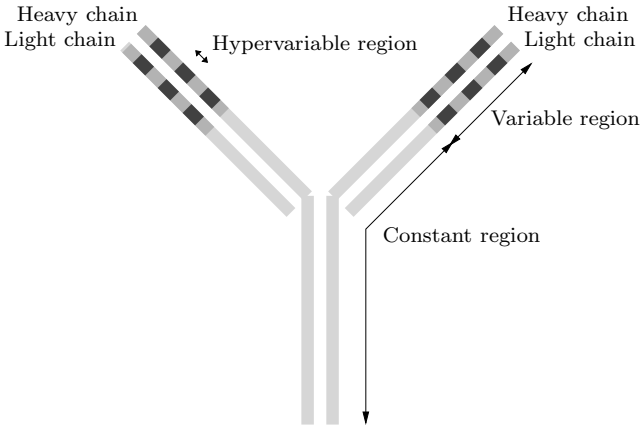



Figure 3.19: B-cell receptor

**T-cell receptor** A T-cell receptor is a molecule present in the surface of a T-cell. A T-cell receptor is able to interact with a particular epitope or antigenic determinant of a pathogenic molecule, presented in a histocompatibility molecule by an antigen presenting cell. That interaction together with some other cells signals activate the T-cell.

The T-cell receptor is a protein molecule containing two identical subunits named alpha ($\alpha$) and beta ($\beta$), please see figure 3.20. Each subunit has a variable region and a constant region. The variable regions presents, each, three hypervariable regions, named similarly as for the B-cell receptors, complementary determining regions. The complementary determining regions constitute the binding site, or paratope, which according with its specificity, bind to a particular epitope or antigenic determinant in a pathogenic molecule and the MHC molecule.

The specificity of a T-cell receptor to a particular pathogenic molecule is given before the T-cell encounters that pathogenic molecule. The specificity is encoded in gene segments which by random combination give birth to a big repertoire of T-cell receptors in the T-cells of the body. For that there are four kinds of gene segments: V for variable, D for diverse, J for joining and C for constant, same as for B-cell receptors. For the variable region of an $\alpha$ subunit there are 50 V gene segments and 50 J gene segments in the chromosome 14 of the genetic material of the cell. For the variable region of the $\beta$ subunit there are 20 V gene segments, 13 J gene segments and 2 D gene segments in the chromosome 7. Then there is the possibility of having $50 \times 50 \times 20 \times 13 \times 2 = 1.3 \times 10^6$ different T-cell receptors. That number does not consider the constant region which is responsible for the type of reaction that the cell takes in case of a successful binding. The constant region is not responsible for the T-cell receptor binding specificity.

Some T-cell receptors contain other two subunits named gamma ($\gamma$) and delta

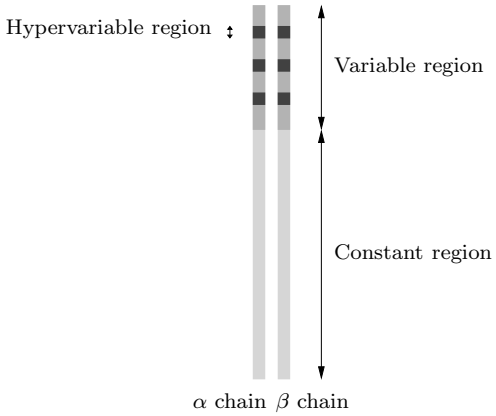



Figure 3.20: T-cell receptor

($\delta$). They are produced transcribing and translating the gene assembled with gene segments stored in chromosome 7 for the $\gamma$ protein subunit and gene segments stored in chromosome 14 for the $\delta$ protein subunit.

#### Cell communication model - Part 2

David Berlo extended the general model of communication in 1960 with the concept of channel, resulting in *Sender→Message→Channel→Receiver*. A channel is the medium where the message travels towards the receiver, statement taken from [Wikipedia, 2010]. The idea of a communication medium applies only to a long distance communication where the sender and the receiver are apart. In the context of cell communication the channel is the extracellular liquid in the cell environment where the ligand travels through before interacting with a receptor. Regarding where the ligand is located and where the ligand-receptor interaction takes place, five types of cell communication can be distinguished:

**Juxtacrine communication** The ligand-receptor interaction is given in cells having physical contact. The ligand is located on the surface of the ligand producing cell and the receptor on the surface of the receiving cell. The ligands in this kind of communication are usually protein molecules on the surface of cells which need to transmit precisely a message.

**Paracrine communication** Communication of cells over short distances. The ligand is released by the ligand producing cell and travels the extracellular liquid. The ligand-receptor interaction in the proximity of the ligand producing cell with a cell having a matching receptor. The ligands in this kind of communication are usually neurotransmitters or cytokines.

**Endocrine communication** Communication of cells over long distance. The ligand travels the extracellular liquid and the ligand-receptor interaction is given far away of the ligand producing cell with a cell having a matching receptor. Ligands are usually hormones.

**Autocrine communication** Communication of the cell with itself. The ligand for this kind of communication, once outside of the ligand producing cell, binds to a specific receptor

in the same cell. This kind of communication can be compared with the internal conversation that a person establish with itself speaking aloud. The ligands for this kind of communication are usually cytokines.

**Intracrine communication** Communication of the cell with itself. The ligand stays inside the ligand producing cell and binds to a matching receptor lying inside the same cell. This kind of communication can be compared with the internal conversation that a person establish with itself in silence. The ligands for this kind of communication are usually are cytokines.

### Cell communication model - Part 3

Finally, Wilbur Schram indicated in 1954 that it is important to consider also in the general model of communication the impact that the message has on the receiver, statement taken from [Wikipedia, 2010]. That idea concerns what happens in cell communication since after a ligand-receptor interaction occurs, the cell having the receptor transduces the signal till its nucleus and decodes from genetic material the respective cell response. In consequence, the process of cell communication can be modeled as *Ligand-receptor interaction→Signal transduction→Cell response*.

**Signal transduction** A surface cell receptor has a side on the internal surface of the cell membrane that binds with molecules inside the cell across a signal transduction pathway. There are different signal transduction pathways. Signal transduction can help amplifying the signal or merge the signals of multiple receptors. The ligand-receptor interaction initiates a determined signal transduction pathway, which at the end activates a determined transcription factor able to turn a respective gen or set of genes on in the nucleus of the cell. The information of that genes is taken into messenger ribonucleic acid, in short mRNA, process named as transcription.

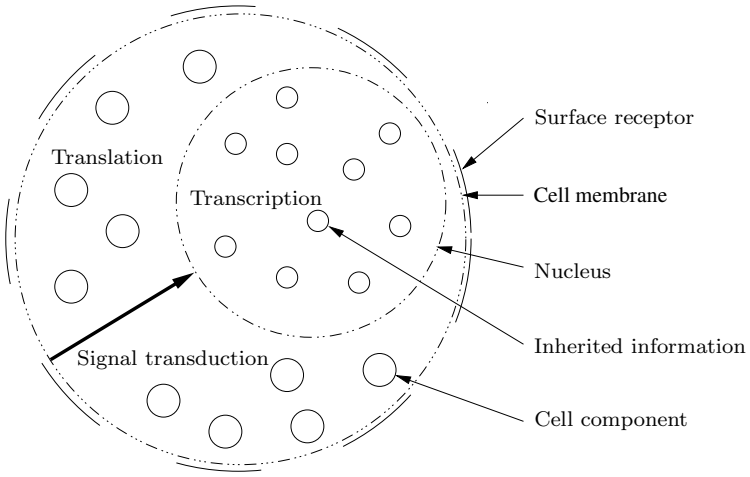



Figure 3.21: Signal transduction, transcription and translation

**Cell response** At the structural level, the response of a cell is codified by genes, stored in the genetic material of the organism in the nucleus of the cell, made of deoxyribonucleic acid, in short DNA. A gene is considered the unit of heredity in any living organism. A gene

defines the sequence of amino acids of one protein. Note that the entire genetic material of an organism is inside all cells and is named genome, however, in each cell only some parts of that genetic material is expressed, reason why it is possible to have different cells. The entire set of proteins expressed by a cell at a defined time and conditions is named proteome. The information of the composition of an specific amino acid is contained in a triplet of nucleotides named codon, encoded in the genetic code. A nucleotide is a molecule that can be defined as the structural unit in the inherited information in a cell, please see figure 3.22. The genetic code consider the four existing nucleotides: adenine (A), cytosine (C), guanine (G) and Thymine (T) or uracil (U) as components of codons of three nucleotides. Uracil is present in transcribed genetic material outside the nucleus or genetic material of viruses or cells without nucleus made of ribonucleic acid, in short RNA, instead of DNA. That gives the possibility of having $4^3 = 64$ kinds of amino acids. But, the genetic code is redundant since there exist 20 kinds of amino acids in the structure of cell proteins. A cell is a kind of molecular machine, it produces molecules, mostly proteins, from the transcribed mRNA in ribosomes, process called as translation. A ribosomes is a component in the cytoplasm of a cell. Please refer to figure 3.21 for getting the idea where each process, signal transduction, transcription and translation, take place inside the cell.

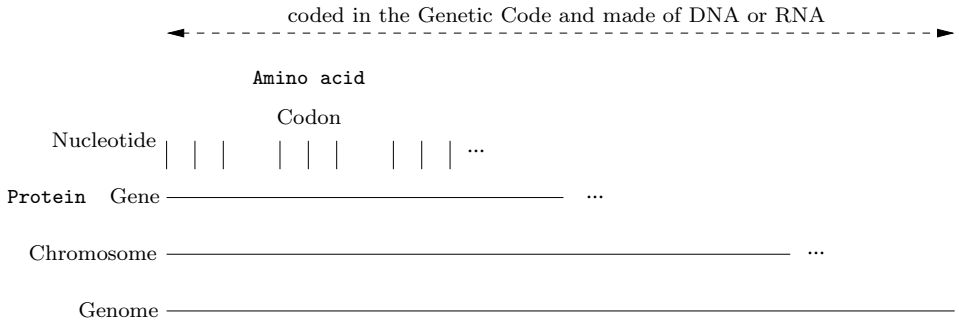



Figure 3.22: Genetic material

At a functional level, a ligand-receptor interaction triggers in the target cell a response that depends on: the characteristics of the ligand, the number of ligands around the cell, the number of receptors available on the cell for such ligand, the affinity between ligand and receptor and the activated pathway for signal transduction. The most known target cell responses to its environment after ligand-receptor interaction are: secretion of identical or different ligands, increment of the number of receptors for the same or different ligands and suppression of its response. Ligand-receptor interaction can produce also an effect on the target cell itself. Thus, according to that reaction, ligands can be classified and named as growth, death or survival factors. A growth factor stimulates or inhibit the growth of the cell, its proliferation by cell division, its maturation or its differentiation. A death factor causes a cell to undergo a programmed cell death. A survival factor is a factor that looks for the maintenance of the functional activities of the target cells and which deprivation can lead to a programmed cell death.

### 3.1.4 Immune system infrastructure

The immune system infrastructure is a set of organs and ducts connected together. The immune system agents, the leukocytes, use such infrastructure to move around the body and perform the immune response under the presence of pathogens in the body. The immune system infrastructure consist of the cardiovascular system and the lymphatic system. Both systems are generally presented in the literature apart, therefore they will be presented separately below. After that, the use of such infrastructure for the immune response is exposed.

**Cardiovascular system** The cardiovascular system is composed of the blood, the heart and the blood vessels, see figure 3.23. The cardiovascular vessels transport blood throughout the body being pumped by the heart. The cardiovascular system is a closed system, meaning that the blood flows in a circular way without leaving the blood vessels. Small permeable blood vessels in the surroundings from tissues allow the delivery of nutrients from the blood to the tissues and the gathering of waste products from the tissues to the blood. The blood is important for the immune system because it transports ready to combat leukocytes to the tissues. Furthermore, since the blood is spread throughout the body, it also serves as a communication medium transporting substances which act as messages for the leukocytes.

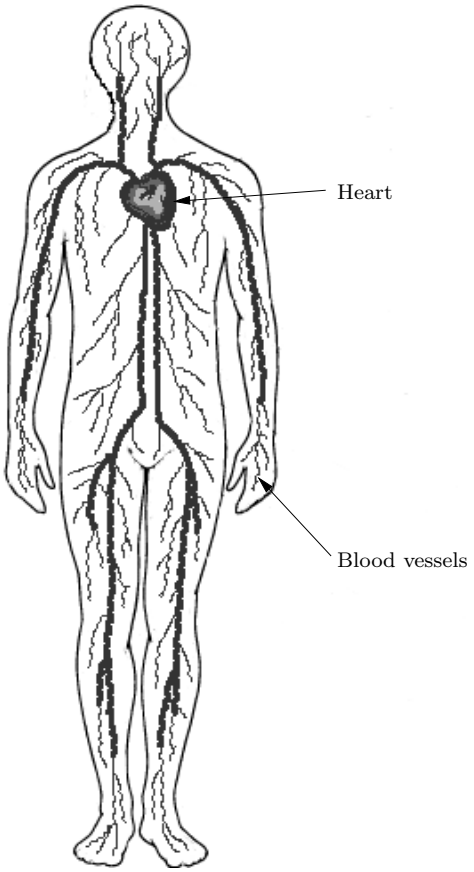



Figure 3.23: Cardiovascular system

**Lymphatic system** The lymphatic system is composed of the lymph, the lymph nodes and the

lymph vessels. Small permeable lymph vessels closed at one end and located in-between the cells of tissues, allow the gathering of the lymph. The lymph, term which comes from the latin word *lympha* that means clear water, is composed of leukocytes which trapped pathogen and cellular debris. The lymphatic vessels transport the lymph from the tissues to the lymph nodes. It is important to note that, there are not lymph vessels that bring the lymph back to the tissues, instead new leukocytes produced in the lymph nodes are directed to the blood vessels for moving into the blood. Consequently, the lymphatic system is considered a one way open system. Furthermore, the lymphatic system relies on some organs like the bone marrow, the thymus and the spleen for producing leukocytes and the tonsils for battling with inhaled or digested pathogen, please see figure 3.24.

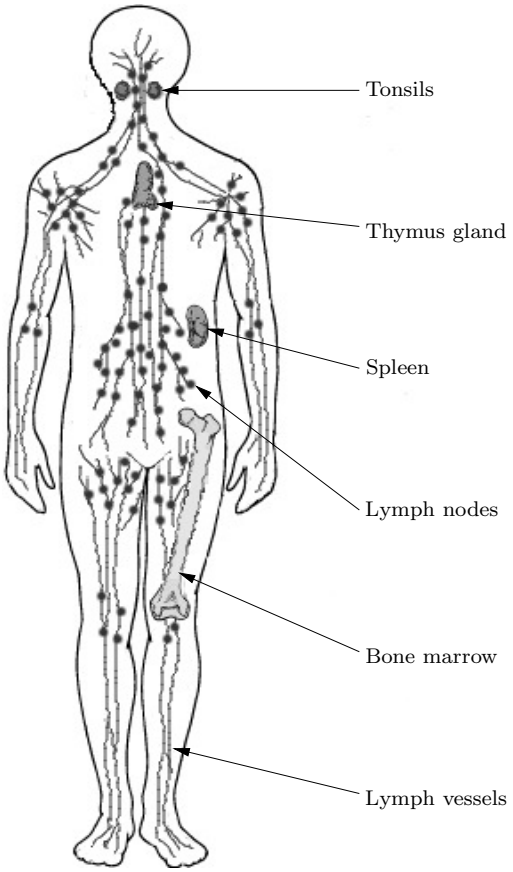



Figure 3.24: Lymphatic system

During the immune response, the immune system infrastructure is used actively in the following way. Leukocytes are produced in the bone marrow, a tissue inside the largest bones. Some of them migrate to the spleen and some others to the thymus in order to become fit for recognizing pathogen. Leukocytes enter the blood, flow in the blood vessels throughout the body, and guard for the presence of pathogens. In case pathogens invade the body through a wound, called site of infection, leukocytes migrate to the tissues being affected by pathogens and combat with them. In the other case when pathogens are ingested or inhaled, the first

organ of defense that they encounter are the tonsils. The tonsils have leukocytes on their outermost layer that scans for the presence of pathogens. The cellular debris, product of the battle between leukocytes and pathogens, and the leukocytes that trapped pathogen and communicate the situation on the site of infection, enter the lymph and flow in the lymphatic vessels towards the lymph nodes. The lymph nodes get rid of cellular debris and process the information communicated by the arriving leukocytes. With the processed information, new more skilled leukocytes are produced. The newly produced leukocytes enter the lymph towards blood vessels were the lymph is mixed with the circulating blood. The leukocytes that entered the blood move again to the site of infection or compensate the volume of leukocytes that left the blood for fighting with pathogens.

### 3.1.5 Immune system agents

The immune system agents are cells responsible of executing the immune response in the immune system. Such cells are named leukocytes. The term leukocyte means white cell coming from the Greek words *leukos*, that means white, and *kutos* that means cell. They were named white cells since in laboratory they show up in white color.

Leukocytes are cells which are mainly produced from the hematopoietic stem cell in the bone marrow. An hematopoietic stem cell, term which comes from the Greek words *haimat* that means blood and *poietic* that means create, is a type of stem cell that gives origin to all cells of the blood. Stem cells are cells in multicellular organisms that have the ability to reproduce through cell division and then differentiate into several types of cells. Stem cells can be embryonic or adult. Embryonic stem cells are found on embryos and adult stem cell on tissues of the body. Adult stem cells replenish cells regenerating in this way damaged tissues.

Leukocytes produced from the hematopoietic stem cell differentiate into several types of cells with different physical functions. Thus, leukocytes can be classified regarding the presence of granules inside the cell in granulocytes and agranulocytes, see figure 3.25. Granulocytes are small sacks which serve for storing cell produced substances, for transporting substances inside the cells or as compartment where chemical reactions take place, such is the case of decomposition of complex substances. Among granulocyte leukocytes are the basophil cells, the neutrophil cells and the eosinophil cells. Agranulocytes leukocytes are the monocytes and the lymphocytes. Basophil cells, neutrophil cells, eosinophil cells and monocytes perform mainly the innate immune response. Lymphocytes, except natural killer cells, are responsible of the adaptive immune response, see figure 3.26. The function of each kind of leukocyte is described below.

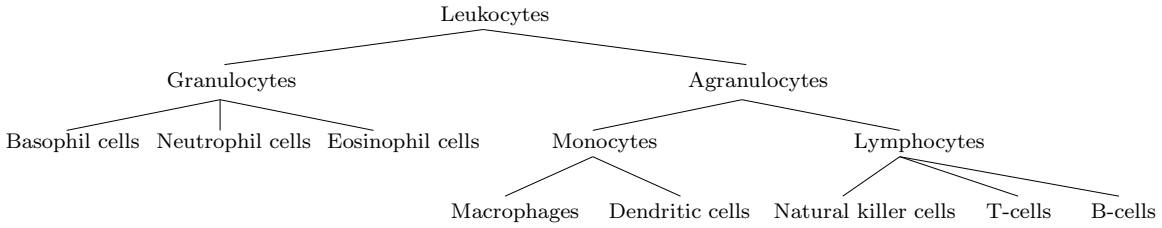



Figure 3.25: Leukocyte classification

**Basophil cells** These cells have granules that change color with basic dyes. A dye is a coloring liquid. Basophil cells derive their name from the word basic and from the Greek word

| | | | |
|---|---|---|---|
| Innate immune response | Basophil cells | Neutrophil cells | Eosinophil cells |
| | Macrophages | Dendritic cells | Natural killer cells |
| Adaptive immune response | T-cells | | B-cells |

Figure 3.26: Leukocytes in the innate and adaptive immune response

*philos*, that means beloved. Basophil cells release the organic compound histamine that causes inflammation, see figure 3.27. Histamine act as a chemokine. Inflammation is a process that can be defined as the movement of leukocytes out of the blood vessels to the site of infection. Big amounts of leukocytes in the site of infection are the reason of swelling. The histamine is an organic compound that serves for informing other leukocytes over the site where the infection takes place.

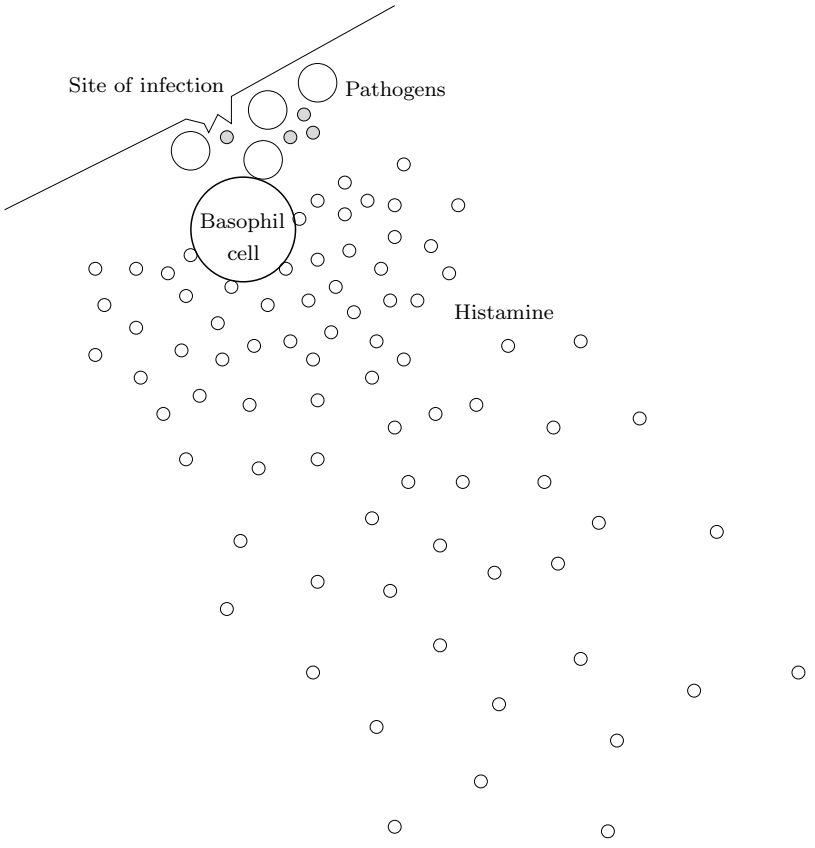



Figure 3.27: Basophil cell releases histamine for attracting other leukocytes

**Neutrophil cells** These cells have granules which change color with neutral dyes. Neutrophil cells move by chemotaxis to the site of infection and engulf pathogens, i.e. bacteria, please see figure 3.28. Neutrophil cells die after ingesting pathogen forming a yellow-

white substance called pus around the site of infection.

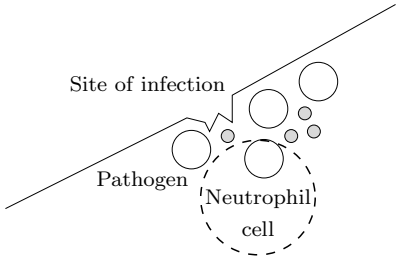



Figure 3.28: Neutrophil cell engulfs pathogen and dies

**Eosinophil cells** These cells have granules which change color with the acid dye eosin. Eosinophil cells release toxins which are poisonous organic compounds that produce the death of pathogen infected cells, i.e. virus infected cells, please see figure 3.29.

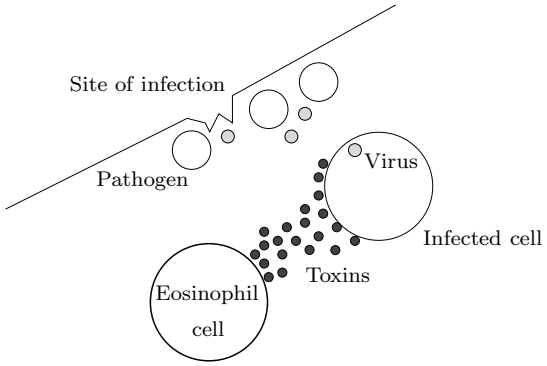



Figure 3.29: Eosinophil cell releases toxins for killing infected cell

**Monocytes** The term monocyte is derived from the word *mono* that means one. Monocytes are cells characterized for having one large nucleus. Monocytes move by chemotaxis out from the blood vessels towards the site of infection. Once in the site of infection, they execute the process of phagocytosis. The term phagocytosis comes from the Greek words *phagein*, that means to eat and *osis*, that mean process. Phagocytosis is the process of engulfing solid material such as cellular material and pathogens. Monocytes differentiate in the site of infection into macrophages and dendritic cells.

**Macrophages** The term macrophage comes from the Greek words *makro*, that means large, and *phagein*, that means to eat. Macrophages are cells that mainly execute phagocytosis removing dead cells, such as neutrophils that engulfed pathogen and then died, worn-out cells and cellular debris, please see figure 3.30.

**Dendritic cells** These are cells located in the interfaces between the external and internal environment of the body such as the skin and the lining of the gastrointestinal tract. They reach such tissues transported by the blood. Dendritic cells are said to have three states immature, semi-mature and mature. In a mature state they present lots of branched projections called dendrites, reason why they got the name of dendritic cells. Dendritic cells have the function of collecting, processing and

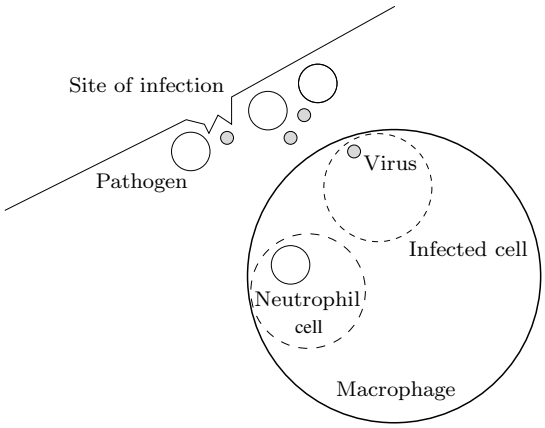



Figure 3.30: Macrophage cell removes dead cells

presenting antigen material to cells responsible of the adaptive immune response, mainly to CD4+ T-cells. Therefore, they are considered as the interface between the innate immune system and the adaptive immune system.

A dendritic cell collects antigenic material by receptor endocytosis. In receptor endocytosis, a cell ingests a molecule when one of its receptors binds to that molecule. To support the task of collecting antigenic material, a dendritic cell has toll like receptors for a variety of typical ligands present in pathogen. Besides, a dendritic cell has a set of receptors for further pathogen-associated molecular patterns, for damage-associated molecular patterns produced by macrophages and other injured or death cells, and for inflammatory cytokines. Those signals help the dendritic cell by assessing the danger situation around the antigenic material.

The collected material is processed in the lysosomes of the dendritic cell. A lysosome is a component in the cytoplasm of the cell that fragments the ingested material. Those fragments are arranged together with histocompatibility molecules class II and located in the external surface of its cell membrane.

Then, the dendritic cell migrates from the tissue to the lymph node or spleen. T-cells and B-cells reside in the lymph nodes or the spleen, being the probability of encountering a CD4+ T-cell with a T-cell receptor that matches the pathogenic fragment together with the MHC-II molecule high. In case the T-cell receptor of some CD4+ T-cell binds the presented compound of the dendritic cell, the T-cell waits for a co-stimulatory signal in order to become activated. For that signal, the CD4+ T-cell has a receptor named CD-28 and the dendritic cell produces a ligand named B7 when the situation at the site of infection is dangerous. Once activated, the CD4+ T-cell starts to differentiate according to additional ligands secreted also by the dendritic cell. Figure 3.33 gives a pictured idea of how a dendritic cell works.

A dendritic cell besides having the ability of activating a CD4+ T-cell is also able to inform the T-cell where to go for combating the antigen. For that, a dendritic cell that collects antigen in the skin, also collects vitamin $D_3$-calciferol and convert it in calcitriol. Calcitriol is secreted by the dendritic cell and induces the activated CD4+ T-cell to produce CCR10 receptors. CCR10 receptors bind to

CCL27 chemokines present in the skin. That means, the CD4+ T-cell is attracted to move into direction of the skin by chemotaxis. A dendritic cell that collects antigen in the gastrointestinal tract takes vitamin A-retinol, converts it to retinoic acid and secretes it nearby the activated CD4+ T-cell. The CD4+ T-cell produces CCR9 receptors that bind to CCL25 chemokines which lead the CD4+T-cell to the intestine by chemotaxis.

There are also dendritic cells which origin are not monocytes. Those dendritic cells are named lymphoid dendritic cells or plasmacytoid dendritic cells, because they follow the lymphoid differentiation of the hematopoietic stem cell, not the myeloid differentiation which gives origin to monocytes. They have TLR receptors, for example the TLR-7 receptor which binds the single stranded ribonucleic acid of viruses. Under a viral infection they produce high amounts of a cytokine named interferon. Interferons are ligands which trigger the immune system activating natural killer cells and increasing the reproduction of cytotoxic cells, in that way interfering the viral or tumor cell replication.

Dendritic cells are not completely understood. Many questions are open like: at which point a dendritic cell decides to migrate to the lymph node, how much antigen the dendritic cell collects in the tissue, which concentration of external signals is necessary for producing a co-stimulatory signal able to activate a CD4+ T-cell, what is the life span of the dendritic cell after presenting an antigen to a CD4+ T-cell, how many CD4+ T-cells can a dendritic cell activate, how dendritic cells present antigen to B-cells. Those questions where reported open at the time of developing the algorithm presented in [Greensmith, 2007] and in section 3.2.4 of this chapter.

**Lymphocytes** These are a type of leukocyte responsible of executing the adaptive immune response. Natural killer cells, B-cells and T-cells belong to the group of lymphocytes. Below a description of those cells.

**Natural killer cells** These are cells specialized in killing virus infected cells and tumor cells. Natural killer cells recognize their target cells using a set of receptors. Their receptors are classified in activating receptors and inhibiting receptors. Activating receptors are specialized in recognizing suspicious molecules, however, activator receptors in natural killer cells are not as specialized and diverse as the receptors of B-cells and T-cells. Natural killer cells do not develop an adaptive immune response, reason why they belong to the innate immune system rather to the adaptive immune system. Inhibitor receptors bind to MHC-I molecules. Viruses normally suppress the expression of MHC-I molecules in the cell they have infected. The same happens with cancer cells that present a reduced amount of MHC-I molecules. Thus, when the number of activating signals is higher that the number of inhibitory signals, the natural killer cell becomes activated and starts performing actions for killing the cell. For that, natural killer cells release toxic substances such as perforins and granzymes near the cell to be killed. Perforins perforate the cell membrane of the cell allowing granzymes enter the cell. Granzymes induce the cell to die by apoptosis and to produce the so called reactive oxygen species (ROS). Reactive oxygen species are free radicals, or better known as oxidants, that have the ability of damaging the cell reacting quickly with molecules in cell structures

for reaching an stable state. When activated, natural killer cells release also ligands that stimulate macrophages to kill the bacteria they have phagocyted and attract other immune cells to the site of infection producing inflammation.

**T-cells** A T-cell is a cell of the immune system that participates in the adaptive immune response. Its surface is covered with thousands of identical copies of receptors named T-cell receptors. T-cell receptors are specific, that means, they bind to a particular epitope of a pathogenic molecule. When that happens, a T-cell may become activated, starting processes that contribute with the adaptive immune response. A T-cell with T-cell receptors able to bind to a particular pathogenic molecule exists even before the T-cell encounters that pathogenic molecule. The particular structure of a T-cell receptor is encoded in a gene which is assembled from randomly taken gene segments of the genetic material in the nucleus of the cell during T-cell maturation.

T-cells mature in the thymus, reason why they got their name. First, a population of hematopoietic stem cells produced in the bone marrow migrates through the blood into the thymus. The hematopoietic stem cells begin cell division for generating a population of immature T-cells with a T-cell receptor of random specificity and both CD8 and CD4 receptors. Then, the immature T-cells are confronted, in the cortex of the thymus, to a wide variety of molecules that belong the body, mostly proteins, inserted in histocompatibility molecules. When a T-cell is not able to bind the body molecule–histocompatibility molecule compound successfully, it is removed by apoptosis. Otherwise, the T-cell is conserved down-regulating one of its receptors CD4 or CD8, becoming a CD8+ T-cell or a CD4+ T-cell respectively. The T-cell becomes a CD4+ T-cell when it binds successfully a body molecule–MHC-II molecule compound or a CD8+ T-cell when it binds successfully to a body molecule–MHC-I molecule compound. That process of maturation is named positive selection, since a T-cell, under successfully recognition, is positively selected. This process has two objectives, first to train the T-cells for recognizing body molecules, and second to prove the functionality of the T-cells, particularly the ability to bind to the histocompatibility molecules produced by body cells. Histocompatibility molecules produced by body cells are named as self-histocompatibility molecules in order to differentiate with histocompatibility molecules of cells of foreign tissues or cells of external organisms.

Auto-immunization is the process by which the immune system produces an immune response against molecules that belong the body, mostly own proteins. Auto-immunization can occur when a T-cell receptor of a T-cell binds with a body molecule–self-histocompatibility compound so strongly that the T-cell initiates an adaptive immune response against that molecule, damaging the own body or even killing it. In order to avoid that situation, T-cells are confronted to body molecules inserted in histocompatibility molecules in the medulla of the thymus. When the T-cell binds too strongly to any presented body molecule–histocompatibility molecule compound, it is considered as auto-reactive and therefore it is eliminated. That process is named negative selection, since under receptor interaction, the T-cell is negatively selected, that means removed form the T-cell set. That process follows the idea that T-cell receptors of a T-cell that bind with compounds of body molecule–self-histocompatibility molecule with low affinity, may bind compounds

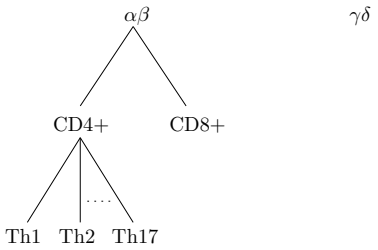



Figure 3.31: T-cell classification

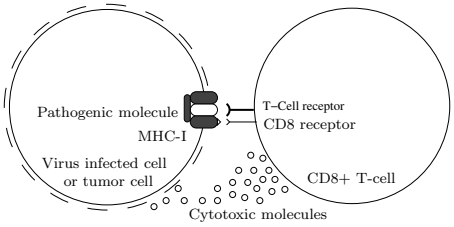



Figure 3.32: Pathogen - cell - CD8+ T-cell

of pathogenic molecule–self-histocompatibility molecule with high affinity. Finally, the remaining T-cells are considered as mature and leave the thymus. On those T-cells the body relies to mount immune responses on. Even though auto-reactive T-cells have been detected before leaving the thymus, some of them may pass unrecognized through. In that case, when any body molecules–MHC-I molecules compound of an antigen presenting cell interacts with the T-cell receptor of a T-cell in the lymph nodes or the spleen, the dendritic cell starts a process to get rid of that T-cell. Auto-reactive T-cells are potential initiators of auto-immune diseases, therefore the body tries to eliminate them following diverse paths. That is not the case for B-cells, since B-cells need T-cells for being activated.

Regarding the type of T-cell receptor a T-cell has, there are two kinds of T-cells, $\alpha\beta$ T-cells and $\gamma\delta$ T-cells, please see figure 3.31. The function of $\alpha\beta$ T-cells is better understood than the function of $\gamma\delta$ T-cells. $\gamma\delta$ T-cells have been identified binding directly pathogenic molecules in tissues of the body, acting actually as pathogen collectors instead of relying in antigen presenting cells. That is also the reason why $\gamma\delta$ T-cells present neither CD4 nor CD8 receptors that interact with the side epitope of histocompatibility molecules. $\gamma\delta$ T-cells migrate from the thymus to tissues that connect the body with the external world like the skin and the gastrointestinal tract, constituting a first line of defense against pathogenic molecules.

$\alpha\beta$ T-cells can have either a CD8 receptor or a CD4 receptor being named CD8+ T-cells and CD4+ T-cells respectively. The T-cell receptors of CD8+ T-cells bind with pathogenic molecule–MHC-I molecule compounds. Compounds body molecule–MHC-I molecule are present in almost all cells of the body showing every protein of the cell or produced inside the cell. A virus that enters a cell produces unnoticed pathogenic molecules, but using MHC-I molecules, the cell can inform outside about the anomaly. Then, a CD8+ T-cell which T-cell receptor bind such

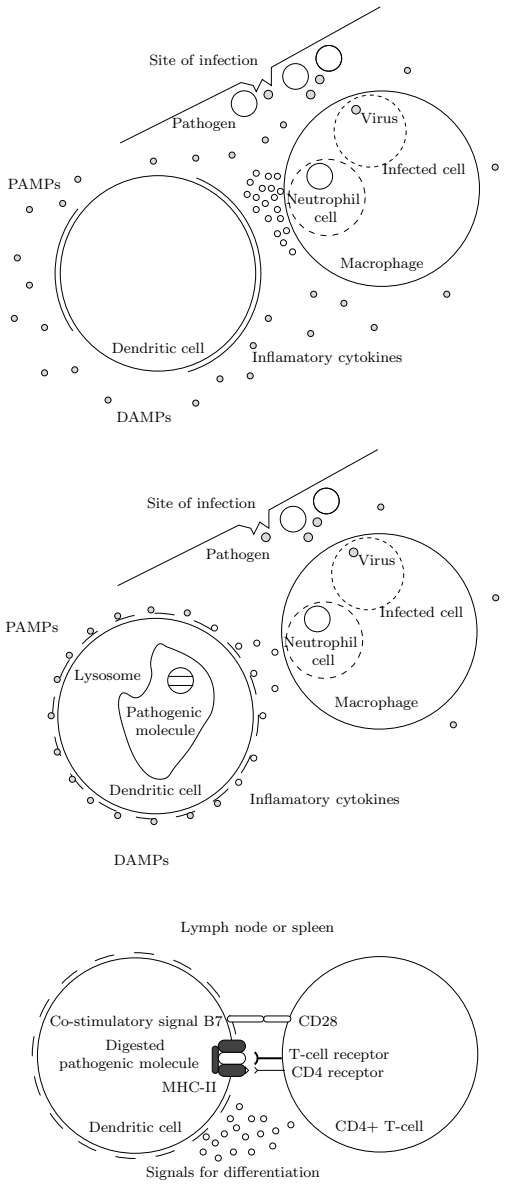



Figure 3.33: Pathogen - dendritic cell - CD4+ T-cell

a compound becomes activated and starts secreting cytotoxic molecules that kill the infected cell, see figure 3.32. The T-cell receptors of CD4+ T-cells bind with pathogenic molecule–MHC-II molecule compounds of antigen presenting cells, such as dendritic cells. Besides CD4+ T-cells require a co-stimulation signal, named B7, for being activated. For that signal the CD4+ T-cell presents a receptor named CD28, see figure 3.33. If no co-stimulation signal is present, the CD4+ T-cell becomes anergic. An anergic CD4+ T-cell is unable to mount an immune response. The lack of co-stimulation is given normally when a T-cell presents a T-cell receptor that matches a body molecule-MHC-II molecule compound. That mechanism allows to stop auto-reactive cells. When both, several hundreds of T-cell receptors in a CD4+ T-cell bind to pathogenic molecule-MHC-II molecule compounds and the receptors CD28 bind to B7, the CD4+ T-cell becomes activated and starts cell division and differentiation into helper CD4+ T-cells. Helper CD4+ T-cells are cells that help other cells in the immune system secreting ligands that stimulate that cells. Activated CD4+ T-cell differentiate into different kinds of helper CD4+ T-cells according to the kind of ligands that the dendritic cell secretes at the time of binding. The most known differentiated helper CD4+ T-cells are Th1, Th2, Tfh, Th17 and Treg and are produced due to the presence of the ligands IL-12, IL-4, IL-21, IL-23 and IL-10 respectively. Th1 cells help combating intracellular pathogen by stimulating macrophages to kill bacteria. Th2 cells help combating extracellular pathogen by binding B-cells activating them for synthesizing and secretion of antibodies. Tfh cells help B-cells to reproduce by cell division, undergo affinity maturation and differentiate into memory B-cells and plasma B-cells. Plasma B-cells are cells capable of synthesizing huge amounts of antibodies. They are found in follicles in the lymph node, reason why they got that name. Th17 cells help combating fungi and bacteria by attracting neutrophil cells to places like the skin and the gastrointestinal tract and producing inflammation. They also stimulate epithelial cells to release antimicrobial proteins in the skin or mucosal barriers against bacteria. Treg cells are cells that suppress the immune response. When activated they release a ligand named IL-10 that inhibits all other helper T-cells or kills antigen presenting cells. Some other regulatory T-cells like Tr1 and Tr3 are found in the intestine. There, they care for making the body tolerant to molecules ingested in the diet. Some of the produced cells by CD4+ T-cell differentiation become memory CD4+ T-cells. A memory CD4+ T-cell gets the same T-cell receptors as its parent CD4+ T-cell and has the ability for being activated with an smaller amount of pathogenic molecules binding its T-cell receptors. The intrusion of the same pathogen a time later will generate an earlier and effective immune response because of the presence of that memory CD4+ T-cells.

There are also $\alpha\beta$ T-cells which have on their surfaces activating and inhibiting receptors, like those found in natural killer cells. Therefore, these cells are named natural killer T-cells. They do not bind pathogenic molecule–histocompatibility molecule I or II compounds, instead they bind a pathogenic molecule–CD1 molecule compounds. MHC-II molecules form compounds with ingested pathogenic molecules with carbohydrate components while CD1 molecules form compounds with ingested pathogenic molecules with lipid components. Lipid molecules are hydrophobic. When a natural killer T-cell is activated, it secretes high amounts of

ligands that stimulate CD4+ T-cells to differentiate into Th1 and Th2, accelerating the adaptive immune response.

**B-cells** B-cells are immune cells that participate in the adaptive immune response. The surface of a B-cell is covered with thousands of identical copies of B-cell receptors which bind to an specific epitope or antigenic determinant of a pathogen.

B-cells are cells produced and maturated in the bone marrow, reason why they got their name. Over half of the produced B-cells have B-cell receptors able to bind with body molecules. Fact which would start a fight against the own body tissues. Therefore, the produced B-cells are confronted with body molecules in the bone marrow. When the B-cell receptor of a B-cell binds too strongly with a body molecule, the receptor undergoes a process of receptor editing. Receptor editing consists in transcribing again the pool of gene segments that encode the receptor specificity and produce a receptor with another binding site. When the B-cell receptor binds a body molecule again, the cell dies by apoptosis. Although some B-cells with B-cell receptors able to bind to body molecules still leave the bone marrow, they are stopped by further processes like the absence of helper CD4+ T-cells with matching T-cell receptors able to activate them.

When a pathogen enters the body, it encounters a pool of B-cells with B-cell receptors of different specificity. The binding site, or paratope, of the B-cell receptors of some B-cells will bind the antigenic determinants, or epitopes, of that pathogen with different affinities. Affinity is the strength of binding. Only the B-cells whose B-cell receptors bind with a determined strength will participate in the adaptive immune response, process named as selection. Selected B-cells phagocyte the pathogenic molecules by receptor endocytosis, digest them in lysosomes and form digested pathogenic molecule–MHC-II molecule compounds to be shown on their surfaces, please see figure 3.34. When the T-cell receptors of a helper CD4+ T-cell bind the digested pathogenic molecule–MHC-II molecule compound of a B-cell, it starts releasing ligands which activates the B-cell. An activated B-cell starts to reproduce by mitosis. Process named as clonal expansion.

By clonal expansion, two kinds of cells are produced, plasma B-cells and memory B-cells. A plasma B-cells is a cell able to synthesize and secrete B-cell receptors massively. Those free B-cell receptors, called also antibodies, help fighting against the pathogen by binding pathogenic molecules for further removal. A memory B-cell is a clone of the parent B-cell having identical receptors as those of its parent B-cell. Memory cells have a long life span, therefore an earlier and effective response to a second intrusion of the same pathogen into the body is possible. That process is named immunological memory and has been the basis for the development of vaccines. A vaccine is an small amount of a determined pathogen given to a person, which triggers an immune response that produces memory B-cells.

During clonal expansion B-cells are first cloned and afterwards they differentiate. During differentiation a group of memory B-cells get B-cell receptors with different specificity as those of their parent, [Allen et al., 1987]. The change of specificity in the B-cell receptors is given by mutation. Since mutation occurs in multiple points and it occurs in immune cells, not in reproductive cells of the body, the process is named somatic hyper-mutation. That process helps increasing the diversity of the B-cell receptor repertoire. Thus, it is possible an every-time-increased affinity of

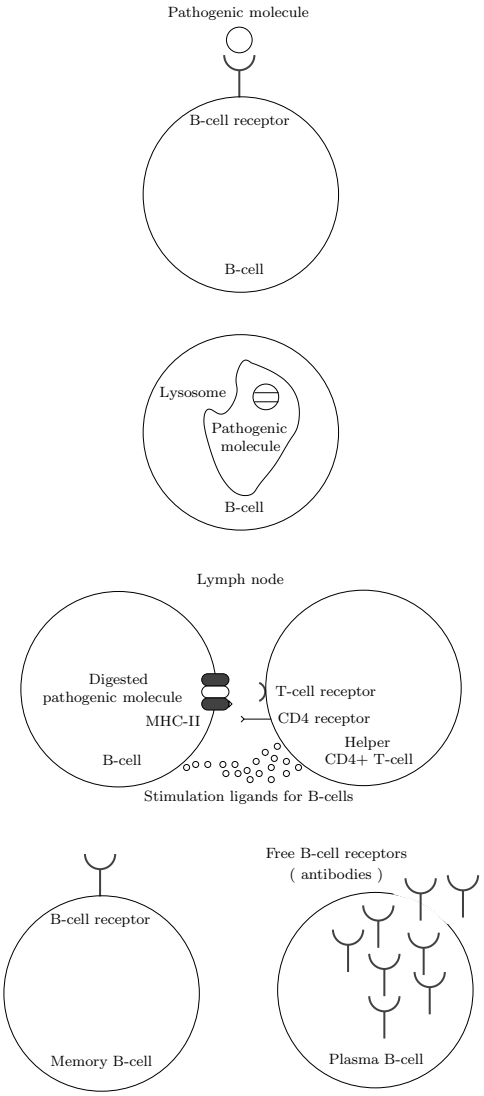



Figure 3.34: Pathogen - B-cell - helper CD4+ T-cell

the B-cell receptors of B-cells that meet the antigenic determinant again. That process is named affinity maturation and makes possible an evolution of the immune response in order to combat definitely and more effectively with a pathogen.

Affinity maturation implies producing B-cells with a different B-cell receptor. But why if that B-cell receptor binds too strongly to body molecules. B-cells with mutated receptors that show an improved affinity to the antigenic determinant are selected. All other B-cells which present B-cell receptors with unchanged or diminished affinity to the antigenic determinant are removed, [Tarlinton, 1998]. That process is said to happen in germinal centers placed in the cortex of the lymph nodes.

## 3.2 Artificial immune system models and algorithms

Artificial immune system models are a simplified version of the biological immune system models presented in literature about Immunology. The aim of this section is to show the application of such artificial immune system models for the development of artificial immune system algorithms which can be useful for solving computational problems, machine learning and pattern recognition. The main artificial immune models are: positive and negative selection, clonal selection, immune network, dendritic cells and the formal immune network. Next subsections present those artificial immune system models by its immunological background together with an algorithm. Algorithms can vary according to the application, however, the presented pseudo-codes for each algorithm intend to give a hands-on idea of how immunological theories can be used.

### 3.2.1 Positive and negative selection

Positive and negative selection are processes present during the maturation of B-cells and T-cells. B-cells are produced and maturated in the bone marrow. Since B-cells are produced with a wide repertoire of B-cell receptors, some B-cell receptors can bind body molecules tightly inducing the production of antibodies against the body. Therefore, B-cells are confronted in the bone marrow with body molecules. All B-cells whose B-cell receptors bind tightly to body molecules, rearrange their receptors or die by apoptosis. That process is named negative selection. The same process happens in the lymph nodes when through somatic hyper-mutation B-cells with mutated receptors are produced. There, auto-reactive T-cells are detected and induced to die by apoptosis. The maturation of T-cells is a little more complicated since the T-cell receptors are able to recognize compounds of antigen–histocompatibility molecule shown by antigen presenting cells. T-cells are produced in the bone marrow, but migrate to the thymus to maturate. There, they are first confronted with compounds of body molecule–histocompatibility molecules. T-cells able to bind to any of those compounds are positively selected. That process assures that only T-cells with functionally working T-cell receptors survive, all other die by apoptosis. Besides, T-cells are also confronted with compounds of body molecule–histocompatibility molecules, in order to eliminate those T-cells that are reactive to the body, same as the negative selection in B-cell maturation.

The negative selection process has been used for the development of an algorithm for the fist time in [Forrest et al., 1994]. The proposed algorithm has been applied for detecting changes in a computer caused by viruses, unauthorized use, etc. It considers known data in a

computer as the self and any deviation or change of the known data as the non-self. The goal is to have a mechanism that discriminates self from non-self. In order to generate a detector set of strings representing deviation or changes of the known data in a computer, negative selection is used. Given a random set of strings, the affinity of those strings in relation to strings representing the known data in the computer is calculated. If any string of the random set has a considerably affinity with any of the strings representing the normal state of the computer, the string is removed from the set, otherwise is conserved in the set. The remaining strings become the set of detectors of possible deviations in the data of a computer.

Although the negative selection process can be applied for generating a set of detectors when the string representing the self is provided, positive selection can be used when the strings representing the non-self are also provided. Therefore, both processes positive selection and negative selection, can be executed similarly to the maturation of T-cells in the thymus. Positive selection can be applied for getting a set of T-cells that bind correctly with antigens, in other words the non-self. The algorithm combining positive and negative selection is presented and explained below.

Algorithm 3.1 presents the pseudo-code of the positive and negative selection. Note that for the algorithm, instead of considering compounds of antigen–histocompatibility molecules and body molecule–histocompatibility molecule, only antigen and body molecule are written for reasons of simplicity. A T-cell can be interpreted as a number or a vector with a determined number of parameters. The algorithm presents two parts. The first part is the positive selection of T-cells and the second the negative selection of T-cells.

First a random set of immature T-cells with a specified number of parameters is generated, please see line 2 in the pseudo-code. The idea is, provided a set of antigens, compute for each random generated immature T-cell, the binding affinity to each antigen in the given antigen set, please see line 8 in the pseudo-code. The affinity can be computed as, for instance, the Euclidean distance between the antigen and the immature T-cell. The idea is, positively select the immature T-cell that binds, within a given threshold, with at least one of the antigens, see lines 9 to 16 of the pseudo-code. Since the same procedure is executed for each immature T-cell of the random set of immature T-cells, the number of bindings variable is reset in line 6 of the pseudo-code. The resulting set of semi-mature T-cells, is then the input for the negative selection part of the algorithm.

The negative selection part of the algorithm computes the binding affinity of each semi-mature T-cell with each body molecule of a given set of body molecules. When a T-cell binds to any of the body molecules, it is negatively selected from the set. All remaining T-cells constitute thereafter, the set of mature T-cells, please see lines 21 to 32 of the pseudo-code. Since the same procedure is executed for each semi-mature T-cell of the set of semi-mature T-cells, the number of bindings variable is reset in line 22 of the pseudo-code.

The positive and the negative selection processes can be executed a number of iterations times. Then, the mature T-cells set can be accumulated in each iteration as shown in line 36 of the pseudo-code. For that, the number of semi-mature and mature T-cells variables should be reset and the sets of semi-mature and mature T-cell sets emptied, see lines 3, 19, 4 and 20 in the pseudo-code.

### 3.2.2 Clonal selection

The clonal selection is a theory initiated in 1954 by the immunologist Niels Jerne, statement taken from [Wikipedia, 2010]. He stated that in the body a pool of immune cells with specific

**Algorithm 3.1:** Positive and negative selection

**Input**: Number of immature T-cells, set of antigens, set of body molecules, threshold for the binding affinity with the antigen, threshold for the binding affinity with the body molecule, number of iterations

**Output**: Set of mature T-cells

```
1:  foreach iteration do
2:      Generate randomly immature T-cells
3:      Reset the number of semi-mature T-cells variable
4:      Empty the set of semi-mature T-cells
5:      foreach immature T-cell do
6:          Reset the number of bindings variable
7:          foreach antigen do
8:              Compute the binding affinity between the immature T-cell and the antigen
9:              if binding affinity <= threshold for the binding affinity with the antigen then
10:                 Increment the number of bindings variable
11:             end
12:         end
13:         if number of bindings > 0 then
                % Positive selection
14:             Copy the immature T-cell to the set of semi-mature T-cells
15:             Increment the number of semi-mature T-cells variable
16:         end
17:     end
18:     if number of semi-mature T-cells ≠ 0 then
19:         Reset the number of semi-mature T-cells variable
20:         Empty the set of mature T-cells
21:         foreach semi-mature T-cell do
22:             Reset the number of bindings variable
23:             foreach body molecule do
24:                 Compute the binding affinity between the semi-mature T-cell and the body
                    molecule
25:                 if binding affinity <= threshold for the binding affinity with the body molecule
                    then
26:                     Increment the number of bindings variable
27:                 end
28:             end
29:             if number of bindings = 0 then
                    % Negative selection
30:                 Copy the semi-mature T-cell to the set of mature T-cells
31:                 Increment the number of semi-mature T-cells variable
32:             end
33:         end
34:     end
35:     if number of mature T-cells ≠ 0 then
36:         Accumulate the mature T-cells
37:     end
38: end
```

receptors is present before an antigen enters the body. First when the antigen enters the body, the immune cells which bind the antigen start to reproduce for combating the antigen. In 1958 Frank Macfarlane Burnet named the theory for the first time as the clonal selection theory and added the idea that the immune cells reproduce by cloning two types of cells, one type that combats with the pathogen and the other type with a longer lifespan. The clones cells with longer lifespan ensure a future immunity to the antigen. In the following years the theory has been enhanced with the idea that immune cells, in particular B-cells, differentiate into cells which are able to bind the antigen in the course of time with higher binding affinity. That effect has been attributed to the mutation of the binding sites of receptors of B-cell clones during differentiation. B-cells are cells of the adaptive immune system which take a big role in the development of an specific immune response against antigens. The clonal selection algorithm takes inspiration on the processes associated to the clonal selection theory for B-cells, explained in 3.1.5 and defined briefly below.

**Clonal selection** Only the B-cells whose B-cell receptors bind to the antigen with a determined strength start an immune response.

**Clonal expansion** Selected B-cells start to produce B-cell clones.

**Immunological memory** Some of the B-cell clones get a longer lifespan and are named memory B-cells.

**Somatic hypermutation** Some of the B-cell clones get mutated B-cell receptors.

**Affinity maturation** B-cell clones whose mutated receptors bind with higher strength to the antigen in reference to their parent B-cell survive, all other die. An iterated production of B-cell clones causes an ever increasing binding strength with the antigen.

The most known clonal selection algorithm is the CLONALG proposed by Leandro N. de Castro and Fernando Von Zuben and presented in [Castro and Timmis, 2002]. That algorithm sets the number of B-cell clones proportional to the binding strength of the B-cell receptors of the selected B-cell to the antigen and the mutation rate inverse proportional to the same binding strength, all that in the clonal selection and somatic hypermutation processes respectively.

The pseudo-code of the clonal selection algorithm is listed in Algorithm 3.2. That pseudo-code considers a B-cell having only one B-cell receptor, therefore for simplicity the pseudo-code is written in terms of B-cells only, not in terms of B-cell receptors. Besides, the binding strength is named binding affinity. The algorithm gives as output a set of memory B-cells. The set of memory B-cells is a set with B-cells that present the highest binding affinity found to a set of given antigens. There is the possibility to generate only one memory B-cell for every antigen or a group of memory B-cells for each antigen. There is also the possibility of having only one or a set of antigens in the given set. The implementation of the algorithm is application dependent. The application at the end determines which exactly is the output of the algorithm, the number representation of B-cells and antigens and the binding affinity function to use. This algorithm has been frequently used for learning patterns and optimizing functions. In the first case the antigens would represent the patterns. In the second case, the memory B-cells represent the arguments that minimize or maximize a given function.

In Algorithm 3.2, first of all, an initial random set of B-cells of a given size is created, please see line 1 in the pseudo-code. A B-cell represents the specificity of its B-cell receptor.

**Algorithm 3.2:** Clonal selection

**Input**: Size of the initial random B-cell set, set of antigens, number of cloning loops, cloning factor, mutation factor, number of worst B-cells for replacing
**Output**: Set of memory B-cells

```
1:  Create an initial random set of B-cells of the given size
2:  foreach antigen do
3:      foreach B-cell do
4:          Compute the binding affinity function between the antigen and the B-cell
5:      end
6:      Sort the set of B-cells by descending binding affinity
7:      Select the B-cell with highest binding affinity as a parent B-cell for cloning
8:      foreach cloning loop do
9:          Set the number of B-cell clones proportional to the binding affinity of the parent B-cell
            using a cloning factor if given
10:         Set the mutation rate inverse proportional to the binding affinity of the parent B-cell
            using a mutation factor if given
11:         foreach B-cell clone do
12:             Mutate the B-cell clone
13:             Compute the binding affinity function between the antigen and the B-cell clone
14:             if affinity of the B-cell clone > current highest affinity then
15:                 Select the B-cell clone as a parent B-cell for next cloning loop
16:                 Set the current highest binding affinity with the binding affinity of the B-cell
                    clone
17:             end
18:         end
19:     end
20:     if B-cell clone with better binding affinity exists then
21:         Replace the B-cell parent with the B-cell clone with better binding affinity
22:     end
23:     Copy the B-cell with best binding affinity after cloning in the set of memory B-cells
24:     Replace B-cells with the worst binding affinity to the antigen with random B-cells
25: end
```

Next, the binding affinity of one antigen with each B-cell is computed, see lines 2 and 4 in the pseudo-code. The binding affinity function could be implemented as the Euclidean distance between the antigen and the B-cell, hence, the higher the distance the lower the binding affinity. The B-cell with the highest binding affinity is selected as a parent B-cell, see line 7 in the pseudo-code. Note that instead selecting only the B-cell, a set of B-cell with the highest binding affinity can be selected as parent cells for producing B-cell clones. The biological counterpart of this procedure is the clonal selection.

The number of B-cell clones to produce is set proportional to the binding affinity. Because of the proportionality, a given cloning factor could be used, see line 9 in the pseudo-code. The mutation rate is inversely proportional to the binding affinity and a given mutation factor could also be used, see line 10 in the pseudo-code. Each one of the B-cell clones is mutated, see line 12. The binding affinity between the mutated B-cell clone and the antigen is computed, see 13 in the pseudo-code. If the binding affinity of the mutated B-cell clone to the antigen is higher in comparison to the binding affinity of its parent B-cell to the antigen, the mutated B-cell clone is selected as parent B-cell and the highest binding affinity value is updated, see lines 15 and 16 in the pseudo-code. In the next iteration, the next B-cell clone is mutated, its binding affinity to the antigen computed and if its binding affinity is higher than the current highest binding affinity, the mutated B-cell clone is selected as parent B-cell and its affinity becomes the new highest affinity. The same for the next B-cell clone, and so on. In the next cloning loop the mutated B-cell clone with highest affinity will be the one that produces new B-cell clones. That procedure is repeated a given number of cloning loops. The biological counterparts of that procedure are the clonal expansion, somatic hyper-mutation and affinity maturation processes. Note that a cloning loop is performed for the B-cell with the best binding affinity to the antigen and not for the whole population of B-cells. The algorithm does not match in this point its biological counterpart since it has been optimized. In the non optimized case, the cloning loop should be inserted between lines 2 and 3 of the pseudo-code.

When a better B-cell clone exist, it replaces the parent B-cell in the population, see line 21 in the pseudo-code, and a given number of B-cells with the lowest binding affinity to the antigen are replaced with random B-cells, see lines 20 to 24 in the pseudo-code. The new set of B-cells now will serve to compute the binding affinity to the next antigen. Lines 20 to 24 make sense when the antigens are correlated or similar.

The mutated B-cell clone with the best binding affinity to the antigen is copied into the memory B-cells set, see line 23 in the pseudo-code. The memory B-cell set besides being the solution set, it also ensures that a B-cell with best binding affinity is not removed in next iterations. The biological counterpart of that procedure is the immunological memory process.

The quality of the result, that is to say, B-cells with the best binding affinity to the antigen or set of antigens can be reached with a high number of cloning loops. Whenever the application provides only one antigen the whole algorithm could be executed a number of iterations. So an even better affinity can be reached.

### 3.2.3 Immune network

The immune network algorithm is based in the immune network theory proposed by Niels K. Jerne. A quotation of some of his important ideas published in [Jerne, 1974] are cited below:

> What is needed, I believe, is to incorporate clonal selection into a broader immune

> theory which accounts for the properties of the essential regulatory mechanism. It is a tentative approach to this task that I have proposed that clonal selection operates in the frame of a lymphocyte network. .... The immune system is a network of antibody molecules and lymphocytes that recognize and are recognized by other antibody molecules and lymphocytes. ... This network is not just a formal curiosity but it is a functional network and the properties of this network represent the essential regulatory mechanism of the immune system. ... Apart from the antigen-driven regulation, regulatory mechanisms that must discriminate between such lymphocytes, enhancing some and inhibiting others, can therefore act only (a) via the idiotopes (by means of combining sites of other molecules or receptors that recognize these idiotopes) or (b) via the combining sites (by means of idiotopes on other molecules or receptors that are recognized by these combining sites). ... Are cells influenced when the combining sites of their receptors recognize an idiotope?. Yes. ... Are cells influenced when the idiotypes of their receptors are recognized by a combining site?. Yes.

He states that the number of cells in the immune system not only increases with cloning and affinity maturation, but it is also regulated by a sort of immune network. Such network is not just the network of immune cells like B-cells or T-cells, but a network that integrates antibodies too. Therefore, the number of immune cells and the number of antibodies increase and decrease towards an stable state. In such network, not only programmed cell death and antigen are the causes of the decreasing of immune cells and antibodies, but also cells or antibodies that act as inhibitors of cell proliferation. That can be possible because of the presence of idiotopes and anti-idiotipic antibodies, please see figure 3.35 and 3.36

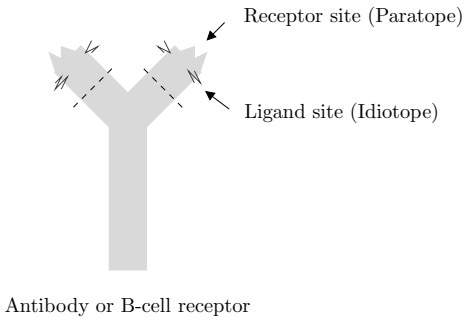



Figure 3.35: Idiotope

An idiotope is a binding site acting as a ligand in a B-cell receptor or antibody. That idiotope binds with an anti-idiotipic B-cell receptor or anti-idiotipic antibody. The question now is how that anti-idiotypic antibodies are produced. I have not found a clear answer but it is assumed that a separate antibody, that is able to bind with the idiotope of the antibody or B-cell receptor created against a determined antigen, is produced by some cell or maybe by the same antibody producing cell, taken from [Wikipedia, 2010]. Figure 3.37 taken from [Jerne, 1985], shows a B-cell receptor Ab2 whose paratope binds to the idiotope of an antibody Ab1. Consequently, the B-cell produces and secretes antibodies with such paratope, anti-idiotipic antibodies. We also see in figure 3.37, that the paratope of the antibody Ab1 binds to the idiotope of the B-cell receptor Ab2. In that case, the question is, can a B-cell be opsonized

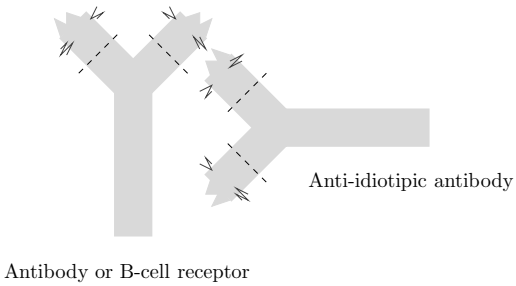



Figure 3.36: Anti-idiotipic antibody

by an antibody?. That idea sounds not much convincing, instead the B-cell can be inhibited to produce antibodies Ab2. In consequence, some cells can be stimulated by some other cells and inhibited by some other cells which are also stimulated by other cells and inhibited by other, giving the idea of a dynamic network.

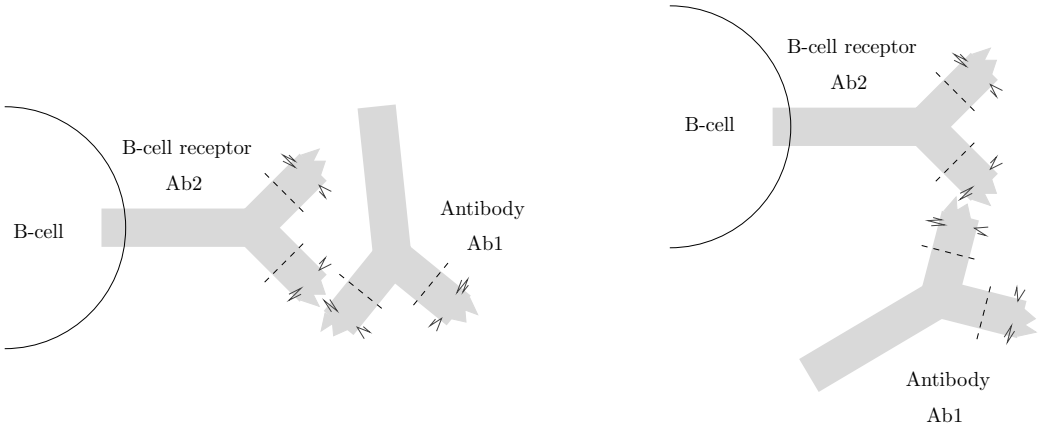



Figure 3.37: Idiotope-paratope interaction

Considering the immune system have approximately $R$ lymphocytes and it is sensitive to $m$ antigens, the repertoire of lymphocytes for a determined antigen is $\frac{R}{m}$. The repertoire of lymphocytes for an specific antigen presents $n$ sets of lymphocytes with different receptors recognizing that antigen with different degrees of precision or affinity. Each of those sets of lymphocytes with identical receptors is dynamic since the number of lymphocytes in the set changes with time. The dynamics of such set has been modeled quantitatively by Niels K. Jerne in [Jerne, 1974] in the formula 3.2.

$$\frac{dL}{dt} = \alpha - \beta L + L\sum_{1}^{n}\varphi(E_i, K_i, t) - L\sum_{1}^{n}\psi(I_i, K_i, t) \tag{3.2}$$

Where:

The first term $\alpha$ is the rate at which the lymphocytes enter into the set $L$ from other compartments of the immune system. Even though it is not clear what the author refers with "other compartments", I assume that it is referred to cells produced in the bone marrow.

The second term $\beta L$ is the rate at which the lymphocytes decay or leave the set $L$. I assume that lymphocyte decay can be originated by programmed cell death because of cell aging or insufficient nutrients as examples.

The third term is the summation of some function $\varphi$ of all excitatory signals of idiotopes of the sets $E$ that bind with association constants $K$ to paratopes of the receptors of cells in the set $L$. I assume that excitatory signals initiate cloning or the production of antibodies with identical paratopes, therefore the term is be positively added.

The fourth term is the summation of some function $\psi$ of all inhibitory signals of paratopes of the sets $I$ that bind to the idiotopes of the set $L$ with association constants $K$. I assume again that inhibitory signals represent the inhibition of cell cloning or antibody production, hence, the negative sign for that term.

Although the mathematical model in 3.2 has been constructed for a set of identical lymphocytes, some ideas for modeling the rate of change of the population of all lymphocytes in the body have been taken. So, Alan S. Perelson in [Perelson, 1989] presents the idea of reproduction of cells by cell stimulation and the death of cells by lack of stimulation as is to seen in the formula 3.3

$$\begin{aligned} \textit{Rate of lymphocyte population change} \quad = \quad & \textit{influx from bone marrow} \\ & + \textit{reproduction of stimulated cells} \\ & - \textit{death of unstimulated cells} \end{aligned} \tag{3.3}$$

Where the reproduction of stimulated cells is assumed to take place by means of cloning and the cells which receive no stimulation in a time frame die.

An immune network is a dynamic network of internal agents which also receive stimuli form external agents. Therefore there are two types of interactions: the interaction of internal agents with themselves inside the organism and the interactions of internal agents with external agents. The stimulation of internal agents (lymphocytes) produced by external agents (antigen) is not present in the mathematical model proposed by Niels K. Jerne in equation 3.2. However, it is assumed that cloning and affinity maturation by means of somatic hypermutation follows the interaction of the paratope of any B-cell receptor or antibody with the epitope of an antigen, as seen in figure 3.38, and that this event produces the increasing of the population of lymphocytes.

In summary, all those ideas have been taken for constructing immune network algorithms for computer programs. By the implementation the main goals have been to show clusters and inter-relationships among data items and to analyze the evolution and stability of such networks. There exist many algorithms following different interpretations of the given mathematical models. Equation 3.4 shows a collection of all possible factors for the increasing or the decreasing of a lymphocyte population. That equation helps, in the following paragraph, to explain better the differences among all existing algorithms.

$$\textit{Rate of lymphocyte population change} \quad = \quad N_{bm} - D_n + R_i - D_i + R_e - D_e - D_p \tag{3.4}$$

Where:

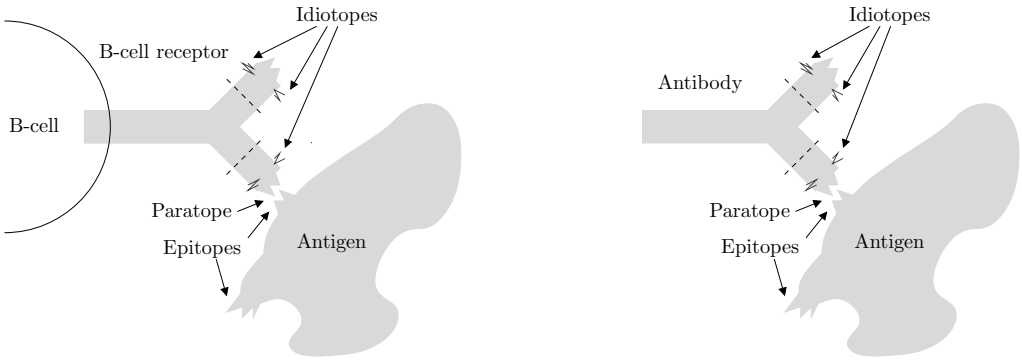



Figure 3.38: Epitope-paratope interaction

$N_{bm}$ is the influx of new lymphocytes from the bone marrow

$D_n$ is the out-flux of lymphocytes due to natural death like aging or nutrient deficiency

$R_i$ is the influx of new cells due to reproduction of cells stimulated by other cells, in other words internal stimulation

$D_i$ is the out-flux of cells due to the lack of stimulation of cells by other cells

$R_e$ is the influx of new cells due to reproduction of cells stimulated by antigens, in other words external stimulation

$D_e$ is the out-flux of cells due to the lack of stimulation of cells by antigens

$D_p$ is the outflux of cells due to death produced by antigens

The artificial immune network proposed in [Timmis et al., 2000] and [Timmis and Neal, 2001] considers the computation of the stimulation of a cell by other cells together with the stimulation of a cell by the presented antigen or group of antigens. First of all, cells with a low stimulation value are removed from the network. Cells which have an stimulation level higher that a given threshold are reproduced proportionally to their stimulation. Such clone cells are mutated with a determined mutation rate. A percentage of cells with weak stimulation are removed. The clone cells are inserted in the network and the whole procedure is executed iteratively again.

Differently, the artificial immune network named aiNet, presented in [Castro and Timmis, 2002], computes the stimulation of all cells by the presented antigen and then, like in the clonal selection algorithm, selects cells with higher stimulation for reproducing. Clone cells are mutated with a strength inversely proportional to the stimulation level. After cell reproduction, the stimulation of each cell by all other cells is computed. The stimulation is measured by means of affinity or similarity, one of the cells that presents a similarity with other cell higher than a given threshold is eliminated from the network. In this way, the outflux of cells helps to maintain the number of cells not out of bounds and within a given range. Besides, new random cells are added at the end of the iteration in order to insert diversity in the cell population.

As we could perceive in last examples, some factors for the increasing or the decreasing of a lymphocyte population, shown in equation 3.4 have been taken into account. Some factor

are missing and the main difference is the way of computation of the stimulation level of a cell for reproduction.

Algorithm 3.3 shows the pseudo-code of an immune network algorithm. The algorithm gives as output, in each iteration, the actual population of B-cells and the set of memory B-cells. The algorithm starts with a random initial set of B-cells of a given size, see line 1 in the pseudo-code. The binding affinity of an antigen with each of the B-cells in the network is computed, see lines 3 and 5 in the pseudo-code. The binding affinity function can be, for instance, the Euclidean distance between the antigen and the B-cell. However, it is to note that the lower the distance, the higher the binding affinity. B-cells with the lowest binding affinity to the antigen, much lower than a given stimulation threshold, are replaced with random B-cells, see line 8 in the pseudo-code. B-cells with binding affinity to the antigen higher than the given stimulation threshold, are selected for cloning and mutation of clones is executed, see lines 7 to 16 in the pseudo-code. For each B-cell clone the binding affinity with the antigen is also computed. A given number of B-cell clones with binding affinity higher than the given stimulation threshold is copied into the set of memory B-cells, see lines 17 to 24 in the pseudo-code. The B-cell clones copied to the B-cell memory set are also copied into the B-cell set, see line 25. For each B-cell in the set is computed the binding affinity with all other B-cells in the set. When any B-cell of the set stimulates the B-cell, with a binding affinity at least of suppression threshold, then the B-cell is removed from the B-cell set because of lack of stimulation, see lines 26 to 37 in the pseudo-code. A given number of B-cells are inserted into the B-cell set, see line 38. The same procedure is executed for all antigens, line 3 in the pseudo-code, during a given number of iterations, line 2 in the pseudo-code.

Algorithm 3.3 considers only B-cells since mutation of receptors for affinity maturation happens only by B-cells. Moreover, the influx of new B-cells in the network comes from the reproduction of B-cells stimulated by antigen and from new B-cells. The outflux of B-cells happen due to lack of stimulation of B-cells by other B-cells and by antigen. An extra B-cell memory set which cells are the ones which bind with all antigens the best has been created in order not to loose that B-cells in future iterations because of lack of stimulation by other antigens or by other B-cells.

In comparison with the clonal selection algorithm, the immune network algorithm aims in a way to control the variation of the number of cells in the population, maintaining in equilibrium the influx and the outflux of cells. That comes from the idea that the immune network is considered an homeostatic system, that is to say, a system that tends to maintain a stable condition. The clonal selection algorithm produces clones with the objective of finding clones with better binding affinity without caring much about the rate of variation of the population.

As conclusion, I can say, that immune network algorithms give a big range of possibilities since the interpretation of the biological principles can vary, some details of the immunological processes are still not well understood and the implementation into a program put some other restrictions. From my point of view, one question is still open, how an idiotope exactly stimulates or inhibits an immune cell after binding with the paratope of an antibody or another cell? Such question generates, at the end, diversity in the biological interpretations of an immune network and finally in these algorithms.

**Algorithm 3.3:** Immune network

**Input**: Initial size of the B-cell set, number of iterations, set of antigens, stimulation threshold, number of B-cells with lowest binding affinity for replacing, number of B-cell clones per antigen for the memory set, number of new B-cells for inserting, suppression threshold
**Output**: Set of memory B-cells, set of B-cells

```
1:  Create an initial random set of B-cells of the given size
2:  foreach iteration do
3:      foreach given antigen do
4:          foreach B-cell do
5:              Compute the binding affinity function between the antigen and the B-cell
6:          end
7:          Sort the set of B-cells by descending binding affinity
8:          Replace a given number of B-cells with the lowest binding affinity, much lower than the
            stimulation threshold, with random B-cells
9:          Select the B-cells with binding affinity higher than the given stimulation threshold as parent
            B-cells for cloning
10:         foreach parent B-cell do
11:             Determine the number of B-cell clones proportional to the binding affinity of the parent
                B-cell
12:             Set the mutation rate inverse proportional to the binding affinity of the parent B-cell
13:             foreach B-cell clone do
14:                 Mutate the parameters of the B-cell clone
15:             end
16:         end
17:         if set of B-cell clones exist then
18:             foreach B-cell clone do
19:                 Compute the binding affinity function between the antigen and the B-cell clone
20:             end
21:             Sort the set of B-cell clones by descending binding affinity
22:             Select a given number of B-cell clones with affinity higher than the stimulation threshold
23:             Copy the selected B-cell clones into the set of memory B-cells
24:         end
25:         Add the set of selected B-cell clones to the set of B-cells
26:         for i = 1 to number of B-cells−1 do
27:             for j = i + 1 to number of B-cells do
28:                 Compute the binding affinity of B-cell i and B-cell j
29:                 if binding affinity < suppression threshold then
30:                     Increase idiotope stimulation variable
31:                 end
32:             end
33:             if idiotype stimulation variable ≠ 0 then
34:                 Maintain B-cell i in the set
35:             end
36:             Eliminate B-cell i from the set
37:         end
38:         Insert random B-cells to the set
39:     end
40: end
```

### 3.2.4 Dendritic cells

Dendritic cells are present in the tissues that connect the body with the external world. Since external pathogens enter the body through that tissues, dendritic cells patrol the tissues collecting suspicious molecules and sampling a manifold set of other molecules in order to assess the situation around such suspicious molecules. Once the dendritic cell is prepared to give information about the state in the tissue, it migrates to the lymph node to communicate that information to more specialized cells in the immune system. The dendritic cells algorithm has been inspired in the behavior of the dendritic cells of the human body and largely presented by her author Julie Greensmith in her Doctor Thesis, please refer [Greensmith, 2007]. The following paragraphs present the pseudo-code of the algorithm, its biological counterpart and the most important details for getting it programed.

The algorithm can be divided into two parts:

- The collection of antigens and sampling of context signals followed by their processing by a set of immature dendritic cells, lines 1 to 23 of the algorithm 3.4
- The analysis of the sampled antigens and context final values delivered by migrated dendritic cells, lines 24 to 42 of the algorithm 3.4

Considering the biological counterpart, the first part takes place in a tissue of the body. There, immature dendritic cells digest antigens and receptors of that cells interact with ligands identified as the context signals. The digestion of antigens is translated in the algorithm as the collection of antigens and accumulation of them into a vector, line 6 of the pseudo-code. In each cycle an immature dendritic cell can collect only one antigen or a set of antigens. The collected antigens are exclusive for that immature dendritic cell. All other cells should look for the remaining antigens in the pool. For reason of simplicity, the algorithm considers an equal amount of antigens collected per cycle for all immature dendritic cells in the set. The vector of antigens in line 4 of the pseudo-code should have the size of the number of antigens collected per cycle times the size of the immature dendritic cell set.

Four categories of input context signals are considered: pathogen associated molecular patterns (PAMPs), danger signals, safe signals and inflammatory signals. PAMPs are molecules produced exclusively by pathogen which are recognized by receptors in the dendritic cell. Danger signals are the signals produced by cells which died by necrosis. Safe signals are the signals produced by cells which died by apoptosis. Inflammatory signals are ligands released by some cells in the site of infection in order to attract cells of the immune system for combating an infection. Dendritic cells have receptors for all those signals, therefore they are able to assess whether the situation in the tissue is dangerous or not. The danger theory has been proposed by the immunologist Polly Matzinger, who proposes that an immune response is initiated and regulated not only by the presence of the antigen but by the damage of the body detected by means of signals produced by injured or stressed cells. The input context signals in the application where this algorithm is to be applied can be manifold, however they should be grouped into the four listed categories. Such signal categorization is application dependent and requires expert knowledge. In the pseudo-code of the algorithm presented in 3.4, the number of input signal per category is one for the sake of simplicity. The algorithm also considers that in a cycle all dendritic cells get the same values of the input context signals, see line 3 in the pseudo-code. The last simulates that the context signals, also called ligands, are uniformly distributed to all cells and each cell posses enough receptors for such ligands.

**Algorithm 3.4:** Dendritic cells

**Input**: Streams of the input context signals, stream of antigens, number of cycles, number of antigens collected per cycle, size of the set of immature dendritic cells, migration threshold for each immature dendritic cell, weights matrix, antigen types, mature context antigen value (MCAV) threshold for each antigen type

**Output**: Dangerousness of each antigen type

Reset the number of migrated dendritic cells variable
**foreach** cycle **do**
  Update the values PAMPs(P), danger(D), safe(S) and inflammatory(I) signals in the vector of input context signals
  Update the vector of antigens
  **foreach** immature dendritic cell **do**
    Take a number of antigens collected per cycle from the vector of antigens and accumulate them into the vector of collected antigens of the immature dendritic cell
    **foreach** Output $p$: CSM, semi-mature, mature **do**
      $\text{Output}(p) = [\text{W}_\text{P}(p)\text{P} + \text{W}_\text{D}(p)\text{D} + \text{W}_\text{S}(p)\text{S}] \times (1 + \text{I})$
    **end**
    Accumulate the computed output signals into the vector of outputs of the immature dendritic cell
    **if** Output(CSM) > migration threshold of the immature dendritic cell **then**
      **if** Output(mature) > Output(semi-mature) **then**
        Set the immature dendritic cell context variable to 1
      **else**
        Set the immature dendritic cell context variable to 0
      **end**
      Copy the immature dendritic cell into the set of migrated dendritic cells
      Increment the number of migrated dendritic cells variable
      Remove the immature dendritic cell from the set
      Add a new immature dendritic cell to the set
    **end**
  **end**
**end**
Reset the vector of presentations of antigen types
Reset the vector of mature presentations of antigen types
**if** number of migrated dendritic cells $\neq 0$ **then**
  **foreach** migrated dendritic cell **do**
    Count the antigens per type in the vector of collected antigens
    Increment the vector of presentations of antigen types accordingly
    **if** migrated dendritic cell context variable = 1 **then**
      Increment the vector of mature presentations for each antigen type accordingly
    **end**
  **end**
**end**
**foreach** antigen type **do**
  Calculate the mature context antigen value MVAC using:

$$\text{MVAC(antigen type)} = \frac{\text{number of mature presentations(antigen type)}}{\text{number of presentations(antigen type)}}$$

  **if** MVAC(antigen type) > MVAC threshold(antigen type) **then**
    The antigen type is dangerous
  **else**
    The antigen type is not dangerous
  **end**
**end**

The ligand-receptor interactions initiate signal transduction that leads to different reactions, i.e. production of receptors, secretions of ligands, etc. The immature dendritic cell computes and accumulates, as the time passes, three output signals: the co-stimulatory output signal (CSM), the semi-mature output signal and the mature output signal, see line 8 to 10 in the pseudo-code. The processing of the input signals for obtaining the output signals is given by the formula 3.5.

$$\text{Output}(p) = [\text{W}_\text{P}(p) \sum_{k}^{K} \text{P}_k + \text{W}_\text{D}(p) \sum_{l}^{L} \text{D}_L + \text{W}_\text{S}(p) \sum_{m}^{M} \text{S}_m] \times (1 + \text{I}) \tag{3.5}$$

Where:

$P_k$ represents the $K$ PAMP signals

$D_l$ represents the $L$ danger signals

$P_m$ represents the $M$ safe signals

I represents the inflammatory signal

$\text{W}_\text{P}(p)$ represents the weight for PAMP signals for the output $p$

$\text{W}_\text{D}(p)$ represents the weight for danger signal for the output $p$

$\text{W}_\text{S}(p)$ represents the weight for safe signals for the output $p$

$p$ output signals can be computed as Output(CSM), Output(Semi-mature signal) and Output(mature signal)

The weights for the computation of the output signals should be provided beforehand. The weight assignment is application dependent and requires also expert knowledge. Nevertheless, relative weight values were derived empirically from immunological data [Greensmith, 2007] and give an idea how to define such values, please see table 3.1.

| | **Input context signals** | | |
|---|---|---|---|
| **Output signal** $p$ | **PAMPs** (P) | **danger signals** (D) | **safe signals** (S) |
| **CSM** | $\text{W}_\text{P}$(CSM) | $\frac{\text{W}_\text{P}(\text{CSM})}{2}$ | $1.5 \times \text{W}_\text{P}$(CSM) |
| **semi-mature** | 0 | 0 | 1 |
| **mature** | $\text{W}_\text{P}$(mature) | $\frac{\text{W}_\text{P}(\text{mature})}{2}$ | $-1.5 \times \text{W}_\text{P}$(mature) |

Table 3.1: Dendritic cells algorithm relative weights [Greensmith, 2007]

The co-stimulatory output is a ligand in the surface of the dendritic cell named in biological terms B7. That signal attaches to a receptor named CD28 in the specialized T-cell activating it after having presented the antigen within an MHC molecule. The signature of the antigen in the MHC molecule is not enough for activating that specialized T-cell for that specific antigen. A co-stimulatory signal is absolutely necessary otherwise the T-cell is unable to mount an immune response against that presented antigen. In the pseudo-code, when the

co-stimulatory output reaches a threshold given for that dendritic cell, the dendritic cell is ready to migrate to the lymph node, see line 11 in the pseudo-code. One could say that the co-stimulatory signal is powerful enough for activating a T-cell in the lymph node.

All dendritic cells in the population can be assigned different migration threshold values, so a kind of diversity in the dendritic cell population is assured. A rule for assigning a migration threshold is to give a value around the half of the maximum possible value of co-stimulatory output in one cycle given by the formula 3.6.

$$\text{Migration threshold} > 0.5 \times (\mathrm{W_P}(p)\mathrm{Max_P} + \mathrm{W_D}(p)\mathrm{Max_D} + \mathrm{W_S}(p)\mathrm{Max_S}) \times (1 + \mathrm{I}) \tag{3.6}$$

Where:

$p$ is CSM

$\mathrm{Max_P}$ maximum value of $P_k$ expected

$\mathrm{Max_D}$ maximum value of $D_l$ expected

$\mathrm{Max_S}$ maximum value of $S_m$ expected

The semi-mature and mature outputs are the ligands secreted by the dendritic cell, named IL-10 and IL-12 respectively in the biology terminology. IL-10 is a ligand that inhibits the synthesis of pro-inflammatory ligands regulating the immune response and making the body tolerant to some antigens, i.e. bacteria in the gastrointestinal tract. IL-12 is a ligand that promotes the differentiation of the T-cell in helper T-cells which are able to activate other immune cells. The dendritic cell have a context variable assigned in the pseudo-code. That variable inform whether the situation in the tissue is dangerous or not. When the accumulated mature output is higher than the accumulated semi-mature output, the dendritic cell becomes a context variable of 1, which means in the biological counterpart that the immune response against the antigen should be mounted, see lines 12 and 13 in the pseudo-code. A context variable of 0 means that the antigen should be tolerated, see line 15 in the pseudo-code. In reality the dendritic cell in the lymph node releases ligands that promote or inhibit the immune response according to the situation observed in the tissue.

A dendritic cell in a program can be implemented as a structure with the following fields: vector of collected antigen, input context signals vector, weights matrix, migration threshold and context value. For migrating the dendritic cell from the tissue to the lymph node, the dendritic cell structure is copied into another structure, see line 17 in the pseudo-code. An array of such structures forms the migrated dendritic set. Since an immature dendritic cell should be removed from the immature dendritic cell set, line 19 in the pseudo-code, and a new immature dendritic cell should be added to the immature dendritic cell set, line 20 in the pseudo-code, it is sufficient to set the immature dendritic cell with initial values and reuse that structure in the next cycle.

The second part of the algorithm takes place in the lymph node. Only dendritic cells that passed a maturation phase in the tissue are able to migrate to the lymph node. A variable counts the number of migrated dendritic cells, see lines 1 and 18 in the pseudo-code. In the lymph node, dendritic cells present the collected antigen to T-cells in order to mount an immune response. The collected antigens of all migrated dendritic cells are counted per type, see lines 24, 28 and 29 in the pseudo-code. The collected antigens of all mature migrated

dendritic cells are also counted per type, see lines 25 and 31 in the pseudo-code. The ratio between number of antigens sampled by mature migrated dendritic cells over the number of antigens sampled by all migrated dendritic cells is the mature context value which is calculated for all antigen types, see line 36 in the pseudo-code. When that ratio is compared to a defined threshold, the antigen type can be declared as dangerous or not dangerous, see lines 37 to 41 in the pseudo-code.

The fist part of the algorithm has as inputs an stream of input signals sampled at the frequency allowed by the time the input context signals processing allows. The second part of the pseudo-code can take place in parallel or off-line, maybe after a determined number of cycles passed away, see line 2 in the pseudo-code, in order to be able to asses the dangerousness of every antigen. Although, the assignation of the number of cycles after which the assessment of the antigens should be executed is application dependent. It is from benefit to maintain such number small enough in order to react timely against the antigen.

### 3.2.5 Formal immune network

A formal immune network is the formal mathematical representation of immune networks presented in [Tarakanov et al., 2003]. The notable issue of these networks is that they take a protein as the most elementary part for modeling. Thus, a formal protein is the abstract model of a B-cell receptor, an antigen, an antibody or a T-cell receptor, since all are just proteins. The formal model of a protein considers a three dimensional arrangement of its atoms. Because of the complexity of the model it is not presented here. The model of the formal protein is not absolutely necessary for understanding and implementing formal immune networks, since by applying formal immune networks for a determined application, a protein can be represented as a set of numbers. The set of number representing a protein can get different meanings depending on the application.

A formal immune network is a network of bindings among formal proteins. The BB-Network and the AB-Network are presented in [Tarakanov et al., 2003]. The BB-Network, is a network among B-cell receptors and the AB-Network a network of B-cell receptors with antigens. Since the strong binding of a B-cell receptor attached to a B-cell with an antigen or another B-cell receptor triggers cell reproduction, a B-cell has been also modeled as a quadruplet presented in 3.7.

$$B - cell = \langle P, Ip, Is, Im \rangle \tag{3.7}$$

Where:

$P$ represents the B-cell receptor

$Ip$ is the state indicator of the B-cell receptor from $n$ states $0, 1, 2, ..., n-1$

$Is$ is the cell state indicator

$Im$ is a mutation indicator

That model can be understood as the model of an agent which behavior is supported by the cell and mutation state indicators in the following way.

$Is = 0$ death, the B-cell is destroyed

$Is = 1$ recognition, the receptor $P$ can bind another protein

$Is = 2$ proliferation, the B-cell is converted into other to B-cell copies with $Is = 1$ and $Ip$ determined by $Im$

$Im = 0$ without mutation, inherit child receptor from cell parent

$Im = 1$ with mutation, change child receptor state

The binding energy is given by:

$$w(P(i), P(j)) = min(i - j)mod(n), (j - i)mod(n) \tag{3.8}$$

Given $n_h$ as the threshold of binding of the immune network, for a determined B-cell in the state $Is = 1$ compute the binding energy with their neighbors, if at least one $w <= n_h$ then the B-cell proliferates, otherwise it dies.

The BB-Network with B-cells with two neighbors is described by:

$$B_1, ..., B_{k-1}, B_k, B_{k+1}, ..., B_m \tag{3.9}$$

If cell $B_k$ proliferates in $B_{k1}$ and $B_{k2}$:

$$B_1, ..., B_{k-1}, B_{k1}, B_{k2}, B_{k+1}, ..., B_m \tag{3.10}$$

And if $B_k$ dies:

$$B_1, ..., B_{k-1}, \emptyset, B_{k+1}, ..., B_m \tag{3.11}$$

Empty spaces are filled shifting the B-cells accordingly.

In the case of an AB-Network, the antigens are placed above the B-cells, as it can be seen below:

$$A_1, ..., A_k B_1, ..., B_k, ..., B_m \tag{3.12}$$

A B-cell dies if there is not matching antigen or if it does not bind with any neighbor B-cell under. If the B-cell binds with a binding energy $w = n_h$, the B-cell proliferates without mutation. If the B-cell binds with a binding energy $w < n_h$, the B-cell proliferates with mutation.

The described BB-Network is a one dimensional B-cell network with parameters $n$ as the number of receptor types and $n_h$ the binding threshold. So it can be represented as $1D - BB(n, n_h)$. A BB-Network where its $B_k$ cells have four neighbors is considered a two dimensional B-cell network and represented as $2D - BB(n, n_h)$. A BB-Network where its $B_k$ cells have six neighbors is considered a three dimensional B-cell network and can be represented as $3D - BB(n, n_h)$. It is to imagine that an AB-Network could also exist as a two or three dimensional network, although not presented in [Tarakanov et al., 2003].

The BB-Networks can evolve to the following states: death of all B-cells, unlimited proliferation of B-cells or the cyclic reproduction of the population. Those states can be compared with the biological states of immune networks: immunodeficiency, allergy or immune memory, respectively. In the case of AB-Networks, a population of antigen of the same type leads that the whole population of B-cells in the end matches that antigen, that is to say, all B-cells result having the same receptor.

A formal model of a T-cell is also presented in [Tarakanov et al., 2003], however it is not really included in the constructed networks. In the same way as presented above, immune cells could be modeled and a dynamic formal immune network with different cells as agents could be constructed and simulated.

## 3.3 Comparison of artificial immune algorithms

Table 3.2 presents a summary of the biological analogy, inputs and outputs that each of the presented algorithms consider. You can also refer to the listed pseudo-codes, in last section, in order to verify those entries. Besides, the table contains a line about the main application of each algorithm. Please note that the algorithms have been and can be applied to a huge variety of application. However, applications which have been thoroughly implemented in the literature or are straight forward to implement are listed in the table.

Table 3.2 has no entries in inputs and outputs for formal immune networks since no algorithm has been presented in the respective section. The section referred to formal immune networks contains just the formal representation of the immune cells and the immune network. The implementation could take place considering the formal B-cells, formal T-cells or free formal proteins as agents inside a formal network. To show the implementation of such agent network has not been the aim, just to present how the immune cells and an immune network can be modeled in a formal way. The algorithm coined as cytokine formal immune network does not consider any of the formal representation of the B-cells, T-cells or free formal proteins, instead it has a mathematical background rather than a biological one. That algorithm can be applied for learning multidimensional patterns and for recognizing on-line those learned patterns. Cytokine formal immune networks is explained in chapter 4.

| Algorithm | Biological analogy | Input | Output | Application |
|---|---|---|---|---|
| Self/Non-self | T-cells maturation | body molecules/antigens | T-cell set | learning |
| Clonal selection | B-cells reproduction | antigens | B-cell memory set | optimization |
| Immune networks | B-cells reproduction, idiotopes | antigens | B-cell memory set | clustering |
| Dendritic cells | Dendritic cells | context signals/antigens | dangerousness of antigens | recognition |
| Formal immune network | B-cells reproduction, proteins | | | agents |

Table 3.2: Comparison of artificial immune algorithms

## 3.4 Bibliography

Allen, D., Cumano, A., Dildrop, R., Kocks, C., Rajewsky, K., Tajewsky, N., Roes, J., Sablitzky, F., and Siekevitz, M. (1987). Timing, Genetic Requirements and Functional Consequences of Somatic Hypermutation during B-Cell Development. *Immunological Reviews*, 96(1):5–22. Blackwell Publishing Ltd.

Castro, L. N. and Timmis, J. (2002). *Artificial Immune Systems. A new Computational Intelligence Approach*. Springer.

Forrest, S., Perelson, A. S., Allen, L., and Cherukuri, R. (1994). Self-Nonself Discrimination in a Computer. In *Symposium on Research in Security and Privacy*, pages 202–212. IEEE.

Greensmith, J. (2007). *The Dendritic Cell Algorithm*. PhD thesis, University of Nottingham.

Jerne, N. K. (1974). Clonal selection in a lymphocyte network. In Edelman, G. M., editor, *Cellular selection and regulation in the immune response*, pages 39–48. Raven Press, New York.

Jerne, N. K. (1985). The Generative Grammar of the Immune System, Nobel lecture, 8 December 1984. *Bioscience Reports*, 5(6):439–451. Springer.

Kimball, J. W. (1994). *Biology*. Addison-Wesley, 6 edition.

Perelson, A. S. (1989). Immune Network Theory. *Immunologival Reviews*, 110(1):5–36. Munksgaard.

Tarakanov, A. O., Skormin, V. A., and Sokolova, S. P. (2003). *Immunocomputing, Principles and Applications*. Springer.

Tarlinton, D. (1998). Germinal centers: form and function. *Current Opinion in Immunology*, 10(3):245–251. Elsevier.

Timmis, J. and Neal, M. (2001). A resource limited artificial immune system for data analysis. *Knowledge Based Systems*, 14(3-4):121–130. Elsevier.

Timmis, J., Neal, M., and Hunt, J. (2000). An artificial imune system for data analysis. *BioSystems*, 55(1-3):143–150. Elsevier.

Wikipedia (2010). Searched words: artificial intelligence, disease, cell, software agent, intelligent agent, apoptosis, necrosis, pathogen, virus, bacterion, fungus, lymphatic system, lymph, thymus, spleen, tonsils, leukocyte, protein, enzyme, allosteric regulation, cell signaling, communication, intracrine, autocrine signalling, juxtacrine signalling, paracrine signalling, idiotope, interferon, histamine, tumor necrosis factor, toxin, growth factor, hormone, cytokine, neurotransmitter, DAMPs, PAMPs, major histocompatibility complex, pattern recognition receptor, toll like receptor, signal transduction, gene, genetic, genetic code, dendritic cell, T-cell, B-cell, T helper 17 cell, lymph node, clonal selection, affinity maturation.

# 4

# Fault recognition

Fault recognition is the most important task in a self-repairing system. In the self-repairing architecture depicted in chapter 1, this task is performed by the fault recognition module. The fault recognition module has as inputs the input and output signals of the circuit for self repairing. With the observation of those signals in a determined point of time, the fault recognition module intends to find whether the behavior of the circuit is right or wrong. When a wrong behavior is detected, a repairing mechanism, represented as an integer number, is given to the recovery procedure module. The repairing mechanism value informs which recovery procedure has to be executed for repairing the circuit. This fault recognition method can be seen as a multiclass classifier. Of course, methods for the design of self-checking circuits can be used for determining first whether a system is faulty or not, but the method presented here helps for determining which repairing mechanism should be executed.

This chapter is devoted to explain issues related to the design of such a fault recognition module such as: the fault representation, the fault recognition procedure, the fault repairing mechanism assignation and the learning method for making the fault recognition module able to recognize faults and to assign repairing mechanisms. Fault recognition has been largely implemented by using linear transformation methods such as Principal Component Analysis or Singular Value Decomposition, techniques presented extensively in [Krishnan and Kerkhoff, 2012], [Mardia et al., 1979], [Theodoridis and Koutroumbas, 2008], [Theodoridis, 2009] and [Oja, 2003] among others. However, fault recognition can be implemented by using empirical methods such as artificial immune system algorithms as shown in [Amaral, 2011] and [Tarakanov et al., 2005]. The method presented in [Tarakanov et al., 2005], named cytokine Formal Immune Network, is a non-linear transformation which fuses ideas from the mathematical method Principal Component Analysis and the binding energy in proteins which is the basis of any immune network inside a biological entity. This chapter starts with the explanation of the Principal Component Analysis and the Singular Value Decomposition as a starting point for understanding the method cytokine Formal Immune Network. Those

methods are useful for designing a multiclass classifier able to determine a fault repairing mechanism under a determined permanent fault. That is why, they are presented in form of algorithms for a precise understanding, an then in the next chapter they are compared with experimental data in order to demonstrate its usefulness and performance in terms of percentage of recognized faults.F

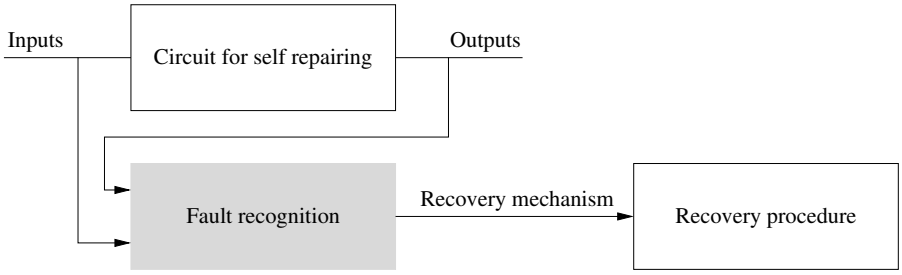



Figure 4.1: Inputs and outputs of the fault recognition module

## 4.1 Fault representation

The fault recognition module recognizes faults which are patterns that have a determined representation. If we consider a fault being defined as a vector $[inputs|outputs]$, formed with input values to the circuit for self repairing and corresponding values of the output signals expressing a wrong behavior of the circuit for self repairing, that fault can be seen as a pattern representing a point in a multidimensional space. Generalizing, a fault pattern can be seen as a vector $X = [x_1, x_2, ....., x_p, x_{p+1}, x_{p+2}, ...., x_{p+q}]$ belonging to the multidimensional space of dimension $p+q$ named $\{X\}$, where $p$ is the number of inputs and $q$ is the number of outputs to and from the circuit for self repairing respectively. Such a multidimensional vector is difficult to represent graphically because of the number of dimensions. Nevertheless, considering a circuit for self repairing has one input and one output, a fault pattern vector $X = [x_1, x_2]$ can be represented graphically as a point in a two dimensional space $\{X\}$ as shown in figure 4.2.

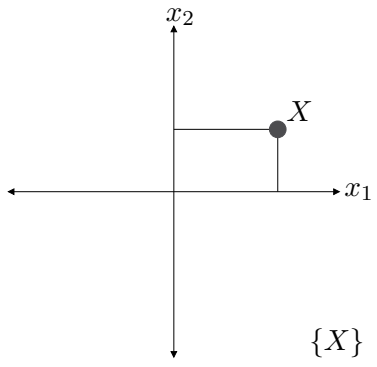



Figure 4.2: Fault vector $X = [x_1, x_2]$ represented in the two dimensional space $\{X\}$

Furthermore, the representation of a fault is dependent on how the inputs and outputs of the circuit for self repairing are. Figures 4.3 and 4.4 show the possibilities of inputs and outputs of the circuit for self repairing. We can be confronted with digital or analog input

and output signals. In the case of digital signals, a signal can have a single line or multiple lines. In the case the circuit has single-line digital inputs and outputs, we can be confronted with a combinational or a sequential circuit for self repairing, see figure 4.4 a) and b). In the case of a circuit for self repairing with multiple-line input and output signals, multiple-line signals can have independent lines, which is not very common because of the huge quantity of lines that this implicates, see figure 4.4 c). It is more often to have values stored in buffers which are sent to other modules through a serial or parallel bus, see figure 4.4 d). Those values can have integer, fixed point or floating point format. In the case of analog signals, each signal should be converted to a digital signal through a sampler and an analog to digital converter, see figure 4.4 e). After the conversion, values can be expressed in an integer, fixed point or floating point format which can be stored in buffers and be sent serially or in parallel to other modules.

Thus, in practice, in the case of digital single-line inputs and outputs of the circuit for self repairing, a fault pattern looks like this example $[0;1;1;1;1;1;0|1;1;0]$. When digital multiple-line signals or analog signals converted to digital is the case, a fault pattern looks like this other example $[1,34;8,98;0,45;1,09|0,02;3,78;8,46]$. In brief, values in a fault pattern vector $X$, representing inputs and outputs of the self repairing circuit, could be binary 0 and 1, integer numbers or decimal numbers expressed in fixed or floating point format.

Looking at the output signal values in a fault pattern, they can contain output values that represent an explicit fault, $[inputs|incorrect\ outputs]$, or can contain correct output values, $[inputs|correct\ outputs]$. In the case of output values that represent an explicit fault, at the moment of comparing the fault pattern of an explicit fault with an observed output, that explicit fault can be recognized uniquely or a similar fault to that explicit fault can also be detected using a similarity threshold. But, in the case of correct output values, at the moment of comparing the fault pattern of correct outputs with an observed output, a deviation can help to detect a non explicitly defined fault. Examples of fault pattern vectors containing explicit and non explicit faults, are given below.

Given a combinational circuit with three inputs and one output implementing the function $Z = A \cdot B + C$, examples of fault pattern vectors are:

- → $[1,1,1|0]$ represents an explicit fault since the output 0 under inputs 111 is wrong. Then that fault or a similar one can be detected with the help of that stored fault pattern vector.

- → $[1,0,0|0]$ where output 0 is the correct output under inputs 100. Then a deviation of the correct output expresses a fault when that stored fault pattern vector is available.

These two forms of conceiving a fault pattern vector, explicitly and non explicitly, can be combined into a set of fault pattern vectors, but a mark should be given. Fault pattern vectors with correct outputs can be marked as *self* and fault pattern vectors with outputs containing an explicit fault can be marked as *non self*. This mark can be attached to the fault pattern vector $X$ in the following form $[self|inputs|outputs]$ or $[non\ self|inputs|outputs]$.

Having sequential circuits for self repairing, taking care of its internal states is necessary. Sequential circuit outputs depend not only on its inputs but also on its internal states so that any fault pattern vector $X$ should be expressed in the form $[inputs|present\ states|next\ states|outputs]$. But, having the present and next states available outside the circuit, requires the circuit for self repairing to be observable and controllable. A sequential circuit can be made observable and controllable by means of design for test techniques such as scan design,

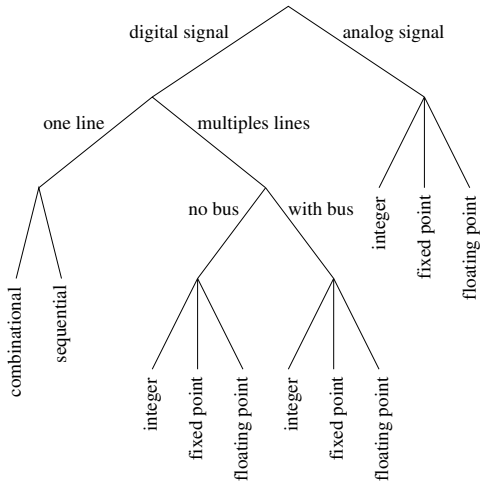



Figure 4.3: Possibilities of inputs and outputs of the circuit for self repairing in a tree representation

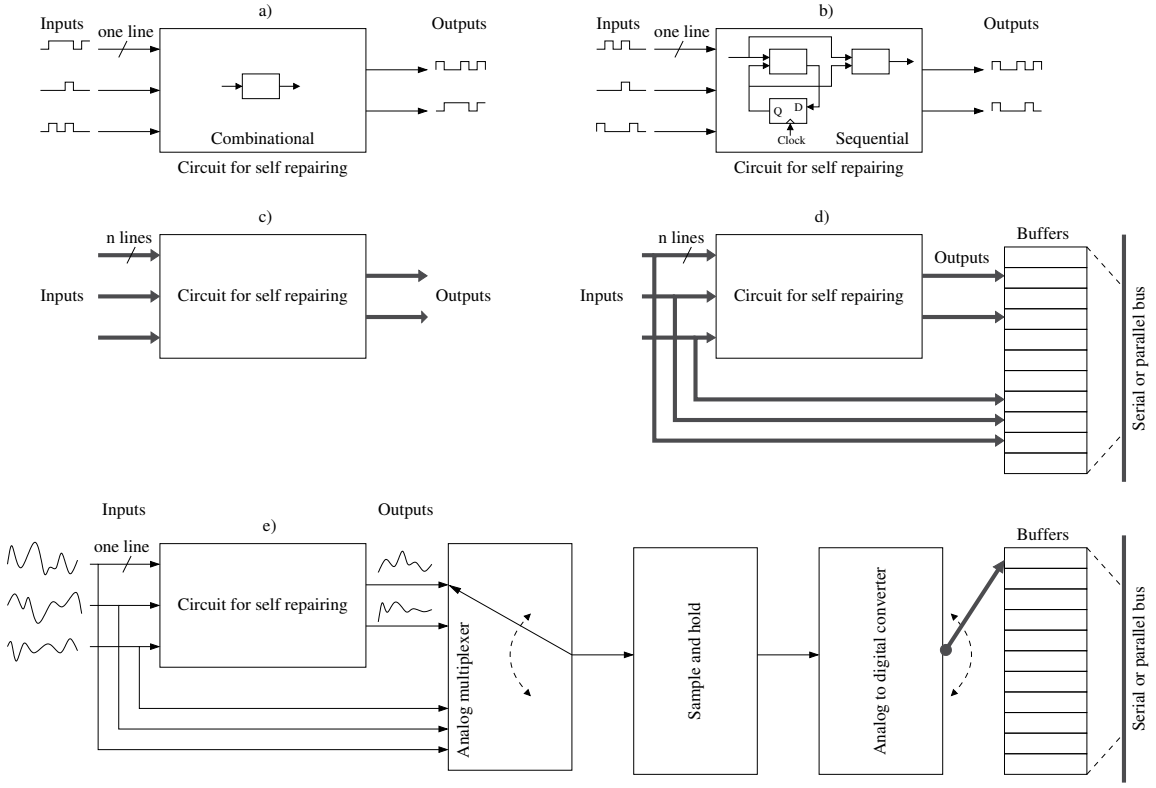



Figure 4.4: Possibilities of inputs and outputs of the circuit for self repairing
a) digital combinational circuit with one line inputs and outputs
b) digital sequential circuit with one line inputs and outputs
c) independent multiple-line inputs and outputs
d) multiple-line inputs and outputs sent to other modules through a bus
e) analog inputs and outputs digitalized by an analog to digital converter and sent to other modules through a bus

see figure 4.5. Scan design uses available registers configured into a serial shift-register chain. That serial shift register chain is used to give from outside desired present state signals to the circuit and receive next state signals back. That is generally done serially through extra signals *serial data in*, *serial data out* and a *test signal*. The test signal controls multiplexers added at the inputs of the registers in order to let either normal signals or desired test signals pass through. However, it should be noted that scan design implies stopping the circuit and entering in a test mode for passing the desired known present state values together with the inputs to the circuit for self repairing.

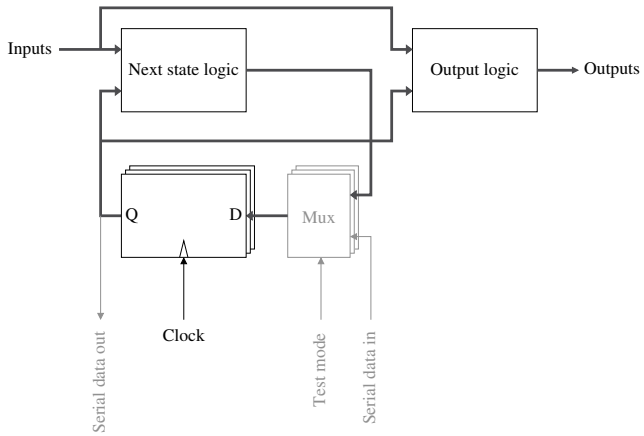



Figure 4.5: Scan design for making sequential circuits observable and controllable

## 4.2 Fault recognition

Fault pattern recognition implies finding out that a *given fault vector* $X_g$ represents a fault. A given fault vector $X_g$ represents a fault when it has determined typical fault features or is equal or at least similar to some other *known stored fault pattern vector* $X_p$. To find out that a given fault vector $X_g$ has determined typical fault features, requires first to perform feature extraction and afterwards compare the extracted features with stored typical fault features. Feature extraction is defined and explained in section 4.6. To find out a given fault vector $X_g$ is equal to another stored known fault pattern vector $X_p$, requires comparing each fault vector component value $x_{g_i}$ and $x_{p_i}$ for equality. That procedure is named pattern matching. But, when similarity of the given fault vector $X_a$ to a fault pattern vector $X_p$ is searched for, a way to measure such similarity is required.

The similarity between two vectors is inversely proportional to the distance they have, see figure 4.6. Therefore, it is possible by means of the measurement of the distance between two vectors to determine whether a given fault vector $X_g$ is similar enough to a fault pattern vector $X_p$ or not. There are distinctive ways of measuring the distance. Some distance measurement methods taken from [Wikipedia, 2010] and [Theodoridis and Koutroumbas, 2008] are defined below.

Given a fault vector $X_g = [x_{g_1}, ..., x_{g_i}, ..., x_{g_n}]$ and a fault pattern vector $X_p = [x_{p_1}, ..., x_{p_i}, ..., x_{p_n}]$, the distance can be defined as:

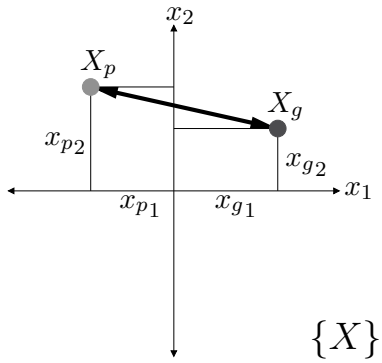



Figure 4.6: Determination of the similarity of a given fault vector $X_g = [x_{g_1}, x_{g_2}]$ and a fault pattern vector $X_p = [x_{p_1}, x_{p_2}]$ through the measurement of its distance in the two dimensional space $\{X\}$

Minkowski distance:

$$distance(X_g, X_p) = (\sum_{i=0}^{n} |x_{g_i} - x_{p_i}|^q)^{\frac{1}{q}} = (|x_{g_1} - x_{p_1}|^q + ... + |x_{g_i} - x_{p_i}|^q + ... + |x_{g_n} - x_{p_n}|^q)^{\frac{1}{q}} \tag{4.1}$$

Euclidean distance: (Minkowski distance for $q = 2$)

$$distance(X_g, X_p) = (\sum_{i=0}^{n} (x_{g_i} - x_{p_i})^2)^{\frac{1}{2}} = ((x_{g_1} - x_{p_1})^2 + .. + (x_{g_i} - x_{p_i})^2 + .. + (x_{g_n} - x_{p_n})^2)^{\frac{1}{2}} \tag{4.2}$$

Manhattan distance: (Minkowski distance for $q = 1$)

$$distance(X_g, X_p) = \sum_{i=0}^{n} |x_{g_i} - x_{p_i}| = |x_{g_1} - x_{p_1}| + ... + |x_{g_i} - x_{p_i}| + ... + |x_{g_n} - x_{p_n}| \tag{4.3}$$

Chebyshev distance: (Minkowski distance for $q = \infty$)

$$\begin{aligned} distance(X_g, X_p) &= \lim_{q \to \infty} (\sum_{i=0}^{n} |x_{g_i} - x_{p_i}|^q)^{\frac{1}{q}} = \max_{i=1}^{n} |x_{g_i} - x_{p_i}| \\ &= max(|x_{g_1} - x_{p_1}|, ..., |x_{g_i} - x_{p_i}|, ..., |x_{g_n} - x_{p_n}|) \end{aligned} \tag{4.4}$$

Hamming distance:

$$distance(X_g, X_p) = \sum_{i=0}^{n} (x_{g_i} \oplus x_{p_i}) \tag{4.5}$$

The Minkowski distance is suitable for fault vectors which components are integer or decimal numbers expressed in fixed or floating point format. And the Hamming distance is suitable for binary fault vector components, case of digital one line inputs and outputs of the circuit for self repairing. The Hamming distance of two fault vectors is the number of positions at

which the corresponding coding symbols, "1" or "0" are different. i.e. given $X_g = [11100101]$ and $X_p = [11110100]$, the Hamming distance is 2, because the fault vector components in the 4th and 8th positions are different.

Once the similarity of a given fault vector $X_g$ with every other known stored fault pattern vector $X_p$ has been measured, the fault pattern vector $X_p$ which has the smallest distance to the given fault vector $X_g$ is declared the most similar.

Furthermore, if similarity of the given fault vector $X_g$ is searched for, not to a single fault pattern vector but to a *set c of fault pattern vectors* $\{X\}_c$, that similarity can be measured computing the Mahalanobis distance.

Given a fault vector $X_g = [x_{g_1}, ..., x_{g_i}, ..., x_{g_n}]^T$ and a set of fault pattern vectors $\{X\}_c$ with mean $\mu_c = (\mu_{c1}, ..., \mu_{ci}, ..., \mu_{cn})^T$ and covariance matrix $Cov_c$, the Mahalanobis distance of the given fault vector $X_g$ to the set of fault pattern vectors $\{X\}_c$ is defined as:

$$distance(X_g, \mu_c) = \sqrt{(X_g - \mu_c)^T Cov_c^{-1} (X_g - \mu_c)} \tag{4.6}$$

If the covariance matrix is the identity matrix, Mahalanobis distance reduces to the Euclidean distance of the given fault vector $X_g$ to the set of fault pattern vectors $\{X\}_c$ in the following way:

$$distance(X_g, \mu_c) = \sqrt{(X_g - \mu_c)^T (X_g - \mu_c)} \tag{4.7}$$

In some special cases the covariance matrix of the set of fault pattern vectors $\{X\}_c$ can be a scalar multiple of the identity matrix $Cov_c = a_c I$ or a diagonal matrix $Cov_c = diag(\sigma_{c1}^2, ..., \sigma_{ci}^2, ..., \sigma_{cn}^2)$. Where $\sigma_{ci}$ is the standard deviation and $\sigma_{c1}^2$ is the variance of the vector component $x_i$ within the fault patten vector set $\{X\}_c$.

Mahalanobis distance can also be applied for measuring the similarity between the given fault vector $X_g$ and a fault pattern vector $X_{cp}$ that belongs to a set of fault pattern vectors $\{X\}_c$ with covariance $Cov_c$.

$$distance(X_g, X_{cp}) = \sqrt{(X_g - X_{cp})^T Cov_c^{-1} (X_g - X_{cp})} \tag{4.8}$$

Again, if the covariance matrix is the identity matrix, the Mahalanobis distance between the given fault vector $X_g$ and a fault pattern vector $X_{cp}$ reduces to the Euclidean distance:

$$distance(X_g, X_{cp}) = \sqrt{(X_g - X_{cp})^T (X_g - X_{cp})} = \sqrt{\sum_{i=0}^{n} (x_{g_i} - x_{cp_i})^2}$$

And finally, if the covariance matrix is the diagonal matrix $Cov_c = diag(\sigma_{c1}^2, ..., \sigma_{ci}^2, ..., \sigma_{cn}^2)$, the distance reduces to the normalized Euclidean distance:

$$distance(X_g, X_{cp}) = \sqrt{\sum_{i=0}^{n} \frac{(x_{g_i} - x_{cp_i})^2}{\sigma_{cp_i}^2}} \tag{4.9}$$

## 4.3 Fault repairing mechanisms assignation

We have seen above that a given fault vector $X_g$ can be encountered to represent a fault finding out equality or similarity of its components or extracted features to a fault pattern

vector $X_p$ or a set of fault pattern vectors $\{X\}_c$. Once a given fault vector $X_g$ has been encountered to represent a fault, it is expected the assignation of a fault name or fault class to the given fault vector $X_g$. If each stored fault pattern vector $X_p$ or each set of fault pattern vectors $\{X\}_c$ has a repairing mechanism assigned then, instead of giving a fault name or a fault class, a repairing mechanism can be assigned to the given fault vector $X_g$.

Whenever equality of the given fault vector $X_g$ and a stored fault pattern vector $X_p$ has been identified during fault pattern recognition, the repairing mechanism that the fault pattern vector $X_p$ has, will be assigned to the given fault vector $X_g$ when it is encountered that it represents a fault.

When similarity between the given fault vector $X_g$ and a fault pattern vector $X_p$ or set of fault pattern vectors $\{X\}_c$ has been identified by fault pattern recognition then, the repairing mechanism of the closest fault pattern vector $X_p$ or set of fault pattern vectors $\{X\}_c$ is assigned to the given fault vector $X_g$. The closest fault pattern vector $X_p$ to the given fault vector $X_g$ is the one that has the smallest distance measured with any of the Minkowski distance equations given in last section 4.2. The closest set of fault pattern vectors $\{X\}_c$ to the given fault vector $X_g$ is that one with the smallest distance measured with the Mahalanobis distance equation given also in last section 4.2.

A method which uses the computed distances between the given fault vector $X_g$ and each of the vectors $X_{cp}$ of sets of fault pattern vectors $\{X\}_c$ measured with any of the Minkowski distance equations is the k-nearest neighbor procedure, please see [Theodoridis, 2009] and [Wikipedia, 2010]. The k-nearest neighbor procedure uses the $k$ closest fault pattern vectors $X_{cp}$ to the given fault vector $X_g$. The repairing mechanisms of the given fault vector $X_g$ can be determined by the majority vote of its neighbor fault pattern vectors $X_{cp}$. The most common repairing mechanism among its k-nearest neighbors $X_{cp}$ is assigned to the given fault vector $X_g$. The number $k$ is a positive integer, generally small. If $k = 1$, then the repairing mechanism of the nearest neighbor $X_{cp}$ is assigned to the given fault vector $X_g$. This method eliminates the need of computing the mean $\mu_c$ and the covariance matrix $\Sigma_c$ of each set $\{X\}_c$.

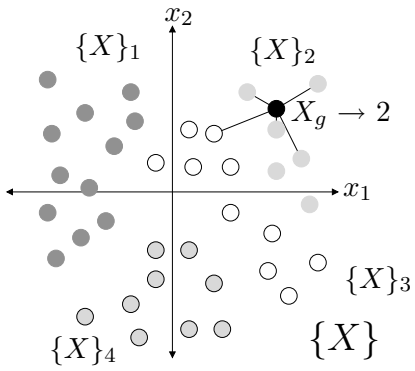



Figure 4.7: k-nearest neighbor procedure with $k = 5$ assigns $X_g$ to the subset $\{X\}_2$

Figure 4.7 shows graphically the procedure followed by the k-nearest neighbor algorithm in a two dimensional $\{X\}$ space for $k = 5$. In this example, the repairing mechanism number coincide with the subset number. Therefore, the assigned repairing mechanism for $X_g$ is 2.

Even though the repairing mechanism is not expressed as an integer number but as a real decimal, the average of the values of the k-nearest neighbors can be assigned to the given fault vector. However this assumption is not applicable to our problem, since we defined repairing

mechanisms to be represented as integer numbers.

## 4.4 Fault space partitioning

If we assign a *repairing mechanism* to each fault pattern vector $X$, then, the space $\{X\}$ can be partitioned into subsets $\{X\}_c$, where $c = 1, 2, ...., l$ are the integer numbers allotted to the repairing mechanisms and $l$ the total number of repairing mechanisms. Figure 4.8 shows such partitioning in a $\{X\}$ space of two dimensions. The process of partitioning the space $\{X\}$ in subsets is named learning [Tarakanov et al., 2003]. Provided that the repairing mechanisms are given beforehand, then the learning process is named *supervised learning*, since the space subsets are already defined. If the repairing mechanisms are unknown, the partitioning of the space can be done by means of *unsupervised learning* i.e. clustering, for more information please see [Theodoridis and Koutroumbas, 2008].

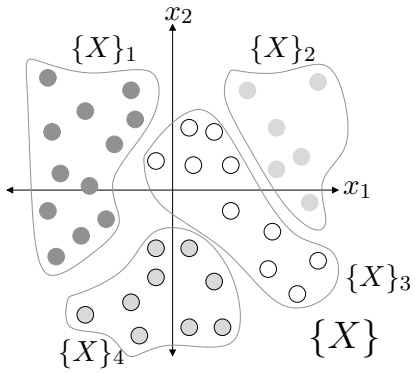



Figure 4.8: Partitioning of the space $\{X\}$ in subset spaces $\{X\}_c$

To execute the fault space partitioning or the already defined learning procedure, fault pattern vectors should be provided. It has to be noted that fault pattern vectors of all possible $[self|inputs|outputs]$ combinations, in reality are hard to obtain and even if available, it is too much data for being processed. Therefore, the fault recognition module should try to generalize from the given incomplete amount of fault patterns vectors in order to determine the repairing mechanism of a given fault pattern correctly.

When the repairing mechanisms are known, the fault pattern vectors for learning have the form $[repairing\ mechanism|self|inputs|outputs]$. In the case the fault pattern vectors are given only in the form $[self|inputs|outputs]$, they should be classified by unifying similar vectors and separating distinctive vectors and further assigning a repairing mechanism to each of the resulting clusters. By means of the measurement of the distance between two fault vectors, defined in section 4.2, it is possible to determine whether fault pattern vectors are similar enough for being unified into a cluster or separated. It would also be possible to be provided with both forms of fault pattern vectors ($[repairing\ mechanism|self|inputs|outputs]$ and $[self|inputs|outputs]$) for which supervised learning algorithms and unsupervised learning algorithms like clustering can be executed to extend the already given repairing mechanisms set.

Once the space $\{X\}$ has been partitioned into subsets $\{X\}_c$, were each subset is assigned to a determined repairing mechanism, the recognition of a given fault vector $X_g$ is the determination of a *repairing mechanism* $c$, such as $X_g \subset \{X\}_c$.

The fault pattern vectors provided for learning, where each vector is allotted a determined repairing mechanism, is stored for later use in the recognition process within the fault recognition module memory.

## 4.5 Fault recognition time

The fault pattern recognition time is dependent on the size and number of fault pattern vectors stored within the fault recognition module memory. Each time a given fault vector $X_g$ is presented for determining whether it represents a fault or not, distances of that vector with each fault pattern vector stored in memory are computed. After a given fault vector $X_g$ is declared to represent a fault, a repairing mechanism is assigned. Figure 4.9 shows the total time needed before a repairing mechanism can be executed. In order that under a fault, repairing takes place on time, small fault recognition times should be assured.

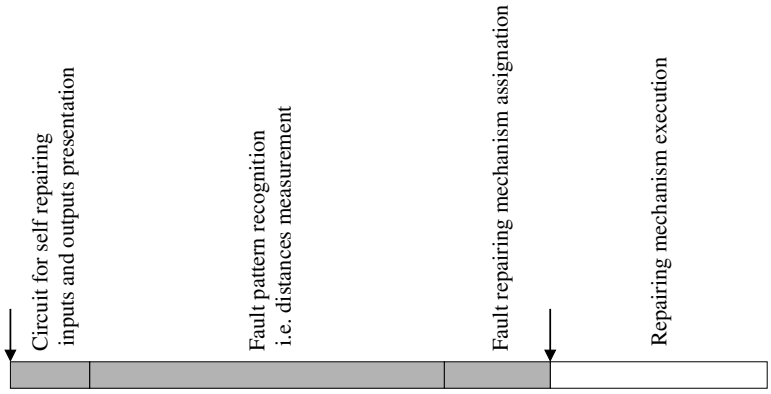



Figure 4.9: Total time required before a repairing mechanism can be executed

Assume that each data element in memory has the format [*repairing mechanism*|*self*| *inputs*|*outputs*]. A determined and limited number of such elements can be stored in the available memory. But, as stated above, a large amount of such data elements increases the recognition time because of the number of required computations. For this reason, a strategy to obtain a compact data set of fault pattern vectors with a high recognition capability is searched for. First, it is intended to reduce the size of each data element. The number of *inputs* plus *outputs* gives the dimension of the data, i.e. fault vector dimension reduction techniques are needed. Second, it is intended to store only data elements with a high recognition capability, consequently methods for fault pattern vector number reduction are also required.

While fault vector dimension reduction can be solved by mathematical techniques and fault pattern vector number reduction can be performed by conventional algorithms, among others algorithms inspired in principles of the biological immune system, a cytokine Formal Immune Network, an algorithm from Immunocomputing, get both, the fault vector dimension and the fault pattern vectors number, reduced. Fault vector dimension reduction and fault pattern vectors number reduction, highlighting cytokine Formal Immune Networks at the end, are presented in the following sections.

## 4.6 Fault vector dimension reduction

In our context, the dimension of a fault vector is the number of *inputs* plus *outputs* signals. Dimension reduction implies to reduce the number of signals in order to ease the distance measurement computations which have to be made for fault pattern recognition and repairing mechanism assignation [Theodoridis and Koutroumbas, 2008].

Given an $n$ dimensional fault vector[1] $X = [x_1, ..., x_i, ..., x_n]^T$, the aim is to find a lower dimensional vector $R = [r_1, ..., r_i, ..., r_t]^T$, with $t <= n$, that represents that fault preserving the content of the original fault vector according to some criterion, see figure 4.10. If we consider that we have $m$ fault pattern vectors $X = [x_1, ..., x_i, ..., x_n]^T$, a matrix $\mathbf{X}_{n \times m}$ can be formed with those fault vectors. Then, the aim of dimension reduction is to find a reduced matrix $\mathbf{R}_{t \times m}$, with $t <= n$, see figure 4.11.

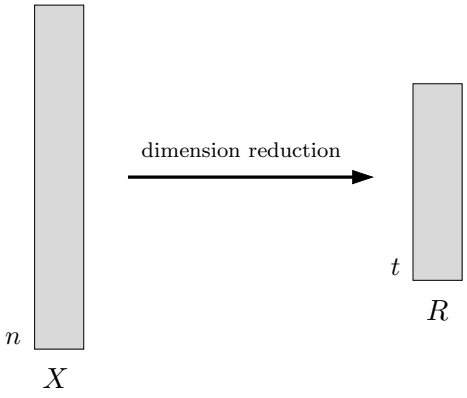



Figure 4.10: Dimension reduction of vector $X_{n \times 1}$ into vector $R_{t \times 1}$

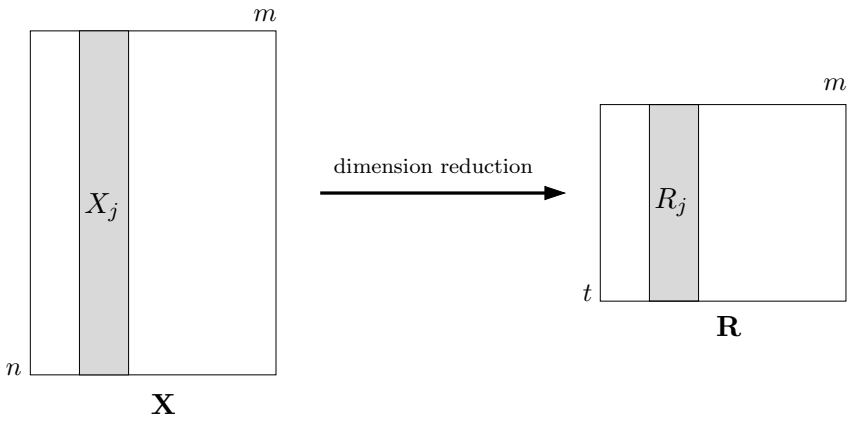



Figure 4.11: Dimension reduction of matrix $\mathbf{X}_{n \times m}$ into matrix $\mathbf{R}_{t \times m}$

Dimension reduction can be carried out by feature selection or feature extraction. On the one hand, by feature selection it is intended to find a subset of the most relevant original input and output signals. Methods of feature selection can be applied for reducing the dimension of fault vectors with one-bit binary value components "0" or "1". Algorithms of feature

[1] A vector is usually represented as a column, therefore the transpose is applied.

selection can be classified into feature ranking and subset feature selection. Feature ranking ranks the input and output signals by a metric and eliminates the signals that do not achieve an adequate score. Subset feature selection searches for an optimal subset of signals. On the other hand, by feature extraction the data in the high dimensional space is transformed to a space of fewer dimensions. Data transformation can be linear or non-linear. Feature extraction can be used for reducing the dimension of fault pattern vectors with integer or decimal number components expressed in fixed or floating point format.

Linear transformation for feature extraction considers each of the new $t$ elements $r_i$ of the reduced fault vector $R$ as a linear combination of the $n$ components $x_i$ of the original fault vector $X$ in the form:

$$R_{t\times 1} = \mathbf{W}^T_{n\times t} \times X_{n\times 1} \text{ where } \mathbf{W}_{n\times t} \text{ is the linear transformation weight matrix} \quad (4.10)$$

Figure 4.12 shows the linear transformation of vector $X$, where each element $r_i$ of the new reduced fault vector $R$ can be computed by:

$$r_i = \mathbf{W}_{i,1}x_1 + ... + \mathbf{W}_{i,n}x_n \quad \text{where} \quad i = 1, ..., t$$

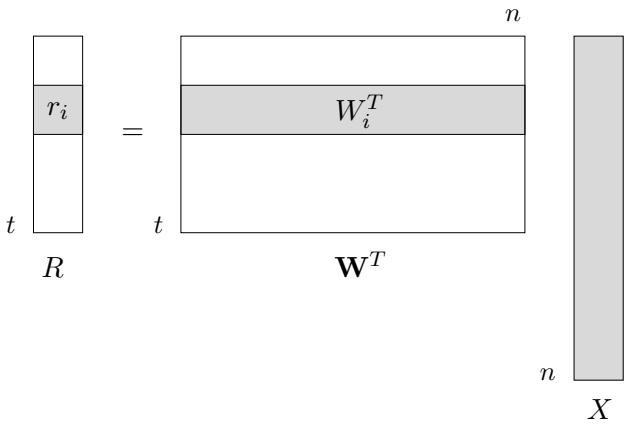



Figure 4.12: Linear transformation $R_{t\times 1} = \mathbf{W}_{t\times n} \times X_{n\times 1}$ of vector $X_{n\times 1}$ into vector $R_{t\times 1}$

Further, the matrix representation of the linear transformation is:

$$\mathbf{R}_{t\times m} = \mathbf{W}^T_{n\times t} \times \mathbf{X}_{n\times m} \quad (4.11)$$

Figure 4.13 shows the linear transformation of matrix $\mathbf{X}_{n\times m}$, where each element $\mathbf{R}_{ij}$ of the reduced matrix $\mathbf{R}_{t\times m}$ can be computed by:

$$\mathbf{R}_{i,j} = \mathbf{W}_{i,1}\mathbf{X}_{1,j} + .... + \mathbf{W}_{i,n}\mathbf{X}_{n,j} \quad \text{where} \quad i = 1, ..., t \text{ and } j = 1, ..., m$$

The most known linear transformation method for dimension reduction is the Principal Component Analysis which is explained next in subsection 4.6.1

### 4.6.1 Principal component analysis

Principal Component Analysis is a dimension reduction technique that executes a linear transformation of the fault pattern vectors $X = [x_1, ..., x_i, ..., x_n]^T$, arranged in the matrix $\mathbf{X}_{n\times m}$,

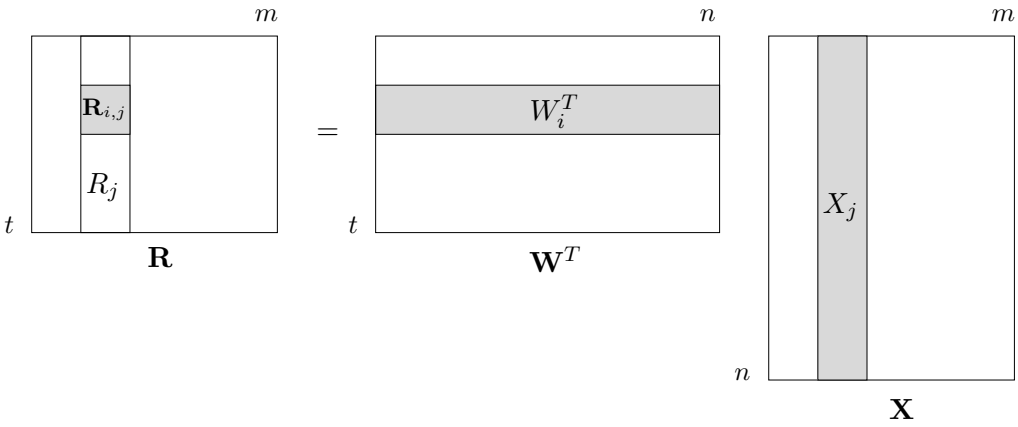



Figure 4.13: Linear transformation $\mathbf{R}_{t\times m} = \mathbf{W}^T_{n\times t}\mathbf{X}_{n\times m}$ of matrix $\mathbf{X}_{n\times m}$ into matrix $\mathbf{R}_{t\times m}$

to reduced fault pattern vectors $R = [r_1, ..., r_i, ..., r_t]^T$, arranged in the matrix $\mathbf{R}_{t\times m}$, refer [Theodoridis and Koutroumbas, 2008] and [Mardia et al., 1979]. The linear transformation of a fault pattern vector $X_{n\times 1}$ has been already defined above as:

$$R_{t\times 1} = \mathbf{W}^T_{n\times t} \times X_{n\times 1}$$

and the linear transformation of the fault pattern matrix $\mathbf{X}_{n\times m}$ has also been already defined above as:

$$\mathbf{R}_{t\times m} = \mathbf{W}^T_{n\times t} \times \mathbf{X}_{n\times m}$$

Given initially a not reduced transformation:

$$\mathbf{R}_{n\times m} = \mathbf{W}^T_{n\times n} \times \mathbf{X}_{n\times m} \tag{4.12}$$

the first goal of Principal Component Analysis is to find a proper transformation matrix $\mathbf{W}_{n\times n}$ such that a transformed fault pattern vector $R_{n\times 1}$ has uncorrelated components $r_i$, i.e. covariance equal to 0. The second goal is to reduce the dimension of the transformed fault pattern vector $R_{n\times 1}$ by retaining just $t$ components $r_i$ in vector $R_{t\times 1}$ called *principal components*. The principal components $r_i$ should describe as much of the variability as possible of fault pattern vector components $x_i$ of the original fault pattern vector $X$.

The first vector $W_1$, from the $t$ vectors in $\mathbf{W}_{n\times t}$, is used to calculate the first principal component $r_1$. $W_1$ should be the argument that describes the most of the total variance of the transformed matrix $\mathbf{R}_{n\times m} = \mathbf{W}^T_{n\times n} \times \mathbf{X}_{n\times m}$ and can be defined as:

$$W_1 = arg\ max_{\|W_1\|=1} Var\{\mathbf{W}^T_{n\times n} \times \mathbf{X}_{n\times m}\} \tag{4.13}$$

The following $t-1$ vectors $W_i$ should be the next arguments that describe as much of the remaining variance as possible of the transformed matrix $\mathbf{W}^T_{n\times n} \times \mathbf{X}_{n\times m}$. Note that vectors $W_i$ are orthonormal, i.e. norm $\| W_i \| = 1$, and mutually perpendicular, i.e. $W_i \cdot W_j = 0$ with $i \neq j$. Orthonormal vectors $W_i$ form an orthogonal matrix $\mathbf{W}_{n\times n}$, which constitute the basis for the transformation. An orthogonal matrix has the following properties: $\mathbf{W}\mathbf{W}^T = \mathbf{W}^T\mathbf{W} = I$ and $\mathbf{W}^T = \mathbf{W}^{-1}$. Where $I$ is the identity matrix and $\mathbf{W}^{-1}$ is the inverse matrix of $\mathbf{W}$.

The variance is the mean of the square of the amount of variation $\mathbf{X}_{i,j} - \bar{x}_i$ from the mean $\bar{x}_i$ of values $\mathbf{X}_{i,j}$ corresponding to the fault pattern vector component $x_i$. It can be also

calculated by the square of the standard deviation $\sigma_{x_i}$. Please see figures 4.14 and 4.15 for a better understanding. Besides, terms like mean $\bar{x}_i$, standard deviation $\sigma_{x_i}$ and variance $Var(x_i)$ are defined formally below.

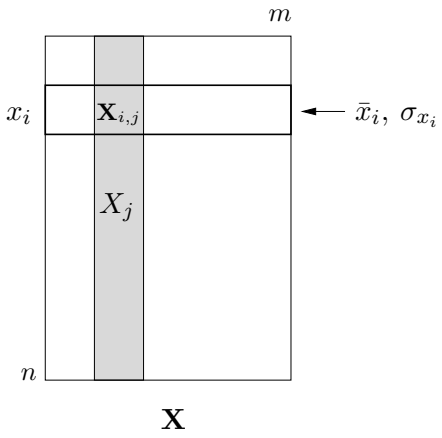



Figure 4.14: Mean $\bar{x}_i$ and standard deviation $\sigma_{x_i}$ of a fault pattern vector components $x_i$ in matrix $\mathbf{X}_{n\times m}$

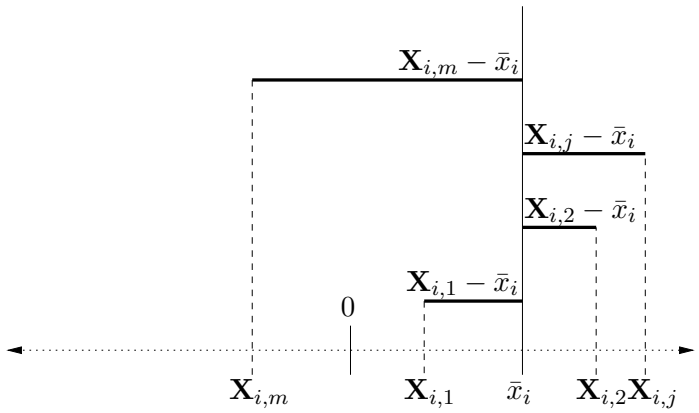



Figure 4.15: Deviation $\mathbf{X}_{i,j} - \bar{x}_i$ of fault pattern component values $\mathbf{X}_{i,j}$ from the fault pattern vector component mean $\bar{x}_i$

The mean $\bar{x}_i$ of fault pattern vector components $x_i$ is also called expected value of a fault pattern vector component $E[x_i]$ and can be calculated by:

$$\bar{x_i} = E[x_i] = \frac{1}{m}\sum_{j=1}^{m}\mathbf{X}_{i,j} \tag{4.14}$$

The standard deviation $\sigma_{x_i}$ of a fault pattern vector component $x_i$ can be calculated by:

$$\sigma_{x_i} = \sqrt{E[(x_i - \bar{x}_i)^2]} = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(\mathbf{X}_{i,j} - \bar{x}_i)^2}$$

The square of the standard deviation gives the variance as shown below:

$$Var(x_i) = E[(x_i - \bar{x}_i)^2] = \frac{1}{m}\sum_{j=1}^{m}(\mathbf{X}_{i,j} - \bar{x}_i)^2$$

Moreover the mean $\bar{X}$, standard deviation $\sigma_X$ and variances $Var(X)$ of all $n$ fault pattern vector components $x_i$ of fault pattern vector $X$ can be defined as:

$$\bar{X} = (\bar{x}_1, ..., \bar{x}_i, ..., \bar{x}_n)^T = E[X] = \frac{1}{m}\sum_{j=1}^{m} X_j$$

$$\sigma_X = (\sigma_{x_1}, ..., \sigma_{x_i}, ..., \sigma_{x_n})^T = \sqrt{E[(X - \bar{X})^2]} = \sqrt{\frac{1}{m}\sum_{j=1}^{m}(X_j - \bar{X})^2}$$

$$Var(X) = (Var(x_1), ..., Var(x_i), ..., Var(x_n))^T = E[(X - \bar{X})^2] = \frac{1}{m}\sum_{j=1}^{m}(X_j - \bar{X})^2$$

If the variances $Var(x_i)$ of vector $Var(X)$ are arranged in the diagonal of a $n \times n$ matrix, the values off the diagonal $Cov(x_i, x_l)$ are called covariances between fault pattern vector components $x_i$ and $x_l$ and the resulting matrix is called covariance matrix. First, the covariance between fault pattern vector components $x_i$ and $x_l$ can be calculated by:

$$Cov(x_i, x_l) = E[(x_i - \bar{x}_i)(x_j - \bar{x}_l)] = \frac{1}{m}\sum_{j=1}^{m}(\mathbf{X}_{i,j} - \bar{x}_i)(\mathbf{X}_{l,j} - \bar{x}_l)$$

The covariance expresses how much fault pattern vector components $x_i$ and $x_l$ vary together. A covariance of zero means that both variables do not show any tendency to vary together. A covariance greater than zero means big values of $x_i$ correspond to big values of $x_l$. And a covariance less than zero means big values of $x_i$ correspond to small values of $x_l$. The covariance gets the dimension obtained by multiplying the units of fault pattern vector component $x_i$ with fault pattern vector component $x_l$. Second, the covariance matrix can be computed by means of matrices operations as follows:

$$\mathbf{Cov}(\mathbf{X})_{n\times n} = E[(X - \bar{X})(X - \bar{X})^T] = \frac{1}{m}\sum_{j=1}^{m}(X_j - \bar{X})(X_j - \bar{X})^T = \frac{1}{m}\mathbf{B}_{n\times m}\mathbf{B}^T_{n\times m} \quad (4.15)$$

Where $\mathbf{B}_{n\times m}$ is the matrix of deviations from the mean $\bar{X}$ of matrix $\mathbf{X}$ computed by:

$$\mathbf{B}_{n\times m} = \mathbf{X}_{n\times m} - (\bar{X}_{n\times 1} \times [1, ..., 1, ...1]_{1\times m}) \quad (4.16)$$

After all definitions, the first goal of Principal Component Analysis can be redefined as getting a transformed matrix $\mathbf{R}_{n\times m}$ such that its covariance matrix looks like:

$$Cov(\mathbf{R})_{n\times n} = \begin{pmatrix} Var(r_1) & & & \cdots & \mathbf{0} \\ & \ddots & & & \vdots \\ & & Var(r_i) & & \\ \vdots & & & \ddots & \\ \mathbf{0} & \cdots & & & Var(r_n) \end{pmatrix}$$

Now, considering the not reduced transformed matrix $\mathbf{R}_{n\times m}$ is the matrix of deviations from the mean $\bar{R}$ of matrix $\mathbf{R}$ with $\bar{R} = [0]$, the covariance matrix of $\mathbf{R}_{n\times m}$ can be expressed as:

$$\mathbf{Cov}(\mathbf{R})_{n\times n} = E(\mathbf{R}_{n\times m}\mathbf{R}^T_{n\times m})$$

Substituting $\mathbf{R}_{n\times m}$ by $\mathbf{W}^T_{n\times n}\mathbf{X}_{n\times m}$ and making operations we get:

$$\mathbf{Cov}(\mathbf{R})_{n\times n} = E((\mathbf{W}^T_{n\times n}\mathbf{X}_{n\times m})(\mathbf{W}^T_{n\times n}\mathbf{X}_{n\times m})^T)$$

$$\mathbf{Cov}(\mathbf{R})_{n\times n} = E((\mathbf{W}^T_{n\times n}\mathbf{X}_{n\times m}\mathbf{X}^T_{n\times m}\mathbf{W}_{n\times n}))$$

$$\mathbf{Cov}(\mathbf{R})_{n\times n} = \mathbf{W}^T_{n\times n}E(\mathbf{X}_{n\times m}\mathbf{X}^T_{n\times m})\mathbf{W}_{n\times n}$$

$$\mathbf{Cov}(\mathbf{R})_{n\times n} = \mathbf{W}^T_{n\times n}\mathbf{Cov}(\mathbf{X}_{n\times m})\mathbf{W}_{n\times n}$$

Multiplying to the left of the terms of the equation by $\mathbf{W}_{n\times n}$ and to the right by $\mathbf{W}^T_{n\times n}$ we get:

$$\mathbf{W}_{n\times n}\mathbf{Cov}(\mathbf{R})_{n\times n}\mathbf{W}^T_{n\times n} = \mathbf{Cov}(\mathbf{X}_{n\times m})$$

The equation resembles the eigenvalue decomposition of matrix $\mathbf{Cov}(\mathbf{X}_{n\times m})$ defined as:

$$\mathbf{W}_{n\times n}\mathbf{E}_{n\times n}\mathbf{W}^T_{n\times n} = \mathbf{Cov}(\mathbf{X}_{n\times m})$$

where $\mathbf{E}_{n\times n}$ is a diagonal matrix that contains eigenvalues and the matrix $\mathbf{W}_{n\times n}$ contains corresponding eigenvectors. Then we have $\mathbf{Cov}(\mathbf{R})_{n\times n} = \mathbf{E}_{n\times n}$, equality which meets the first goal of the Principal Component Analysis, covariances equal to 0 off the diagonal. If the eigenvalues are arranged in a decreasing way and the eigenvectors correspondingly, the first eigenvalue expresses the most of the variance and its corresponding eigenvector can be taken as the first transformation vector $W_1$. Considering the total variance of matrix $\mathbf{R}_{n\times m}$ is the sum of the variances $Var(x_i)$, correspondingly the sum of the eigenvalues $\mathbf{E}_{i,i}$ of the diagonal of matrix $\mathbf{E}_{n\times n}$, $t$ eigenvalues which contribute to maintain determined amount of variance in percent can be chosen and their respective eigenvectors can be taken to form the transforming matrix $\mathbf{W}_{t\times m}$.

Furthermore, the transforming matrix $\mathbf{W}_{n\times n}$ can also be calculated by means of the singular value decomposition of the matrix of deviations from the mean $\bar{X}$ of matrix $\mathbf{X}_{n\times m}$, named above $\mathbf{B}_{n\times m}$. If we define the singular value decomposition of the matrix of deviations $\mathbf{B}_{n\times m}$ as:

$$\mathbf{B}_{n\times m} = \mathbf{U}_{n\times n}\mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m}$$

Were:

$\mathbf{S}_{n\times m}$ is a matrix with singular values $s_i$, ordered in a descending way, in the main diagonal and zeros off the diagonal

$\mathbf{U}_{n\times n}$ is a matrix which contains the left singular vectors corresponding the singular values $s_i$

$\mathbf{V}_{m\times m}$ is a matrix which contains the right singular vectors corresponding the singular values $s_i$

We can now use the singular value decomposition factorization in the definition of the covariance matrix $\mathbf{Cov}(\mathbf{X})_{n\times n}$ in the following way:

**Algorithm 4.1:** Fault pattern vector dimension reduction using principal component analysis and the covariance

1: Arrange fault pattern vectors $X_{n\times 1}$ together into matrix $\mathbf{X}_{n\times m}$
2: Calculate the mean vector $\bar{X}_{n\times 1}$, which contain the mean of each fault pattern vector component $x_i$ using:
$$\bar{x}_i = \frac{1}{m}\sum_{j=1}^{m}\mathbf{X}_{ij}$$
3: Calculate the matrix of deviations from the mean using:
$$\mathbf{B}_{n\times m} = \mathbf{X}_{n\times m} - (\bar{X}_{n\times 1} \times [1, ..., 1, ..., 1]_{1\times m})$$
4: Calculate the covariance matrix of matrix $\mathbf{X}_{n\times m}$ using:
$$\mathbf{Cov}(\mathbf{X})_{n\times n} = \frac{1}{m}(\mathbf{B}_{n\times m} \times \mathbf{B}^T_{n\times m})$$
5: Calculate the matrix of eigenvectors $\mathbf{W}_{n\times n}$ and the matrix of eigenvalues $\mathbf{E}_{n\times n}$ of the covariance matrix $\mathbf{Cov}(\mathbf{X})_{n\times n}$
6: Rearrange the eigenvalues in the diagonal matrix $\mathbf{E}_{n\times n}$ in a decreasing manner moving their respective eigenvectors in matrix $\mathbf{W}_{n\times n}$ accordingly
7: Calculate the total variance of the transformed matrix $\mathbf{R}_{n\times m}$ using:
$$total\ variance = \sum_{j=1}^{m}\mathbf{E}_{j,j}$$
8: Given an expected *maintained variance* in percent, choose $t$ eigenvectors corresponding to the biggest $t$ eigenvalues such that $partial\ variance \geq maintained\ variance$ using:
$$partial\ variance = (\mathbf{E}_{1,1} + ... + \mathbf{E}_{t,t})\frac{100}{total\ variance}$$
9: Calculate the new compressed matrix using only the $t$ chosen eigenvectors using:
$$\mathbf{R}_{t\times m} = \mathbf{W}^T_{n\times t}\mathbf{X}_{n\times m}$$

$$\mathbf{Cov}(\mathbf{X})_{n\times n} = \frac{1}{m}\mathbf{B}_{n\times m}\mathbf{B}^T_{n\times m} = \frac{1}{m}(\mathbf{U}_{n\times n}\mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m})(\mathbf{U}_{n\times n}\mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m})^T$$

Making some operations we get:

$$\mathbf{Cov}(\mathbf{X})_{n\times n} = \mathbf{U}_{n\times n}(\frac{1}{m}\mathbf{S}_{n\times m}\mathbf{S}^T_{n\times m})\mathbf{U}^T_{n\times n}$$

We can therefore conclude that the left singular vector matrix $\mathbf{U}_{n\times n}$ is equal to the matrix of eigenvectors $\mathbf{W}_{n\times n}$ of the eigenvalue decomposition of the covariance matrix $\mathbf{Cov}(\mathbf{X})_{n\times n}$ and the eigenvalue matrix $\mathbf{E}_{n\times n}$ can be calculated by:

$$\mathbf{E}_{n\times n} = \frac{1}{m}\mathbf{S}_{n\times m}\mathbf{S}^T_{n\times m} \tag{4.17}$$

Then the procedure to select the $t$ left singular vectors of the matrix $\mathbf{U}_{n\times n}$ to form the transforming matrix $\mathbf{W}_{n\times t}$ can be executed in the same way as explained above.

Algorithm 4.1, shown below, serves to compute the reduced matrix $\mathbf{R}_{t\times m}$ calculating the covariance matrix $\mathbf{Cov}(\mathbf{X})_{n\times n}$ and Algorithm 4.2 serves to compute the reduced matrix $\mathbf{R}_{t\times m}$ calculating the singular value decomposition of the matrix of deviations $\mathbf{B}_{n\times m}$.

### 4.6.2 Singular value decomposition

The singular value decomposition, shortened as SVD, is a way to factorize a rectangular matrix $\mathbf{X}_{n\times m}$. This factorization can be used to find the linear transformation weight matrix $\mathbf{W}_{n\times t}$ for reducing fault pattern vectors $X_{n\times 1}$ arranged in the matrix $\mathbf{X}_{n\times m}$ to matrix

**Algorithm 4.2:** Fault pattern vector dimension reduction using principal component analysis and singular value decomposition

1: Arrange fault pattern vectors $X_{n\times 1}$ together into matrix $\mathbf{X}_{n\times m}$

2: Calculate the mean vector $\bar{X}_{n\times 1}$, which contain the mean of each fault pattern vector component $x_i$ using:

$$\bar{x}_i = \frac{1}{m}\sum_{j=1}^{m}\mathbf{X}_{ij}$$

3: Calculate the matrix of deviations from the mean using:

$$\mathbf{B}_{n\times m} = \mathbf{X}_{n\times m} - (\bar{X}_{n\times 1} \times [1, ..., 1, ..., 1]_{1\times m})$$

4: Calculate the matrix of singular values $\mathbf{S}_{n\times m}$, matrix of left singular vectors $\mathbf{U}_{n\times n}$ and matrix of right singular vectors $\mathbf{V}_{m\times m}$ of matrix $\mathbf{B}_{n\times m}$ (The compact singular value decomposition $\mathbf{B}_{n\times m} = \mathbf{U}_{n\times r}\mathbf{S}_{r\times r}\mathbf{V}^T_{m\times r}$ can be also sufficient, here being $r$ the rank of matrix $\mathbf{B}_{n\times m}$)

5: Rearrange the singular values in matrix $\mathbf{S}_{n\times m}$ in a decreasing manner moving their respective left singular vectors in matrix $\mathbf{U}_{n\times n}$ and right singular vectors in matrix $\mathbf{V}_{m\times m}$ accordingly

6: Calculate the covariance matrix of the transformed matrix $\mathbf{R}_{n\times m}$ which corresponds to the matrix of eigenvalues $\mathbf{E}_{n\times n}$ of the eigenvalue decomposition of the covariance of matrix $\mathbf{X}_{n\times m}$ using:

$$\mathbf{E}_{n\times n} = \frac{1}{m}(\mathbf{S}_{n\times m} \times \mathbf{S}^T_{n\times m})$$

7: Calculate the total variance of the transformed matrix $\mathbf{R}_{n\times m}$ using:

$$\textit{total variance} = \sum_{j=1}^{m}\mathbf{E}_{j,j}$$

8: Given an expected *maintained variance* in percent, choose $t$ eigenvectors corresponding to the biggest $t$ eigenvalues such that *partial variance* $\geq$ *maintained variance* using:

$$\textit{partial variance} = (\mathbf{E}_{1,1} + ... + \mathbf{E}_{t,t})\frac{100}{\textit{total variance}}$$

9: Calculate the new compressed matrix using only the $t$ chosen eigenvectors using:

$$\mathbf{R}_{t\times m} = \mathbf{U}^T_{n\times t}\mathbf{X}_{n\times m}$$

$\mathbf{R}_{t\times m}$ by means of the linear transformation given in equation 4.11, as extensively shown in [Theodoridis and Koutroumbas, 2008] and [Theodoridis, 2009].

We consider the singular value decomposition of matrix $\mathbf{X}$ defined as:

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times n}\mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m}$$

Were:

$\mathbf{S}_{n\times m}$ is a matrix with singular values $s_i$, ordered in a descending way, in the main diagonal and zeros off the diagonal

$\mathbf{U}_{n\times n}$ is a matrix which contains the left singular vectors corresponding the singular values $s_i$

$\mathbf{V}_{m\times m}$ is a matrix which contains the right singular vectors corresponding the singular values $s_i$

If we multiply both terms of the singular value decomposition by the transpose of the left singular vector matrix i.e. $\mathbf{U}^T_{n\times n}$, we obtain:

$$\mathbf{U}^T_{n\times n}\mathbf{X}_{n\times m} = \mathbf{U}^T_{n\times n}\mathbf{U}_{n\times n}\mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m}$$

If matrix $\mathbf{U}_{n\times n}$ is orthogonal, then $\mathbf{U}^T_{n\times n}\mathbf{U}_{n\times n} = \mathbf{I}_{n\times n}$. Where $\mathbf{I}_{n\times n}$ is the identity matrix. Note that a matrix multiplied by an identity matrix give as a result the same matrix. Then we obtain the Karhunen-Loéve transform $\mathbf{Y}$ of matrix $\mathbf{X}$ as shown below:

$$\mathbf{Y}_{n\times m} = \mathbf{U}^T_{n\times n}\mathbf{X}_{n\times m} = \mathbf{S}_{n\times m}\mathbf{V}^T_{m\times m}$$

If we compare this matrix equation with the transformation matrix we are looking for $\mathbf{W}_{n\times n}$, matrix $\mathbf{U}_{n\times n}$ can be used to find the transformation matrix $\mathbf{W}_{t\times n}$. But, still matrix $U$ is $n \times n$.

Visualizing the factorization in figure 4.16 for $n > m$, because there exist only $m$ corresponding singular values, the matrix $U$ can be rewritten as an $n \times m$ matrix.

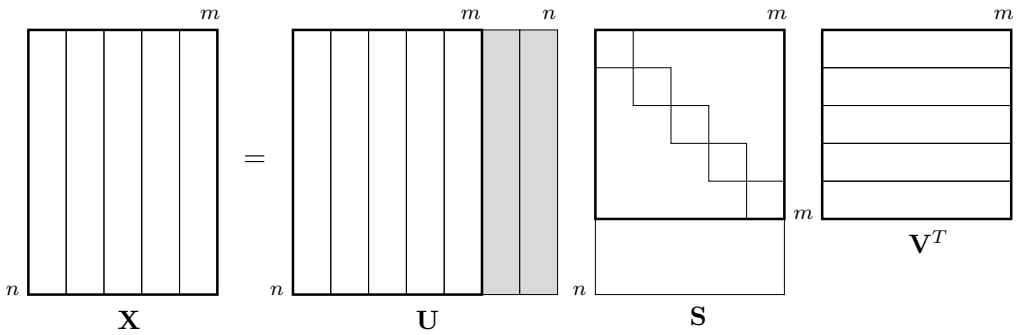



Figure 4.16: Singular value decomposition of matrix $\mathbf{X}_{n\times m} = \mathbf{U}_{n\times m}\mathbf{S}_{m\times m}\mathbf{V}^T_{m\times m}$ for $n > m$

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times m}\mathbf{S}_{m\times m}\mathbf{V}^T_{m\times m}$$

$$\begin{pmatrix} X_{1,1} & \cdots & X_{1,m} \\ \vdots & \ddots & \vdots \\ X_{n,1} & \cdots & X_{n,m} \end{pmatrix} = \begin{pmatrix} U_{1,1} & \cdots & U_{1,m} \\ \vdots & \ddots & \vdots \\ U_{n,1} & \cdots & U_{n,m} \end{pmatrix} \times \begin{pmatrix} S_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & S_{m,m} \end{pmatrix} \times \begin{pmatrix} V_{1,1} & \cdots & V_{m,1} \\ \vdots & \ddots & \vdots \\ V_{1,m} & \cdots & V_{m,m} \end{pmatrix}$$

Also, visualizing the factorization in figure 4.17 for $n < m$, because there exist only $n$ corresponding singular values, the matrix $V^T$ can be rewritten as a $m \times n$ matrix.

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times n}\mathbf{S}_{n\times n}\mathbf{V}^T_{m\times n}$$

$$\begin{pmatrix} X_{1,1} & \cdots & X_{1,m} \\ \vdots & \ddots & \vdots \\ X_{n,1} & \cdots & X_{n,m} \end{pmatrix} = \begin{pmatrix} U_{1,1} & \cdots & U_{1,n} \\ \vdots & \ddots & \vdots \\ U_{n,1} & \cdots & U_{n,n} \end{pmatrix} \times \begin{pmatrix} S_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & S_{n,n} \end{pmatrix} \times \begin{pmatrix} V_{1,1} & \cdots & V_{n,1} \\ \vdots & \ddots & \vdots \\ V_{1,m} & \cdots & V_{n,m} \end{pmatrix}$$

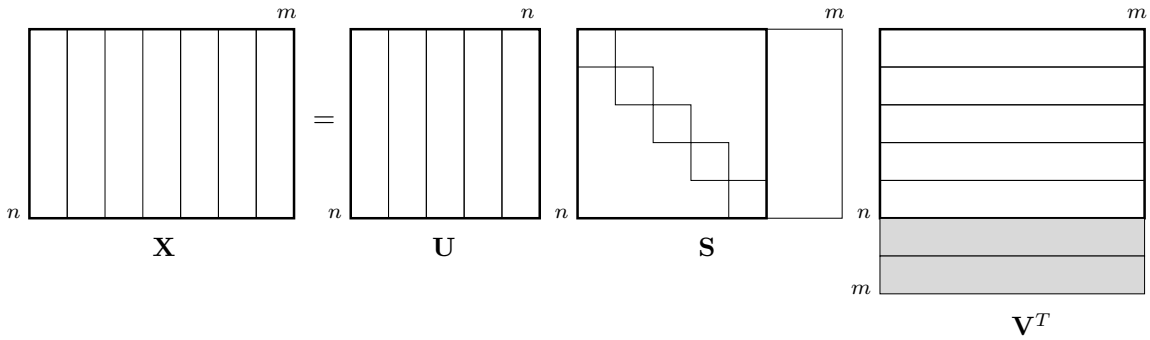



Figure 4.17: Singular value decomposition of matrix $\mathbf{X}_{n\times m} = \mathbf{U}_{n\times n}\mathbf{S}_{n\times n}\mathbf{V}^T_{m\times n}$ for $n < m$

Now, if we multiply $\mathbf{U}_{n\times n} \times \mathbf{S}_{n\times n} \times \mathbf{V}^T_{m\times n}$, the singular value decomposition when $n > m$, we obtain:

$$\begin{pmatrix} X_{1,1} & \cdots & X_{1,m} \\ \vdots & \ddots & \vdots \\ X_{n,1} & \cdots & X_{n,m} \end{pmatrix} = \begin{pmatrix} U_{1,1}S_{1,1}V_{1,1} + ... + U_{1,m}S_{m,m}V_{1,m} & \cdots & U_{1,1}S_{1,1}V_{m,1} + ... + U_{1,m}S_{m,m}V_{m,m} \\ \vdots & \ddots & \vdots \\ U_{n,1}S_{1,1}V_{1,1} + ... + U_{n,m}S_{m,m}V_{1,m} & \cdots & U_{n,1}S_{1,1}V_{m,1} + ... + U_{n,m}S_{m,m}V_{m,m} \end{pmatrix}$$

Alike for $n < m$:

$$\begin{pmatrix} X_{1,1} & \cdots & X_{1,m} \\ \vdots & \ddots & \vdots \\ X_{n,1} & \cdots & X_{n,m} \end{pmatrix} = \begin{pmatrix} U_{1,1}S_{1,1}V_{1,1} + ... + U_{1,n}S_{n,n}V_{1,n} & \cdots & U_{1,1}S_{1,1}V_{m,1} + ... + U_{1,n}S_{n,n}V_{m,n} \\ \vdots & \ddots & \vdots \\ U_{n,1}S_{1,1}V_{1,1} + ... + U_{n,n}S_{n,n}V_{1,n} & \cdots & U_{n,1}S_{1,1}V_{m,1} + ... + U_{n,n}S_{n,n}V_{m,n} \end{pmatrix}$$

As we can see, each singular value adds up a new term to each element in the matrix. Normally the singular values are ordered in a descending way then, the effect of last terms in the sum is minimal.

Therefore, if only $t$ column vectors of $\mathbf{U}$ and $t$ row vectors of $\mathbf{V}^T$ corresponding to the largest singular values $s_1$ to $s_t$ of matrix $\mathbf{S}$ are taken and the rest is discarded, we obtain a truncated singular value decomposition. That is shown in figure 4.18 and defined below.

$$\tilde{\mathbf{X}}_{n\times m} = \mathbf{U}_{n\times t}\mathbf{S}_{t\times t}\mathbf{V}^T_{m\times t}$$

Where $\tilde{\mathbf{X}}_{n\times m}$ is an approximation of matrix $\mathbf{X}_{n\times m}$ since the singular value decomposition is no longer exact.

If we consider then:

$$\mathbf{U}^T_{n\times t}\tilde{\mathbf{X}}_{n\times m} = \mathbf{S}_{t\times t}\mathbf{V}^T_{m\times t}$$

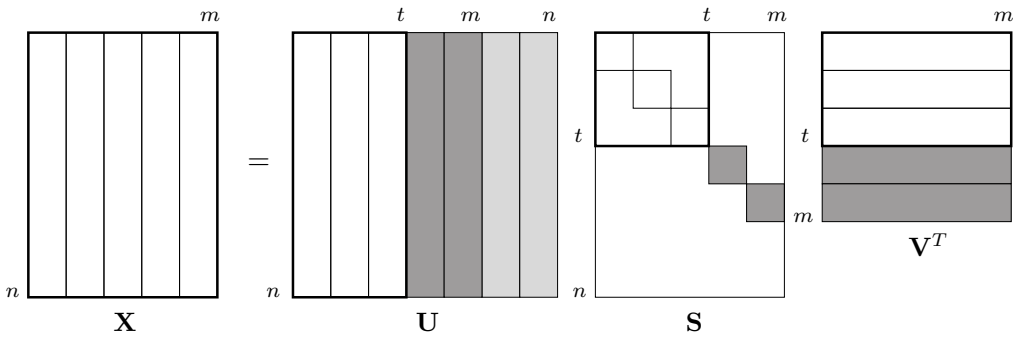



Figure 4.18: Truncated singular value decomposition $\tilde{\mathbf{X}}_{n\times m} = \mathbf{U}_{n\times t}\mathbf{S}_{t\times t}\mathbf{V}^T_{m\times t}$ of matrix $\mathbf{X}$ for $n < m$

We get a reduced transformed matrix $\mathbf{R}_{t\times m}$. Then we can write:

$$\mathbf{R}_{t\times m} = \mathbf{U}^T_{t\times t}\tilde{\mathbf{X}}_{n\times m} = \mathbf{S}_{t\times t}\mathbf{V}^T_{m\times t}$$

The listed algorithm 4.3 shown below serves to compute the reduced matrix $\mathbf{R}_{t\times m}$ calculating the singular value decomposition of matrix $\mathbf{X}_{n\times n}$. A variant of this method to reduce the dimension of fault pattern vectors is used by the cytokine Formal Immune Network algorithm described in section 4.8.

**Algorithm 4.3:** Fault pattern vector dimension reduction using singular value decomposition

1: Arrange fault pattern vectors $X_{n\times 1}$ together into matrix $\mathbf{X}_{n\times m}$
2: Calculate the matrix of singular values $\mathbf{S}_{n\times m}$, matrix of left singular vectors $\mathbf{U}_{n\times n}$ and matrix of right singular vectors $\mathbf{V}_{m\times m}$ of matrix $\mathbf{X}_{n\times m}$ (The compact singular value decomposition $\mathbf{X}_{n\times m} = \mathbf{U}_{n\times r}\mathbf{S}_{r\times r}\mathbf{V}^T_{m\times r}$ can be also sufficient, being $r$ the rank of matrix $\mathbf{X}_{n\times m}$)
3: Rearrange the singular values in matrix $\mathbf{S}_{n\times m}$ in a decreasing manner moving their respective left singular vectors in matrix $\mathbf{U}_{n\times n}$ and right singular vectors in matrix $\mathbf{V}_{m\times m}$ accordingly
4: Choose $t$ left singular vectors corresponding to the biggest $t$ singular values following some criterion
5: Calculate the new compressed matrix using only the $t$ chosen eigenvectors using:
$$\mathbf{R}_{t\times m} = \mathbf{U}^T_{n\times t}\mathbf{X}_{n\times m}$$

## 4.7 Fault pattern vectors number reduction

Fault pattern vectors number is the number $m$ of vectors $X_{n\times 1}$ arranged in matrix $\mathbf{X}_{n\times m}$ or the number $m$ of vectors $R_{t\times 1}$ arranged in matrix $\mathbf{R}_{t\times m}$. For convenience the text below is expressed in terms of reduced fault pattern vectors $R_{t\times 1}$ because vector operations with reduced vectors is much more efficient. However, fault pattern vectors number reduction can also be executed before dimension reduction. Fault pattern vectors number reduction implies to reduce the number $m$ of vectors $R_{t\times 1}$ in such a way that the remaining $q$ fault pattern

vectors $R_{t\times 1}$ are the most representative in the set $\{R\}$. Those $q$ fault pattern vectors $R_{t\times 1}$ should fit into memory available in the fault recognition module.

The biological immune system disposes of immune cells which reproduce and die intensively under the presence of pathogens. However, the immune system maintains the number of immune cells within some limits avoiding an exponential increasing or decreasing of immune cells. That regulatory mechanism is attributed to a sort of network that immune cells build by interacting with each other. The strategies for avoiding an exponential increasing of immune cells are, among others, the death of immune cells with insufficient stimulation and the elimination of auto-reactive immune cells. Those two strategies are used for reducing the fault pattern vectors number $m$ in next subsections.

### 4.7.1 Death of immune cells with insufficient stimulation

Immune cells are stimulated by other immune cells that are also stimulated by other immune cells. Cells which do not receive stimulation die. Stimulation of immune cells is given by affinity or complementarity. An immune cell stimulates another immune cell when the affinity of both cells overpass an stimulation threshold. The immune cells with insufficient stimulation in a time period die. This idea has been transferred to artificial immune networks in [Timmis and Neal, 2001] for bringing dynamics into a population of cells thereby eliminating outliers. That algorithm is explained below.

Given two fault pattern vectors $R_k$ and $R_l$, their affinity or complementarity can be assumed as their similarity. As we have seen before, similarity among two fault pattern vectors $R_k$ and $R_l$ can be measured by any distance measurement method, i.e. Minkowski distance or Hamming distance, where the lower the distance the higher the similarity. Fault pattern vectors $R_k$ which do not present any similarity within an stimulation threshold with any other fault pattern vector $R_l$ are removed from the set $\{R\}$. Algorithm 4.4 shows that procedure. Note that instead of removing fault pattern vectors $R_k$ with stimulation variable equal to 0 from matrix $\mathbf{R}_{t\times m}$, fault pattern vectors $R_k$ with stimulation variable greater that 0 are included into the reduced fault pattern vectors matrix $\mathbf{Q}$. Then, matrix $\mathbf{Q}_{t\times q}$ contains $q$ fault pattern vectors $R_{t\times 1}$ which are close together within a dense area. All fault pattern vectors $R_{t\times 1}$ far away from that dense area, at least with a distance equal to the stimulation threshold, are eliminated. If only one fault repairing mechanism is assigned to all fault pattern vectors $R_{t\times 1}$ in matrix $\mathbf{R}_{t\times m}$, that is to say, vectors $R_{t\times 1}$ belong to a subset $\{R\}_c$ then, eliminated fault vectors $R_{t\times 1}$ can be interpreted as a sort of outliers of that set $\{R\}_c$, please see figure 4.19 drawn for $t = 2$.

### 4.7.2 Elimination of auto-reactive immune cells

An immune cell which recognizes a molecule of the body and mount an immune response against that molecule is named auto-reactive immune cell. Auto-reactive immune cells can damage the body producing auto-immune diseases. Therefore, the immune system tries to get rid of those immune cells. That idea has been used in the cFIN algorithm for reducing similar vectors and named as *apoptosis*, please see [Tarakanov et al., 2005] and [Tarakanov, 2008] or refer to next section 4.8.

As we have seen in the section referred to fault recognition, given two fault pattern vectors $R_k$ and $R_l$, fault pattern vector $R_k$ recognizes fault pattern vector $R_l$ when fault pattern vectors $R_k$ and $R_l$ are similar enough. That is to say, the distance between fault pattern

**Algorithm 4.4:** Fault pattern vectors number reduction resembling the death of immune cells with insufficient stimulation

Arrange the $m$ fault pattern vectors $R_{t\times 1}$ together into matrix $\mathbf{R}_{t\times m}$
**for** $k = 1$ **to** $m - 1$ **do**
  Reset the stimulation variable of fault pattern vector $R_k$
  **for** $l = k + 1$ **to** $m$ **do**
    Calculate the distance of fault pattern vector $R_k$ with fault pattern vector $R_l$ using i.e. any of the Minkowski distances by:
    $$\text{distance}(R_k, R_l) = (|r_{k1} - r_{l1}|^h + ... + |r_{ki} - r_{li}|^h + ... + |r_{kt} - r_{lt}|^h)^{\frac{1}{h}}$$
    **if** distance$(R_k, R_l)$ < stimulation threshold **then**
      Increase the stimulation variable of fault pattern vector $R_k$ by 1
    **end**
  **end**
  **if** stimulation variable of fault pattern vector $R_k > 0$ **then**
    Include fault pattern vector $R_k$ to the reduced fault pattern vectors matrix $\mathbf{Q}$
  **end**
**end**

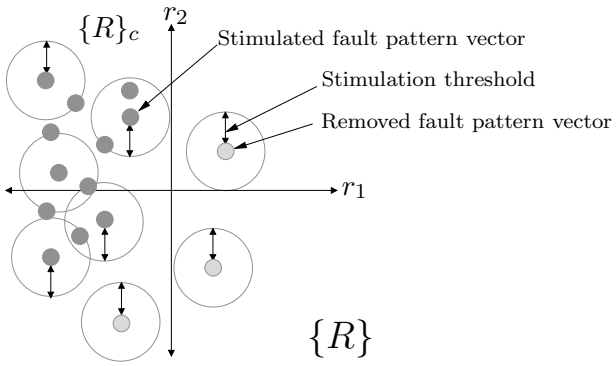



Figure 4.19: Reduction of fault pattern vectors using the principle of insufficient stimulation

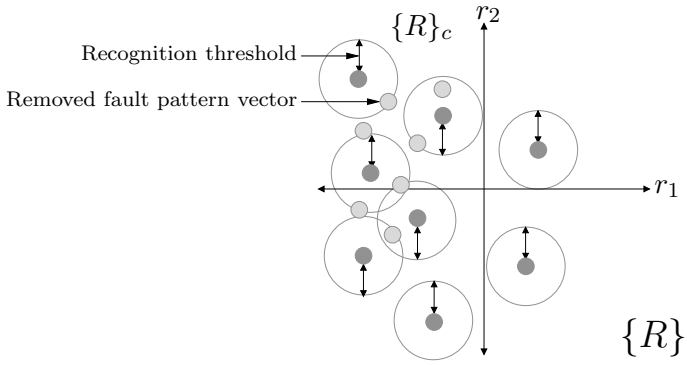



Figure 4.20: Reduction of fault pattern vectors using the principle of auto-reactive immune cells

**Algorithm 4.5:** Removal of fault pattern vectors resembling the elimination of auto-reactive immune cells

1: Arrange the $m$ fault pattern vectors $R_{t\times 1}$ together into matrix $\mathbf{R}_{t\times m}$
2: **for** $k = 1$ **to** $m - 1$ **do**
3: Copy fault pattern vector $k$ of matrix $\mathbf{R}$ in the reduced set of fault pattern vectors $\mathbf{Q}$
4: **for** $l = k + 1$ **to** $m$ **do**
5: Calculate the distance of fault pattern vector $R_k$ with fault pattern vector $R_l$ using i.e. any of the Minkowski distances by:
$$\text{distance}(R_k, R_l) = (|r_{k1} - r_{l1}|^h + ... + |r_{ki} - r_{li}|^h + ... + |r_{kt} - r_{lt}|^h)^{\frac{1}{h}}$$
6: **if** distance$(R_k, R_l) >$ threshold **then**
7: Copy fault patter vector $R_k$ in the reduced set of fault pattern vectors $\mathbf{Q}$
8: **end**
9: **end**
10: Adopt the whole reduced set of fault pattern vectors $\mathbf{Q}$ as the new set of fault pattern vectors $\mathbf{R}$ for reducing in the next loop
11: Calculate the new number of fault pattern vectors $m$ of the new matrix $\mathbf{R}$
12: Maintain $k$ fault pattern vectors in the reduced set $\mathbf{Q}$ and remove the rest fault pattern vectors
13: **end**

vector $R_k$ and fault pattern vector $R_l$ is smaller than a recognition threshold. Then, fault pattern vectors $R_l$ that are recognized by fault pattern vector $R_k$ can be eliminated. This procedure assures to have only one fault pattern vector $R_{t\times 1}$ within a round area of radius the size of the recognition threshold, thereby avoiding redundant fault pattern vectors in the set $\{R\}_c$, please see figure 4.20 drawn for $t = 2$. Algorithm 4.5 serves for reducing fault pattern vectors $R_{t\times 1}$ arranged in a matrix $\mathbf{R}_{t\times m}$. The resultant reduced fault pattern vectors matrix $\mathbf{Q}_{t\times q}$ depends on the order of fault pattern vectors $R_{t\times 1}$ in matrix $\mathbf{R}_{t\times m}$. This method for reducing the fault pattern vectors number is used in the cytokine Formal Immune Network algorithm explained in next section considering that different sets $\{R\}_c$ are available.

## 4.8 Cytokine formal immune network

Cytokine Formal Immune Network is a method for pattern recognition based on molecular recognition, which is the interaction between two molecules that show complementarity. This method is presented in [Tarakanov et al., 2003], [Tarakanov et al., 2005] and [Tarakanov, 2008]. This section aims to show the mathematical and the biological background of this method. This method uses a variant of the singular value decomposition transformation described in this chapter for fault vector dimension reduction named formal immune network and a variant of the elimination of auto-reactive immune cells method for fault pattern vectors number reduction named apoptosis and auto-immunization. The special feature of this method is that by the fault pattern vectors number reduction it considers the class of each fault pattern vector in the so called cytokine vector.

Proteins are molecules that are present in many biological processes. One example of such

biological processes is the antibody-antigen interaction following the principles of molecular recognition. Antibodies are proteins produced by immune cells and released into the extracellular fluid. Antibodies have the task of binding antigens and marking them thereby for being removed by other cells. Antigens are also proteins and are placed on the surfaces of pathogens. They received that name from the construction *anti*body *gen*erator, since the binding of an antigen with an antibody placed on the surface of an immune cell, stimulates that immune cell for producing more antibodies for combating the recognized pathogen.

Antibodies placed on the surface of immune cells or released into the extracellular fluid interact with antigens and also with each other building a complex network. That network serves for communicating signals that promote cell growing, cell differentiation, cell removal and cell functioning. A Formal Immune Network intends to model mathematically such a network of interacting proteins starting with the formal model of a protein-protein interaction.

### 4.8.1 Protein-protein interaction formal model

Firstly, two proteins can be defined in a formal way as the vectors $U_{m\times 1}$ and $V_{n\times 1}$, with different number of elements $m$ and $n$. Elements in the vectors represent binding points. Proteins interact with a determined binding energy where: the lower the energy the stronger the binding, [Tarakanov et al., 2003]. That binding energy $w$ can be defined by the bilinear form:

$$w = -U^T_{1\times m}\mathbf{X}_{m\times n}V_{n\times 1} \tag{4.18}$$

A bilinear form is a function represented as $f(U,V) = U^T\mathbf{X}V$ that maps two vectors $U$ and $V$ to one scalar. Function $f(U,V)$ is named bilinear because given an scalar number $\lambda$, it is possible to apply the properties of linearity such as vector addition and scalar multiplication with respect of both vectors $U$ and $V$ to the function $f(U,V)$ as follows:

$$\begin{aligned} f(U_1+U_2,V) &= f(U_1,V)+f(U_2,V) \\ f(U,V_1+V_2) &= f(U,V_1)+f(U,V_2) \\ f(U,\lambda V) &= f(\lambda U,V)+\lambda f(U,V) \end{aligned}$$

The minus sign in equation 4.18 aims to represent the binding energy as a number $w \leq 0$. A positive value in the binding energy would mean repulsion between the two proteins. In the same equation, matrix $\mathbf{X}_{m\times n}$ can be defined as the binding matrix between the formal proteins $U_{m\times 1}$ and $V_{n\times 1}$.

The problem we have now is to determine those formal proteins $U^*_{m\times 1}$ and $V^*_{n\times 1}$ that have the minimal binding energy $w^*$ given a binding matrix $\mathbf{X}_{m\times n}$. That is to say, to find the formal proteins $U^*_{m\times 1}$ and $V^*_{n\times 1}$ that bind the best. Besides, in order to simplify vector operations, it is required that vectors $U_{m\times 1}$ and $V_{n\times 1}$ are orthonormal. That means their norms should be equal to one, $||U|| = \sqrt{U^TU} = 1$ and $||V|| = \sqrt{V^TV} = 1$, leading that the scalar product of the transposed vector $U^T$ or $V^T$ with the vector $U$ or $V$ respectively should be also equal to one, $U^TU = 1$ and $V^TV = 1$. Then, we have a function $f(U,V)$ with two restrictions for minimizing:

$$\begin{aligned} f(U,V) &= -U_{m\times 1}^T \mathbf{X}_{m\times n} V_{n\times 1} \\ U_{m\times 1}^T U_{m\times 1} &= 1 \\ V_{n\times 1}^T V_{n\times 1} &= 1 \end{aligned} \tag{4.19}$$

The minimization problem can be solved minimizing the Lagrange function $\Lambda(U, V, \lambda_1, \lambda_2)$ given below.

$$\Lambda(U, V, \lambda_1, \lambda_2) = f(U, V) + \lambda_1(g(U, V) - c_1) + \lambda_2(h(U, V) - c_2)$$

That Lagrange function $\Lambda(U, V, \lambda_1, \lambda_2)$ presents two constraint functions $g(U, V) = c_1$ and $h(U, V) = c_2$ with respective Lagrange multipliers $\lambda_1$ and $\lambda_2$. Lagrange multipliers serve for scaling constraint functions. Replacing equations 4.19 we get:

$$\Lambda(U, V, \lambda_1, \lambda_2) = -U_{m\times 1}^T \mathbf{X}_{m\times n} V_{n\times 1} + \lambda_1(U_{m\times 1}^T U_{m\times 1} - 1) + \lambda_2(V_{n\times 1}^T V_{n\times 1} - 1)$$

Now, it is necessary to find the gradient of the Lagrange function $\nabla_{U,V,\lambda_1,\lambda_2}\Lambda(U, V, \lambda_1, \lambda_2)$ for finding the extremal points. That gradient should be made equal to zero for finding variables $U$, $V$, $\lambda_1$ and $\lambda_2$ that minimize the function $\Lambda$.

$$\nabla_{U,V,\lambda_1,\lambda_2}\Lambda(U, V, \lambda_1, \lambda_2) = 0$$

Calculating the gradient $\nabla_{U,V,\lambda_1,\lambda_2}\Lambda(U, V, \lambda_1, \lambda_2)$ by the partial differential equations of function $\Lambda(U, V, \lambda_1, \lambda_2)$ respecting each of its variables $U$, $V$, $\lambda_1$ and $\lambda_2$, we get the following system of linear equations:

$$\begin{aligned} \frac{\partial \Lambda}{\partial U} &= 2\lambda_1 U - \mathbf{X}V = 0 \\ \frac{\partial \Lambda}{\partial V} &= 2\lambda_2 V - \mathbf{X}^T U = 0 \\ \frac{\partial \Lambda}{\partial \lambda_1} &= U^T U - 1 = 0 \\ \frac{\partial \Lambda}{\partial \lambda_2} &= V^T V - 1 = 0 \end{aligned}$$

Multiplying the first partial differential equation left by $U^T$ and using the third partial differential equation $U^T U = 1$ we obtain:

$$2\lambda_1 = U^T \mathbf{X} V$$

Furthermore, multiplying the second partial differential equation also left by $V^T$ and using the fourth partial differential equation $V^T V = 1$, we obtain an equation that transposed gives:

$$2\lambda_2 = U^T \mathbf{X} V$$

Last two equations derive $2\lambda_1 = 2\lambda_2$. If we rewrite those two equations into one assigning $2\lambda_1 = 2\lambda_2 = s$, where $s$ is a scalar number we get:

$$s = U^T \mathbf{X} V \tag{4.20}$$

Reminding the definition of singular value decomposition of matrix $\mathbf{X}_{m\times n}$:

$$\mathbf{X}_{m\times n} = s_1 U_1 V_1^T + ... + s_i U_i V_i^T + ... + s_r U_r V_r^T \tag{4.21}$$

where $s_i$ are singular values, $U_i$ left singular vectors, $V_i$ right singular vectors and $r$ rank of the matrix $\mathbf{X}_{m\times n}$. Every singular value in a singular value decomposition of a matrix $\mathbf{X}_{m\times n}$ can also be expressed as:

$$s_i = U_i^T \mathbf{X} V_i \tag{4.22}$$

That expression is very similar to obtained equation 4.20 since in both cases vectors $U$ and $V$ are orthonormal. That confirms that extremal points of a bilinear form $U_{m\times 1}^T \mathbf{X}_{m\times n} V_{n\times 1}$ are determined by the singular value decomposition of the matrix $\mathbf{X}_{m\times n}$, as in [Tarakanov et al., 2003] stated, having:

$$s_i = -w_i^*$$

Which means that all singular values $s_i$ are assumed as the minimal binding energy values $-w_i^*$ for the pairs of formal proteins $U_i^*$ and $V_i^*$ that bind the best given a binding matrix $\mathbf{X}_{m\times n}$. Figure 4.21 shows such pairs of formal proteins $U_i^*$ and $V_i^*$ as a subset of all possible values of formal proteins $U_i$ and $V_i$ that are arguments of the bilinear function $w = f(U, V) = -U_{m\times 1}^T \mathbf{X}_{m\times n} V_{n\times 1}$. Since with singular value decomposition we address only pairs of formal proteins $U_i^*$ and $V_i^*$ that bind the best, the notation $U_i$ and $V_i$, without asterisk $^*$, from now on assumes such pair of formal proteins $U_i^*$ and $V_i^*$ with minimal binding energy $w_i^*$.

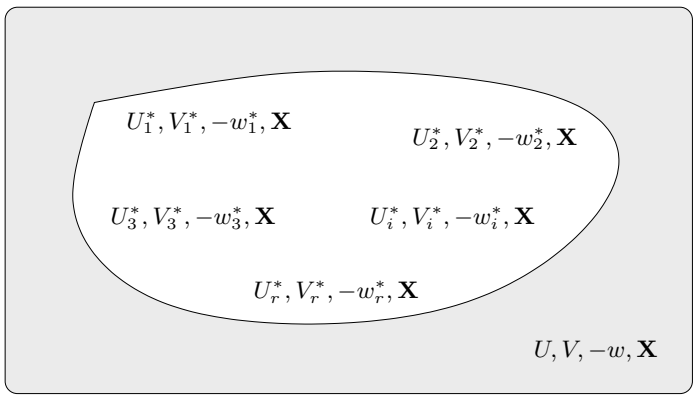



Figure 4.21: Subset of pairs of formal proteins $U^*$ and $V^*$ with minimal binding energy $-w^*$ given a binding matrix $\mathbf{X}_{m\times n}$

### 4.8.2 Formal immune network

The singular value decomposition of a matrix $\mathbf{X}_{m\times n}$ can also be written in a matrix form as:

$$\mathbf{X}_{m\times n} = \mathbf{U}_{m\times m} \mathbf{S}_{m\times n} \mathbf{V}_{n\times n}^T$$

Multiplying both sides of the equation right by $\mathbf{V}_{n\times n}$ and considering $\mathbf{V}_{n\times n}^T \mathbf{V}_{n\times n} = 1$ we obtain:

$$\mathbf{X}_{m\times n}\mathbf{V}_{n\times n} = \mathbf{U}_{m\times m}\mathbf{S}_{m\times n} \tag{4.23}$$

Expressing the matrix equation in terms of vectors $U_i$, $V_i$ and $X_i$ we get:

$$\begin{pmatrix} X_1^T \\ - \\ \vdots \\ - \\ X_m^T \end{pmatrix}_{m\times n} \times \begin{pmatrix} V_1 & | & \cdots & | & V_n \end{pmatrix}_{n\times n} = \begin{pmatrix} U_1 & | & \cdots & | & U_m \end{pmatrix}_{m\times m} \times \begin{pmatrix} s_1 & & & & & & 0 \\ & \ddots & & & & & \\ & & s_i & & & & \\ & & & \ddots & & & \vdots \\ \vdots & & & & s_r & & \\ & & & & & \ddots & \\ & & & & & & 0 \\ & & & & & & \vdots \\ 0 & & \cdots & & & & 0 \end{pmatrix}_{m\times n}$$

Note that for convenience vectors of $n$ dimensions $X_{n\times 1}$ are arranged not as $m$ column vectors in a matrix $\mathbf{X}_{n\times m}$ but as $m$ line vectors in $X^T_{n\times 1}$ in a matrix $\mathbf{X}^T_{n\times m}$. Then, we have as binding matrix $\mathbf{X}_{m\times n}$, the matrix $\mathbf{X}^T_{n\times m}$. This convention helps a better handling during matrix multiplication and eases to find the transformation that we are looking for.

Matrix $\mathbf{S}_{m\times n}$ is shown for $m \geq n$. Note that there exist $r$ non zero singular values $s_i$ placed in the diagonal of matrix $\mathbf{S}_{m\times n}$ of size $min(m,n)$. Number $r$ is the rank of matrix $\mathbf{X}_{m\times n}$ which is less or equal than the minimum between $m$ and $n$, $r \leq min(m,n)$. All other values off the diagonal in matrix $\mathbf{S}_{m\times n}$ are 0.

Now multiplying matrices:

$$\begin{pmatrix} X_1^TV_1 & \cdots & X_1^TV_i & \cdots & X_1^TV_n \\ \vdots & & \vdots & & \vdots \\ X_p^TV_1 & \cdots & X_p^TV_i & \cdots & X_p^TV_n \\ \vdots & & \vdots & & \vdots \\ X_m^TV_1 & \cdots & X_m^TV_i & \cdots & X_m^TV_n \end{pmatrix}_{m\times n} = \begin{pmatrix} U_1s_1 & | & \cdots & | & U_rs_r & | & \cdots & | & U_n0 \end{pmatrix}_{m\times n}$$

And expressing vectors $U_i$ by their elements:

$$\begin{pmatrix} X_1^TV_1 & \cdots & X_1^TV_i & \cdots & X_1^TV_r & \cdots & X_1^TV_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ X_p^TV_1 & \cdots & X_p^TV_i & \cdots & X_p^TV_r & \cdots & X_p^TV_n \\ \vdots & & \vdots & & \vdots & & \vdots \\ X_m^TV_1 & \cdots & X_m^TV_i & \cdots & X_m^TV_r & \cdots & X_m^TV_n \end{pmatrix}_{m\times n} = \begin{pmatrix} U_{11}s_1 & \cdots & U_{i1}s_i & \cdots & U_{r1}s_r & \cdots & U_{n1}0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ U_{1p}s_1 & \cdots & U_{ip}s_i & \cdots & U_{rp}s_r & \cdots & U_{np}0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ U_{1m}s_1 & \cdots & U_{im}s_i & \cdots & U_{rm}s_r & \cdots & U_{nm}0 \end{pmatrix}_{m\times n}$$

Considering $n \leq m$, matrix $\mathbf{U}_{m\times m}$ multiplied by matrix $\mathbf{S}_{m\times n}$ produces a matrix $m \times n$. Further, when the rank of matrix $\mathbf{X}_{m\times n}$ is $r \leq n \leq m$, matrix $\mathbf{U}_{m\times m}\mathbf{S}_{m\times n}$ can be written as a reduced matrix of dimension $m \times r$. Even further away, equation 4.21 has shown that by decomposing matrix $\mathbf{X}_{m\times n}$ into a sum of matrices $s_iL_iR_i^T$ by means of its singular values $s_i$ and right and left singular vectors $V_i$ and $U_i$, the first sum terms contribute the matrix $\mathbf{X}_{m\times n}$ the most because singular values are ordered falling down $s_1 \leq s_2 \leq ... \leq s_r$. Then, when considering only $t$ singular values in the singular value decomposition of matrix $\mathbf{X}_{m\times n}$, with $t \leq r$, it is sufficient to write matrices $\mathbf{U}_{m\times m}\mathbf{S}_{m\times n}$ and $\mathbf{X}_{m\times n}\mathbf{V}_{n\times n}$ of equation 4.23 as matrices of dimensions $m \times t$ obtaining:

$$\begin{pmatrix} X_1^T V_1 & \cdots & X_1^T V_i & \cdots & X_1^T V_t \\ \vdots & & \vdots & & \vdots \\ X_p^T V_1 & \cdots & X_p^T V_i & \cdots & X_p^T V_t \\ \vdots & & \vdots & & \vdots \\ X_m^T V_1 & \cdots & X_m^T V_i & \cdots & X_m^T V_t \end{pmatrix}_{m\times t} = \begin{pmatrix} U_{11}s_1 & \cdots & U_{i1}s_i & \cdots & U_{t1}s_t \\ \vdots & & \vdots & & \vdots \\ U_{1p}s_1 & \cdots & U_{ip}s_i & \cdots & U_{tp}s_t \\ \vdots & & \vdots & & \vdots \\ U_{1m}s_1 & \cdots & U_{im}s_i & \cdots & U_{tm}s_t \end{pmatrix}_{m\times t}$$

Observing carefully the elements of that matrices, vector $X_p^T$ of binding matrix $\mathbf{X}_{m\times n}$ contributes only to components $p$ of all formal proteins $U_i$. Besides, although only $t$ singular values, $t$ lefts singular vectors and $t$ right singular vectors can be seen, all $m$ vectors $X_{n\times 1}^T$ are present. Then looking at the central elements in the matrices, we can derive the following equation:

$$U_{ip} = \frac{1}{s_i} X_p^T V_i \tag{4.24}$$

That equation shows that the components $U_{ip}$ of left singular vectors $U_i$ are a function of the vector $X_p^T$ having $V_i$ and $s_i$, as the following expression illustrates:

$$\begin{pmatrix} \frac{X_1^T V_1}{s_1} & \cdots & \frac{X_1^T V_i}{s_i} & \cdots & \frac{X_1^T V_t}{s_t} \\ \vdots & & \vdots & & \vdots \\ \frac{X_p^T V_1}{s_1} & \cdots & \frac{X_p^T V_i}{s_i} & \cdots & \frac{X_p^T V_t}{s_t} \\ \vdots & & \vdots & & \vdots \\ \frac{X_m^T V_1}{s_1} & \cdots & \frac{X_m^T V_i}{s_i} & \cdots & \frac{X_m^T V_t}{s_t} \end{pmatrix}_{m\times t} = \begin{pmatrix} U_{11} & \cdots & U_{i1} & \cdots & U_{t1} \\ \vdots & & \vdots & & \vdots \\ U_{1p} & \cdots & U_{ip} & \cdots & U_{tp} \\ \vdots & & \vdots & & \vdots \\ U_{1m} & \cdots & U_{im} & \cdots & U_{tm} \end{pmatrix}_{m\times t}$$

And the same expression in terms of the matrix elements of the matrix of left singular vectors $\mathbf{U}_{m\times t}$ for a better understanding given below:

$$\begin{pmatrix} \frac{X_1^T V_1}{s_1} & \cdots & \frac{X_1^T V_i}{s_i} & \cdots & \frac{X_1^T V_t}{s_t} \\ \vdots & & \vdots & & \vdots \\ \frac{X_p^T V_1}{s_1} & \cdots & \frac{X_p^T V_i}{s_i} & \cdots & \frac{X_p^T V_t}{s_t} \\ \vdots & & \vdots & & \vdots \\ \frac{X_m^T V_1}{s_1} & \cdots & \frac{X_m^T V_i}{s_i} & \cdots & \frac{X_m^T V_t}{s_t} \end{pmatrix}_{m\times t} = \begin{pmatrix} \mathbf{U}_{1,1} & \cdots & \mathbf{U}_{1,i} & \cdots & \mathbf{U}_{1,t} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{U}_{p,1} & \cdots & \mathbf{U}_{p,i} & \cdots & \mathbf{U}_{p,t} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{U}_{m,1} & \cdots & \mathbf{U}_{m,i} & \cdots & \mathbf{U}_{m,t} \end{pmatrix}_{m\times t}$$

Then vectors $X_p^T$ can be transformed into vectors formed with that components $U_{ip}$ of left singular vectors $U_i$. Considering that $t < n$, then vectors $X_p$ of dimension $n \times 1$ can be expressed as vectors from now on $R_p$ of dimension $t \times 1$. That means a vector dimension reduction of $X_{n\times 1}$ to $R_{t\times 1}$. We can take either vectors $R_{t\times 1}$ with components $(U_{1p}, ..., U_{ip}, ..., U_{tp})$ from the left singular vectors matrix $\mathbf{U}_{m\times t}$ or recompute component values $U_{ip}$ by means of equation 4.24.

A single vector $R_p$ with $i = 1, ..., t$ components represents all $p$ binding points of $t$ formal proteins $U_i$ that bind with $t$ formal proteins $V_i$ with minimal binding energy $w_i^*$ respectively, having a binding matrix $\mathbf{X}_{m\times n}$. In consequence vectors $R_p$ contain information of a sort of network of interacting formal proteins. That network of formal proteins has been named in [Tarakanov et al., 2003] as a Formal Immune Network and the space where the vectors $R_p$ are drawn is named Formal Immune Network space. The dimension of the space is determined by the number $t$.

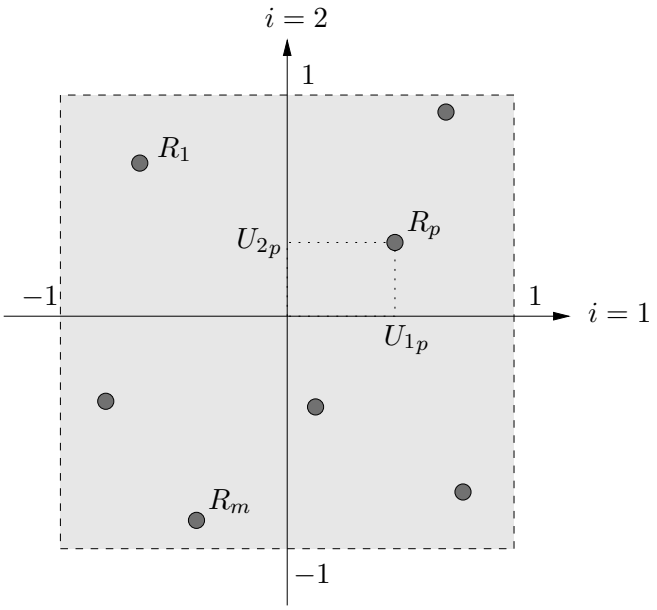



Figure 4.22: Formal immune network space of dimension $t = 2$

As an example we can see in figure 4.22 $m$ vectors $R_p$ in a Formal Immune Network space of dimension $t = 2$. A single vector $R_p = (U_{1p}, U_{2p})$ of that set of $m$ vectors $R_p$ represents the links of formal proteins $U_1$ and $U_2$ that bind to formal proteins $V_1$ and $V_2$ with binding energies $s_1 = -w_1^*$ and $s_2 = -w_2^*$ respectively, given a binding matrix $\mathbf{X}_{m\times n}$. Vectors $R_p$ contain the information of a sort of network of formal proteins $U_1$, $U_2$, $V_1$ and $V_2$.

We can take that theory for reducing the dimension of fault pattern vectors in the following way. Given fault pattern vectors $X_{n\times 1}$ arranged in a matrix $\mathbf{X}^T_{n\times m}$, the task is first to calculate the singular value decomposition of that matrix $\mathbf{X}^T_{n\times m}$. Once having computed the singular values $s_i$, left singular vectors $U_i$ and right singular vectors $V_i$, we can take only $t$ left singular vectors $U_i$. Elements $p$ of left singular vectors $U_i$ are the coordinates of transformed vectors $R_p$. Algorithm 4.6 presents that procedure.

### 4.8.3 Molecular recognition

In last subsection we have found a set of $t$ formal proteins $V_i$ that bind the best with other $t$ formal proteins $U_i$ given a binding matrix $\mathbf{X}_{m\times n}$. Let us consider those $t$ formal proteins $V_i$ as antibodies. Number $t$ determines how many antibodies $V_i$ we have computed with given matrix $\mathbf{X}_{m\times n}$ and the dimension of the space where the binding points of proteins $U_i$ in vectors $R_p = (U_{1p}, ..., U_{tp})$ can be drawn. Antibodies interact with antigens by molecular recognition. If antibodies are represented by proteins $V_i$, then an antigen is represented by the proteins $U_i$. In this method of molecular recognition, an antigen is not given directly as vectors $U_i$ but as a vector $X_g$ of dimension $n \times 1$ of the binding matrix $\mathbf{X}_{m\times n}$. The problem now is to find which network of proteins $U_i$ represent that antigen and interact with the already found network of antibodies $V_i$. With the help of the already computed minimal binding energies $-w_i = s_i$ and the antibodies $V_i$, vector $X_g$ can be transformed to the Formal Immune Network space as vector $R_g$ with coordinates $(U_{1g}, ..., U_{tg})$ using equation 4.24 rewritten for $X_g$ as:

**Algorithm 4.6:** Fault pattern vector dimension reduction by means of a formal immune network

1: Arrange $m$ fault pattern vectors $X_{n\times 1}$ together into matrix $\mathbf{X}_{n\times m}$
2: Calculate the matrix of singular values $\mathbf{S}_{m\times n}$, matrix of left singular vectors $\mathbf{U}_{m\times m}$ and matrix of right singular vectors $\mathbf{V}_{n\times n}$ of matrix $\mathbf{X}^T_{n\times m}$ (The compact singular value decomposition $\mathbf{X}^T_{n\times m} = \mathbf{U}_{m\times t}\mathbf{S}_{t\times t}\mathbf{V}_{t\times n}$ is sufficient, being $t \leq$ rank of matrix $\mathbf{X}^T_{n\times m}$ and $s_1 \geq ... \geq s_t$)
3: Take the $m$ rows of $t$ elements of the matrix of left singular vectors $\mathbf{U}_{m\times t}$ as the $m$ reduced fault pattern vectors $R_p$ of dimension $t \times 1$ as follows:

$$\begin{pmatrix} R_1^T \\ \vdots \\ R_p^T \\ \vdots \\ R_m^T \end{pmatrix}_{m\times t} = \begin{pmatrix} \mathbf{U}_{1,1} & \cdots & \mathbf{U}_{1,i} & \cdots & \mathbf{U}_{1,t} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{U}_{p,1} & \cdots & \mathbf{U}_{p,i} & \cdots & \mathbf{U}_{p,t} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{U}_{m,1} & \cdots & \mathbf{U}_{m,i} & \cdots & \mathbf{U}_{m,t} \end{pmatrix}_{m\times t}$$

$$U_{ig} = \frac{1}{s_i} X_g^T V_i \tag{4.25}$$

If vector $X_g$ is equal to any vector $X_p$ of binding matrix $\mathbf{X}_{m\times n}$, then the singular value decomposition and equation 4.25 applies. But, if $X_g$ is different to any other vector $X_p$ of matrix vector $\mathbf{X}_{m\times n}$, transforming vector $X_g$ to $R_g = (U_{1g}, ..., U_{tg})$ by means of equation 4.25 helps to find the most similar vector $X_p$ of binding matrix vector $\mathbf{X}_{m\times n}$ in the Formal Immune Network space. That is the idea of this method of molecular recognition.

Figure 4.23 shows vector of binding points $R_g = (U_{1g}, U_{2g})$ in a Formal Immune Network space of dimension $t = 2$.

Now the problem is to determine which vector $R_p$ is the most similar to vector $R_g$ in order to recognize the antigen. Similarity is measured by means of distance. Any distance measurement method can be taken for computing the distance among the vector $R_g$ and each vector $R_p$. The vector $R_p$ with the minimal distance to vector $R_g$ is elected, please see figure 4.24.

The method of molecular recognition can be employed for fault recognition in the following way. Given $m$ reduced fault pattern vectors $R_{t\times 1}$ and a fault vector for being recognized $X_g$, first fault vector $X_g$ of dimension $n \times 1$ should be reduced to vector $R_g$ of dimension $t \times 1$. Computing the distance of vector $R_g$ to all reduced fault pattern vectors $R_p$, the fault vector $R_g$ is the type of the reduced fault pattern vector $R_p$ with minimal distance to the reduced fault vector $R_g$. Algorithm 4.7 shows that procedure.

### 4.8.4 Cytokine formal immune network

Cytokines are proteins produced by immune cells and released into the extracellular fluid for communicating with other cells. The production of cytokines by an immune cell is triggered by the interaction of an antibody placed on the surface of the immune cell with an antigen. The cytokine that is released after antibody-antigen binding is dependent on the encountered antigen. Since every vector $X_p$ in a binding matrix $\mathbf{X}_{m\times n}$ has the capability of recognizing

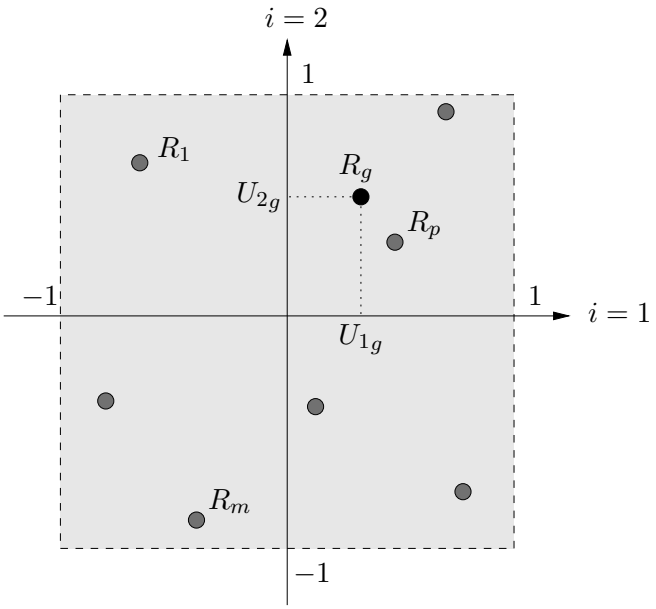



Figure 4.23: Vector $R_g$ in a formal immune network space of dimension $t = 2$

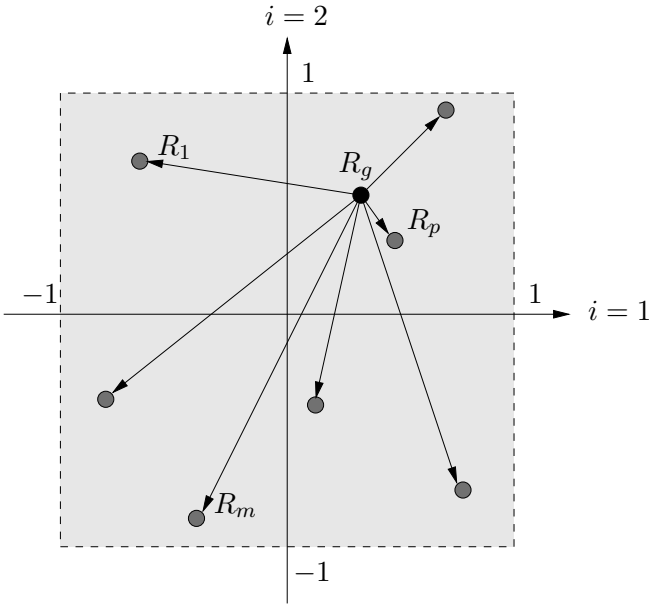



Figure 4.24: Recognition in a formal immune network space of dimension $t = 2$ (adapted from figure in [Tarakanov, 2008] ©2008 Springer)

**Algorithm 4.7:** Fault recognition by means of molecular recognition

1: Reduce fault vector $X_g$ calculating $t$ coordinates of reduced fault vector $R_g$ using:
$$r_{g_i} = \frac{1}{s_i} X_g^T V_i$$
2: Calculate the distance of the reduced fault vector $R_g$ with all other reduced fault pattern vectors $R_p$ using i.e. any of the Minkowski distances by:
$$\text{distance}(R_g, R_p) = (|r_{g_1} - r_{p_1}|^h + ... + |r_{g_i} - r_{p_i}|^h + ... + |r_{g_t} - r_{p_t}|^h)^{\frac{1}{h}}$$
3: Fault pattern vector $X_g$ is the type of the nearest reduced fault pattern vector $R_p$ to $R_g$

an antigen $X_g$, a cytokine Formal Immune Network considers an assigned cytokine, in other words a class $c$, for each vector $X_p$ in the binding matrix $\mathbf{X}_{m \times n}$ or vector $R_p$ in matrix $\mathbf{R}_{m \times t}$. A cytokine or class $c$, can be an integer number or a real number.

That procedure is very helpful for learning algorithms. In a supervised learning procedure, a class $c$ assigned to a vector $X_p$ is inherited by the reduced vector $R_p$. If the classes $c$ are unknown we are in front of an unsupervised learning procedure where the assignment of classes $c$ to vectors $R_p$ can be executed by clustering in the Formal Immune Network space. Clustering means grouping close together vectors $R_p$ into a class $c$.

Molecular recognition in a cytokine Formal Immune Network implies assigning the class $c$ of the nearest vector $R_p$ to vector $R_g$. Other methods for assigning a class $c$ to vector $R_g$ can also be executed such as the nearest neighbor procedure already presented in section 4.3.

A cytokine Formal Immune Network can be employed for fault repairing mechanism assignation when the repairing mechanisms are represented by classes $c$ and assigned by molecular recognition to fault vectors. Algorithm 4.8 presents that procedure.

**Algorithm 4.8:** Fault repairing mechanism assignation by means of a cytokine formal immune network

1: Assign a fault repairing mechanism $c$ to every fault pattern vector $X_p$ of matrix $\mathbf{X}_{n \times m}$ if supervised learning is applied, or to every fault pattern vector $R_p$ in the Formal Immune Network space by clustering if unsupervised learning is applied
2: Calculate $i = 1, ..., t$ coordinates of a given fault vector $X_g$ for getting a reduced fault vector $R_g$ using:
$$r_{g_i} = \frac{1}{s_i} X_g^T V_i$$
3: Calculate the distance of the reduced given fault vector $R_g$ with all other reduced fault pattern vectors $R_p$ using i.e. any of the Minkowski distances by:
$$\text{distance}(R_g, R_p) = (|r_{g_1} - r_{p_1}|^h + ... + |r_{g_i} - r_{p_i}|^h + ... + |r_{g_t} - r_{p_t}|^h)^{\frac{1}{h}}$$
4: Assign the fault repairing mechanism $c$ of the nearest reduced fault pattern vector $R_p$ to the given fault vector $X_g$

### 4.8.5 Apoptosis and auto-immunization

An immune cell whose surface receptors interact with a molecule that belongs to the body can trigger the production of antibodies that are able to interact with more body molecules,

**Algorithm 4.9:** Fault pattern vectors number reduction by means of apoptosis and auto-immunization

1: Arrange the $m$ fault pattern vectors $R_{t\times 1}$ together into matrix $\mathbf{R}_{t\times m}$
2: **for** $e = 1$ **to** $m - 1$ **do**
3: Copy fault pattern vector $e$ of matrix $\mathbf{R}$ into the reduced matrix of fault pattern vectors $\mathbf{Q}$
4: **for** $f = e + 1$ **to** $m$ **do**
5: Calculate the distance of fault pattern vector $R_e$ with fault pattern vector $R_f$ using i.e. any of the Minkowski distances by:
$$\text{distance}(R_e, R_f) = (|r_{e1} - r_{f_1}|^h + ... + |r_{ei} - r_{f_i}|^h + ... + |r_{et} - r_{f_t}|^h)^{\frac{1}{h}}$$
6: **if** distance$(R_e, R_f) <$ threshold **and** $c_e = c_f$ **then**
7: Copy fault pattern vector $R_f$ into the matrix of deleted fault pattern vectors $\mathbf{D}$
8: **else**
9: Copy fault pattern vector $R_f$ into the reduced matrix of fault pattern vectors $\mathbf{Q}$
10: **end**
11: **end**
12: Adopt matrix of fault pattern vectors $\mathbf{Q}$ as the new matrix of fault pattern vectors $\mathbf{R}$ for reducing in the next loop
13: Calculate the new number of fault pattern vectors $m$ of the new matrix $\mathbf{R}$
14: Maintain $e$ fault pattern vectors in the reduced matrix $\mathbf{Q}$ and remove the rest fault pattern vectors
15: **end**
16: Have ready $m - z$ deleted fault pattern vectors $D_{t\times 1}$ together into matrix $\mathbf{D}_{t\times m-z}$ and $z$ fault pattern vectors $Q_{t\times 1}$ together into matrix $\mathbf{Q}_{t\times z}$
17: **for** $e = 1$ **to** $m - z$ **do**
18: **for** $f = 1$ **to** $z$ **do**
19: Calculate the distance of deleted fault pattern vector $D_e$ with fault pattern vector $Q_f$ using i.e. any of the Minkowski distances by:
$$\text{distance}(D_e, Q_f) = (|d_{e1} - q_{f_1}|^h + ... + |d_{ei} - q_{f_i}|^h + ... + |d_{et} - q_{f_t}|^h)^{\frac{1}{h}}$$
20: Save the distance together with the index $f$
21: **end**
22: Choose the lowest distance and its respective index $f$
23: **if** $c_e \neq c_f$ **then**
24: Insert fault patter vector $D_e$ in the reduced matrix of fault pattern vectors $\mathbf{Q}$
25: **end**
26: **end**

killing the body in that way. The immune system get rid of those auto-reactive immune cells programming their death. A programmed death of cells is named apoptosis. In the Formal Immune Network space vectors $R_f$ with class $c_f$ that are near enough to another vector $R_e$ with the same class $c_e$ are removed form the set. That procedure is named apoptosis and it tries to get a reduced set of vectors $\{Q\}$ eliminating redundant vectors $R_{t\times 1}$ in the set $\{R\}$ in order to accelerate molecular recognition. Since each vector $R_e$ should be compared using a determined distance measurement method with all other vectors $R_f$ in the Formal Immune Network space, the resulting reduced set $\{Q\}$ changes according with the initial order of vectors $R_{t\times 1}$ in matrix $\mathbf{R}_{t\times m}$ before comparison. Therefore, the cytokine Formal Immune Network method proposes correcting the wrong removal of vectors $R_e$ performing a second comparison between removed vectors $D_{t\times 1}$ and vectors $Q_{t\times 1}$ in the actual reduced set $\{Q\}$. Vector $D_e$ with class $c_e$ of the set of deleted vectors $\{D\}$ is inserted into the set $\{Q\}$ if the nearest vector $Q_f$ in the reduced set $\{Q\}$ has a different class $c_f$ assigned. That method has been named as auto-immunization since it is indeed the insertion of vectors $D_{t\times 1}$ which have been marked already as auto-reactive. Algorithm 4.9 presents the procedure for reducing the number of fault pattern vectors by apoptosis and auto-immunization.

## 4.9 Conclusions

This chapter presented fault pattern recognition and repairing mechanism assignment after giving a fault representation convention. Because of memory and fault recognition time constraints it is required to have a set able to recognize fault pattern vectors the best as possible with the minimum information as possible. For that, dimension reduction of fault pattern vectors and reduction of the number of fault pattern vectors have been presented as the main objectives to follow. Dimension reduction can be obtained by methods such as the Principle Component Analysis or the Formal Immune Network. Principle Component Analysis offers a way to reduce the dimension of fault pattern vectors minimizing the covariance matrix, which implies to compute the eigenvalue decomposition of the covariance matrix. It has been demonstrated that using the Singular Value Decomposition of the set of fault pattern vectors lead to the same result. A Formal Immune Network reduces fault pattern vectors also using Singular Value Decomposition, however the computation of reduced fault pattern vectors has another approach. That method takes as reduced coordinates of the fault pattern vectors elements of the matrix of left singular vectors. Methods of fault pattern vectors number reduction have been also presented. Those resemble some biological processes. Please have in mind that the method named cytokine Formal Immune Network method, presented extensively in section 4.8 takes additionally into consideration the assignment of classes to the fault pattern vectors, which allows performing repairing mechanism assignment after fault pattern recognition. The cytokine Formal Immune Network method executes both, fault vector dimension reduction and fault pattern vectors number reduction, using variants of the methods presented in sections 4.6 and 4.7, thus it has been devoted a single section for that method. The algorithms presented in this chapter have been elaborated in the context of this thesis with the sight to fault recognition. Those algorithms are implemented and analyzed for some applications in next chapter.

## 4.10 Bibliography

Amaral, J. L. M. (2011). Fault Detection in Analog Circuits Using a Fuzzy Dendritic Cell Algorithm. In *10th International Conference on Artificial Immune Systems - ICARIS 2011*. Springer.

Fodor, I. K. (2002). A survey of dimension reduction techniques. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.

Krishnan, S. and Kerkhoff, H. G. (2012). A Robust Metric for Screening Outliers from Analogue Product Manufacturing Tests Responses. In *6th European Test Symposium - ETS 2011*. IEEE.

Mardia, K. V., Kent, J. T., and Bibby, J. M. (1979). *Multivariate Analysis*. Probability and Mathematical Statistics. Academic Press.

Oja, E. (2003). Principal Component Analysis. In Arbib, M. A., editor, *The Handbook of Brain Theory and neural Networks*. The MIT Press.

Tarakanov, A., Goncharova, L., and Tarakanov, O. (2005). A Cytokine Formal Immune Network. In Capcarrère, M., Freitas, A., Bentley, P., Johnson, C., and Timmis, J., editors, *Advances in Artificial Life*, volume 3630 of *Lecture Notes in Computer Science*, pages 510–519. Springer.

Tarakanov, A. O. (2008). Formal Immune Networks: Self-Organization and Real-World Applications. In Wu, X. and Prokopenko, M., editors, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 271–290. Springer.

Tarakanov, A. O., Skormin, V. A., and Sokolova, S. P. (2003). *Immunocomputing, Principles and Applications*. Springer.

Theodoridis, S. (2009). *Introduction to Pattern Recognition: A MATLAB Approach*. Academic Press.

Theodoridis, S. and Koutroumbas, K. (2008). *Pattern Recognition*. Academic Press, 4 edition.

Timmis, J. and Neal, M. (2001). A resource limited artificial immune system for data analysis. *Knowledge Based Systems*, 14(3-4):121–130. Elsevier.

Wikipedia (2010). Searched words: dimension reduction, singular value decomposition, Karhunen-Loève theorem, Hamming distance, feature selection k-nearest neighbor algorithm, Mahalanobis distance, pattern recognition, pattern matching, machine learning, molecular recognition, bilinear form, Lagrange multiplier.

# 5

# Evaluation of fault recognition methods

The circuit for self-repairing, shown in figure 4.1, can be a digital combinational circuit, a digital sequential circuit, a complex digital system or a sampled analog system, as presented in detail in section 4.1. On the first side, digital combinational circuits and digital sequential circuits are able to provide to the fault recognition module one line digital signals containing binary values, as can be seen in figure 5.1. On the other side, complex digital systems and sampled analog systems are able to provide to the fault recognition module multiple line digital signals containing real values, in a number representation format i.e. fixed or floating point format, as can be seen in figure 5.2.

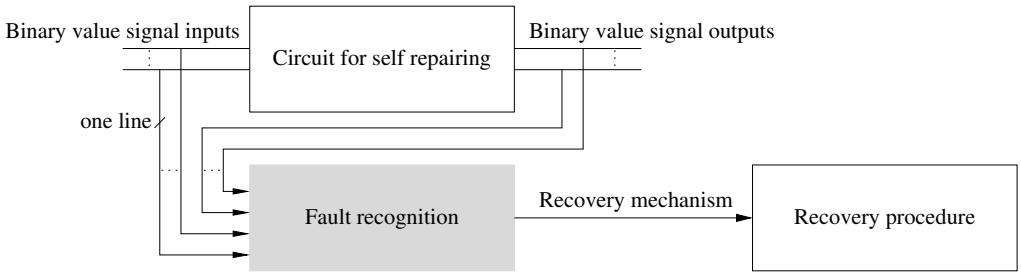



Figure 5.1: Binary value inputs and outputs to the fault recognition module

Since fault vectors in the fault recognition module are formed aggregating the inputs and

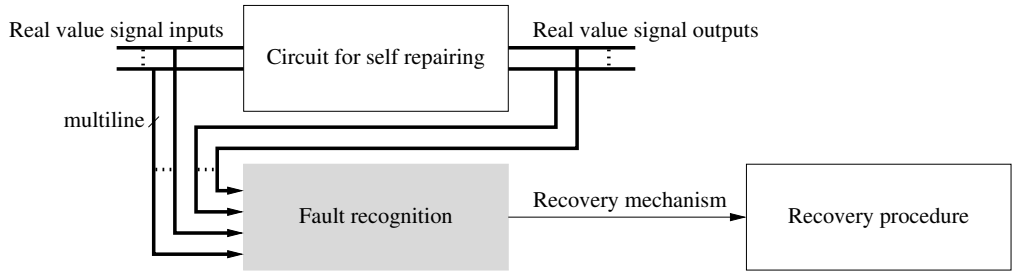



Figure 5.2: Real value inputs and outputs to the fault recognition module

outputs of the circuit for self-repairing, as can be seen in figures 5.1 and 5.2, the type of the signals provided by the circuit for self-repairing to the fault recognition module determines the type of the fault vector elements. Consequently, considering the value of the inputs and outputs of the circuit for self-repairing there are two fault vector types, fault vectors with binary value elements and fault vectors with real value elements, as can be seen in figure 5.3.

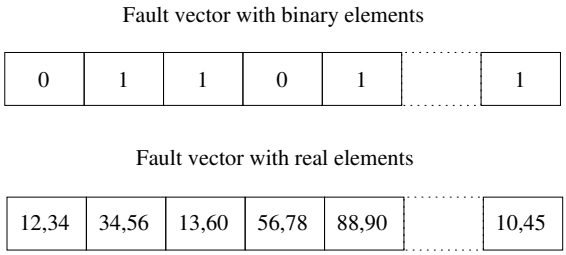



Figure 5.3: Fault vector types

Last chapter presented issues related to the design of the fault recognition module for a self-repairing system theoretically. In order to evaluate the exposed fault recognition methods, this chapter presents the same issues but practically. Since the implementation of algorithms for fault recognition, fault vector dimension reduction and fault pattern vectors number reduction are different for binary and real fault vector elements, they are presented separately in the next two sections and in the last section conclusions are derived.

## 5.1 Fault recognition module with real fault vector elements

This section presents the implementation of algorithms for the design of a fault recognition module which gets from the circuit for self-repairing real value signal inputs and outputs. Those real value signals become the elements of the fault vector in the fault recognition module. It is important to note that the arrangement of signals within a fault vector for the design of a fault recognition module is application dependent. Some ideas over the arrangement of signals into a fault vector have been given in section 4.1.

In order to show the output of the implemented algorithms, a set of 87 fault pattern vectors with real value elements has been made available. The set of fault pattern vectors comes from a real application. Every single fault pattern vector of the set is formed with samples of three sensor analog signals provided by a wire-bonding machine. The samples are placed as elements in the fault pattern vector which has the following fault vector arrangement: [*wire-bond failure*

*value | first sensor samples, second sensor samples, third sensor samples*]. The first sensor provides with 380 samples, the second sensor with 80 samples and the third sensor with 380 samples. The *wire-bond failure value* contains the value assigned to one of three wire-bond failures. A single fault pattern vector is shown in figure 5.4. Note that the *wire-bond failure* value has been taken out for the graphic.

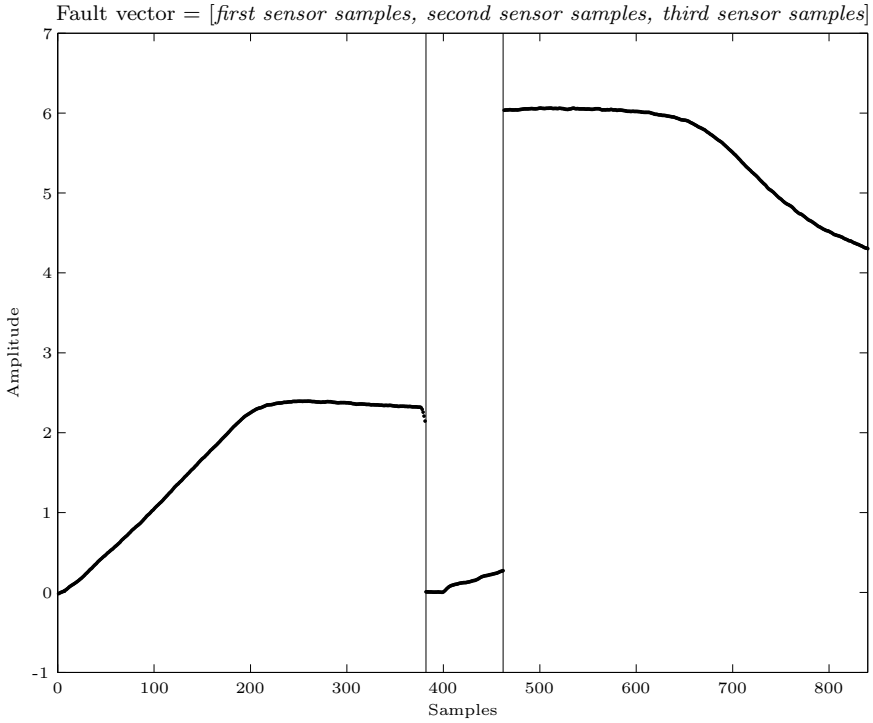



Figure 5.4: Fault vector arrangement for a wire-bonding machine application

The available set contains 40 different fault pattern vectors corresponding to acceptable wire-bonds, 40 different fault pattern vectors corresponding to wire-bonds failed because of a contaminated bonding surface and 7 different fault pattern vectors corresponding to wire-bonds failed because of bad placement of the bonding point. For the evaluation of the algorithms another set with 113 fault vectors has been also made available. The available set contains 46 different fault pattern vectors corresponding to acceptable wire-bonds, 60 different fault pattern vectors corresponding to wire-bonds failed because of a contaminated bonding surface, and 7 different fault pattern vectors corresponding to wire-bond failed because of bad placement of the bonding point. Table 5.1 shows the number of fault vectors per wire-bond failure in the available data for the design of the fault recognition module and for its testing.

Methods for fault recognition, fault vector dimension reduction and fault pattern vectors number reduction presented in sections 4.5, 4.6 and 4.7 were implemented in Matlab®. The implementation and evaluation of those methods, using the available data, are presented in next subsections. Please note that the results to be presented are specifically for the available data, nevertheless, the same methods can be used for any other set of fault pattern vectors coming from a different application.

Table 5.1: Available fault vectors with real elements

| Class value | Wire-bond failure | Design | Test |
|---|---|---|---|
| 0 | Acceptable wire-bond | 40 | 46 |
| 1 | Failed wire-bond - contaminated surface | 40 | 60 |
| 2 | Failed wire-bond - bad placement | 7 | 7 |
| | Total vectors | 87 | 113 |
| | Data matrix size | $840 \times 87$ | $840 \times 113$ |

### 5.1.1 Fault recognition

For recognizing a fault, we require a method for determining that a given fault vector represents a fault. For that purpose, section 4.2 presented different distance measurement methods. Those distance measurement methods measure the degree of similarity of the given fault vector to each fault pattern vector of a stored set of fault pattern vectors. However, it is important to have in mind that the similarity is inversely proportional to the measured distance. For real fault vectors, the Minkowski distance, the Euclidean distance, the Manhattan distance and the Chebyshev distance have been programmed as functions in Matlab. The Matlab program codes of those functions are shown in the code listing 5.1.

Program Code 5.1: Minkowski, Euclidean, Manhattan and Chebyshev distance functions

```
function [distance] = distanceminkowski(a,b,n)
distance = ( sum( ( a - b ).^n ) ).^( 1/n );

function [distance] = distanceeuclidean(a,b)
distance = sqrt( sum( ( a - b ).^2 ) );

function [distance] = distancemanhattan(a,b)
distance = sum( abs( a - b ) );

function [distance] = distancechebyshev(a,b)
distance = max( abs( a - b ) );
```

Please note that although all functions are written together for prettiness of the page, each function is implemented as a single Matlab file. The function definition contains the reserved word **function** next to the outputs of the function in square brackets. The equal sign is followed by the name of the function next to the inputs to the function in round brackets. In these functions, the inputs are represented by the variables a, b and n. Functions taken from the Matlab library appear in bold face in the code listing, i.e. **sum**, **abs** and **sqrt**. For more information over those functions, please refer to the online documentation of Matlab.

Additionally, for determining that a given fault vector represents a fault, there is also the possibility of measuring the similarity using the Mahalanobis distance. Since each fault pattern vector has assigned a fault class, the whole fault pattern vector set mX can be partitioned into subsets mXc. One subset for each fault class. The Mahalanobis distance measures the distance of the given fault vector a to any subset of fault patten vectors mXc. For that, the inputs to the Mahalanobis distance function, shown in code listing 5.2, are the given fault vector a, the mean vector meanmXc of the subset mXc computed in Matlab by meanmXc = **mean**(mXc,2),

and the covariance matrix covmXc computed using the function displayed in code listing 5.3. Please note that the covariance matrix is computed using equations 4.15 and 4.16, already explained in chapter 4. Note also that the mean vector and covariance matrix are computed for the respective subset of fault pattern vectors mXc, not for the whole set mX.

Program Code 5.2: Mahalanobis distance function

```
function [distancetoc] = distancemahalanobis(a,meanmXc,covmXc)
distancetoc = sqrt( ( ( a - meanmXc )'/covmXc )*( a - meanmXc ) );
```

Program Code 5.3: Covariance matrix function

```
function [covmXc] = covariance(mXc)
mdfmeanmXc = mXc - ( mean(mXc,2) * ones(1,size(mXc,2)) );
covmXc = ( mdfmeanmXc * mdfmeanmXc' )/size(mXc,2);
```

Section 4.2 in chapter 4 presented many formulas for computing the Mahalanobis distance. The selection of the formula depends on the covariance matrix of the addressed subset of fault pattern vectors. Usually the covariance matrix of a real fault pattern vector subset is far from an identity matrix or a diagonal matrix of variances, like shown in formula 4.7 and subsequent paragraph. Therefore, firstly, the Mahalanobis distance function has been implemented using the most general formula 4.6.

Once having the distances from the given fault vector to each fault pattern vector of the whole set of fault pattern vectors, assigning a fault class to the given fault vector is the next task. That task can be executed by assigning the class of the nearest fault pattern vector or applying the k-nearest neighbor algorithm explained in section 4.3. In the case of using the Mahalanobis distance method, the fault class of the subset that reported the minimum distance to the given fault vector is assigned to that fault vector.

Code listing 5.4 shows the Matlab function for assigning the fault class of the nearest fault pattern vector to the given fault vector. Note that the variable input variable distances, is a vector that contains the distances of the given fault vector to all fault pattern vectors in the set of fault pattern vectors mX. Additionally, vc is a vector that contains the fault classes of all fault pattern vectors in the set mX. In the end, the variable faultvectorclass gets the fault class of the nearest fault pattern vector to the given fault vector.

Program Code 5.4: Nearest neighbor class function

```
function [faultvectorclass] = nearestneighborclass(distances,vc)
[~,minplace] = min(distances);
faultvectorclass = vc(minplace);
```

The k-nearest neighbor algorithm is a bit more complicated. Code listing 5.5 presents the Matlab program code of that algorithm as a function. The function gives as outputs k nearest neighbors. That means, k fault pattern vectors with the minimal distance to the given fault vector. The inputs to the function are the distances of the given fault vector to all fault pattern vectors into the vector distances, the vector vc with the classes of all fault patten vectors in the set mX, the number of nearest neighbors to search for in variable k, the number of classes in variable numberclasses and the vector classesvector with the numerical values assigned for every existing class. The function **min** in Matlab gives the value and place of the minimal element in a vector. With that function it is possible to find the place of the

nearest fault pattern vector and copy its class in vector nearestneighborsclasses. That process is shown in lines 5 to 10 of the code listing 5.5. In order to find the k most nearest fault pattern vectors, the value of the minimal distance in the vector distances is replaced with the value of the maximal distance. Then, it is possible to find the place of the next most nearest fault pattern vector. Now, the next task is to find the most common class in the vector nearestneighborsclasses. That process is shown in code lines 12 to 17 of the code listing 5.5. Given two vectors, the Matlab function **eq** gives a vector with elements 1 for equal vector elements, otherwise 0. Using that function and function **sum**, it is possible to get in the vector nearestneighborsperclass the number of nearest neighbors of each fault class. Then, with the help of function **max**, the most common class is found and assigned to the given fault vector in the variable faultvectorclass.

Program Code 5.5: k-nearest neighbors function

```
1  function [faultvectorclass] = knearestneighborsclass(distances,vc,k,\
2                                                        numberclasses,
3                                                        classesvector)
4  nearestneighborsclasses = zeros(1,k);
5  for i = 1:k
6     [~,minplace] = min(distances);
7     nearestneighborsclasses(i) = vc(minplace);
8     [maxvalue,~] = max(distances);
9     distances(minplace) = maxvalue;
10 end
11 nearestneighborsperclass = zeros(1,numberclasses);
12 for j = 1:numberclasses
13    nearestneighborsperclass(j) = sum( eq( (c*ones(1,k)),\
14                                           nearestneighborsclasses ) );
15 end
16 [~,maxplace] = max(nearestneighborsperclass);
17 faultvectorclass = classesvector(maxplace);
```

Now, using the available data summarized in table 5.1, it is possible to compare all distance measurement methods and class assignation methods towards fault recognition. For that, the class of every of the 113 fault vectors of the test set were searched using the 87 fault pattern vectors of the design set. Table 5.2 shows the number of wrong class recognitions from the 113 recognized fault vector classes. Likewise, the number of wrong class recognitions per existing class, that means classes 0, 1 and 2, are also shown in that table. Class recognition using the nearest neighbor class assignation method and the Manhattan distance measurement method reports to be the best, having failed only once in recognizing the class of the 113 test vectors. On the contrary, class recognition using the minimal distance class assignation method and the Mahalanobis distance measurement method failed 57 times, that means that only 56 test vectors were recognized correctly. Finally, the nearest neighbor class assignation method reported a better recognition comparing with the k-nearest neighbor class assignation method for $k = 3$ neighbors.

Table 5.3 is an extended version of table 5.2. It also includes the number of wrong class recognitions using the Mahalanobis distance measurement method with the following covariance matrices covmXc: all equal to the identity matrix (Mahalanobis*) and all matrices with the variances in the diagonal and zeros off the diagonal (Mahalanobis**). Those forms of Mahalanobis distances have been explained in detail in section 4.2 of chapter 4. There, it

Table 5.2: Wrong class recognitions per distance measurement and class assignation method

| Class assignation method | | Nearest neighbor | | | | k-nearest neighbor | | | | Minimal distance |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | k = 3 | | | | |
| Distance measurement method | | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis |
| Wrong class recognitions | Total | 1 | 3 | 5 | 7 | 4 | 5 | 7 | 9 | 57 |
| | Class 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 8 |
| | Class 1 | 0 | 0 | 2 | 3 | 0 | 0 | 1 | 3 | 43 |
| | Class 2 | 0 | 2 | 2 | 4 | 4 | 5 | 6 | 6 | 6 |
| Total test vectors | | 113 | | | | | | | | |

has also been mentioned that the Mahalanobis distance can also be applied for measuring the distance of a fault vector to a fault pattern vector that belongs to a subset of fault pattern vectors with covariance covmXc, as shown in formula 4.8. Table 5.3 also comprises the number of wrong class recognitions for that case (Euclidean*) and additionally the case taking all covariances matrices covmXc with the variances in the diagonal and zeros of the diagonal (Euclidean**), case known as the normalized Euclidean distance.

Table 5.3 shows on one side a high number of wrong class recognitions using the distance measurement methods that apply the whole covariance matrices, such is the case of the methods Mahalanobis and Euclidean*. On the other side, a low number of wrong class recognitions can be seen using the Manhattan distance measurement method and the distance measurement methods that considered a covariance matrix with the variances in the diagonal and zeros off the diagonal. The effect of such diagonal matrix is that the computation of the distances takes place with standardized variables, which are in fact dimensionless. That is to say, instead of using the difference $x_{g_i} - x_{p_i}$ of the variable values in the Euclidean distance $(\sum_{i=1}^{n}(x_{g_i}-x_{p_i})^2)^{\frac{1}{2}}$, normalized variable values in the form $\frac{x_{g_i}-x_{p_i}}{\sigma_i} = \frac{x_{g_i}}{\sigma_i} - \frac{x_{p_i}}{\sigma_i}$ are applied getting $(\sum_{i=1}^{n}(\frac{x_{g_i}-x_{p_i}}{\sigma_i})^2)^{\frac{1}{2}}$, where $\sigma_i$ is the standard deviation for variable $i$. The standard deviation computed by $\sigma_i = (\frac{1}{m}\sum_{j=1}^{m}(x_{i,j} - \mu_i)^2)^{\frac{1}{2}}$, where $\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_{i,j}$ is the mean value, is very similar to the vector norm $\|x_{i,j} - \mu_i\| = (\sum_{j=1}^{m}(x_{i,j} - \mu_i)^2)^{\frac{1}{2}}$, considering as vector all $i$ components of the $m$ available fault pattern vectors with its mean taken off $[x_{i,1} - \mu_i, x_{i,2} - \mu_i, ..., x_{i,m} - \mu_i]$. The only difference is the factor $\frac{1}{m}$ in the standard deviation. In a more generalized form, the p-norm defined as $\|x_{i,j} - \mu_i\| = (\sum_{i=0}^{m}(x_{i,j} - \mu_i)^p)^{\frac{1}{p}}$ could also be used, as shown later on in table 5.4. In the Mahalanobis distance that uses a covariance matrix with the variances in the diagonal and zeros off the diagonal (Mahalanobis**), the following normalization takes

Table 5.3: Wrong class recognitions using variants of the Mahalanobis distance

| Class assignation method | | Nearest neighbor | | | | | | k-nearest neighbor | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | covmXc | | | | | | covmXc | | covmXc | | |
| Distance measurement method | | Manhattan | Euclidean | Minkowski | Chebyshev | Euclidean* | Euclidean** | Manhattan | Euclidean | Minkowski | Chebyshev | Euclidean* | Euclidean** | Mahalanobis | Mahalanobis* | Mahalanobis** |
| | | | | | | | $diag(\sigma_{c1}^2, ..., \sigma_{ci}^2, ..., \sigma_{cn}^2)$ | | | | | | $diag(\sigma_{c1}^2, ..., \sigma_{ci}^2, ..., \sigma_{cn}^2)$ | | $I$ | $diag(\sigma_{c1}^2, ..., \sigma_{ci}^2, ..., \sigma_{cn}^2)$ |
| | Total | 1 | 3 | 5 | 7 | 57 | 1 | 4 | 5 | 7 | 9 | 57 | 1 | 57 | 7 | 1 |
| Wrong class recognitions | Class 0 | 1 | 1 | 1 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 8 | 0 | 0 |
| | Class 1 | 0 | 0 | 2 | 3 | 43 | 1 | 0 | 0 | 1 | 3 | 43 | 0 | 43 | 3 | 0 |
| | Class 2 | 0 | 2 | 2 | 4 | 6 | 0 | 4 | 5 | 6 | 6 | 6 | 1 | 6 | 4 | 1 |
| Total test vectors | | 113 | | | | | | | | | | | | | | |

place $\frac{x_{gi}-\mu_i}{\sigma_i} = \frac{x_{gi}}{\sigma_i} - \frac{\mu_i}{\sigma_i}$, which is also known as the standard score. The standard score measures how many standard deviations, the measured variable is above or below the mean. In conclusion, to work by the distance measurement and subsequent class assignation with normalized variables had a positive effect in the recognition, producing thereby a low number of wrong class recognitions.

Table 5.4: Wrong class recognitions using normalization through the p-norm

| Class assignation method | | Nearest neighbor | | | | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | standard score | | |
| Distance measurement method | | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | | | | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | | | | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| | Total | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | | | | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | | | | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| Wrong class recognitions | Class 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 3 | 1 | 0 | 0 | 7 | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | | | | 0 | 0 | 0 | 5 | 8 | 0 | 0 |
| | Class 1 | 0 | 0 | 2 | 3 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | | | | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | | | | 0 | 0 | 0 | 0 | 43 | 3 | 0 |
| | Class 2 | 0 | 2 | 2 | 4 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | | | | 4 | 5 | 6 | 6 | 1 | 1 | 1 | 2 | | | | 1 | 1 | 1 | 3 | 6 | 4 | 1 |
| Total test vectors | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

In the last paragraph, the normalized Euclidean distance has been presented as a variant of the Mahalanobis distance. In the same way, Manhattan, Minkowski and Chebyshev distances can be normalized with the help of the standard deviation $\sigma$ or its similar, the Euclidean norm. Besides, a generalized form of norm, named the p-norm, can be used for normalizing

the distances. The p-norm is a general form of norm, the same way as the Minkowski distance is a general form of distance. Using that p-norm with $p = 1$, the Manhattan distance can be normalized. The same way, using the p-norm with $p = \infty$, the Chebyshev distance can be normalized. Table 5.4 has been elaborated normalizing the distance measurement through both ways. Firstly, using the normalized distance measurement methods through the standard deviation, also known as Euclidean norm or p-norm with $p = 2$. Secondly, using the normalized distance measurement methods through the p-norm, applying the $p$ employed in the respective distance measurement method i.e. Manhattan: $p = 1$ for the distance measurement method and $p = 1$ for the norm, similarly Chebyshev: $p = \infty$ for the distance measurement method and $p = \infty$ for the norm.

Table 5.4 shows that the class recognition improves using such normalized distances. The number of wrong class recognitions drops even to zero using the normalized Minkowski distance measurement method and the nearest neighbor class assignation method. Furthermore, the recognition per class looks to be independent of the number of fault pattern vectors per class available in the design set by using the nearest neighbor class assignation method and normalized distance measurement methods. Next section shows how to accelerate the recognition time reducing the dimensions of the fault vectors. That means, the distance measurement methods can be applied for less dimensions, speeding up the class recognition in that way.

### 5.1.2 Fault vector dimension reduction

The total time taken for computing the distances between a given fault vector to each of the stored fault pattern vectors, depends on the number of components of the given fault vector and the stored fault pattern vectors in consideration. A way of reducing that time is reducing the number of components of those vectors. The number of components of a vector is also called dimension, since it is the maximal number of dimensions in which the vector can be represented graphically. That is why, the process of reducing the components of a vector is called vector dimension reduction. The Matlab program codes for all methods of fault vector dimension reduction presented in section 4.6 of chapter 4 are presented in the following sections of this subsection. Furthermore, using the available data summarized in table 5.1, the implemented algorithms for fault vector dimension reduction could be tested. The obtained results are presented at the end in a comparative manner by meas of a table.

#### Principal component analysis

Principal component analysis tries to reduce the number of components of vectors performing a linear transformation which produces vectors with fewer components, called principal components. The principal components contain the most of the information that a vector comprises. There are two ways of finding those principal components. One way is performing eigenvalue decomposition on the covariance matrix covmX of the matrix mX formed with the set of fault pattern vectors. And the other way is performing singular value decomposition on the matrix of deviations from the mean mmeanmX also of the matrix mX formed with the set of fault pattern vectors. The Matlab program code for both forms of performing such transformation is shown in code listings 5.6 and 5.7.

Program Code 5.6: PCA transformation using eigenvalue decomposition function

```
1 function [mR,W] = transfpcacoveigs(mX,covmX,maintainedvariance)
```

```
2  [W,E] = eigs(covmX,rank(covmX));
3  totalvariance = sum(diag(E));
4  t = 1;
5  partialvariance = (E(t,t)/totalvariance);
6  while partialvariance < maintainedvariance
7      t = t + 1;
8      partialvariance = partialvariance + (E(t,t)/totalvariance);
9  end
10 if t == 1
11     t = 2;
12 end
13 W = W(:,1:t);
14 mR = W' * mX;
```

Code listing 5.6 shows the implementation of algorithm 4.1: "Fault pattern vector dimension reduction using principal component analysis and the covariance" presented in section 4.6.1 of chapter 4. That algorithm has been implemented as the function transfpcacoveigs. The outputs that the function can hand out are: the transformed matrix mR, and the transformation matrix W. The function requires as inputs: the matrix to be transformed mX, the covariance covmX of the matrix mX, and the expected maintained variance between 0 and 100 in the argument maintainedvariance. The maintained variance relates to the amount of information which should be maintained in the reduced matrix mR, where a value of 100 produces a matrix mR equal to the original matrix mX, that is to say no information is lost and the whole variance is conserved.

The function **eigs** in line 2 of code listing 5.6 computes the eigenvalues and eigenvectors of the covariance matrix covmX of matrix mX and gives them out in matrices E and W respectively. The number of eigenvalues and eigenvectors computed by the function **eigs** is equal to the rank of the matrix covmX, which is the maximum possible number and can be given in the second argument of that function. However, the number of eigenvectors t taken into account in the transformation matrix W is computed according to the expected maintained variance in code lines 3 to 12. Then, the new compressed matrix mR in code line 14 comes out from transforming matrix mX with a transformation matrix W. That transformation matrix contains only the first t eigenvectors that correspond to the biggest eigenvalues in matrix E.

In contrast to algorithm 4.1, code listing 5.6 does not implement the steps for computing the covariance covmX of matrix mX, since that computation can be done using the covariance matrix function explained already in detail and shown in code listing 5.3. Besides, it is not necessary to rearrange the eigenvalues in the diagonal matrix E in a decreasing manner and to move their respective eigenvectors in matrix W accordingly, since the function **eigs** gives out the eigenvalues in matrix E already arranged in a decreasing manner.

Program Code 5.7: PCA transformation using singular value decomposition function

```
1  function[mR,W] = transfpcacovsvd(mX,mdfmeanmX,maintainedvariance)
2  [U,S,~]=svd(mdfmeanmX);
3  E = (S.^2)./size(mX,2);
4  totalvariance = sum(diag(E));
5  t = 1;
6  partialvariance = (E(t,t)/totalvariance);
7  while partialvariance < maintainedvariance
8      t = t + 1;
```

```
9       partialvariance = partialvariance + (E(t,t)/totalvariance);
10  end
11  if t == 1
12      t = 2;
13  end
14  W = U(:,1:t);
15  mR = W' * mX;
```

Code listing 5.7 shows the implementation of algorithm 4.2: "Fault pattern vector dimension reduction using principal component analysis and singular value decomposition" presented in section 4.6.1 of chapter 4. That algorithm has been implemented as the function transfpcacovsvd. The outputs and inputs of that function are the same comparing with the function transfpcacoveigs presented in code listing 5.6, except for the input mdfmeanmX. That is because, instead of computing the eigenvalues and eigenvectors of the covariance covmX of matrix mX, the singular values and singular vectors of the matrix of deviations from the mean mdfmeanmX of matrix mX are computed.

The function **svd** in line 2 of code listing 5.7 computes the singular values and the left singular vectors of matrix mdfmeanmX and gives them out in matrices S and U respectively. Note that the output corresponding to the matrix of right singular vectors has been replaced by a symbol because that matrix is not needed. The number t of left singular vectors of matrix U, to take into account into the transformation matrix W, is computed in code lines 4 to 13. For that computation, it is necessary to have the matrix of eigenvalues E, which is possible to get with the equation 4.17 implemented in code line 3 and explained extensively in section 4.6.1 of chapter 4. Then, the new compressed matrix mR in code line 15 comes out from transforming matrix mX with a transformation matrix W that contains only the first t left singular vectors that correspond to the biggest singular values in matrix S. It is to remark that the output matrix mR handed out by both functions transfpcacoveigs and transfpcacovsvd is the same, since the matrix of left singular vectors U and the matrix of eigenvectors W come out with equal values, as demonstrated in section 4.6.1 of chapter 4.

In contrast to algorithm 4.2, code listing 5.7 does not implement the steps for computing the matrix of deviations from the mean mdfmeanmX of matrix mX, since that computation can be done using the Matlab expression mdfmeanmX = mX − ( **mean**(mX,2) ∗ **ones**(1,**size**(mX,2)) ), already shown in code listing 5.3. Besides, it is not necessary to rearrange the singular values in the diagonal matrix S in a decreasing manner and to move their respective left singular vectors in matrix U accordingly, since the function **svd** gives the singular values in matrix S out already arranged in a decreasing manner.

### Singular value decomposition

The singular value decomposition is a method of matrix factorization that can be used for reducing the number of components of vectors by performing a linear transformation. This linear transformation requires a transformation matrix W, that can be obtained computing the singular value decomposition of the matrix mX formed with the available set of fault pattern vectors. The singular value decomposition of matrix mX produces the following three matrices: a diagonal matrix with singular values S, a matrix with left singular vectors U, and a matrix with right singular vectors V. In this method, a submatrix of the matrix of left singular vectors U is taken as transformation matrix W. Thereby, only t left singular vectors that correspond to the t biggest singular values, chosen according to some criterion, are taken

into matrix W. The Matlab program code for performing such transformation is shown in code listing 5.8.

Program Code 5.8: Singular value decomposition transformation function

```
1  function [mR,W] = transfsvd(mX, expectedenergy)
2  [U,S,~]=svd(mX);
3  totalenergy = sum(diag(S));
4  t = 1;
5  partialenergy = (S(t,t)/totalenergy);
6  while partialenergy < expectedenergy
7      t = t+1;
8      partialenergy = partialenergy + (S(t,t)/totalenergy);
9  end
10 if t == 1
11     t = 2;
12 end
13 W = U(:,1:t);
14 mR = W' * mX;
```

Code listing 5.8 shows the implementation of algorithm 4.3: "Fault pattern vector dimension reduction using singular value decomposition" presented in section 4.6.2 of chapter 4. That algorithm has been implemented as the function transfsvd. The outputs that this function can hand out are: the transformed matrix mR, and the transformation matrix W. For that, the function requires as inputs: the matrix mX, and the expected energy, in variable expectedenergy, which is the amount of information that it is expected to be maintained in the reduced matrix mR.

The function **svd** in line 2 of code listing 5.8 computes the singular values and the left singular vectors of matrix mX and gives them out in matrices S and U respectively. The singular values in the diagonal matrix S with their corresponding left and right singular vectors, come already arranged in a decreasing manner out. The number t of left singular vectors to take into account into the transformation matrix W is computed according to the expected energy in code lines 3 to 12. Thereafter, the new compressed matrix mR is obtained in code line 14 transforming matrix mX with the transformation matrix W, which contains only the first t left singular vectors that correspond to the t biggest singular values.

It is to remark that unlike function transfpcasvd, where the transformation matrix W is computed performing the singular value decomposition of the matrix of deviation from the mean mmeanmX of matrix mX, function transfsvd computes the transformation matrix W performing the singular value decomposition of the matrix mX directly. Besides, instead of considering the t eigenvectors that correspond to the t biggest eigenvalues chosen given an expected maintained variance, function transfsvd takes the t singular vectors that correspond to the t biggest singular values chosen given an expected maintained energy.

#### Formal immune network

Formal immune network is a method for reducing the dimensions of vectors by performing a special transformation, That transformation requires the computation of the singular value decomposition of the transposed matrix mX', which has been formed with the available set of fault pattern vectors. The Matlab implementation of that algorithm is presented in code listing 5.9.

Program Code 5.9: Formal immune network transformation function (a)

```
1 function [mR,S,V] = transffin(mX,dimension)
2 [U,S,V] = svd(mX');
3 mR = U(:,1:dimension)';
```

Code listing 5.9 shows the implementation of algorithm 4.6: "Fault pattern vector dimension reduction by means of a formal immune network" presented in detail in subsection 4.8.2 of chapter 4. That algorithm has been implemented as the function transffin. The outputs that this function can hand out are: the transformed matrix mR, the matrix of singular values S, and the matrix of right singular vectors V. For that, the function requires as inputs: the matrix mX, and the number of dimensions to be considered, normally 2 or 3.

The function **svd** in line 2 of code listing 5.10 computes the singular values, the left singular vectors, and the right singular vectors of the transpose of the matrix mX, and gives them out into matrices S, U, and V respectively. The singular values in the diagonal matrix S, with their corresponding left and right singular vectors, come out already arranged in a decreasing manner. At the end, the new compressed matrix mR is the transpose of a submatrix of the matrix of left singular vectors U. That submatrix is formed with all the rows of the matrix of left singular vectors U, and only with a reduced number of columns equal to the given number of dimensions as shown in code line 3.

Program Code 5.10: Formal immune network transformation function (b)

```
1 function [mR,S,V] = transffin(mX,dimension)
2 [U,S,V] = svd(mX');
3 mR = zeros(size(mX,2),dimension);
4 for i = 1:size(mX,2)
5         for q = 1:dimension
6                 mR(i,q) = ( 1/S(q,q) ) * A(i,:) * V(:,q);
7         end
8 end
9 mR = mR';
```

Unlike functions transfpcaeigs, transfpcasvd, and transfsvd, function transffin does not obtain the transformed matrix mR computing a transformation matrix W. However, the transformed matrix mR can also be computed using the formula 4.24 in subsection 4.8.2 of chapter 4, whose code is shown in code lines 3 to 9 of code listing 5.10. This method is useful for the transformation of single vectors and requires only a number equal to the number of dimensions of singular values and their respective right singular vectors.

### Comparison of methods

Table 5.4 presented the number of wrong class recognitions using different distance measurement and class assignation methods. For elaborating that table, the vectors of table 5.1 with all their 840 components have been employed. Now, the intention is to extend table 5.4 using vectors with reduced number of components. For that, four functions for dimension reduction have been presented above in this subsection. The functions transfcoveigs, transfcovsvd, transfsvd, and transffin require as input the matrix mX. The matrix mX can be formed with the 87 fault pattern vectors of the design set, introduced in table 5.1. Those functions hand out the matrix mR. The matrix mR contains the fault pattern vectors with reduced number

of components. Figure 5.5 shows the fault pattern vectors of the design set which dimensions have been reduced to two using all four functions. In that figure, it can be observed that all methods try to cluster the points of a common class.

In order to analyze whether those reduced fault pattern vectors are able to recognize fault vectors of the test set, the distances between fault pattern vectors and fault vectors of the test set should be measured. For that, both sets, the design set and the test set, should contain vectors with reduced number of components. Therefore, it is necessary to reduce also the components of the fault vectors of the test set. For that, a transformation function is necessary. On the one side, the functions transfcoveigs, transfcovsvd and transfsvd hand out a transformation matrix W. On the other side, the function transffin gives out instead, a matrix with singular values S and a matrix with right singular vectors V. Therefore, two functions for transforming the fault vectors of the test set are presented below.

The first function for reducing the components of fault vectors of the test set is transftestdata and it is shown in code listing 5.11. It serves for transforming a single fault vector or the whole test set. It requires as inputs: a transformation matrix W, and the fault vectors of the test set arranged into the matrix mT. The transformation matrix W is the matrix provided by any of the functions transfcoveigs, transfcovsvd, or transfsvd. Function transftestdata hands out the fault vectors with reduced number of components in matrix mTt.

Program Code 5.11: Transformation function for the fault vectors of the test set

```
1 function[mTt] = transftestdata(mT,W)
2 mTt = W'*mT;
```

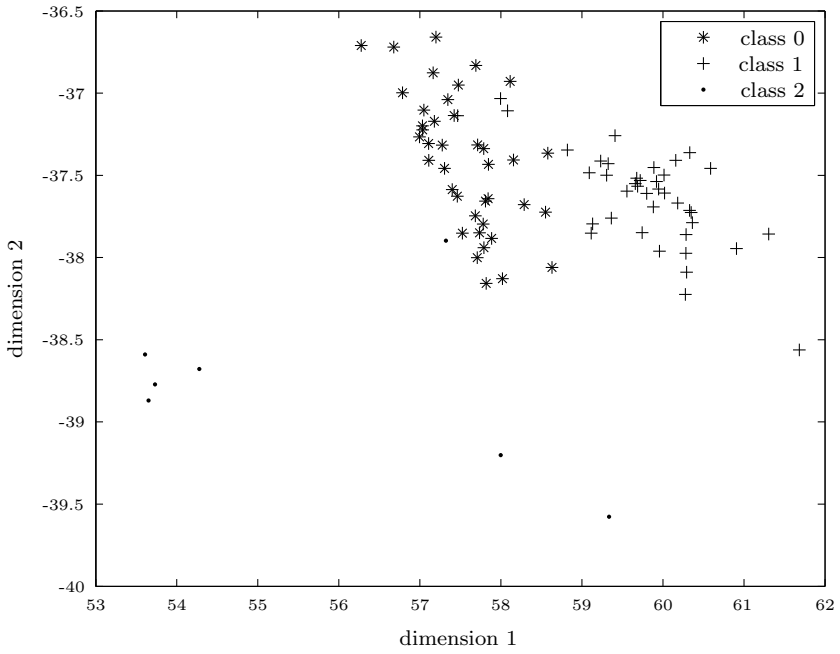
The second function for reducing the components of the fault vectors of the test set is transftestdatafin and it is shown in code listing 5.12. It serves for transforming a single fault vector or the whole test set. It requires as inputs: the matrix of singular values S, the matrix of right singular vectors V, and the fault vectors of the test set arranged into the matrix mT. Matrices S and V are the matrices provided by the function transffin. Function transftestdatafin also hands out the fault vectors with reduced number of components into matrix mTt. It is to remark that, this function implements the equation 4.25 explained in subsection 4.8.3 of chapter 4.

Program Code 5.12: Transformation function for the fault vectors of the test set for FIN

```
1 function[mTt] = transftestdatafin(mT,S,V)
2 dimensionfin = size(S,2);
3 mR = zeros(size(mT,2),dimensionfin);
4 for i = 1:size(mT,2)
5         for q = 1:dimensionfin
6                 mR(i,q) = ( 1/S(q,q) ) * mT(:,i)' * V(:,q);
7         end
8 end
9 mTt = mR';
```

Fault vector dimension reduction should reduce the time taken in computing the distances between vectors. That happens at expense of increasing the number of wrong class recognitions. Subtables 5.6a, 5.6b, 5.6c and 5.6d show the results obtained executing all fault vector dimension reduction functions exposed above in this subsection. The results have been obtained varying the maintained variance (*) and maintained energy (**) for getting one to

(a) transfpcacoveigs




(b) transfpcacovsvd

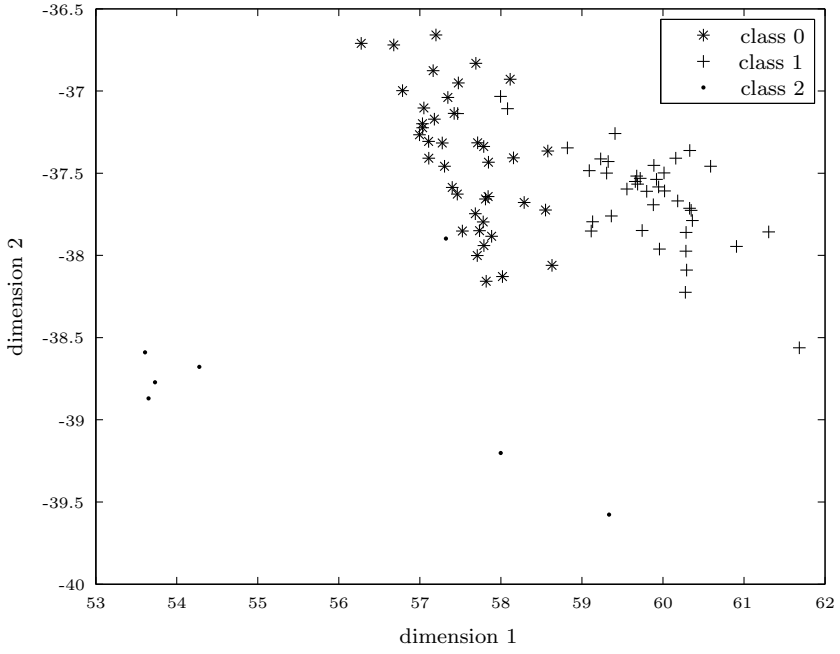
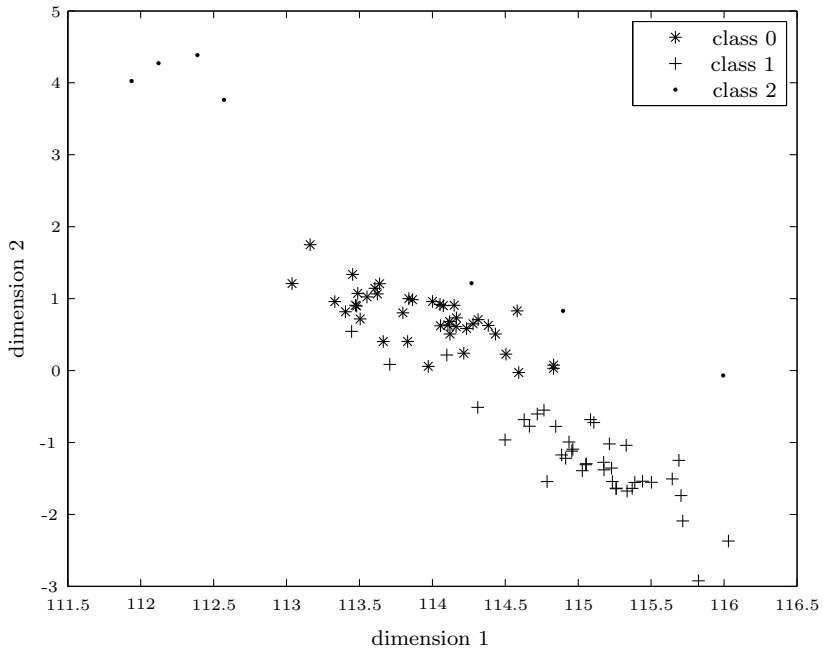



Figure 5.5: Fault pattern vectors with reduced dimensions

(c) transfsvd




(d) transffin

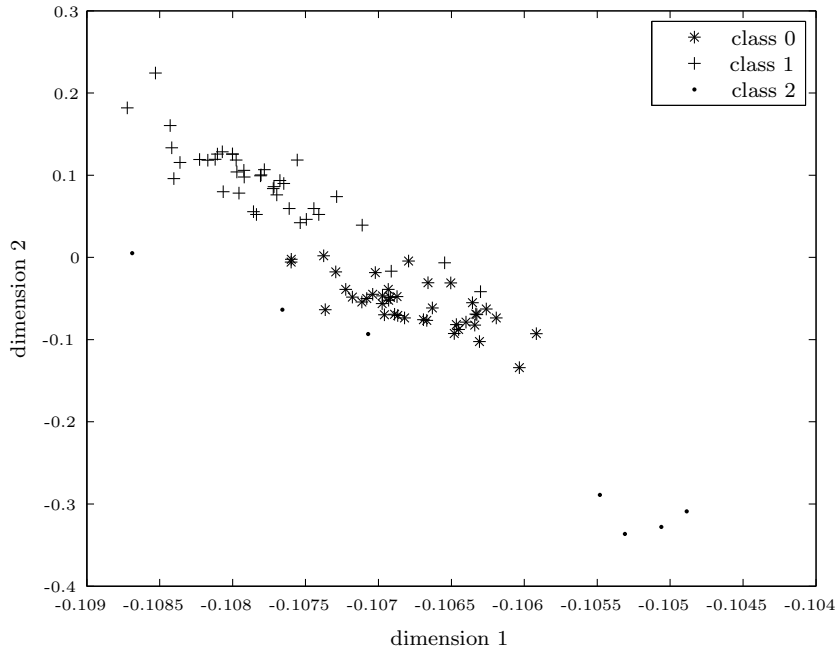



Figure 5.5: Fault pattern vectors with reduced dimensions

four dimensions for each subtable respectively. The line notransf in all subtables presents obviously the lowest number of wrong class recognitions. Fault vector dimension reduction using the functions transfpcacoveigs and transfpcacovsvd report the same results. That is not a surprise, since in subsection 4.6.1 of chapter 4, it has been demonstrated mathematically that both methods of dimension reduction lead to the same results. The recognition of fault vectors using the methods of dimension reduction showed even with only 1 to 4 dimensions, less wrong class recognition than the recognition executed without reducing dimensions. Especially using the Chebyshev and Mahalanobis distance measurement methods. Those cases are remarked with dark gray in the subtables.

Subsection 5.1.1 showed through table 5.4 that normalization of the fault vectors produces a smaller number of wrong class recognitions. Applying that idea, four new functions transfpcacorreigs, transfpcacorrsvd, transfsvdnormalized and transffinnormalized introduce normalization in the process of dimension reduction and are presented separately in the following paragraphs.

Function transfpcacorreigs computes the eigenvalue decomposition of the correlation matrix corrmX of the matrix mX, instead of the covariance matrix covmX of the matrix mX, executing **eigs**(corrmX). The correlation is a statistical measure that denotes dependence between two variables $x_i$ and $x_l$. The correlation can be obtained dividing the covariance of the variables $x_i$ and $x_l$ by the product of their standard deviations, as shown in equation 5.1. Given a vector of variables $X$, a correlation matrix can be formed with the correlations of all the variables with all other variables. The computing of such a correlation matrix requires a term by term division of the covariance matrix, defined in equation 4.15, by a matrix formed with the standard deviations of all the variables. The standard deviations, can be obtained in a form of a vector computing the square root of the variances taken from the diagonal of the covariance matrix. Then, the product of the vector of standard deviations by its transpose, $\sigma_X * \sigma_X^T$, produces the required matrix, as can be shown in the implementation in code listing 5.13. It is to remark that the correlation presents dimensionless values between +1 and −1. A value of 0 means no dependence between variables. A value of +1 is obtained when the correlation is computed for a variable with itself, unless the variance of that variable is 0, case when all the values for that variable do not change and the correlation consequently does not exist.

$$Corr(x_i, x_l) = \frac{Cov(x_i, x_l)}{\sigma_{x_i}\sigma_{x_l}} = \frac{E[(x_i - \bar{x}_i)(x_l - \bar{x}_l)]}{\sigma_{x_i}\sigma_{x_l}} = \frac{1}{m}\sum_{j=1}^{m}\frac{(x_{ij} - \bar{x}_i)(x_{lj} - \bar{x}_l)}{\sigma_{x_i}\sigma_{x_l}} \tag{5.1}$$

Program Code 5.13: Correlation matrix function

```
function [corrmX] = correlation(mX,covmX)
vsdmX = sqrt(diag(covmX));
corrmX = ( (mdfmeanmX*mdfmeanmX')./(vsdmX*vsdmX') )/numbervectorsdata;
```

Function transfpcacorrsvd computes the singular value decomposition of the matrix of standard scores mszmX of matrix mX, instead of the singular value decomposition of the matrix mX, executing **svd**(mszmX). The standard score of the given value of a variable can be computed by the expression $\frac{x_{g_i}-\mu_i}{\sigma_i} = \frac{x_{g_i}}{\sigma_i} - \frac{\mu_i}{\sigma_i}$, that represents how many standard deviations the measured variable is above or bellow the mean, as already explained in subsection

Table 5.5: Wrong class recognitions using vector dimension reduction

| Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| Distance measurement method | | Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | notransf | 840 | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| | transfpcacoveigs* | 1 | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| | transfpcacovsvd* | 1 | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| | transfsvd** | 1 | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| | transffin | 1 | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

(a) * maintained variance = 0.84, ** maintained energy = 0.96, fin dimension = 1

| Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| Distance measurement method | | Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | notransf | 840 | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| | transfpcacoveigs* | 2 | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| | transfpcacovsvd* | 2 | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| | transfsvd** | 2 | 5 | 5 | 5 | 4 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 14 | 38 |
| | transffin | 2 | 9 | 12 | 13 | 13 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 8 | 38 |
| Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

(b) * maintained variance = 0.93, ** maintained energy = 0.98, fin dimension = 2

| Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| Distance measurement method | | Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | notransf | 840 | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| | transfpcacoveigs* | 3 | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| | transfpcacovsvd* | 3 | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| | transfsvd** | 3 | 3 | 3 | 3 | 4 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 7 | 9 |
| | transffin | 3 | 8 | 8 | 8 | 7 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 3 | 3 | 3 | 3 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 5 | 9 |
| Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

(c) * maintained variance = 0.965, ** maintained energy = 0.984, fin dimension = 3

| Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| Distance measurement method | | Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | notransf | 840 | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| | transfpcacoveigs* | 4 | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| | transfpcacovsvd* | 4 | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| | transfsvd** | 4 | 3 | 4 | 4 | 5 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 3 | 4 | 5 | 6 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 7 | 8 |
| | transffin | 4 | 5 | 4 | 3 | 3 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 4 | 4 | 3 | 4 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 6 | 8 |
| Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

(d) * maintained variance = 0.978, ** maintained energy = 0.986, fin dimension = 4

5.1.1. In Matlab the matrix of standard scores can be computed by the following expression mzsmX = mdfmeanmX./(vsdmX∗**ones**(1,numbervectorsdata)).

Function transfsvdnormalized computes the singular value decomposition of the matrix of standard scores mszmX of matrix mX, instead of the singular value decomposition of the matrix mX, executing **svd**(mszmX).

Function transffinnormalized computes the singular value decomposition of the transpose of the matrix of standard scores mszmX of matrix mX, instead of the singular value decomposition of the transpose of the matrix mX, executing **svd**(mszmX'). However, the computation of the transformed matrix mR employing equation 4.24 and the matrix mX instead of matrix mszmX, same as in the function transfin, has implemented because it delivered better results.

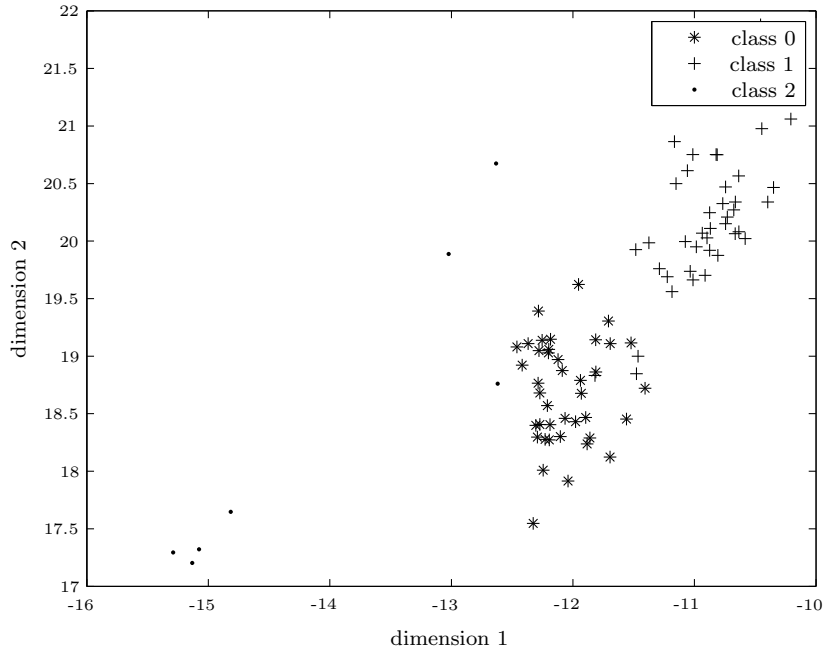
Figure 5.6 shows the fault pattern vectors of the given design set presented in table 5.1, which dimensions have been reduced to two, using all four functions transfpcacorreigs, transfpcacorrsvd, transfsvdnormalized and transffinnormalized. In that figure, it can be observed that the functions transfpcacorreigs, transfpcacorrsvd and transfsvdnormalized produce with normalization the same results. In the case of the fault pattern vectors which dimensions has been reduced with function transffinnormalized, the fault pattern vectors look equal placed as the other methods but with scaled down components.

In order to see the effect of normalization and to analyze whether the reduced fault pattern vectors are able to recognize fault vectors of the test set presented in table 5.1, table 5.6 has been prepared. Thereby, functions transfpcacorreigs and transfpcacorrsvd have been executed setting the same maintained variance as in functions transfpcacoveigs and transfpcacovsvd. Similarly, function transfsvdnormalized has been executed setting the same maintained energy as in function transfsvd. Finally, function transffinnormalized has been executed with the same dimension as for the function transffin. In table 5.6, it can be noticed that although the intention has been to implement a similar normalization for both functions transfpcacorreigs and transfpcacorrsvd, they are not equivalent. Besides, the number of dimensions by the functions transfpcacorreigs, transfpcacorrsvd and transffinnormalized increases. Despite of that, an improvement in the recognition in comparison to the results with no transformation and transformation without normalization can observed by the values highlighted with dark color. That is not the case of function transffinnormalized, where even with the same number of dimensions as their similar function transffin, lower number of wrong class recognitions are obtained. It is to notice that for a dimension of 4, highlighted with dark color, functions transfpcacorreigs, transfpcacorrsvd and transffinnormalized deliver zero wrong class recognitions. Therefore, for a fairer comparison, table 5.7 presents the wrong class recognitions for all functions with the same number of dimensions from 1 to 6.

In that table 5.7 it can be observed that the wrong class recognitions is the same by the functions transfpcacorreigs, transfpcacorrsvd and transffinnormalized for all distance measurement methods. That is because using normalization, the transformation of the fault pattern vectors gives the same results, as could be observed in subfigures (a), (b) and (c) of figure 5.6. Moreover, in table 5.7, the number of wrong class recognitions using the k-nearest neighbor class assignation method using the Euclidean norm and the p-norm is zero for dimensions from 2 to 6. All cases of zero wrong class recognitions are highlighted with dark color.

In order to see how the number of wrong class recognitions varies increasing the number of dimensions, figure 5.7 presents a plot of the number of wrong class recognitions versus the number of dimensions from 1 to 87 for the k-nearest neighbor class assignation method considering the Euclidean norm normalization. 87 is the maximum number of dimension because the maximum number of dimension can only be the rank of the available matrix of

(a) transfpcacorreigs




(b) transfpcacorrsvd

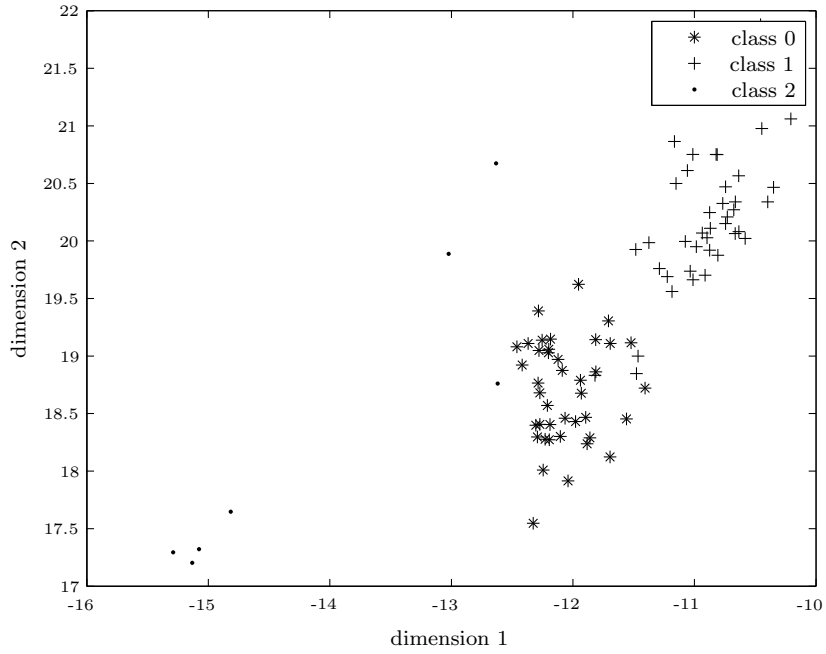
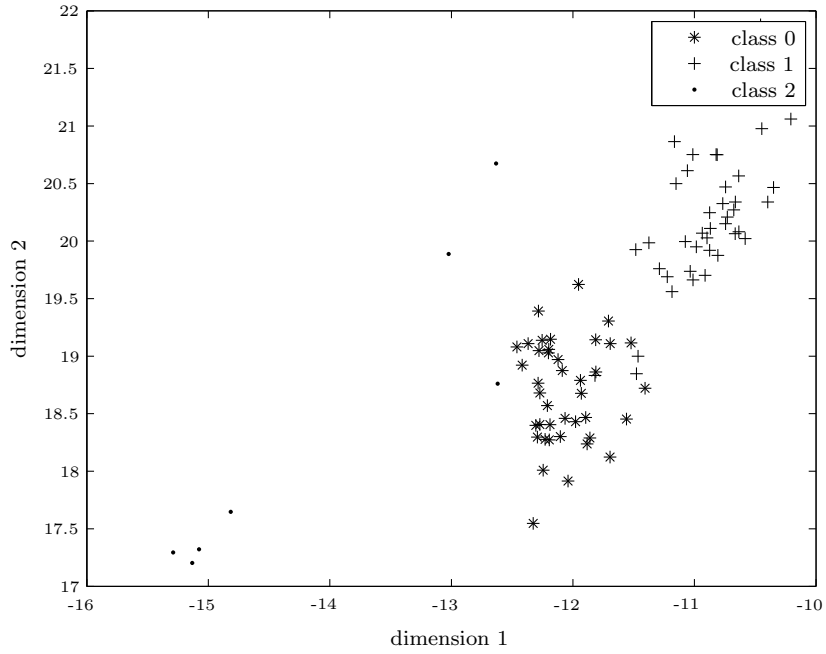



Figure 5.6: Fault pattern vectors with reduced dimensions using normalization

(c) transfsvdnormalized




(d) transffinnormalized

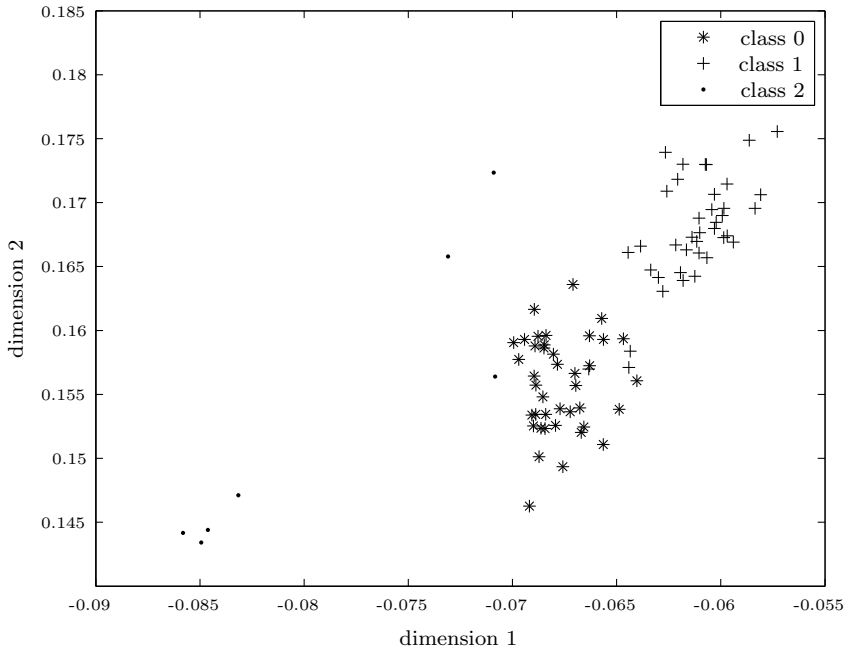



Figure 5.6: Fault pattern vectors with reduced dimensions using normalization

Table 5.6: Wrong class recognitions using normalized vector dimension reduction. (a) Same maintained variance and maintained energy values

| Class assignation method | | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance standard score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | | | |
| Distance measurement method | maintained variance/energy | Nr. of dimensions | | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| notransf | | 840 | | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| transfpcacoveigs* | 0.84 | 1 | | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| transfpcacovsvd* | 0.84 | 1 | | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| transfsvd** | 0.96 | 1 | | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| transffin | | 1 | | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| transfpcacorreigs* | 0.84 | 3 | | 6 | 3 | 3 | 3 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 5 | 4 | 4 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 7 | 29 |
| transfpcacorrsvd* | 0.84 | 4 | | 5 | 4 | 4 | 4 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 6 | 4 | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 7 | 11 |
| transfsvdnormalized** | 0.96 | 68 | | 2 | 3 | 4 | 4 | 30 | 20 | 12 | 6 | 36 | 20 | 11 | 8 | 7 | 5 | 4 | 4 | 5 | 5 | 4 | 4 | 7 | 5 | 4 | 6 | 34 | 7 | 15 |
| transffinnormalized | | 1 | | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 6 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 7 | 12 | 4 | 12 |
| transfpcacoveigs* | 0.93 | 2 | | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| transfpcacovsvd* | 0.93 | 2 | | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| transfsvd** | 0.98 | 2 | | 5 | 5 | 5 | 4 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 14 | 38 |
| transffin | | 2 | | 9 | 12 | 13 | 13 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 8 | 38 |
| transfpcacorreigs* | 0.93 | 4 | | 5 | 4 | 4 | 4 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 6 | 4 | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 7 | 11 |
| transfpcacorrsvd* | 0.93 | 10 | | 4 | 3 | 4 | 4 | 37 | 37 | 33 | 35 | 40 | 37 | 35 | 27 | 6 | 4 | 5 | 4 | 3 | 2 | 2 | 1 | 5 | 2 | 2 | 2 | 8 | 7 | 25 |
| transfsvdnormalized** | 0.98 | 76 | | 2 | 3 | 4 | 4 | 33 | 23 | 14 | 4 | 37 | 23 | 13 | 4 | 7 | 5 | 4 | 4 | 5 | 5 | 4 | 2 | 7 | 5 | 4 | 6 | 51 | 7 | 16 |
| transffinnormalized | | 2 | | 7 | 7 | 5 | 5 | 16 | 16 | 15 | 15 | 18 | 16 | 15 | 11 | 8 | 7 | 7 | 6 | 2 | 3 | 3 | 3 | 5 | 3 | 1 | 0 | 2 | 6 | 18 |
| transfpcacoveigs* | 0.965 | 3 | | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| transfpcacovsvd* | 0.965 | 3 | | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| transfsvd** | 0.984 | 3 | | 3 | 3 | 3 | 4 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 7 | 9 |
| transffin | | 3 | | 8 | 8 | 8 | 7 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 3 | 3 | 3 | 3 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 5 | 9 |
| transfpcacorreigs* | 0.965 | 5 | | 4 | 4 | 4 | 4 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 5 | 4 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| transfpcacorrsvd* | 0.965 | 19 | | 4 | 3 | 4 | 4 | 32 | 29 | 25 | 13 | 36 | 29 | 24 | 3 | 6 | 5 | 4 | 4 | 3 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 9 | 7 | 22 |
| transfsvdnormalized** | 0.984 | 78 | | 2 | 3 | 4 | 4 | 33 | 26 | 15 | 5 | 37 | 26 | 13 | 4 | 7 | 5 | 4 | 4 | 5 | 5 | 5 | 2 | 8 | 5 | 4 | 7 | 37 | 7 | 16 |
| transffinnormalized | | 3 | | 9 | 5 | 5 | 5 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 8 | 8 | 8 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 13 | 29 |
| transfpcacoveigs* | 0.978 | 4 | | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| transfpcacovsvd* | 0.978 | 4 | | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| transfsvd** | 0.986 | 4 | | 3 | 4 | 4 | 5 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 3 | 4 | 5 | 6 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 7 | 8 |
| transffin | | 4 | | 5 | 4 | 3 | 3 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 4 | 4 | 3 | 4 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 6 | 8 |
| transfpcacorreigs* | 0.978 | 5 | | 4 | 4 | 4 | 4 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 5 | 4 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| transfpcacorrsvd* | 0.978 | 26 | | 3 | 3 | 4 | 4 | 41 | 39 | 36 | 19 | 43 | 39 | 33 | 3 | 7 | 5 | 4 | 4 | 7 | 5 | 3 | 0 | 9 | 5 | 1 | 1 | 8 | 7 | 21 |
| transfsvdnormalized** | 0.986 | 79 | | 2 | 3 | 4 | 4 | 33 | 26 | 15 | 5 | 37 | 26 | 13 | 4 | 7 | 5 | 4 | 4 | 5 | 5 | 5 | 2 | 9 | 5 | 4 | 7 | 50 | 7 | 15 |
| transffinnormalized | | 4 | | 7 | 8 | 7 | 6 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 8 | 8 | 9 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 8 | 11 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

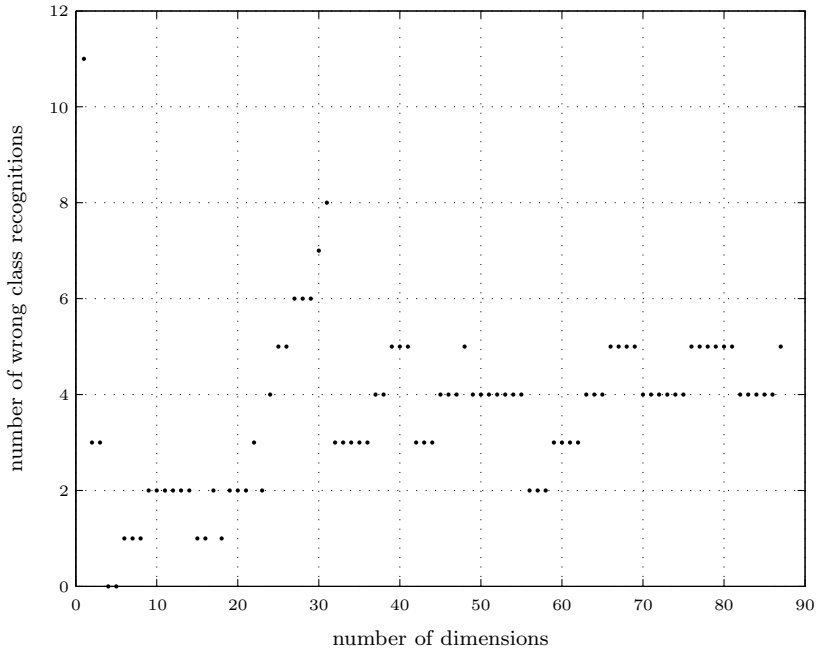
fault pattern vectors $\mathsf{mX}_{87\times840}$. This unique graph refers to all the functions transfpcacorreigs, transfpcacorrsvd, transfsvdnormalized and transffinnormalized because their behavior is the same for that class assignation method, as has been already noticed in table 5.7. Since a low number of wrong class recognitions can be observed from two dimensions upwards, for many numbers of dimensions, then, obviously, it is recommended to take the minimal number of dimensions for a minimal number of wrong class recognitions, in this case four dimensions. However, that should be decided according to the tolerable number of wrong class recognitions and the available memory for saving the reduced fault pattern vectors.

The number of wrong class recognitions without normalizations in the process of dimension reduction increases as the number of dimensions increases, as can be seen in figure 5.7 using function transfpcacoveigs. This shows that a lower number of wrong class recognitions can be obtained employing the normalization implemented in the process of dimension reduction by functions transfpcacorreigs, transfpcacorrsvd, transfsvdnormalized and transffinnormalized.

Table 5.7: Wrong class recognitions using normalized vector dimension reduction. (b) Same dimensions

| Class assignation method | | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| Distance measurement method | | maintained variance/energy | Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| Wrong class recognitions | notransf | | 840 | 1 | 3 | 5 | 7 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 5 | 7 | 9 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 7 | 1 |
| | transfpcacoveigs* | 0.84 | 1 | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| | transfpcacovsvd* | 0.84 | 1 | 11 | 11 | 11 | 11 | 28 | 28 | 28 | 28 | 33 | 28 | 25 | 18 | 11 | 11 | 11 | 11 | 16 | 16 | 16 | 16 | 16 | 16 | 14 | 12 | 37 | 19 | 37 |
| | transfsvd** | 0.96 | 1 | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| | transffin | | 1 | 35 | 35 | 35 | 35 | 40 | 40 | 40 | 40 | 41 | 40 | 40 | 38 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 35 | 30 | 30 | 30 | 65 | 49 | 65 |
| | transfpcacorreigs* | | 1 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 6 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 7 | 12 | 4 | 12 |
| | transfpcacorrsvd* | | 1 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 6 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 7 | 12 | 4 | 12 |
| | transfsvdnormalized* | | 1 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 6 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 7 | 12 | 4 | 12 |
| | transffinnormalized* | | 1 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 6 | 6 | 6 | 6 | 11 | 11 | 11 | 11 | 11 | 11 | 9 | 7 | 12 | 4 | 12 |
| | transfpcacoveigs* | 0.93 | 2 | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| | transfpcacovsvd* | 0.93 | 2 | 4 | 7 | 7 | 8 | 6 | 7 | 7 | 10 | 5 | 7 | 9 | 6 | 4 | 7 | 9 | 9 | 1 | 4 | 4 | 5 | 2 | 4 | 3 | 4 | 2 | 7 | 8 |
| | transfsvd** | 0.98 | 2 | 5 | 5 | 5 | 4 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 14 | 38 |
| | transffin | | 2 | 9 | 12 | 13 | 13 | 10 | 9 | 7 | 11 | 15 | 9 | 5 | 3 | 7 | 7 | 7 | 7 | 7 | 7 | 8 | 6 | 18 | 7 | 4 | 2 | 1 | 8 | 38 |
| | transfpcacorreigs* | | 2 | 6 | 6 | 6 | 5 | 16 | 16 | 15 | 15 | 18 | 16 | 15 | 11 | 8 | 4 | 4 | 4 | 2 | 3 | 3 | 3 | 5 | 3 | 1 | 0 | 2 | 5 | 18 |
| | transfpcacorrsvd* | | 2 | 6 | 6 | 6 | 5 | 16 | 16 | 15 | 15 | 18 | 16 | 15 | 11 | 8 | 4 | 4 | 4 | 2 | 3 | 3 | 3 | 5 | 3 | 1 | 0 | 2 | 5 | 18 |
| | transfsvdnormalized* | | 2 | 6 | 6 | 6 | 5 | 16 | 16 | 15 | 15 | 18 | 16 | 15 | 11 | 8 | 4 | 4 | 4 | 2 | 3 | 3 | 3 | 5 | 3 | 1 | 0 | 2 | 5 | 18 |
| | transffinnormalized* | | 2 | 7 | 7 | 5 | 5 | 16 | 16 | 15 | 15 | 18 | 16 | 15 | 11 | 8 | 7 | 7 | 6 | 2 | 3 | 3 | 3 | 5 | 3 | 1 | 0 | 2 | 6 | 18 |
| | transfpcacoveigs* | 0.965 | 3 | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| | transfpcacovsvd* | 0.965 | 3 | 3 | 7 | 7 | 9 | 5 | 6 | 6 | 7 | 5 | 6 | 6 | 9 | 5 | 5 | 7 | 7 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 7 | 7 |
| | transfsvd** | 0.984 | 3 | 3 | 3 | 3 | 4 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 7 | 9 |
| | transffin | | 3 | 8 | 8 | 8 | 7 | 14 | 14 | 12 | 14 | 14 | 14 | 12 | 11 | 3 | 3 | 3 | 3 | 2 | 3 | 4 | 3 | 3 | 3 | 2 | 3 | 0 | 5 | 9 |
| | transfpcacorreigs* | | 3 | 6 | 3 | 3 | 3 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 5 | 4 | 4 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 7 | 29 |
| | transfpcacorrsvd* | | 3 | 6 | 3 | 3 | 3 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 5 | 4 | 4 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 7 | 29 |
| | transfsvdnormalized* | | 3 | 6 | 3 | 3 | 3 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 5 | 4 | 4 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 7 | 29 |
| | transffinnormalized* | | 3 | 9 | 5 | 5 | 5 | 16 | 15 | 16 | 16 | 20 | 15 | 15 | 7 | 8 | 8 | 8 | 8 | 3 | 3 | 2 | 3 | 9 | 3 | 1 | 0 | 0 | 13 | 29 |
| | transfpcacoveigs* | 0.978 | 4 | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| | transfpcacovsvd* | 0.978 | 4 | 6 | 5 | 5 | 8 | 11 | 11 | 11 | 10 | 9 | 11 | 13 | 18 | 5 | 5 | 7 | 7 | 4 | 5 | 6 | 6 | 3 | 5 | 7 | 13 | 2 | 7 | 8 |
| | transfsvd** | 0.986 | 4 | 3 | 4 | 4 | 5 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 3 | 4 | 5 | 6 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 7 | 8 |
| | transffin | | 4 | 5 | 4 | 3 | 3 | 11 | 16 | 15 | 15 | 12 | 16 | 15 | 14 | 4 | 4 | 3 | 4 | 1 | 1 | 1 | 3 | 1 | 1 | 2 | 4 | 0 | 6 | 8 |
| | transfpcacorreigs* | | 4 | 5 | 4 | 4 | 4 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 6 | 4 | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 7 | 11 |
| | transfpcacorrsvd* | | 4 | 5 | 4 | 4 | 4 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 6 | 4 | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 7 | 11 |
| | transfsvdnormalized* | | 4 | 5 | 4 | 4 | 4 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 6 | 4 | 4 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 7 | 11 |
| | transffinnormalized* | | 4 | 7 | 8 | 7 | 6 | 13 | 14 | 15 | 17 | 16 | 14 | 15 | 11 | 7 | 8 | 8 | 9 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 8 | 11 |
| | transfpcacoveigs* | | 5 | 4 | 3 | 3 | 7 | 7 | 7 | 7 | 6 | 6 | 7 | 7 | 12 | 5 | 4 | 6 | 6 | 5 | 5 | 5 | 6 | 5 | 5 | 5 | 11 | 2 | 7 | 8 |
| | transfpcacovsvd* | | 5 | 4 | 3 | 3 | 7 | 7 | 7 | 7 | 6 | 6 | 7 | 7 | 12 | 5 | 4 | 6 | 6 | 5 | 5 | 5 | 6 | 5 | 5 | 5 | 11 | 2 | 7 | 8 |
| | transfsvd** | | 5 | 3 | 4 | 5 | 6 | 19 | 18 | 16 | 15 | 17 | 18 | 16 | 25 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 2 | 1 | 2 | 5 | 15 | 4 | 7 | 8 |
| | transffin | | 5 | 8 | 8 | 9 | 11 | 19 | 18 | 16 | 15 | 17 | 18 | 16 | 25 | 4 | 4 | 6 | 8 | 3 | 2 | 3 | 2 | 1 | 2 | 5 | 15 | 4 | 7 | 8 |
| | transfpcacorreigs* | | 5 | 4 | 4 | 4 | 4 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 5 | 4 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| | transfpcacorrsvd* | | 5 | 4 | 4 | 4 | 4 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 5 | 4 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| | transfsvdnormalized** | | 5 | 4 | 4 | 4 | 4 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 5 | 4 | 4 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| | transffinnormalized | | 5 | 4 | 6 | 6 | 6 | 22 | 24 | 25 | 24 | 22 | 24 | 25 | 15 | 6 | 6 | 8 | 8 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3 | 7 | 12 |
| | transfpcacoveigs* | | 6 | 5 | 4 | 3 | 5 | 6 | 4 | 4 | 4 | 5 | 4 | 6 | 10 | 6 | 5 | 6 | 6 | 3 | 2 | 1 | 2 | 2 | 2 | 1 | 8 | 9 | 7 | 8 |
| | transfpcacovsvd* | | 6 | 5 | 4 | 3 | 5 | 6 | 4 | 4 | 4 | 5 | 4 | 6 | 10 | 6 | 5 | 6 | 6 | 3 | 2 | 1 | 2 | 2 | 2 | 1 | 8 | 9 | 7 | 8 |
| | transfsvd** | | 6 | 3 | 4 | 4 | 6 | 7 | 7 | 6 | 5 | 7 | 7 | 7 | 6 | 6 | 5 | 5 | 6 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 6 | 9 | 7 | 12 |
| | transffin | | 6 | 6 | 6 | 5 | 7 | 7 | 7 | 6 | 5 | 7 | 7 | 7 | 6 | 5 | 5 | 6 | 7 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 6 | 9 | 4 | 12 |
| | transfpcacorreigs* | | 6 | 5 | 3 | 4 | 4 | 22 | 27 | 27 | 24 | 24 | 27 | 26 | 16 | 6 | 4 | 5 | 4 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 7 | 17 |
| | transfpcacorrsvd* | | 6 | 5 | 3 | 4 | 4 | 22 | 27 | 27 | 24 | 24 | 27 | 26 | 16 | 6 | 4 | 5 | 4 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 7 | 17 |
| | transfsvdnormalized** | | 6 | 5 | 3 | 4 | 4 | 22 | 27 | 27 | 24 | 24 | 27 | 26 | 16 | 6 | 4 | 5 | 4 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 7 | 17 |
| | transffinnormalized | | 6 | 4 | 3 | 3 | 4 | 22 | 27 | 27 | 24 | 24 | 27 | 26 | 16 | 6 | 6 | 6 | 7 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 11 | 17 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

(a) transfpcacorreigs, transfpcacorrsvd, transfsvdnormalized and transffinnormalized




(b) transfpcacorreigs

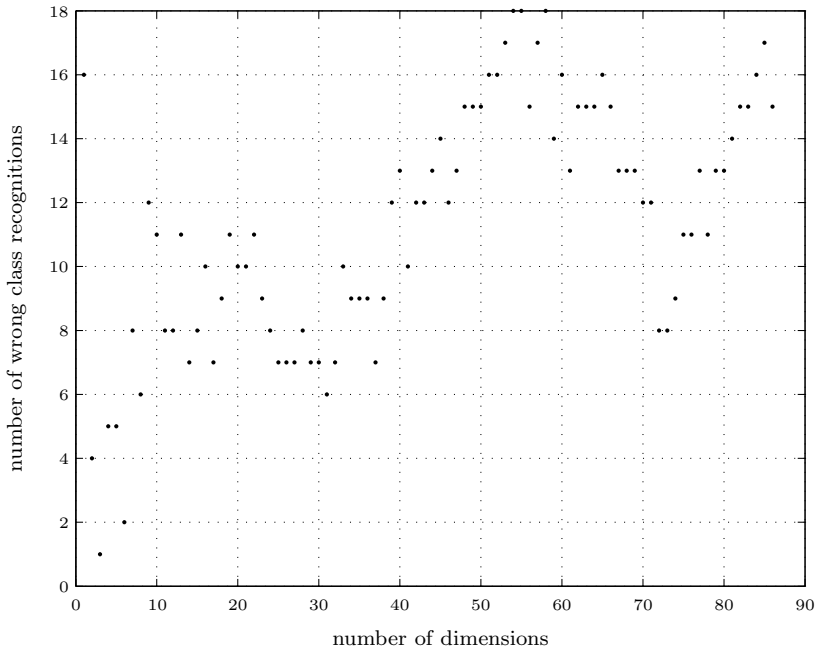



Figure 5.7: Wrong class recognitions vs dimensions

### 5.1.3 Fault pattern vectors number reduction

Last section has shown that working with fault vectors that have reduced components, can not only reduce the time taken in computing the distances between vectors, but can also reduce the number of wrong fault recognitions. This section aims to show that reducing the number of fault pattern vectors to which the given fault vector should be compared, can also reduce the fault recognition time. The reduction of the number of fault pattern vectors can be carried out before or after the fault vector dimension reduction has been executed. Next subsections show the implementation of the methods: "Death of immune cells with insufficient stimulation", "Elimination of auto-reactive immune cells", and "Apoptosis and auto-immunization", shown in section 4.7 and subsection 4.8.5 of chapter 4. Their results are presented by means of tables and two dimensional graphs.

#### Death of immune cells with insufficient stimulation

The reduction of the number of fault pattern vectors resembling the death of immune cells with insufficient stimulation, has been presented in section 4.7 of chapter 4 by means of algorithm 4.4. That algorithm is inspired in the control of population of immune cells and tries to reduce the number of fault pattern vectors in the set, removing all fault pattern vectors that do not present any affinity with all other fault pattern vectors.

Program Code 5.14: Death of immune cells with insufficient stimulation function

```
function [mXr, vcr] = removalbylackofstimulation(mX, vc, threshold, \
                                                 distancemethod)
mXr = []; vcr = [];
for i = 1:( size(mX,2) - 1 )
   stimulation = 0;
    for j = (i+1):size(mX,2)
        distance = distancevectortovector(mX(:,i),mX(:,j),\
                                          distancemethod);
        if distance < threshold
            stimulation = stimulation + 1;
        end
    end
    if stimulation ~= 0
        mXrtemp = [mXr,mX(:,i)]; vcrtemp = [vcr,vc(i)];
        mXr = mXrtemp; vcr = vcrtemp;
    end
    if i == ( size(mX,2) - 1 )
        mXrtemp = [mXr,mX(:,i+1)]; vcrtemp = [vcr,vc(i+1)];
        mXr = mXrtemp; vcr = vcrtemp;
    end
end
```

The algorithm 4.4 named "Fault pattern vectors number reduction resembling the death of immune cells with insufficient stimulation" has been implemented in Matlab as the function removalbylackofstimulation and is shown in code listing 5.14. This function requires as inputs: all fault pattern vectors arranged into the matrix mX, the fault class of each of those fault pattern vectors arranged into the vector vc, a value in the variable threshold which allows to tune the algorithm, and the distance measurement method to be applied for measuring

the affinity between vectors in variable distancemethod. The function gives as outputs: the reduced number of fault pattern vectors into the matrix mXr, and the fault classes of the leftover fault pattern vectors into the vector vcr.

In the function presented in code listing 5.14, the affinity between fault vectors is measured through the distance between them, as can be seen in code line 7. A small distance means a high affinity. Therefore, following the logic of the algorithm, a low value in the variable threshold can produce the removal of a high number of fault pattern vectors, since the area in which a fault pattern vector can be stimulated is too small. As distance measurements methods to define in variable distancemethod, could be selected one of the Minkowski distance measurement methods, or one of the Minkowski distance measurement methods normalized with the Euclidean norm or the p-norm, as explained in section 5.1.1. The Mahalanobis distance is not appropriate, since that one is a distance measured between a vector and a set of vectors.

The fault pattern vectors which have an affinity greater than the threshold with the fault pattern vector in turn in the loop, are counted in the variable stimulation, as can be seen in the code lines 9 to 11. The fault pattern vectors of the matrix mX, which stimulation variable is zero, are not copied in the new reduced matrix mXr, see code lines 13 to 16. Please note that the fault classes of the vectors contained in the new reduced matrix mXr are copied accordingly. Finally, the last fault pattern vector is copied into the new reduced matrix mXr directly since there are not other vectors to be compared with. That can be seen in line codes 17 to 20.

### Elimination of auto-reactive immune cells

The reduction of the number of fault pattern vectors through auto-reactive immune cells, has been presented in section 4.7 of chapter 4 by means of algorithm 4.5. That algorithm is inspired in the removal of immune cells which are able to react against other cells of the body. In that way, that algorithm tries to reduce the number of fault pattern vectors by removing fault pattern vectors that recognize any other fault pattern vector in the set.

Program Code 5.15: Elimination of auto-reactive immune cells function

```
1  function[mXr,vcr] = removalbyautoreactivity(mX,vc,threshold,\
2                                                distancemethod)
3  mXr = mX(:,1); vcr = vc(:,1);
4  numbervectors = size(mX,2);
5  for i = 1:( numbervectors - 1 )
6      for j = ( i + 1 ):numbervectors
7          distance = distancevectortovector(mX(:,i),mX(:,j),\
8                                             distancemethod);
9          if distance > threshold
10             mXrtemp = [mXr,mX(:,j)]; vcrtemp = [vcr,vc(j)];
11             mXr = mXrtemp; vcr = vcrtemp;
12         end
13     end
14     mX = mXr; vc = vcr;
15     numbervectors = size(mXr,2);
16     if i < numbervectors
17         mXr = mXr(:,1:( i + 1 )); vcr = vcr(1:( i + 1 ));
18     end
```

```
19 end
```

The algorithm 4.5 named "Fault pattern vectors number reduction through auto-reactive immune cells" has been implemented in Matlab as the function removalbyautoreactivity and is shown in code listing 5.15. This function requires as inputs: all fault pattern vectors arranged into the matrix mX, the fault class of each of those fault pattern vectors arranged into the vector vc, a value in the variable threshold which allows to tune the algorithm, and the distance measurement method in the variable distancemethod to be applied for determining if a fault pattern recognizes another fault pattern vector or not. The function gives as outputs: the reduced number of fault pattern vectors into the matrix mXr, and the fault classes of the leftover fault pattern vectors into the vector vcr.

In the function presented in code listing 5.15, a fault pattern vector recognizes another fault pattern vector whenever the distance between these fault pattern vectors is less than a given threshold, as can be seen in code lines 7 and 9. In consequence, a high value in the variable threshold can produce that a higher number of fault pattern vectors are recognized by the fault pattern vector in turn and do not be copied in the new reduced matrix mXr, see code lines 9 to 12. Please note that the fault classes of the vectors contained in the new reduced matrix mXr are copied accordingly into vector vcr. As distance measurements methods for assigning in variable distancemethod, could be selected one of the Minkowski distance measurement methods, or one of the Minkowski distance measurement methods normalized with the Euclidean norm or the p-norm, as explained in section 5.1.1. The Mahalanobis distance is here again not appropriate, since that one is a distance measured between a vector and a set of vectors.

Following algorithm 4.5, the reduced set of fault pattern vectors mXr is adopted as the new set of fault pattern vectors mX for being reduced in the next loop, as can be seen in code line 14. The new number of fault pattern vectors in the matrix mX is updated. And in code line 17, only the fault pattern vectors that passed the loop and have not recognized any prior fault pattern vector are copied into matrix mXr. Please note that the conditional expression in code lines 16 and 18, prevents getting a problem with the last fault pattern vector in the row which does not have any fault pattern vector for recognizing. This is the reason why, in code listing 5.15, unlike algorithm 4.5, the fault pattern vector in turn in the loop, is copied into the matrix of reduced fault pattern vectors mXr in the prior loop, instead of copying it at the beginning of the loop, see code line 17. As a consequence, it is necessary to copy the first fault pattern vector in the row into the matrix mXr at the beginning of the code, see code line 3. Finally it is important to remark, that the initial order of the fault pattern vectors in the matrix mX may affect the final matrix mXr given as result.

### Apoptosis and auto-immunization

The reduction of the number of fault pattern vectors by means of apoptosis and auto-immunization, has been presented in section 4.8.5 of chapter 4 by means of algorithm 4.9. That algorithm is inspired in the programmed death of cells, called apoptosis, and in the emergence of cells that are auto-reactive in the body, named auto-immunization. It tries to reduce the number of fault pattern vectors by removing fault pattern vectors that recognize any other fault pattern vector in the set. However, comparing with the algorithm 4.5, which eliminates auto-reactive immune cells, this algorithm considers also the class of the fault pattern vectors for deciding about the removal of the fault pattern vectors.

Program Code 5.16: Apoptosis and auto-immunization function

```
function [mXr, vcr] = removalbyapoptosisautoimmunization(mX, vc, \
                                                         threshold, \
                                                         distancemethod)
%% Apoptosis
mXr = mX(:,1); vcr = vc(1);
mXd = []; vcd = [];
numbervectors = size(mX,2);
for i = 1:( numbervectors - 1 )
    for j = ( i + 1 ):numbervectors
        distance = distancevectortovector(mX(:,i),mX(:,j),\
                                          distancemethod);
        if ( distance < threshold ) && ( vc(i) == vc(j) )
            mXdtemp = [mXd,mX(:,j)]; vcdtemp = [vcd,vc(j)];
            mXd = mXdtemp; vcd = vcdtemp;
        else
            mXrtemp = [mXr,mX(:,j)]; vcrtemp = [vcr,vc(j)];
            mXr = mXrtemp; vcr = vcrtemp;
        end
    end
    mX = mXr; vc = vcr;
    numbervectors = size(mXr,2);
    if i < numbervectors
        mXr = mXr(:,1:( i + 1 )); vcr = vcr(1:( i + 1 ));
    end
end
%% Autoimmunization
numbervectorsdeleted = size(mXd,2);
numbervectorsreduced = size(mXr,2);
for i = 1:numbervectorsdeleted
    for j = 1:numbervectorsreduced
        distances(j) = distancevectortovector(mXd(:,i),mXr(:,j),\
                                              distancemethod);
    end
    [~,minplace] = min(distances);
    if vcd(i) ~= vcr(minplace)
        mXrtemp = [mXr,mXd(:,i)]; vcrtemp = [vcr,vcd(i)];
        mXr = mXrtemp; vcr = vcrtemp;
    end
end
```

The algorithm 4.9 named "Fault pattern vectors number reduction by means of apoptosis and auto-immunization" has been implemented in Matlab as the function removalbyapoptosisautoimmunization and is shown in code listing 5.16. This function requires the same inputs as prior algorithms: all fault pattern vectors arranged into the matrix mX, the fault class of each of those fault pattern vectors arranged into the vector vc, a value in the variable threshold which allows to tune the algorithm, and the distance measurement method in the variable distancemethod to be applied for determining if a fault pattern recognizes another fault pattern vector or not. This function gives also as outputs: the reduced number of fault pattern vectors into the matrix mXr, and the the fault classes of the leftover fault pattern vectors into the vector vcr.

In the function presented in code listing 5.16, the first part of the code implementing apoptosis looks very similar to the code presented in code listing 5.15, except that the fault pattern vector recognizes another fault pattern vector whenever the distance between these fault pattern vectors is less than a given threshold and the fault pattern vector classes are the same, as can be seen in code line 12. Besides, the fault vectors that are removed from the set are saved for being used in the code lines that implement the auto-immunization, as can be seen in code lines 13 and 14. Similar to the code listing 5.15, a high value in the variable threshold can produce the removal of a high number of fault pattern vectors, see code lines 12 to 18. Furthermore, as distance measurements methods for assigning in variable distancemethod, could be selected one of the Minkowski distance measurement methods, or one of the Minkowski distance measurement methods normalized with the Euclidean norm or the p-norm, as explained in section 5.1.1. The Mahalanobis distance is here again is not appropriate, since that one is a distance measured between a vector and a set of vectors.

The second part of the code that implements auto-immunization tries to find the fault pattern vector in the reduced matrix of fault pattern vectors mXr, that approaches the most a fault pattern vector of the matrix of deleted fault pattern vectors mXd. Then, it is proved whether both fault pattern vectors have the same class. If that is not the case, the deleted fault pattern vector is reinserted in the reduced matrix of fault pattern vectors mXr. That procedure can be seen in code lines 27 to 39, and the reason why it is executed is for reinserting some characteristic fault pattern vectors whose deletion has been produced because of the initial order of the matrix mX. Finally, the reduced number of fault pattern vectors are given into the matrix mXr together with their corresponding classes into the vector vcr.

### Comparison of methods

The functions removalbylackofstimulation, removalbyautoreactivity and removalbyapoptosisauto-immunization have been executed for different thresholds as can be seen in tables 5.8 to 5.16 respectively. Thereby, fault pattern vectors with five dimensions has been employed since for the given data set, fault pattern vectors with five dimensions showed in subsection 5.1.2, in many cases, to be able to recognize all fault vectors of the test set producing zero wrong class recognitions.

Tables 5.8 to 5.16 show besides the wrong class recognitions also their corresponding reduced number of fault pattern vectors. Looking at those tables, it is desirable to find the lowest reduced number of fault pattern vectors by the lowest number of wrong class recognitions. That is difficult to find since the lower the reduced number of fault pattern vectors, the higher the number of wrong class recognitions. Therefore, an optimization objective in form of a formula is required for finding the case where both, the number of wrong class recognitions and the reduced number of fault pattern vectors, are minimal. For that, the number of wrong class recognitions and the reduced number of fault pattern vectors can be added up considering that we look for the minimal of both. But, before that, the number of fault pattern vectors should be multiplied by the number of their dimensions. That, in order to be able to compare properly the obtained results with the results of the line notransf obtained without executing dimension reduction, i.e. with 840 dimensions. Besides, since the number of wrong class recognitions has normally a low value, in comparison with the values of the reduced number of fault pattern vectors, a weight should be multiplied to it. The resulting formula is show in equation 5.2

Table 5.8: Results of the algorithm "Death of immune cells with insufficient stimulation" a

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | Distance measur. method | | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1 | A | notransf | 840 | 106 | 6 | 5 | 7 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 23 | 106 | 6 | 7 | 9 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 26 | 67 | 7 | 1 |
| | | transfpcacoveigs | 5 | 7 | 6 | 6 | 10 | 53 | 8 | 8 | 6 | 53 | 8 | 8 | 14 | 9 | 6 | 6 | 6 | 53 | 7 | 3 | 7 | 53 | 7 | 6 | 17 | 8 | 7 | 3 |
| | | transfpcacovsvd | 5 | 7 | 6 | 6 | 10 | 53 | 8 | 8 | 6 | 53 | 8 | 8 | 14 | 9 | 6 | 6 | 6 | 53 | 7 | 3 | 7 | 53 | 7 | 6 | 17 | 8 | 7 | 3 |
| | | transfsvd | 5 | 5 | 7 | 8 | 9 | 67 | 17 | 27 | 21 | 67 | 17 | 21 | 21 | 10 | 6 | 6 | 7 | 67 | 4 | 5 | 5 | 67 | 4 | 5 | 18 | 7 | 7 | 7 |
| | | transffin | 5 | 8 | 8 | 9 | 11 | 67 | 17 | 27 | 21 | 67 | 17 | 21 | 21 | 4 | 4 | 6 | 8 | 67 | 4 | 5 | 5 | 67 | 4 | 5 | 18 | 4 | 7 | 8 |
| | | transfpcacorreigs | 5 | 7 | 6 | 6 | 5 | 67 | 60 | 40 | 45 | 67 | 60 | 28 | 12 | 8 | 7 | 7 | 6 | 67 | 6 | 5 | 12 | 67 | 6 | 2 | 0 | 7 | 8 | 9 |
| | | transfpcacorrsvd | 5 | 7 | 6 | 6 | 5 | 67 | 60 | 40 | 45 | 67 | 60 | 28 | 12 | 8 | 7 | 7 | 6 | 67 | 6 | 5 | 12 | 67 | 6 | 2 | 0 | 7 | 8 | 9 |
| | | transfsvdnormalized | 5 | 7 | 6 | 6 | 5 | 67 | 60 | 40 | 45 | 67 | 60 | 28 | 12 | 8 | 7 | 7 | 6 | 67 | 6 | 5 | 12 | 67 | 6 | 2 | 0 | 7 | 8 | 9 |
| | | transffinnormalized | 5 | 4 | 6 | 6 | 6 | 67 | 60 | 40 | 45 | 67 | 60 | 28 | 12 | 6 | 6 | 8 | 8 | 67 | 6 | 5 | 12 | 67 | 6 | 2 | 0 | 3 | 7 | 12 |
| | B | notransf | 840 | 1 | 81 | 86 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 22 | 1 | 81 | 86 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 22 | 1 | 81 | 86 |
| | | transfpcacoveigs | 5 | 66 | 81 | 82 | 82 | 4 | 22 | 34 | 54 | 3 | 22 | 49 | 82 | 66 | 81 | 82 | 82 | 4 | 22 | 34 | 54 | 3 | 22 | 49 | 82 | 66 | 81 | 82 |
| | | transfpcacovsvd | 5 | 66 | 81 | 82 | 82 | 4 | 22 | 34 | 54 | 3 | 22 | 49 | 82 | 66 | 81 | 82 | 82 | 4 | 22 | 34 | 54 | 3 | 22 | 49 | 82 | 66 | 81 | 82 |
| | | transfsvd | 5 | 65 | 81 | 82 | 83 | 4 | 34 | 47 | 56 | 2 | 34 | 56 | 84 | 65 | 81 | 82 | 83 | 4 | 34 | 47 | 56 | 2 | 34 | 56 | 84 | 65 | 81 | 82 |
| | | transffin | 5 | 87 | 87 | 87 | 87 | 4 | 34 | 47 | 56 | 2 | 34 | 56 | 84 | 87 | 87 | 87 | 87 | 4 | 34 | 47 | 56 | 2 | 34 | 56 | 84 | 87 | 87 | 87 |
| | | transfpcacorreigs | 5 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 |
| | | transfpcacorrsvd | 5 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 |
| | | transfsvdnormalized | 5 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 | 84 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 73 | 82 | 83 |
| | | transffinnormalized | 5 | 87 | 87 | 87 | 87 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 87 | 87 | 87 | 87 | 5 | 32 | 46 | 59 | 4 | 32 | 59 | 84 | 87 | 87 | 87 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 1900 | 68100 | 72290 | 73150 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 18710 | 1900 | 68100 | 72310 | 73170 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 18740 | 1510 | 68110 | 72250 |
| | | transfpcacoveigs | 5 | 400 | 465 | 470 | 510 | 550 | 190 | 250 | 330 | 545 | 190 | 325 | 550 | 420 | 465 | 470 | 470 | 550 | 180 | 200 | 340 | 545 | 180 | 305 | 580 | 410 | 475 | 440 |
| | | transfpcacovsvd | 5 | 400 | 465 | 470 | 510 | 550 | 190 | 250 | 330 | 545 | 190 | 325 | 550 | 420 | 465 | 470 | 470 | 550 | 180 | 200 | 340 | 545 | 180 | 305 | 580 | 410 | 475 | 440 |
| | | transfsvd | 5 | 375 | 475 | 490 | 505 | 690 | 340 | 505 | 490 | 680 | 340 | 490 | 630 | 425 | 465 | 470 | 485 | 690 | 210 | 285 | 330 | 680 | 210 | 330 | 600 | 395 | 475 | 480 |
| | | transffin | 5 | 515 | 515 | 525 | 545 | 690 | 340 | 505 | 490 | 680 | 340 | 490 | 630 | 475 | 475 | 495 | 515 | 690 | 210 | 285 | 330 | 680 | 210 | 330 | 600 | 475 | 505 | 515 |
| | | transfpcacorreigs | 5 | 435 | 470 | 475 | 470 | 695 | 760 | 630 | 745 | 690 | 760 | 575 | 540 | 445 | 480 | 485 | 480 | 695 | 220 | 280 | 415 | 690 | 220 | 315 | 420 | 435 | 490 | 505 |
| | | transfpcacorrsvd | 5 | 435 | 470 | 475 | 470 | 695 | 760 | 630 | 745 | 690 | 760 | 575 | 540 | 445 | 480 | 485 | 480 | 695 | 220 | 280 | 415 | 690 | 220 | 315 | 420 | 435 | 490 | 505 |
| | | transfsvdnormalized | 5 | 435 | 470 | 475 | 470 | 695 | 760 | 630 | 745 | 690 | 760 | 575 | 540 | 445 | 480 | 485 | 480 | 695 | 220 | 280 | 415 | 690 | 220 | 315 | 420 | 435 | 490 | 505 |
| | | transffinnormalized | 5 | 475 | 495 | 495 | 495 | 695 | 760 | 630 | 745 | 690 | 760 | 575 | 540 | 495 | 495 | 515 | 515 | 695 | 220 | 280 | 415 | 690 | 220 | 315 | 420 | 465 | 505 | 555 |
| | | minimum | | 375 | 465 | 470 | 470 | 550 | 190 | 250 | 330 | 545 | 190 | 325 | 540 | 420 | 465 | 470 | 470 | 550 | 180 | 200 | 330 | 545 | 180 | 305 | 420 | 395 | 475 | 440 |
| | Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1

Table 5.9: Results of the algorithm "Death of immune cells with insufficient stimulation" b

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | Distance measur. method | | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 0.25 | A | notransf | 840 | 106 | 106 | 9 | 7 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 9 | 9 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 7 |
| | | transfpcacoveigs | 5 | 12 | 9 | 9 | 12 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 22 | 67 | 11 | 10 | 10 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 21 | 67 | 7 | 24 |
| | | transfpcacovsvd | 5 | 12 | 9 | 9 | 12 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 22 | 67 | 11 | 10 | 10 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 21 | 67 | 7 | 24 |
| | | transfsvd | 5 | 91 | 9 | 8 | 10 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 24 | 67 | 12 | 9 | 9 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 32 | 67 | 7 | 13 |
| | | transffin | 5 | 12 | 8 | 9 | 11 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 24 | 9 | 7 | 9 | 9 | 106 | 106 | 106 | 53 | 106 | 106 | 53 | 32 | 9 | 7 | 10 |
| | | transfpcacorreigs | 5 | 12 | 6 | 6 | 7 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 33 | 13 | 10 | 10 | 10 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 2 | 67 | 7 | 11 |
| | | transfpcacorrsvd | 5 | 12 | 6 | 6 | 7 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 33 | 13 | 10 | 10 | 10 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 2 | 67 | 7 | 11 |
| | | transfsvdnormalized | 5 | 12 | 6 | 6 | 7 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 33 | 13 | 10 | 10 | 10 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 2 | 67 | 7 | 11 |
| | | transffinnormalized | 5 | 4 | 6 | 6 | 6 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 33 | 6 | 6 | 8 | 8 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 2 | 3 | 7 | 12 |
| | B | notransf | 840 | 1 | 1 | 64 | 86 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 64 | 86 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 64 |
| | | transfpcacoveigs | 5 | 3 | 17 | 27 | 38 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 14 | 3 | 17 | 27 | 38 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 14 | 3 | 17 | 27 |
| | | transfpcacovsvd | 5 | 3 | 17 | 27 | 38 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 14 | 3 | 17 | 27 | 38 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 14 | 3 | 17 | 27 |
| | | transfsvd | 5 | 2 | 18 | 24 | 35 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 25 | 2 | 18 | 24 | 35 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 25 | 2 | 18 | 24 |
| | | transffin | 5 | 74 | 81 | 84 | 84 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 25 | 74 | 81 | 84 | 84 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 25 | 74 | 81 | 84 |
| | | transfpcacorreigs | 5 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 |
| | | transfpcacorrsvd | 5 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 |
| | | transfsvdnormalized | 5 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 | 47 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 8 | 28 | 42 |
| | | transffinnormalized | 5 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 26 | 87 | 87 | 87 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 1900 | 1900 | 53850 | 72310 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 53850 | 72330 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1510 | 1900 | 53830 |
| | | transfpcacoveigs | 5 | 135 | 175 | 225 | 310 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 290 | 685 | 195 | 235 | 290 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 280 | 685 | 155 | 375 |
| | | transfpcacovsvd | 5 | 135 | 175 | 225 | 310 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 290 | 685 | 195 | 235 | 290 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 280 | 685 | 155 | 375 |
| | | transfsvd | 5 | 920 | 180 | 200 | 275 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 365 | 680 | 210 | 210 | 265 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 445 | 680 | 160 | 250 |
| | | transffin | 5 | 490 | 485 | 510 | 530 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 365 | 460 | 475 | 510 | 510 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 445 | 460 | 475 | 520 |
| | | transfpcacorreigs | 5 | 160 | 200 | 270 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 460 | 170 | 240 | 310 | 335 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 150 | 710 | 210 | 320 |
| | | transfpcacorrsvd | 5 | 160 | 200 | 270 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 460 | 170 | 240 | 310 | 335 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 150 | 710 | 210 | 320 |
| | | transfsvdnormalized | 5 | 160 | 200 | 270 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 460 | 170 | 240 | 310 | 335 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 150 | 710 | 210 | 320 |
| | | transffinnormalized | 5 | 475 | 495 | 495 | 495 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 460 | 495 | 495 | 515 | 515 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 150 | 465 | 505 | 555 |
| | | minimum | | 135 | 175 | 200 | 275 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 290 | 170 | 195 | 210 | 265 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 540 | 150 | 460 | 155 | 250 |
| | Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 0.25

Table 5.10: Results of the algorithm "Death of immune cells with insufficient stimulation" c

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 0.1 | A | notransf | 840 | 106 | 106 | 99 | 8 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 9 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfpcacoveigs | 5 | 106 | 106 | 106 | 29 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transfpcacovsvd | 5 | 106 | 106 | 106 | 29 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transfsvd | 5 | 106 | 106 | 106 | 6 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transffin | 5 | 19 | 13 | 12 | 11 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 12 | 23 | 10 | 10 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 10 | 14 | 5 |
| | | transfpcacorreigs | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfpcacorrsvd | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfsvdnormalized | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transffinnormalized | 5 | 4 | 6 | 6 | 6 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 6 | 6 | 8 | 8 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 3 | 7 | 12 |
| | B | notransf | 840 | 1 | 1 | 2 | 80 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 80 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfpcacoveigs | 5 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transfpcacovsvd | 5 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transfsvd | 5 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transffin | 5 | 27 | 61 | 70 | 72 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 27 | 61 | 70 | 72 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 27 | 61 | 70 |
| | | transfpcacorreigs | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfpcacorrsvd | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfsvdnormalized | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transffinnormalized | 5 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 87 | 87 | 87 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 1900 | 1900 | 2670 | 67280 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 2350 | 67290 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1510 | 1900 | 2350 |
| | | transfpcacoveigs | 5 | 1065 | 1065 | 1065 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 685 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transfpcacovsvd | 5 | 1065 | 1065 | 1065 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 685 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transfsvd | 5 | 1065 | 1065 | 1065 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 690 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transffin | 5 | 325 | 435 | 470 | 470 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 255 | 535 | 450 | 460 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 235 | 445 | 400 |
| | | transfpcacorreigs | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transfpcacorrsvd | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transfsvdnormalized | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transffinnormalized | 5 | 475 | 495 | 495 | 495 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 495 | 495 | 515 | 515 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 465 | 505 | 555 |
| | | minimum | | 325 | 435 | 470 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 255 | 495 | 450 | 460 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 235 | 445 | 400 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 0.1

Table 5.11: Results of the algorithm "Elimination of auto-reactive immune cells" a

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 0.7 | A | notransf | 840 | 1 | 4 | 12 | 67 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 10 | 4 | 7 | 8 | 67 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 3 | 53 |
| | | transfpcacoveigs | 5 | 3 | 3 | 6 | 8 | 7 | 7 | 7 | 9 | 6 | 7 | 7 | 14 | 3 | 7 | 13 | 12 | 5 | 6 | 6 | 3 | 5 | 6 | 5 | 2 | 8 | 3 | 7 |
| | | transfpcacovsvd | 5 | 3 | 3 | 6 | 8 | 7 | 7 | 7 | 9 | 6 | 7 | 7 | 14 | 3 | 7 | 13 | 12 | 5 | 6 | 6 | 3 | 5 | 6 | 5 | 2 | 8 | 3 | 7 |
| | | transfsvd | 5 | 4 | 3 | 4 | 3 | 19 | 19 | 18 | 11 | 17 | 19 | 21 | 37 | 5 | 7 | 11 | 8 | 3 | 5 | 5 | 13 | 1 | 5 | 6 | 16 | 8 | 3 | 6 |
| | | transffin | 5 | 25 | 67 | 67 | 67 | 19 | 19 | 18 | 11 | 17 | 19 | 21 | 37 | 31 | 67 | 67 | 67 | 3 | 5 | 5 | 13 | 1 | 5 | 6 | 16 | 67 | 67 | 67 |
| | | transfpcacorreigs | 5 | 4 | 4 | 5 | 6 | 22 | 26 | 27 | 28 | 22 | 26 | 23 | 23 | 8 | 10 | 13 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 11 | 4 | 6 |
| | | transfpcacorrsvd | 5 | 4 | 4 | 5 | 6 | 22 | 26 | 27 | 28 | 22 | 26 | 23 | 23 | 8 | 10 | 13 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 11 | 4 | 6 |
| | | transfsvdnormalized | 5 | 4 | 4 | 5 | 6 | 22 | 26 | 27 | 28 | 22 | 26 | 23 | 23 | 8 | 10 | 13 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 11 | 4 | 6 |
| | | transffinnormalized | 5 | 67 | 67 | 67 | 67 | 22 | 26 | 27 | 28 | 22 | 26 | 23 | 23 | 67 | 67 | 67 | 67 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 67 | 67 | 67 |
| | B | notransf | 840 | 87 | 25 | 7 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 25 | 7 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 25 | 7 |
| | | transfpcacoveigs | 5 | 54 | 24 | 19 | 16 | 86 | 82 | 76 | 67 | 87 | 82 | 68 | 22 | 54 | 24 | 19 | 16 | 86 | 82 | 76 | 67 | 87 | 82 | 68 | 22 | 54 | 24 | 19 |
| | | transfpcacovsvd | 5 | 54 | 24 | 19 | 16 | 86 | 82 | 76 | 67 | 87 | 82 | 68 | 22 | 54 | 24 | 19 | 16 | 86 | 82 | 76 | 67 | 87 | 82 | 68 | 22 | 54 | 24 | 19 |
| | | transfsvd | 5 | 54 | 23 | 17 | 12 | 86 | 80 | 73 | 61 | 86 | 80 | 65 | 17 | 54 | 23 | 17 | 12 | 86 | 80 | 73 | 61 | 86 | 80 | 65 | 17 | 54 | 23 | 17 |
| | | transffin | 5 | 5 | 1 | 1 | 1 | 86 | 80 | 73 | 61 | 86 | 80 | 65 | 17 | 5 | 1 | 1 | 1 | 86 | 80 | 73 | 61 | 86 | 80 | 65 | 17 | 5 | 1 | 1 |
| | | transfpcacorreigs | 5 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 |
| | | transfpcacorrsvd | 5 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 |
| | | transfsvdnormalized | 5 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 | 12 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 38 | 18 | 14 |
| | | transffinnormalized | 5 | 1 | 1 | 1 | 1 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 1 | 1 | 1 | 1 | 87 | 75 | 69 | 59 | 87 | 75 | 60 | 12 | 1 | 1 | 1 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 21040 | 6000 | 1510 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 73180 | 73120 | 21070 | 5960 | 1510 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 73160 | 73650 | 21030 | 6410 |
| | | transfpcacoveigs | 5 | 300 | 150 | 155 | 160 | 500 | 480 | 450 | 425 | 495 | 480 | 410 | 250 | 300 | 190 | 225 | 200 | 480 | 470 | 440 | 365 | 485 | 470 | 390 | 130 | 350 | 150 | 165 |
| | | transfpcacovsvd | 5 | 300 | 150 | 155 | 160 | 500 | 480 | 450 | 425 | 495 | 480 | 410 | 250 | 300 | 190 | 225 | 200 | 480 | 470 | 440 | 365 | 485 | 470 | 390 | 130 | 350 | 150 | 165 |
| | | transfsvd | 5 | 310 | 145 | 125 | 90 | 620 | 590 | 545 | 415 | 600 | 590 | 535 | 455 | 320 | 185 | 195 | 140 | 460 | 450 | 415 | 435 | 440 | 450 | 385 | 245 | 350 | 145 | 145 |
| | | transffin | 5 | 275 | 675 | 675 | 675 | 620 | 590 | 545 | 415 | 600 | 590 | 535 | 455 | 335 | 675 | 675 | 675 | 460 | 450 | 415 | 435 | 440 | 450 | 385 | 245 | 695 | 675 | 675 |
| | | transfpcacorreigs | 5 | 230 | 130 | 120 | 120 | 655 | 635 | 615 | 575 | 655 | 635 | 530 | 290 | 270 | 190 | 200 | 150 | 445 | 375 | 355 | 305 | 435 | 375 | 300 | 230 | 300 | 130 | 130 |
| | | transfpcacorrsvd | 5 | 230 | 130 | 120 | 120 | 655 | 635 | 615 | 575 | 655 | 635 | 530 | 290 | 270 | 190 | 200 | 150 | 445 | 375 | 355 | 305 | 435 | 375 | 300 | 230 | 300 | 130 | 130 |
| | | transfsvdnormalized | 5 | 230 | 130 | 120 | 120 | 655 | 635 | 615 | 575 | 655 | 635 | 530 | 290 | 270 | 190 | 200 | 150 | 445 | 375 | 355 | 305 | 435 | 375 | 300 | 230 | 300 | 130 | 130 |
| | | transffinnormalized | 5 | 675 | 675 | 675 | 675 | 655 | 635 | 615 | 575 | 655 | 635 | 530 | 290 | 675 | 675 | 675 | 675 | 445 | 375 | 355 | 305 | 435 | 375 | 300 | 230 | 675 | 675 | 675 |
| | | minimum | | 230 | 130 | 120 | 90 | 500 | 480 | 450 | 415 | 495 | 480 | 410 | 250 | 270 | 185 | 195 | 140 | 445 | 375 | 355 | 305 | 435 | 375 | 300 | 130 | 300 | 130 | 130 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 0.7

Table 5.12: Results of the algorithm "Elimination of auto-reactive immune cells" b

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance standard score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | | | |
| | Distance measur. | method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1 | A | notransf | 840 | 1 | 4 | 17 | 67 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 12 | 4 | 8 | 67 | 67 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 9 | 57 | 5 | 67 |
| | | transfpcacoveigs | 5 | 2 | 2 | 2 | 4 | 7 | 9 | 13 | 12 | 6 | 9 | 8 | 3 | 7 | 8 | 5 | 8 | 5 | 6 | 4 | 9 | 5 | 6 | 6 | 9 | 17 | 4 | 32 |
| | | transfpcacovsvd | 5 | 2 | 2 | 2 | 4 | 7 | 9 | 13 | 12 | 6 | 9 | 8 | 3 | 7 | 8 | 5 | 8 | 5 | 6 | 4 | 9 | 5 | 6 | 6 | 9 | 17 | 4 | 32 |
| | | transfsvd | 5 | 3 | 3 | 2 | 3 | 19 | 20 | 22 | 29 | 17 | 20 | 30 | 23 | 10 | 9 | 4 | 4 | 3 | 6 | 6 | 18 | 1 | 6 | 18 | 20 | 15 | 5 | 16 |
| | | transffin | 5 | 58 | 67 | 67 | 67 | 19 | 20 | 22 | 29 | 17 | 20 | 30 | 23 | 67 | 67 | 67 | 67 | 3 | 6 | 6 | 18 | 1 | 6 | 18 | 20 | 67 | 67 | 67 |
| | | transfpcacorreigs | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transfpcacorrsvd | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transfsvdnormalized | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transffinnormalized | 5 | 67 | 67 | 67 | 67 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 67 | 67 | 67 | 67 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 67 | 67 | 67 |
| | B | notransf | 840 | 87 | 11 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 70 | 87 | 11 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 70 | 87 | 11 | 3 |
| | | transfpcacoveigs | 5 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 |
| | | transfpcacovsvd | 5 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 |
| | | transfsvd | 5 | 33 | 11 | 10 | 10 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 33 | 11 | 10 | 10 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 33 | 11 | 10 |
| | | transffin | 5 | 2 | 1 | 1 | 1 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 2 | 1 | 1 | 1 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 2 | 1 | 1 |
| | | transfpcacorreigs | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transfpcacorrsvd | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transfsvdnormalized | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transffinnormalized | 5 | 1 | 1 | 1 | 1 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 1 | 1 | 1 | 1 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 1 | 1 | 1 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9280 | 2690 | 1510 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 58920 | 73120 | 9320 | 3190 | 1510 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 58890 | 73650 | 9290 | 3190 |
| | | transfpcacoveigs | 5 | 190 | 80 | 70 | 90 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 240 | 140 | 100 | 130 | 475 | 405 | 335 | 330 | 475 | 405 | 320 | 150 | 340 | 100 | 370 |
| | | transfpcacovsvd | 5 | 190 | 80 | 70 | 90 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 240 | 140 | 100 | 130 | 475 | 405 | 335 | 330 | 475 | 405 | 320 | 150 | 340 | 100 | 370 |
| | | transfsvd | 5 | 195 | 85 | 70 | 80 | 615 | 520 | 485 | 505 | 600 | 520 | 515 | 290 | 265 | 145 | 90 | 90 | 455 | 380 | 325 | 395 | 440 | 380 | 395 | 260 | 315 | 105 | 210 |
| | | transffin | 5 | 590 | 675 | 675 | 675 | 615 | 520 | 485 | 505 | 600 | 520 | 515 | 290 | 680 | 675 | 675 | 675 | 455 | 380 | 325 | 395 | 440 | 380 | 395 | 260 | 680 | 675 | 675 |
| | | transfpcacorreigs | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transfpcacorrsvd | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transfsvdnormalized | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transffinnormalized | 5 | 675 | 675 | 675 | 675 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 675 | 675 | 675 | 675 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 675 | 675 | 675 |
| | | minimum | | 145 | 70 | 70 | 80 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 205 | 80 | 75 | 90 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 150 | 315 | 90 | 145 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1

Table 5.13: Results of the algorithm "Elimination of auto-reactive immune cells" c

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | Distance measur. method | | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1.2 | A | notransf | 840 | 1 | 2 | 33 | 67 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 12 | 4 | 4 | 67 | 67 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 8 | 57 | 4 | 67 |
| | | transfpcacoveigs | 5 | 3 | 3 | 7 | 8 | 7 | 11 | 11 | 12 | 6 | 11 | 11 | 32 | 7 | 4 | 6 | 7 | 5 | 5 | 6 | 8 | 5 | 5 | 9 | 38 | 27 | 4 | 41 |
| | | transfpcacovsvd | 5 | 3 | 3 | 7 | 8 | 7 | 11 | 11 | 12 | 6 | 11 | 11 | 32 | 7 | 4 | 6 | 7 | 5 | 5 | 6 | 8 | 5 | 5 | 9 | 38 | 27 | 4 | 41 |
| | | transfsvd | 5 | 5 | 4 | 6 | 12 | 19 | 22 | 27 | 23 | 17 | 22 | 31 | 61 | 8 | 4 | 6 | 24 | 4 | 6 | 15 | 11 | 1 | 6 | 18 | 63 | 45 | 4 | 17 |
| | | transffin | 5 | 67 | 67 | 67 | 67 | 19 | 22 | 27 | 23 | 17 | 22 | 31 | 61 | 67 | 67 | 67 | 67 | 4 | 6 | 15 | 11 | 1 | 6 | 18 | 63 | 67 | 67 | 67 |
| | | transfpcacorreigs | 5 | 3 | 5 | 11 | 10 | 21 | 20 | 25 | 11 | 22 | 20 | 28 | 61 | 9 | 4 | 9 | 9 | 1 | 0 | 0 | 6 | 0 | 0 | 0 | 61 | 50 | 5 | 10 |
| | | transfpcacorrsvd | 5 | 3 | 5 | 11 | 10 | 21 | 20 | 25 | 11 | 22 | 20 | 28 | 61 | 9 | 4 | 9 | 9 | 1 | 0 | 0 | 6 | 0 | 0 | 0 | 61 | 50 | 5 | 10 |
| | | transfsvdnormalized | 5 | 3 | 5 | 11 | 10 | 21 | 20 | 25 | 11 | 22 | 20 | 28 | 61 | 9 | 4 | 9 | 9 | 1 | 0 | 0 | 6 | 0 | 0 | 0 | 61 | 50 | 5 | 10 |
| | | transffinnormalized | 5 | 67 | 67 | 67 | 67 | 21 | 20 | 25 | 11 | 22 | 20 | 28 | 61 | 67 | 67 | 67 | 67 | 1 | 0 | 0 | 6 | 0 | 0 | 0 | 61 | 67 | 67 | 67 |
| | B | notransf | 840 | 87 | 10 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 42 | 87 | 10 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 42 | 87 | 10 | 3 |
| | | transfpcacoveigs | 5 | 25 | 10 | 9 | 8 | 83 | 56 | 47 | 41 | 85 | 56 | 39 | 9 | 25 | 10 | 9 | 8 | 83 | 56 | 47 | 41 | 85 | 56 | 39 | 9 | 25 | 10 | 9 |
| | | transfpcacovsvd | 5 | 25 | 10 | 9 | 8 | 83 | 56 | 47 | 41 | 85 | 56 | 39 | 9 | 25 | 10 | 9 | 8 | 83 | 56 | 47 | 41 | 85 | 56 | 39 | 9 | 25 | 10 | 9 |
| | | transfsvd | 5 | 25 | 10 | 8 | 6 | 81 | 51 | 41 | 34 | 85 | 51 | 35 | 7 | 25 | 10 | 8 | 6 | 81 | 51 | 41 | 34 | 85 | 51 | 35 | 7 | 25 | 10 | 8 |
| | | transffin | 5 | 1 | 1 | 1 | 1 | 81 | 51 | 41 | 34 | 85 | 51 | 35 | 7 | 1 | 1 | 1 | 1 | 81 | 51 | 41 | 34 | 85 | 51 | 35 | 7 | 1 | 1 | 1 |
| | | transfpcacorreigs | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 |
| | | transfpcacorrsvd | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 |
| | | transfsvdnormalized | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 | 7 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 19 | 8 | 7 |
| | | transffinnormalized | 5 | 1 | 1 | 1 | 1 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 1 | 1 | 1 | 1 | 77 | 46 | 37 | 26 | 84 | 46 | 28 | 5 | 1 | 1 | 1 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 8420 | 2850 | 1510 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 35400 | 73120 | 8440 | 3190 | 1510 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 35360 | 73650 | 8440 | 3190 |
| | | transfpcacoveigs | 5 | 155 | 80 | 115 | 120 | 485 | 390 | 345 | 325 | 485 | 390 | 305 | 365 | 195 | 90 | 105 | 110 | 465 | 330 | 295 | 285 | 475 | 330 | 285 | 425 | 395 | 90 | 455 |
| | | transfpcacovsvd | 5 | 155 | 80 | 115 | 120 | 485 | 390 | 345 | 325 | 485 | 390 | 305 | 365 | 195 | 90 | 105 | 110 | 465 | 330 | 295 | 285 | 475 | 330 | 285 | 425 | 395 | 90 | 455 |
| | | transfsvd | 5 | 175 | 90 | 100 | 150 | 595 | 475 | 475 | 400 | 595 | 475 | 485 | 645 | 205 | 90 | 100 | 270 | 445 | 315 | 355 | 280 | 435 | 315 | 355 | 665 | 575 | 90 | 210 |
| | | transffin | 5 | 675 | 675 | 675 | 675 | 595 | 475 | 475 | 400 | 595 | 475 | 485 | 645 | 675 | 675 | 675 | 675 | 445 | 315 | 355 | 280 | 435 | 315 | 355 | 665 | 675 | 675 | 675 |
| | | transfpcacorreigs | 5 | 125 | 90 | 145 | 135 | 595 | 430 | 435 | 240 | 640 | 430 | 420 | 635 | 185 | 80 | 125 | 125 | 395 | 230 | 185 | 190 | 420 | 230 | 140 | 635 | 595 | 90 | 135 |
| | | transfpcacorrsvd | 5 | 125 | 90 | 145 | 135 | 595 | 430 | 435 | 240 | 640 | 430 | 420 | 635 | 185 | 80 | 125 | 125 | 395 | 230 | 185 | 190 | 420 | 230 | 140 | 635 | 595 | 90 | 135 |
| | | transfsvdnormalized | 5 | 125 | 90 | 145 | 135 | 595 | 430 | 435 | 240 | 640 | 430 | 420 | 635 | 185 | 80 | 125 | 125 | 395 | 230 | 185 | 190 | 420 | 230 | 140 | 635 | 595 | 90 | 135 |
| | | transffinnormalized | 5 | 675 | 675 | 675 | 675 | 595 | 430 | 435 | 240 | 640 | 430 | 420 | 635 | 675 | 675 | 675 | 675 | 395 | 230 | 185 | 190 | 420 | 230 | 140 | 635 | 675 | 675 | 675 |
| | | minimum | | 125 | 80 | 100 | 120 | 485 | 390 | 345 | 240 | 485 | 390 | 305 | 365 | 185 | 80 | 100 | 110 | 395 | 230 | 185 | 190 | 420 | 230 | 140 | 425 | 395 | 90 | 135 |
| | Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1.2

Table 5.14: Results of the algorithm "Apoptosis and auto-immunization" a

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance standard score | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1.5 | A | notransf | 840 | 1 | 21 | 39 | 42 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 14 | 4 | 15 | 22 | 13 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 22 | 57 | 4 | 52 |
| | | transfpcacoveigs | 5 | 7 | 25 | 26 | 27 | 7 | 13 | 19 | 18 | 6 | 13 | 20 | 93 | 6 | 15 | 15 | 14 | 5 | 6 | 12 | 4 | 5 | 6 | 8 | 100 | 53 | 4 | 47 |
| | | transfpcacovsvd | 5 | 7 | 25 | 26 | 27 | 7 | 13 | 19 | 18 | 6 | 13 | 20 | 93 | 6 | 15 | 15 | 14 | 5 | 6 | 12 | 4 | 5 | 6 | 8 | 100 | 53 | 4 | 47 |
| | | transfsvd | 5 | 9 | 26 | 24 | 41 | 21 | 23 | 23 | 32 | 19 | 23 | 38 | 97 | 5 | 16 | 18 | 22 | 6 | 13 | 12 | 8 | 3 | 13 | 29 | 98 | 53 | 4 | 37 |
| | | transffin | 5 | 42 | 40 | 40 | 41 | 21 | 23 | 23 | 32 | 19 | 23 | 38 | 97 | 30 | 29 | 32 | 37 | 6 | 13 | 12 | 8 | 3 | 13 | 29 | 98 | 53 | 37 | 48 |
| | | transfpcacorreigs | 5 | 11 | 22 | 24 | 41 | 24 | 26 | 32 | 37 | 23 | 26 | 35 | 98 | 5 | 15 | 14 | 24 | 1 | 0 | 4 | 8 | 0 | 0 | 6 | 103 | 53 | 3 | 46 |
| | | transfpcacorrsvd | 5 | 11 | 22 | 24 | 41 | 24 | 26 | 32 | 37 | 23 | 26 | 35 | 98 | 5 | 15 | 14 | 24 | 1 | 0 | 4 | 8 | 0 | 0 | 6 | 103 | 53 | 3 | 46 |
| | | transfsvdnormalized | 5 | 11 | 22 | 24 | 41 | 24 | 26 | 32 | 37 | 23 | 26 | 35 | 98 | 5 | 15 | 14 | 24 | 1 | 0 | 4 | 8 | 0 | 0 | 6 | 103 | 53 | 3 | 46 |
| | | transffinnormalized | 5 | 40 | 42 | 43 | 44 | 24 | 26 | 32 | 37 | 23 | 26 | 35 | 98 | 32 | 33 | 34 | 35 | 1 | 0 | 4 | 8 | 0 | 0 | 6 | 103 | 53 | 1 | 64 |
| | B | notransf | 840 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 19 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 19 | 87 | 11 | 9 |
| | | transfpcacoveigs | 5 | 19 | 11 | 11 | 11 | 78 | 44 | 35 | 25 | 83 | 44 | 26 | 7 | 19 | 11 | 11 | 11 | 78 | 44 | 35 | 25 | 83 | 44 | 26 | 7 | 19 | 11 | 11 |
| | | transfpcacovsvd | 5 | 19 | 11 | 11 | 11 | 78 | 44 | 35 | 25 | 83 | 44 | 26 | 7 | 19 | 11 | 11 | 11 | 78 | 44 | 35 | 25 | 83 | 44 | 26 | 7 | 19 | 11 | 11 |
| | | transfsvd | 5 | 18 | 11 | 11 | 8 | 73 | 39 | 34 | 24 | 80 | 39 | 24 | 8 | 18 | 11 | 11 | 8 | 73 | 39 | 34 | 24 | 80 | 39 | 24 | 8 | 18 | 11 | 11 |
| | | transffin | 5 | 23 | 21 | 21 | 19 | 73 | 39 | 34 | 24 | 80 | 39 | 24 | 8 | 23 | 21 | 21 | 19 | 73 | 39 | 34 | 24 | 80 | 39 | 24 | 8 | 23 | 21 | 21 |
| | | transfpcacorreigs | 5 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 |
| | | transfpcacorrsvd | 5 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 |
| | | transfsvdnormalized | 5 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 | 13 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 16 | 11 | 13 |
| | | transffinnormalized | 5 | 13 | 15 | 15 | 17 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 13 | 15 | 15 | 17 | 65 | 34 | 28 | 23 | 77 | 34 | 21 | 6 | 13 | 15 | 15 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9450 | 7950 | 8820 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 16100 | 73120 | 9390 | 7780 | 8530 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 16180 | 73650 | 9280 | 8080 |
| | | transfpcacoveigs | 5 | 165 | 305 | 315 | 325 | 460 | 350 | 365 | 305 | 475 | 350 | 330 | 965 | 155 | 205 | 205 | 195 | 440 | 280 | 295 | 165 | 465 | 280 | 210 | 1035 | 625 | 95 | 525 |
| | | transfpcacovsvd | 5 | 165 | 305 | 315 | 325 | 460 | 350 | 365 | 305 | 475 | 350 | 330 | 965 | 155 | 205 | 205 | 195 | 440 | 280 | 295 | 165 | 465 | 280 | 210 | 1035 | 625 | 95 | 525 |
| | | transfsvd | 5 | 180 | 315 | 295 | 450 | 575 | 425 | 400 | 440 | 590 | 425 | 500 | 1010 | 140 | 215 | 235 | 260 | 425 | 325 | 290 | 200 | 430 | 325 | 410 | 1020 | 620 | 95 | 425 |
| | | transffin | 5 | 535 | 505 | 505 | 505 | 575 | 425 | 400 | 440 | 590 | 425 | 500 | 1010 | 415 | 395 | 425 | 465 | 425 | 325 | 290 | 200 | 430 | 325 | 410 | 1020 | 645 | 475 | 585 |
| | | transfpcacorreigs | 5 | 190 | 275 | 305 | 475 | 565 | 430 | 460 | 485 | 615 | 430 | 455 | 1010 | 130 | 205 | 205 | 305 | 335 | 170 | 180 | 195 | 385 | 170 | 165 | 1060 | 610 | 85 | 525 |
| | | transfpcacorrsvd | 5 | 190 | 275 | 305 | 475 | 565 | 430 | 460 | 485 | 615 | 430 | 455 | 1010 | 130 | 205 | 205 | 305 | 335 | 170 | 180 | 195 | 385 | 170 | 165 | 1060 | 610 | 85 | 525 |
| | | transfsvdnormalized | 5 | 190 | 275 | 305 | 475 | 565 | 430 | 460 | 485 | 615 | 430 | 455 | 1010 | 130 | 205 | 205 | 305 | 335 | 170 | 180 | 195 | 385 | 170 | 165 | 1060 | 610 | 85 | 525 |
| | | transffinnormalized | 5 | 465 | 495 | 505 | 525 | 565 | 430 | 460 | 485 | 615 | 430 | 455 | 1010 | 385 | 405 | 415 | 435 | 335 | 170 | 180 | 195 | 385 | 170 | 165 | 1060 | 595 | 85 | 715 |
| | | minimum | | 165 | 275 | 295 | 325 | 460 | 350 | 365 | 305 | 475 | 350 | 330 | 965 | 130 | 205 | 205 | 195 | 335 | 170 | 180 | 165 | 385 | 170 | 165 | 1020 | 595 | 85 | 425 |
| | Total test vectors | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1.5

Table 5.15: Results of the algorithm "Apoptosis and auto-immunization" b

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1.75 | A | notransf | 840 | 1 | 21 | 39 | 42 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 59 | 4 | 15 | 22 | 13 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 59 | 57 | 4 | 52 |
| | | transfpcacoveigs | 5 | 6 | 25 | 42 | 42 | 9 | 15 | 25 | 30 | 7 | 15 | 35 | 104 | 7 | 15 | 15 | 15 | 5 | 11 | 7 | 22 | 5 | 11 | 19 | 106 | 53 | 4 | 48 |
| | | transfpcacovsvd | 5 | 6 | 25 | 42 | 42 | 9 | 15 | 25 | 30 | 7 | 15 | 35 | 104 | 7 | 15 | 15 | 15 | 5 | 11 | 7 | 22 | 5 | 11 | 19 | 106 | 53 | 4 | 48 |
| | | transfsvd | 5 | 15 | 26 | 41 | 41 | 22 | 28 | 30 | 40 | 21 | 28 | 35 | 94 | 8 | 16 | 19 | 22 | 6 | 11 | 20 | 15 | 3 | 11 | 22 | 91 | 53 | 4 | 48 |
| | | transffin | 5 | 42 | 40 | 40 | 41 | 22 | 28 | 30 | 40 | 21 | 28 | 35 | 94 | 30 | 29 | 32 | 37 | 6 | 11 | 20 | 15 | 3 | 11 | 22 | 91 | 53 | 37 | 48 |
| | | transfpcacorreigs | 5 | 9 | 39 | 40 | 41 | 19 | 29 | 35 | 38 | 24 | 29 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfpcacorrsvd | 5 | 9 | 39 | 40 | 41 | 19 | 29 | 35 | 38 | 24 | 29 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfsvdnormalized | 5 | 9 | 39 | 40 | 41 | 19 | 29 | 35 | 38 | 24 | 29 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transffinnormalized | 5 | 40 | 42 | 43 | 44 | 19 | 29 | 35 | 38 | 24 | 29 | 37 | 69 | 32 | 33 | 34 | 35 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 64 |
| | B | notransf | 840 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 9 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 9 | 87 | 11 | 9 |
| | | transfpcacoveigs | 5 | 14 | 11 | 10 | 10 | 73 | 36 | 24 | 22 | 80 | 36 | 20 | 5 | 14 | 11 | 10 | 10 | 73 | 36 | 24 | 22 | 80 | 36 | 20 | 5 | 14 | 11 | 10 |
| | | transfpcacovsvd | 5 | 14 | 11 | 10 | 10 | 73 | 36 | 24 | 22 | 80 | 36 | 20 | 5 | 14 | 11 | 10 | 10 | 73 | 36 | 24 | 22 | 80 | 36 | 20 | 5 | 14 | 11 | 10 |
| | | transfsvd | 5 | 16 | 11 | 9 | 7 | 67 | 32 | 25 | 21 | 75 | 32 | 23 | 7 | 16 | 11 | 9 | 7 | 67 | 32 | 25 | 21 | 75 | 32 | 23 | 7 | 16 | 11 | 9 |
| | | transffin | 5 | 23 | 21 | 21 | 19 | 67 | 32 | 25 | 21 | 75 | 32 | 23 | 7 | 23 | 21 | 21 | 19 | 67 | 32 | 25 | 21 | 75 | 32 | 23 | 7 | 23 | 21 | 21 |
| | | transfpcacorreigs | 5 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 |
| | | transfpcacorrsvd | 5 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 |
| | | transfsvdnormalized | 5 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 | 13 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 15 | 11 | 12 |
| | | transffinnormalized | 5 | 13 | 15 | 15 | 17 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 13 | 15 | 15 | 17 | 60 | 28 | 22 | 17 | 71 | 28 | 17 | 9 | 13 | 15 | 15 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9450 | 7950 | 8820 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 8150 | 73120 | 9390 | 7780 | 8530 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 8150 | 73650 | 9280 | 8080 |
| | | transfpcacoveigs | 5 | 130 | 305 | 470 | 470 | 455 | 330 | 370 | 410 | 470 | 330 | 450 | 1065 | 140 | 205 | 200 | 200 | 415 | 290 | 190 | 330 | 450 | 290 | 290 | 1085 | 600 | 95 | 530 |
| | | transfpcacovsvd | 5 | 130 | 305 | 470 | 470 | 455 | 330 | 370 | 410 | 470 | 330 | 450 | 1065 | 140 | 205 | 200 | 200 | 415 | 290 | 190 | 330 | 450 | 290 | 290 | 1085 | 600 | 95 | 530 |
| | | transfsvd | 5 | 230 | 315 | 455 | 445 | 555 | 440 | 425 | 505 | 585 | 440 | 465 | 975 | 160 | 215 | 235 | 255 | 395 | 270 | 325 | 255 | 405 | 270 | 335 | 945 | 610 | 95 | 525 |
| | | transffin | 5 | 535 | 505 | 505 | 505 | 555 | 440 | 425 | 505 | 585 | 440 | 465 | 975 | 415 | 395 | 425 | 465 | 395 | 270 | 325 | 255 | 405 | 270 | 335 | 945 | 645 | 475 | 585 |
| | | transfpcacorreigs | 5 | 165 | 445 | 460 | 475 | 490 | 430 | 460 | 465 | 595 | 430 | 455 | 735 | 155 | 215 | 280 | 305 | 310 | 140 | 170 | 165 | 365 | 140 | 155 | 715 | 605 | 65 | 670 |
| | | transfpcacorrsvd | 5 | 165 | 445 | 460 | 475 | 490 | 430 | 460 | 465 | 595 | 430 | 455 | 735 | 155 | 215 | 280 | 305 | 310 | 140 | 170 | 165 | 365 | 140 | 155 | 715 | 605 | 65 | 670 |
| | | transfsvdnormalized | 5 | 165 | 445 | 460 | 475 | 490 | 430 | 460 | 465 | 595 | 430 | 455 | 735 | 155 | 215 | 280 | 305 | 310 | 140 | 170 | 165 | 365 | 140 | 155 | 715 | 605 | 65 | 670 |
| | | transffinnormalized | 5 | 465 | 495 | 505 | 525 | 490 | 430 | 460 | 465 | 595 | 430 | 455 | 735 | 385 | 405 | 415 | 435 | 310 | 140 | 170 | 165 | 365 | 140 | 155 | 715 | 595 | 85 | 715 |
| | | minimum | | 130 | 305 | 455 | 445 | 455 | 330 | 370 | 410 | 470 | 330 | 450 | 735 | 140 | 205 | 200 | 200 | 310 | 140 | 170 | 165 | 365 | 140 | 155 | 715 | 595 | 65 | 525 |
| Total test vectors | | | | | | | | | | | | | | | | 113 | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1.75

Table 5.16: Results of the algorithm "Apoptosis and auto-immunization" c

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1.78 | A | notransf | 840 | 1 | 21 | 39 | 42 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 60 | 4 | 15 | 22 | 13 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 60 | 57 | 4 | 52 |
| | | transfpcacoveigs | 5 | 6 | 25 | 42 | 42 | 9 | 16 | 27 | 30 | 7 | 16 | 40 | 104 | 7 | 15 | 15 | 15 | 5 | 12 | 8 | 22 | 5 | 12 | 23 | 106 | 53 | 4 | 48 |
| | | transfpcacovsvd | 5 | 6 | 25 | 42 | 42 | 9 | 16 | 27 | 30 | 7 | 16 | 40 | 104 | 7 | 15 | 15 | 15 | 5 | 12 | 8 | 22 | 5 | 12 | 23 | 106 | 53 | 4 | 48 |
| | | transfsvd | 5 | 15 | 26 | 41 | 41 | 22 | 28 | 29 | 40 | 21 | 28 | 37 | 94 | 8 | 16 | 19 | 22 | 7 | 13 | 19 | 35 | 3 | 13 | 28 | 91 | 53 | 4 | 48 |
| | | transffin | 5 | 42 | 40 | 40 | 41 | 22 | 28 | 29 | 40 | 21 | 28 | 37 | 94 | 30 | 29 | 32 | 37 | 7 | 13 | 19 | 35 | 3 | 13 | 28 | 91 | 53 | 37 | 48 |
| | | transfpcacorreigs | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfpcacorrsvd | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfsvdnormalized | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transffinnormalized | 5 | 40 | 42 | 43 | 44 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 32 | 33 | 34 | 35 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 64 |
| | B | notransf | 840 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 8 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 8 | 87 | 11 | 9 |
| | | transfpcacoveigs | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 |
| | | transfpcacovsvd | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 |
| | | transfsvd | 5 | 16 | 11 | 9 | 7 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 16 | 11 | 9 | 7 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 16 | 11 | 9 |
| | | transffin | 5 | 23 | 21 | 21 | 19 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 23 | 21 | 21 | 19 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 23 | 21 | 21 |
| | | transfpcacorreigs | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transfpcacorrsvd | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transfsvdnormalized | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transffinnormalized | 5 | 13 | 15 | 15 | 17 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 13 | 15 | 15 | 17 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 13 | 15 | 15 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9450 | 7950 | 8820 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 7320 | 73120 | 9390 | 7780 | 8530 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 7320 | 73650 | 9280 | 8080 |
| | | transfpcacoveigs | 5 | 130 | 305 | 465 | 465 | 450 | 335 | 385 | 410 | 465 | 335 | 500 | 1065 | 140 | 205 | 195 | 195 | 410 | 295 | 195 | 330 | 445 | 295 | 330 | 1085 | 600 | 95 | 525 |
| | | transfpcacovsvd | 5 | 130 | 305 | 465 | 465 | 450 | 335 | 385 | 410 | 465 | 335 | 500 | 1065 | 140 | 205 | 195 | 195 | 410 | 295 | 195 | 330 | 445 | 295 | 330 | 1085 | 600 | 95 | 525 |
| | | transfsvd | 5 | 230 | 315 | 455 | 445 | 545 | 435 | 415 | 520 | 585 | 435 | 485 | 975 | 160 | 215 | 235 | 255 | 395 | 285 | 315 | 470 | 405 | 285 | 395 | 945 | 610 | 95 | 525 |
| | | transffin | 5 | 535 | 505 | 505 | 505 | 545 | 435 | 415 | 520 | 585 | 435 | 485 | 975 | 415 | 395 | 425 | 465 | 395 | 285 | 315 | 470 | 405 | 285 | 395 | 945 | 645 | 475 | 585 |
| | | transfpcacorreigs | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transfpcacorrsvd | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transfsvdnormalized | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transffinnormalized | 5 | 465 | 495 | 505 | 525 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 385 | 405 | 415 | 435 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 595 | 85 | 715 |
| | | minimum | | 130 | 305 | 455 | 445 | 450 | 335 | 385 | 410 | 465 | 335 | 460 | 735 | 140 | 205 | 195 | 195 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 595 | 65 | 525 |
| Total test vectors | | | | | | | | | | | | | | | | 113 | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1.78

$$\textbf{Optimization objective} = Weight \times \mathbf{A} + Dimensions \times \mathbf{B} \tag{5.2}$$

Where: **A** is the matrix formed with all numbers of wrong class recognitions and **B** is the matrix formed with all the numbers of reduced fault pattern vectors.

Tables 5.8 to 5.16 show the value of the optimization objective considering a weight of 10, which resulted suitable for finding the best results. The tables also show the minimum optimization objective value along each column, that is to say, along all methods of dimension reduction. The minimum value helps to find the best case by a determined threshold, cases highlighted with light and dark gray color. Taking the minimum value among the results for all thresholds, gives the best case highlighted in dark gray color in tables 5.8 to 5.16 and zoomed in tables 5.17, 5.18 and 5.19 in.

Table 5.17 presents four best cases by a value of 80 for the optimization objective. One of the best cases, highlighted in dark gray in that table, has been taken in order to see graphically the fault pattern vectors with their first two dimensions. That case has been obtained using a threshold of 0.1 by the execution of the function removalbylackofstimulation for the reduction of fault pattern vectors, the function transfpcacorreigs for the reduction of dimensions from 480 to 5, the Chebyshev distance measurement method, the nearest neighbor class assignation method, and with only six fault pattern vectors reported just 5 wrong class recognitions from 113 fault vectors of the test set. Those fault pattern vectors can be seen in figure 5.8 in black, and in gray the removed fault pattern vectors. Thereby, the function removalbylackofstimulation reduced the number of fault pattern vectors removing outlier fault pattern vectors. Although a good recognition has been reported, the removal of outlier fault pattern vectors can lead to important loose of information contained in the original set of fault pattern vectors for future fault vector recognitions.

For the same case, it has been analyzed by means of a plot, how the number of wrong class recognitions and number of fault pattern vectors vary for different thresholds by using function removalbylackofstimulation. That plot is shown in figure 5.9. There, it can be observed that the smaller the stimulation threshold that each fault pattern vector has, the smaller is the probability that the vectors are stimulated, reason why a bigger number of fault pattern vectors is removed due to lack of stimulation. In that figure, it can also be verified the case where for a threshold of 0.1, six fault pattern vectors report 5 wrong class recognitions.

Table 5.18 presents six best cases by a value of 70 for the optimization objective. One of the best cases, highlighted in dark gray in that table, has been taken in order to see graphically the fault pattern vectors with their first two dimensions. That case has been obtained using a threshold of 1 by the execution of the function removalbyautoreactivity for the reduction of fault pattern vectors, the function transfpcacorreigs for the reduction of dimensions from 480 to 5, the Euclidean distance measurement method, the nearest neighbor class assignation method, and with only ten fault pattern vectors reported just 2 wrong class recognitions from 113 fault vectors of the test set. Those fault pattern vectors can be seen in figure 5.10 in black, and in gray the removed fault pattern vectors. Thereby, the function removalbylackofstimulation reduced the number of fault pattern vectors removing redundant fault pattern vectors. This method performs quite good. However, the resulting final set of fault pattern vectors is dependent on the order of the initial fault patten vectors.

For the same case, it has been analyzed by means of a plot, how the number of wrong class recognitions and number of fault pattern vectors vary for different thresholds by using function removalbyautoreactivity. That plot is shown in figure 5.11. There, it can be observed that the

Table 5.17: Best case by "Death of immune cells with insufficient stimulation"

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 0.1 | A | notransf | 840 | 106 | 106 | 99 | 8 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 9 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfpcacoveigs | 5 | 106 | 106 | 106 | 29 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transfpcacovsvd | 5 | 106 | 106 | 106 | 29 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transfsvd | 5 | 106 | 106 | 106 | 6 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 106 | 106 | 106 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 67 | 106 | 67 |
| | | transffin | 5 | 19 | 13 | 12 | 11 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 12 | 23 | 10 | 10 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 53 | 10 | 14 | 5 |
| | | transfpcacorreigs | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfpcacorrsvd | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transfsvdnormalized | 5 | 106 | 106 | 61 | 5 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 67 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 67 | 106 | 67 |
| | | transffinnormalized | 5 | 4 | 6 | 6 | 6 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 6 | 6 | 8 | 8 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 106 | 3 | 7 | 12 |
| | B | notransf | 840 | 1 | 1 | 2 | 80 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 80 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfpcacoveigs | 5 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transfpcacovsvd | 5 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transfsvd | 5 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |
| | | transffin | 5 | 27 | 61 | 70 | 72 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 27 | 61 | 70 | 72 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 27 | 61 | 70 |
| | | transfpcacorreigs | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfpcacorrsvd | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transfsvdnormalized | 5 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |
| | | transffinnormalized | 5 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 87 | 87 | 87 | 87 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 87 | 87 | 87 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 1900 | 1900 | 2670 | 67280 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 2350 | 67290 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1900 | 1510 | 1900 | 2350 |
| | | transfpcacoveigs | 5 | 1065 | 1065 | 1065 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 685 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transfpcacovsvd | 5 | 1065 | 1065 | 1065 | 305 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 685 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transfsvd | 5 | 1065 | 1065 | 1065 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 1065 | 1065 | 1065 | 690 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 675 | 1065 | 675 |
| | | transffin | 5 | 325 | 435 | 470 | 470 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 255 | 535 | 450 | 460 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 235 | 445 | 400 |
| | | transfpcacorreigs | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transfpcacorrsvd | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transfsvdnormalized | 5 | 1065 | 1065 | 620 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 680 | 700 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 675 | 1065 | 680 |
| | | transffinnormalized | 5 | 475 | 495 | 495 | 495 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 495 | 495 | 515 | 515 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 465 | 505 | 555 |
| | | minimum | | 325 | 435 | 470 | 80 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 255 | 495 | 450 | 460 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 1065 | 540 | 235 | 445 | 400 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 0.1

Table 5.18: Best case by "Elimination of auto-reactive immune cells"

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1 | A | notransf | 840 | 1 | 4 | 17 | 67 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 12 | 4 | 8 | 67 | 67 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 9 | 57 | 5 | 67 |
| | | transfpcacoveigs | 5 | 2 | 2 | 2 | 4 | 7 | 9 | 13 | 12 | 6 | 9 | 8 | 3 | 7 | 8 | 5 | 8 | 5 | 6 | 4 | 9 | 5 | 6 | 6 | 9 | 17 | 4 | 32 |
| | | transfpcacovsvd | 5 | 2 | 2 | 2 | 4 | 7 | 9 | 13 | 12 | 6 | 9 | 8 | 3 | 7 | 8 | 5 | 8 | 5 | 6 | 4 | 9 | 5 | 6 | 6 | 9 | 17 | 4 | 32 |
| | | transfsvd | 5 | 3 | 3 | 2 | 3 | 19 | 20 | 22 | 29 | 17 | 20 | 30 | 23 | 10 | 9 | 4 | 4 | 3 | 6 | 6 | 18 | 1 | 6 | 18 | 20 | 15 | 5 | 16 |
| | | transffin | 5 | 58 | 67 | 67 | 67 | 19 | 20 | 22 | 29 | 17 | 20 | 30 | 23 | 67 | 67 | 67 | 67 | 3 | 6 | 6 | 18 | 1 | 6 | 18 | 20 | 67 | 67 | 67 |
| | | transfpcacorreigs | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transfpcacorrsvd | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transfsvdnormalized | 5 | 2 | 2 | 3 | 10 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 8 | 3 | 3 | 9 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 42 | 4 | 10 |
| | | transffinnormalized | 5 | 67 | 67 | 67 | 67 | 21 | 24 | 25 | 29 | 22 | 24 | 19 | 19 | 67 | 67 | 67 | 67 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 17 | 67 | 67 | 67 |
| | B | notransf | 840 | 87 | 11 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 70 | 87 | 11 | 3 | 1 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 70 | 87 | 11 | 3 |
| | | transfpcacoveigs | 5 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 |
| | | transfpcacovsvd | 5 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 | 10 | 85 | 69 | 59 | 48 | 85 | 69 | 52 | 12 | 34 | 12 | 10 |
| | | transfsvd | 5 | 33 | 11 | 10 | 10 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 33 | 11 | 10 | 10 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 33 | 11 | 10 |
| | | transffin | 5 | 2 | 1 | 1 | 1 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 2 | 1 | 1 | 1 | 85 | 64 | 53 | 43 | 86 | 64 | 43 | 12 | 2 | 1 | 1 |
| | | transfpcacorreigs | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transfpcacorrsvd | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transfsvdnormalized | 5 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 | 7 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 25 | 10 | 9 |
| | | transffinnormalized | 5 | 1 | 1 | 1 | 1 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 1 | 1 | 1 | 1 | 84 | 57 | 48 | 40 | 85 | 57 | 40 | 8 | 1 | 1 | 1 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9280 | 2690 | 1510 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 58920 | 73120 | 9320 | 3190 | 1510 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 58890 | 73650 | 9290 | 3190 |
| | | transfpcacoveigs | 5 | 190 | 80 | 70 | 90 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 240 | 140 | 100 | 130 | 475 | 405 | 335 | 330 | 475 | 405 | 320 | 150 | 340 | 100 | 370 |
| | | transfpcacovsvd | 5 | 190 | 80 | 70 | 90 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 240 | 140 | 100 | 130 | 475 | 405 | 335 | 330 | 475 | 405 | 320 | 150 | 340 | 100 | 370 |
| | | transfsvd | 5 | 195 | 85 | 70 | 80 | 615 | 520 | 485 | 505 | 600 | 520 | 515 | 290 | 265 | 145 | 90 | 90 | 455 | 380 | 325 | 395 | 440 | 380 | 395 | 260 | 315 | 105 | 210 |
| | | transffin | 5 | 590 | 675 | 675 | 675 | 615 | 520 | 485 | 505 | 600 | 520 | 515 | 290 | 680 | 675 | 675 | 675 | 455 | 380 | 325 | 395 | 440 | 380 | 395 | 260 | 680 | 675 | 675 |
| | | transfpcacorreigs | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transfpcacorrsvd | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transfsvdnormalized | 5 | 145 | 70 | 75 | 135 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 205 | 80 | 75 | 125 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 545 | 90 | 145 |
| | | transffinnormalized | 5 | 675 | 675 | 675 | 675 | 630 | 525 | 490 | 490 | 645 | 525 | 390 | 230 | 675 | 675 | 675 | 675 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 210 | 675 | 675 | 675 |
| | | minimum | | 145 | 70 | 70 | 80 | 495 | 435 | 425 | 360 | 485 | 435 | 340 | 90 | 205 | 80 | 75 | 90 | 430 | 285 | 250 | 210 | 425 | 285 | 200 | 150 | 315 | 90 | 145 |
| Total test vectors | | | | 113 | | | | | | | | | | | | | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1

Table 5.19: Best case by "Apoptosis and auto-immunization"

| Threshold | Class assignation method | | | Nearest neighbor | | | | | | | | | | | | k-nearest neighbor | | | | | | | | | | | | Minimal distance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Euclidean norm | | | | p-norm | | | | | | | | Euclidean norm | | | | p-norm | | | | standard score | | |
| | | Distance measur. method | D = Nr. of dimensions | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Manhattan | Euclidean | Minkowski | Chebyshev | Mahalanobis | Mahalanobis* | Mahalanobis** |
| 1.78 | A | notransf | 840 | 1 | 21 | 39 | 42 | 2 | 1 | 0 | 5 | 2 | 1 | 0 | 60 | 4 | 15 | 22 | 13 | 1 | 1 | 1 | 4 | 1 | 1 | 1 | 60 | 57 | 4 | 52 |
| | | transfpcacoveigs | 5 | 6 | 25 | 42 | 42 | 9 | 16 | 27 | 30 | 7 | 16 | 40 | 104 | 7 | 15 | 15 | 15 | 5 | 12 | 8 | 22 | 5 | 12 | 23 | 106 | 53 | 4 | 48 |
| | | transfpcacovsvd | 5 | 6 | 25 | 42 | 42 | 9 | 16 | 27 | 30 | 7 | 16 | 40 | 104 | 7 | 15 | 15 | 15 | 5 | 12 | 8 | 22 | 5 | 12 | 23 | 106 | 53 | 4 | 48 |
| | | transfsvd | 5 | 15 | 26 | 41 | 41 | 22 | 28 | 29 | 40 | 21 | 28 | 37 | 94 | 8 | 16 | 19 | 22 | 7 | 13 | 19 | 35 | 3 | 13 | 28 | 91 | 53 | 4 | 48 |
| | | transffin | 5 | 42 | 40 | 40 | 41 | 22 | 28 | 29 | 40 | 21 | 28 | 37 | 94 | 30 | 29 | 32 | 37 | 7 | 13 | 19 | 35 | 3 | 13 | 28 | 91 | 53 | 37 | 48 |
| | | transfpcacorreigs | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfpcacorrsvd | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transfsvdnormalized | 5 | 9 | 39 | 40 | 41 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 8 | 16 | 22 | 24 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 61 |
| | | transffinnormalized | 5 | 40 | 42 | 43 | 44 | 19 | 28 | 35 | 38 | 24 | 28 | 37 | 69 | 32 | 33 | 34 | 35 | 1 | 0 | 6 | 8 | 1 | 0 | 7 | 67 | 53 | 1 | 64 |
| | B | notransf | 840 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 8 | 87 | 11 | 9 | 10 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 8 | 87 | 11 | 9 |
| | | transfpcacoveigs | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 |
| | | transfpcacovsvd | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 | 9 | 72 | 35 | 23 | 22 | 79 | 35 | 20 | 5 | 14 | 11 | 9 |
| | | transfsvd | 5 | 16 | 11 | 9 | 7 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 16 | 11 | 9 | 7 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 16 | 11 | 9 |
| | | transffin | 5 | 23 | 21 | 21 | 19 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 23 | 21 | 21 | 19 | 65 | 31 | 25 | 24 | 75 | 31 | 23 | 7 | 23 | 21 | 21 |
| | | transfpcacorreigs | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transfpcacorrsvd | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transfsvdnormalized | 5 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 | 13 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 15 | 11 | 12 |
| | | transffinnormalized | 5 | 13 | 15 | 15 | 17 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 13 | 15 | 15 | 17 | 58 | 27 | 22 | 17 | 69 | 27 | 18 | 9 | 13 | 15 | 15 |
| | ( 10 * A ) + ( B * D ) | notransf | 840 | 73090 | 9450 | 7950 | 8820 | 73100 | 73090 | 73080 | 73130 | 73100 | 73090 | 73080 | 7320 | 73120 | 9390 | 7780 | 8530 | 73090 | 73090 | 73090 | 73120 | 73090 | 73090 | 73090 | 7320 | 73650 | 9280 | 8080 |
| | | transfpcacoveigs | 5 | 130 | 305 | 465 | 465 | 450 | 335 | 385 | 410 | 465 | 335 | 500 | 1065 | 140 | 205 | 195 | 195 | 410 | 295 | 195 | 330 | 445 | 295 | 330 | 1085 | 600 | 95 | 525 |
| | | transfpcacovsvd | 5 | 130 | 305 | 465 | 465 | 450 | 335 | 385 | 410 | 465 | 335 | 500 | 1065 | 140 | 205 | 195 | 195 | 410 | 295 | 195 | 330 | 445 | 295 | 330 | 1085 | 600 | 95 | 525 |
| | | transfsvd | 5 | 230 | 315 | 455 | 445 | 545 | 435 | 415 | 520 | 585 | 435 | 485 | 975 | 160 | 215 | 235 | 255 | 395 | 285 | 315 | 470 | 405 | 285 | 395 | 945 | 610 | 95 | 525 |
| | | transffin | 5 | 535 | 505 | 505 | 505 | 545 | 435 | 415 | 520 | 585 | 435 | 485 | 975 | 415 | 395 | 425 | 465 | 395 | 285 | 315 | 470 | 405 | 285 | 395 | 945 | 645 | 475 | 585 |
| | | transfpcacorreigs | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transfpcacorrsvd | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transfsvdnormalized | 5 | 165 | 445 | 460 | 475 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 155 | 215 | 280 | 305 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 605 | 65 | 670 |
| | | transffinnormalized | 5 | 465 | 495 | 505 | 525 | 480 | 415 | 460 | 465 | 585 | 415 | 460 | 735 | 385 | 405 | 415 | 435 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 595 | 85 | 715 |
| | | minimum | | 130 | 305 | 455 | 445 | 450 | 335 | 385 | 410 | 465 | 335 | 460 | 735 | 140 | 205 | 195 | 195 | 300 | 135 | 170 | 165 | 355 | 135 | 160 | 715 | 595 | 65 | 525 |
| Total test vectors | | | | | | | | | | | | | | | | 113 | | | | | | | | | | | | | | |

A = Wrong class recognitions, B = Nr. of fault pattern vectors, Threshold = 1.78

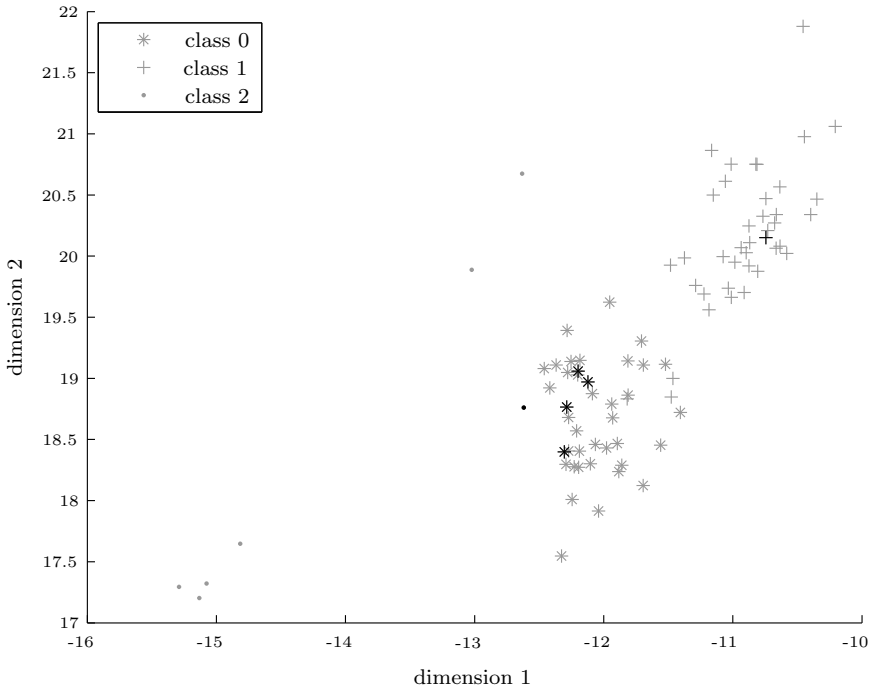



Figure 5.8: Fault pattern vectors reduced by function removalbylackofstimulation

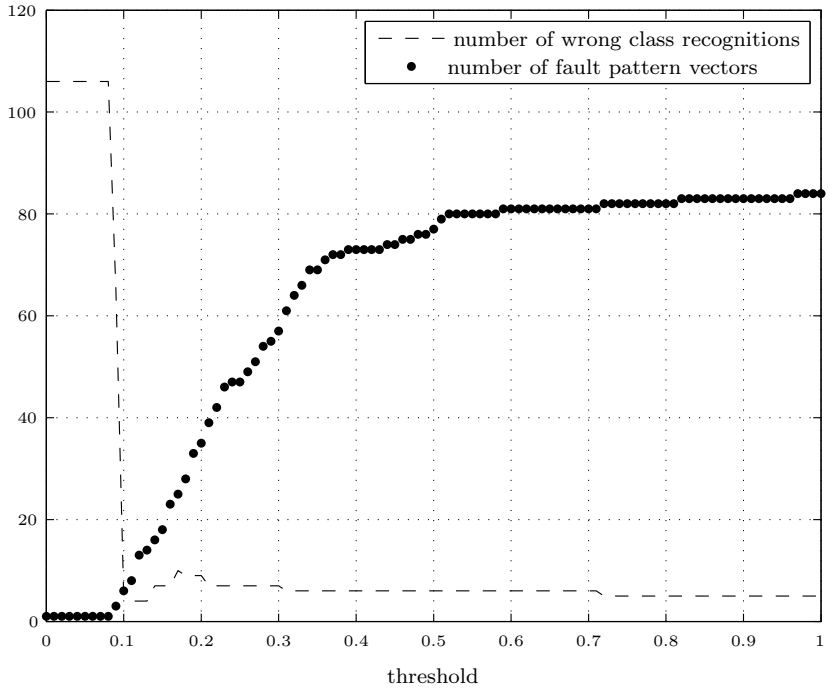



Figure 5.9: Threshold variation by function removalbylackofstimulation

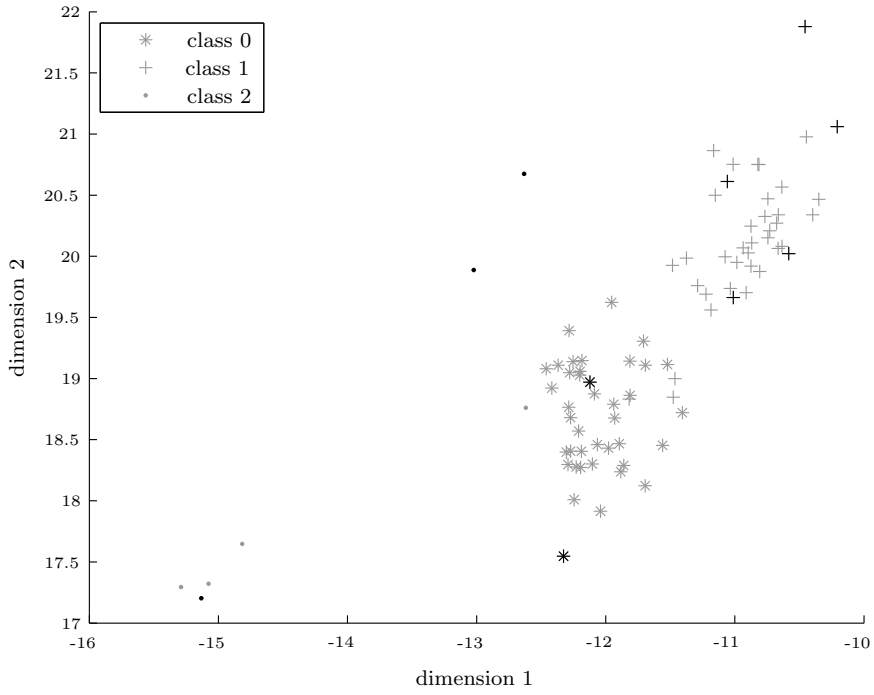



Figure 5.10: Fault pattern vectors reduced by function removalbyautoreactivity

bigger the threshold, the bigger is the probability to find fault pattern vectors for being removed of the area, defined by the given threshold, of another fault pattern vector. Reason why a big number of fault pattern vectors are removed due to the so called autoreactivity. In figure 5.11, it can also be verified the case where for a threshold of 1, ten fault pattern vectors report 2 wrong class recognitions.

Table 5.19 presents three best cases by a value of 65 for the optimization objective. One of the best cases, highlighted in dark gray in that table, has been taken in order to see graphically the fault pattern vectors with their first two dimensions. That case has been obtained using a threshold of 1.78 by the execution of the function removalbyapoptosisautoimmunization for the reduction of fault pattern vectors, the function transfpcacorreigs for the reduction of dimensions from 480 to 5, the Mahalanobis* distance measurement method, the minimal distance class assignation method, and with only eleven fault pattern vectors reported just 1 wrong class recognitions from 113 fault vectors of the test set. Those fault pattern vectors can be seen in figure 5.12 in black, and in gray the removed fault pattern vectors. Thereby, the function removalbyapoptosisautoimmunization reduced the number of fault pattern vectors removing redundant fault pattern vectors. However, this method, in comparison with the one implemented by the function removalbyautoreactivity, tries to get, independently of the order of the initial fault patten vectors, a resulting final set of fault pattern vectors by reinserting already removed fault pattern vectors. Reason why the number of wrong class recognition goes even lower.

For the same case, it has been analyzed by means of a plot, how the number of wrong class recognitions and number of fault pattern vectors vary for different thresholds by using

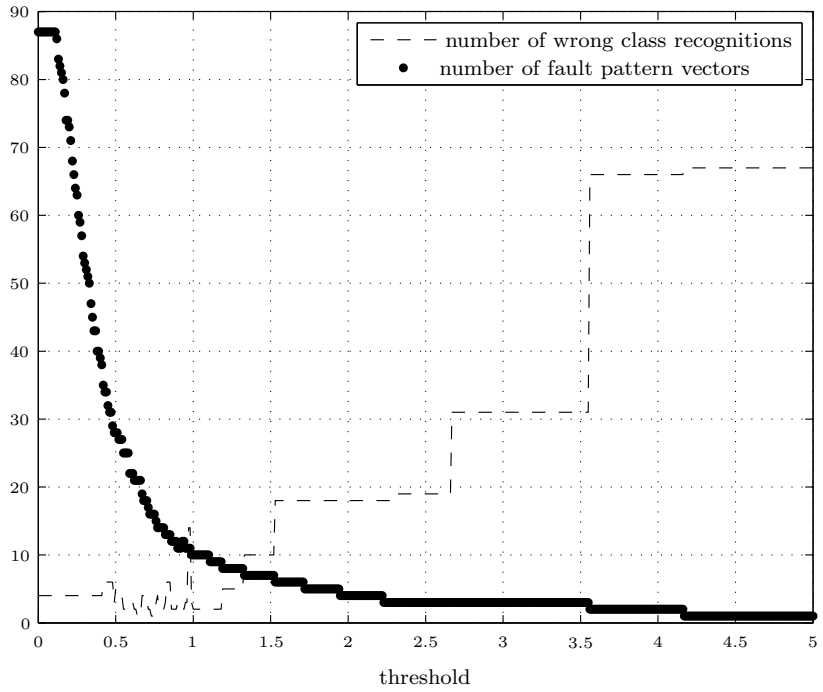



Figure 5.11: Threshold variation by function removalbyautoreactivity

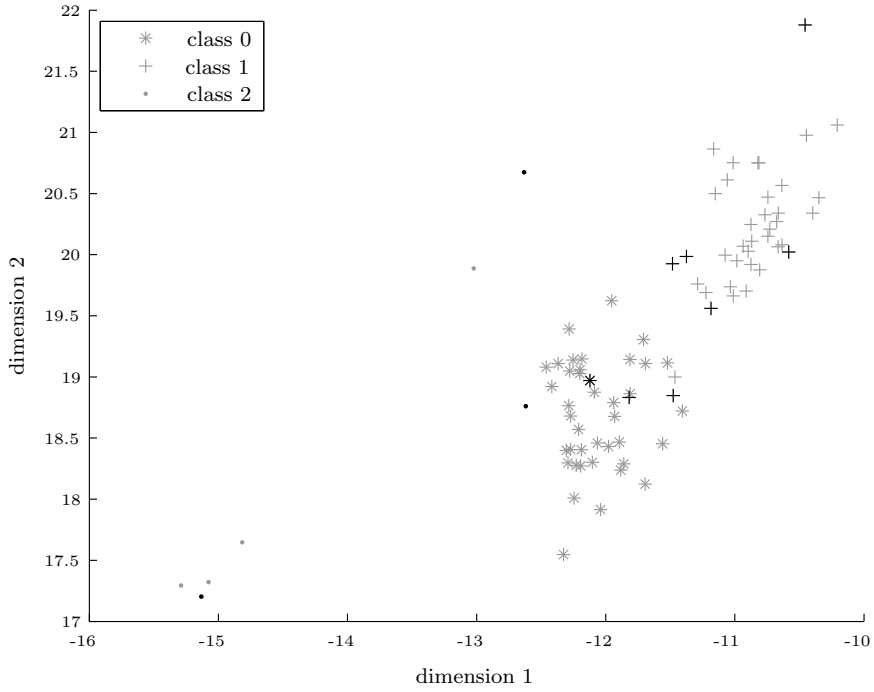



Figure 5.12: Fault pattern vectors reduced by function removalbyapoptosisautoimmunization

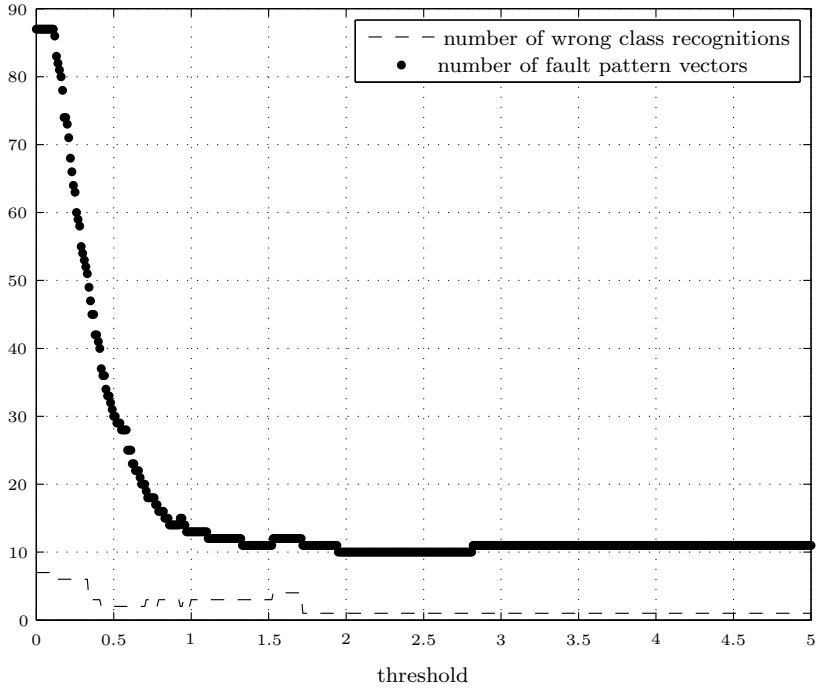



Figure 5.13: Threshold variation by function removalbyapoptosisautoimmunization

the function removalbyapoptosisautoimmunization. That plot is shown in figure 5.13. There, it can be observed that the bigger the threshold, the bigger is the probability to find fault pattern vectors for being removed from the area defined by the given threshold of another fault pattern vector. Reason why a big number of fault pattern vectors are removed by apoptosis. However, some fault pattern vectors are reinserted by the so called autoimmunization to the resulting set of fault pattern vectors again. In figure 5.11, it can also be verified the case where for a threshold of 1.78, eleven fault pattern vectors report 1 wrong class recognition. Furthermore, with a bigger threshold, i.e. 2, the number of vectors would have been reduced to ten maintaining as 1 the number of wrong class recognitions. It is to note that in this method, it is not possible to reduce further, above a limit, the number of fault pattern vectors due to the operation of reinsertion of fault pattern vectors.

With respect to tables 5.8 to 5.16, it is also to remark that the cases that use functions transfpcacorreigs, transfpcacorrsvd and transfsvdnormalized for fault pattern vector dimension reduction, deliver the same results. That is because the same set of fault pattern vectors is obtained after fault pattern vector dimension reduction, as could be seen on subfigures (a), (b) and (c) of figure 5.6 and on table 5.7.

Tables 5.8 to 5.16 can not show appropriate results for all combination of methods since the range of values of the threshold, that produces a variation in the number of fault pattern vectors and number of wrong class recognitions, is different for each combination of class assignation method, distance measurement method and fault pattern vector dimension reduction method. So, for example, using the Manhattan distance measurement method, a rang with high number values for the threshold is required in order to see a variation in the number

of fault pattern vectors and number of wrong class recognitions. Similarly, the use of the transffinnormalized fault pattern vector dimension reduction method, requires a range with low number values for the threshold, in order to see a variation in the number of fault pattern vectors and number of wrong class recognitions. Therefore, the election of the combination of methods should be done first, and then the reduction of number of fault pattern vectors should be executed. This mode to proceed should be more beneficial, since the combination of methods should be determined by the difficulty of the hardware implementation of each one of the methods. The hardware implementation of the method of fault pattern vector dimension reduction is necessary for reducing the vector that comes from a functioning system, and the distance measurement method and class assignation method are necessary for determining whether that vector represents a fault or not. The hardware implementation of those methods will be addressed in chapter 6.

A further optimization could also be done implementing a loop for a set of dimensions over the loop for a set of thresholds. However, since on tables 5.8 to 5.16 the range of values of the threshold, that produces a variation in the number of fault pattern vectors and number of wrong class recognitions, is different for each combination of class assignation method, distance measurement method and fault pattern vector dimension reduction method, it would be better to perform such an optimization for just a subset of methods, e.g. only one method of fault pattern dimension reduction combined with several distance measurement and class assignation methods.

## 5.2 Fault recognition module with binary fault vector elements

This section presents the implementation of algorithms for the design of a fault recognition module which gets from the circuit for self-repairing binary value signal inputs and outputs. Such is the case of combinational and sequential digital systems, as show in section 4.1. Those binary value signals become the elements of the fault vector in the fault recognition module. The arrangement of signals within a fault vector for the design of a fault recognition module depends on the type of digital circuit. A fault vector could have the form [**Inputs**|**Outputs**] for combinational circuits and [**Inputs**|**Present-state**|**Outputs**|**Next-state**] for sequential circuits with full scan design.

Scan design is a technique that inserts test points and additional logic for setting and reading the internal states, i.e., present-state next-state, in a sequential circuit, [Williams and Angell, 1973]. When all the sequential elements, such as flip-flops and latches, are considered in the scan design, the sequential circuit is said to have a full scan design, otherwise it is said to have a partial scan design. A partial scan design tries to reduce the adverse effects on area and performance that a scan design entails for making a sequential circuit testable, [Kalla and Ciesielski, 1998]. In a full scan design, a single test pattern, named here fault pattern vector, can detect a fault. In a sequential circuit without scan design or a sequential circuit with a partial scan design, a sequence of test patterns is necessary for detecting a single fault. Here, we concentrate in a completely observable and controllable digital circuit, i.e., a circuit for which a single fault pattern vector can detect a fault, such is the case of combinational circuits and sequential circuits provided with a full scan design.

For evaluation purposes there exists several ACM/SIGDA benchmark sets, as listed on the page of the [Collaborative Benchmarking and Experimental Algorithmics Laboratory, 2007]. Those sets have been created for analyzing the performance of methods and tools for electronic

design automation, in short EDA. Two of them are the ISCAS 85 and ISCAS 89 sets. The ISCAS 85 is a set of combinational circuits distributed initially at the 1985 International Symposium on Circuits and Systems, [Bryan, 1988]. They have been used for comparing results in the area of test pattern generation. The ISCAS 89 is a set of sequential circuits distributed in the Special Session of Sequential Test Generation at the 1989 International Symposium on Circuits and Systems, [Brglez et al., 1989].

In order to determine if a circuit is faulty or not, fault pattern vectors with binary value elements can be generated for all possible inputs of the digital circuit. That means, when the circuit has $n$ inputs, $2^n$ fault pattern vectors. However, that is only possible when the digital circuit has few inputs. When a digital circuit has a large number of inputs, a reduced set of fault pattern vectors can be generated for finding a given set of faults. Such a reduced set of fault pattern vectors with binary value elements can be generated automatically with the so called automatic test pattern generator, in short ATPG.

There are automatic test pattern generators suitable for different fault models, [Wang et al., 2009]. Those fault models try to reflect the behavior of possible physical defects in a circuit. Some fault models are: the gate-level stuck-at model, which models a digital signal line stuck-at logic levels 0 or 1; the transistor-level stuck-at model, which models a transistor that gets stuck-open or stuck-short; the bridging fault model, that models two signal wires shorted together; the delay fault model, that models the event when the propagation delay of a transition falls outside an specified time limit.

Since a gate-level stuck-at fault in a combinational circuit needs only one fault pattern vector to be detected, and a gate-level stuck-at fault in a sequential circuit needs a sequence of fault pattern vectors to be detected, an ATPG for gate-level stuck-at faults in combinational circuits is different than an ATPG for gate-level stuck-at faults in sequential circuits. However, when a sequential circuit is provided with a full scan design, an ATPG for gate-level stuck-at faults in combinational circuits can be employed for generating fault pattern vectors for that sequential circuits.

Since a stuck-open fault behaves like a level sensitive latch, a sequence of two fault pattern vectors is necessary for detecting a single fault, therefore an ATPG for stuck-open faults in digital circuits usually produces sequences of two fault pattern vectors for detecting stuck-open faults. Since a stuck-short fault is the model of a conducting path between $V_{DD}$, the positive supply voltage, and $V_{SS}$, the negative supply voltage, that fault can be detected by monitoring the power supply current, $I_{DD}$, in its steady state, method named as $I_{DDQ}$. Where Q comes from the term "quiescent state", which is the steady state period when a transistor or other circuit element is not performing an active function in the circuit. In the $I_{DDQ}$ method, a set of fault pattern vectors is necessary to be generated for being applied at the time of monitoring the $I_{DDQ}$ current.

A bridging fault is modeled as a dominant-AND/dominant-OR assuming that one signal dominates the logic value of the shorted signal for one logic value only, the ATPG is similar to the ATPG for gate-level stuck-at faults in combinational circuits but designed with constraints. A delay fault can be caused by resistive opens and shorts in wires and parameter variations in transistors, and can be detected by a sequence of two fault pattern vectors. The first pattern vector initializes the circuit and the second fault pattern vector starts a transition at the start of a path that will propagate in a time that is measured and compared with a determined time limit.

In order to obtain fault pattern vectors for a combinational circuit or a sequential circuit with a full scan design, ATALANTA, an automatic test pattern generator for stuck-at faults

in combinational circuits, [Lee and Ha, 1993], tool developed at the Bradley Department of Electrical and Computer Engineering in the Virginia Polytechnic Institute and State University, is intended to be used.

A fault pattern vector can detect no fault, a single fault, or many faults. In order to evaluate which faults a fault pattern vector is able to detect, a fault simulator is required. A fault simulator injects a determined fault model into the design of a given circuit, applies an input pattern vector on the circuit, and compares the resulting output with the output of the fault-free circuit. When the resulting output and the output of the fault-free circuit are different, then the input pattern vector is able to detect that determined fault. In that way, a fault simulator is able to determine the percentage of faults that a fault pattern vector set is able to find from a pool of inserted faults in the circuit, number called as fault coverage. A fault simulator is an integral part of an automatic test pattern generator since it helps to determine whether a generated input pattern vector is able to detect a fault or not.

The automatic test pattern generator ATALANTA uses the FSIM fault simulator, which is a fault simulator for combinational circuits and sequential circuits with full scan design, [Lee and Ha, 1991]. It has been also developed at the Bradley Department of Electrical and Computer Engineering in the Virginia Polytechnic Institute and State University.

Methods for fault recognition, fault vector dimension reduction and fault pattern vectors number reduction, are different in comparison with the methods shown for fault pattern vectors with real elements. Next sections show some alternatives of how to perform fault recognition, fault vector dimension reduction and fault pattern vectors number reduction, with fault vectors having binary elements.

### 5.2.1 Fault recognition

Now the goal is to find a method for fault recognition in digital circuits. We concentrate here in stuck-at faults. Once a stuck-at fault happened at any input $x_i$, called also primary variable of the circuit, a mathematical method that helps on finding that fault is the so called Boolean difference defined below.

$$\frac{\partial f}{\partial x_i} = f(x_1, ..., x_i = 1, ..., x_n) \oplus f(x_1, ..., x_i = 0, ..., x_n) \tag{5.3}$$

The Boolean difference gives a logic 1, when both terms of the exclusive-or give opposite values. That means that the stuck-at fault can be detected only when $f(x_1, ..., x_i = 1, ..., x_n) \neq f(x_1, ..., x_i = 0, ..., x_n)$. When the Boolean difference gives a logic 0, it means that the fault cannot be detected observing the value of function $f$. In other words the stuck-at fault at $x_i$ is not observable at output $f$.

If a stuck-at fault happens at an internal point $u$ in the circuit, e.g. the output of a gate, called also secondary variable, the Boolean difference looks like the formula below.

$$\frac{\partial f}{\partial u} = f(x_1, ..., x_n, u(x_1, ..., x_n)) \oplus f(x_1, ..., x_n, \bar{u}(x_1, ..., x_n)) \tag{5.4}$$

The same, if $f(x_1, ..., x_n, u(x_1, ..., x_n)) \neq f(x_1, ..., x_n, \bar{u}(x_1, ..., x_n))$, the Boolean difference gives a logic value 1. That means that the stuck-at fault at the secondary variable or internal point $u$ is observable at $f$. When the Boolean difference gives a logic value of 0 then, the stuck-at fault at $u$ is not observable at output $f$.

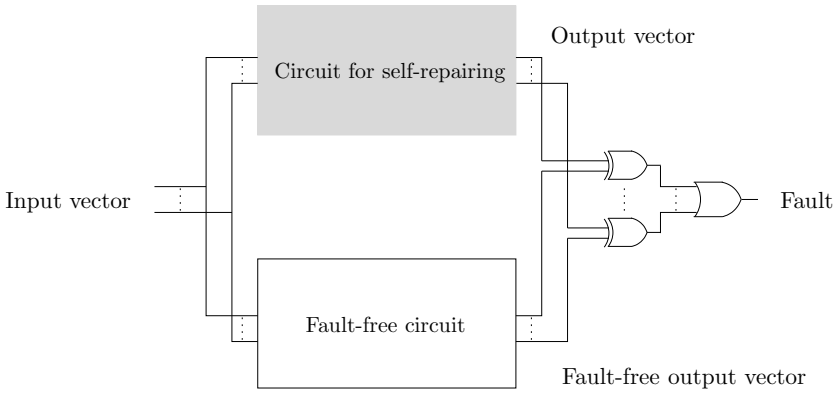



Figure 5.14: Fault recognition module design using duplication

When the stuck-at fault is observable, there exist an input vector $[x_1, ..., x_n]$, or several input vectors, that help on distinguishing a fault-free circuit from a faulty circuit. In the case where the stuck-at is given at a primary variable $x_i$, it is easy to determine that a stuck-at-0 fault can be detected with an input vector $[x_1, ..., x_i = 1, ..., x_n]$ and an stuck-at-1 with an input vector $[x_1, ..., x_i = 0, ..., x_n]$. However, when the stuck-at fault happens at any internal point $u$ of the circuit, it is necessary to find an input vector $[x_1, ..., x_n]$ able to produce a Boolean difference at the fault location $u$, process called fault sensitization, and then show that difference at any output, in our case $f$, process called fault propagation, please see [Kirkland and Mercer, 1988]. Therefore the condition for finding a stuck-at-0 at $u$ joining both processes is $u(x_1, ..., x_n) \cdot f(x_1, ..., x_n, u = 0) \oplus f(x_1, ..., x_n, u = 1) = 1$. Similarly, the condition for finding a stuck-at-1 at $u$ joining both processes is $\bar{u}(x_1, ..., x_n) \cdot f(x_1, ..., x_n, u = 0) \oplus f(x_1, ..., x_n, u = 1) = 1$, please see [Reed, 1973] and [Wang et al., 2009]. If the condition can not be satisfied, the fault can not be detected. That is to say, there exist no input vector able to help on finding that fault.

One of the generated input pattern vectors can be applied to the circuit for self-repairing for detecting the considered fault. Once that input pattern vector has been applied, the output from the circuit for self-repairing should be compared with the output from a fault-free circuit. If those outputs are different, the circuit is considered faulty, otherwise it is considered fault-free. Such output comparison can be implemented with an EXCLUSIVE-OR gate. However, when the circuit has more than one output signal, the outputs of the circuit for self-repairing arranged into a vector should be compared with the output vector of a fault-free circuit. In order to compare those two output vectors, the Hamming distance for vectors with binary value elements, presented in section 4.2 should be zero when the output vectors are equal. Given $n$ outputs in the output vectors, that distance can be implemented using $n$ EXCLUSIVE-OR gates and an $n$ input OR gate as shown in figure 5.14.

In figure 5.14, the fault-free output vector is obtained applying concurrently the generated input pattern vector to a copy of the circuit that is trusted for being fault-free. However, the fault-free output vector can be obtained by implementing a more simple logic designed using a set of fault pattern vectors generated for detecting a determined set of faults, with the arrangement [**Input pattern vector**|**Output pattern vector**]. The logic just monitors the input vector to the circuit, giving a sign whenever an input pattern vector contained in any of the generated fault pattern vector is present, case in which the output vector from the circuit

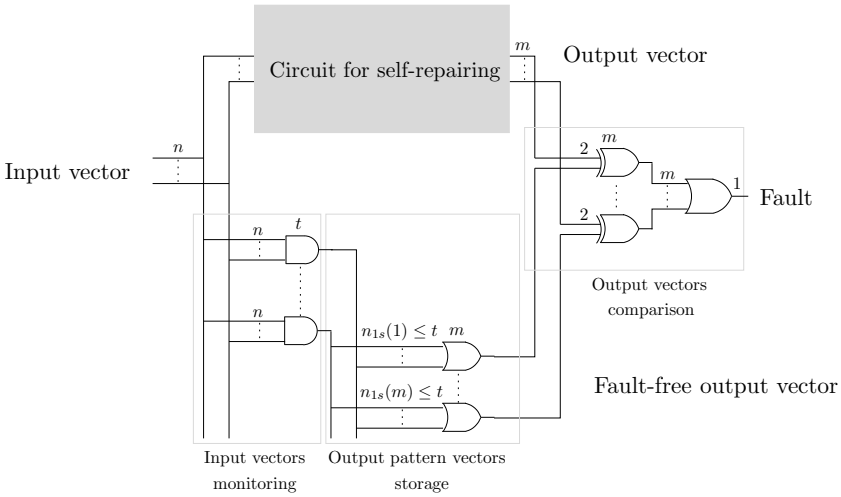



Figure 5.15: Fault recognition module design using only specified values (abstracted and generalized from [Voyiatzis et al., 2009] ©2009 IEEE)

for self-repairing is compared with the output pattern vector contained in the respective fault pattern vector. That architecture has been presented first in [Sharma and Saluja, 1988] and named Built-In Concurrent Self-Test, in short BICST. Given a circuit with $n$ inputs, $m$ outputs and a set with $t$ fault pattern vectors, the logic can be implemented with $t$ $n$-input AND gates for monitoring the inputs, $m$ OR gates for the output pattern vectors storage, with different number of inputs $n_{1s}(i) <= t$ which is the number of 1s in the corresponding output pattern vector $i$, where $1 \leq i \leq m$, and $m$ 2-input EXCLUSIVE-OR gates and one $m$-input OR gate for comparing the output vectors, as shown in figure 5.15 and presented in [Voyiatzis et al., 2009]. Note that the input pattern vectors contained in the fault pattern vectors of the available set serve for designing the input vector monitoring block, and the outputs pattern vectors serve for designing the output pattern vectors storage block.

Since the logic of the inputs monitoring block and the outputs response block have together the form of sums of products, the logic can also be implemented with programmable logic arrays, as proposed in [Sharma and Saluja, 1988]. A programmable logic array, in short PLA, has an AND matrix and an OR matrix for implementing Boolean functions expressed in the form of sums of products.

An alternative for the logic responsible of recognizing faults is to use a memory for storing the output pattern vectors contained in the given fault pattern vectors. Having a circuit with $n$ inputs and $m$ outputs, it is necessary a memory of size $2^n$ times $m$ bits for storing the output pattern vectors. That, if the address word of the memory is taken directly from the $n$ bits input vector of the circuit. If the $n$ bits input vector is decoded to a $p$ bits vector, where $t \leq 2^p$ and $t$ is the number of available fault pattern vectors, it is necessary a memory of size $2^p$ times $m$ bits for storing the output pattern vectors. Since, $n$ or $p$ bit address word points the $m$ bit data word containing the respective output pattern vector as shown in figure 5.16, having a memory of size $2^p$, requires an overflow detector for canceling a possible fault generated when an input vector can not be decoded because a fault pattern vector containing such input pattern vector has not been considered and consequently no output pattern vector exists. For some circuits for self-repairing, the use of a memory could impair a just-in-time

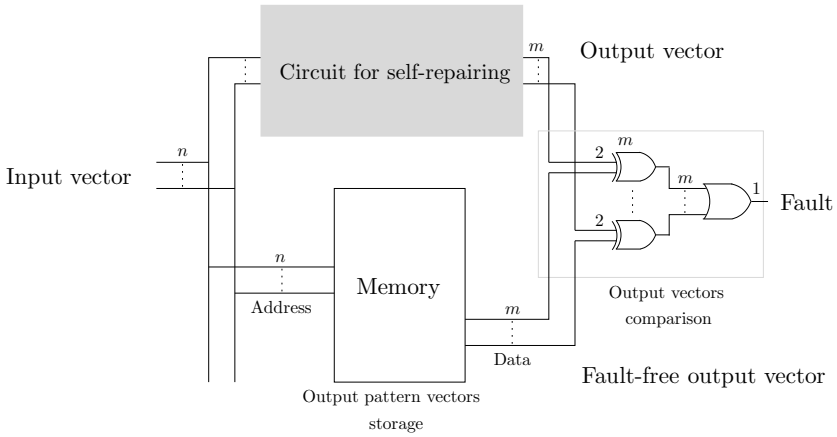



Figure 5.16: Fault recognition module design using a memory

delivery of the fault signal when the reading process of the fault-free output from the memory is slower than the logic of the circuit being tested.

Nevertheless, the use of a memory for storing the output pattern vectors or the whole fault pattern vectors could be useful for highly complex circuits like Systems-On-Chips, where the fault recognition of core or uncore components can be executed in idle times or in a scheduled manner. Although the System-On-Chip in not stopped for performing fault recognition of a core or uncore element, please see [Li et al., 2008] and [Li et al., 2010], the element should be stalled and isolated properly from the system for shifting in the input pattern vectors into the circuit with scan design and comparing the shifted-out output vectors with the corresponding output pattern vectors. In this case, the size of the memory required is dependent on the available number of fault pattern vectors for testing the cores or uncore elements. A reduction of the number of fault pattern vectors could reduce the time the element in stopped.

It is desired to reduce the hardware overhead that fault recognition causes in the circuit for self-repairing. Given $t$ fault pattern vectors with an input pattern vector of $n$ elements and an output pattern vector with $m$ elements, the hardware overhead of the fault recognition module can be reduced by reducing the number of fault pattern vectors $t$ shown in figure 5.15, or by reducing the elements $n$ and $m$ of the input and output pattern vectors respectively. Next subsections deal with this two ways of reducing the hardware overhead of the fault recognition module. The subsection dealing with fault pattern vectors number reduction refers to the reduction of the number $t$ of fault pattern vectors with binary elements in the available set. And the subsection dealing with fault vector dimension reduction refers to the reduction of the elements $n$ and $m$ of the input and output vectors with binary elements from the circuit for self-repairing and the input and output pattern vectors contained in the available set of fault pattern vectors.

### 5.2.2 Fault pattern vectors number reduction

Because of the hardware overhead that fault recognition could cause in the circuit, it is desired to reduce the logic responsible for recognizing faults. One method for reducing the fault recognition logic is to reduce the number of fault pattern vectors as much as possible maintaining the fault coverage. When the fault recognition module is implemented with logic

gates as show in figure 5.15, the number of AND gates required for implementing the input vectors monitoring block is equal to the number of fault pattern vectors $t$. Therefore, a reduced number of fault pattern vectors reduces the number of AND gates required for implementing the input vectors monitoring block. If the fault recognition logic includes a memory as shown in figure 5.16, the vertical size of that memory should be at least $t$. Therefore, a reduced number of fault pattern vectors reduces the size of the memory required for storing the fault-free output vectors.

The fault recognition module for a circuit can be implemented by means of a set of fault pattern vectors generated for detecting a set of faults. Since usually there exist some fault pattern vectors that detect more than one fault in the considered set of faults, redundant fault pattern vectors can be removed from the fault pattern vector set. That method is named fault pattern vector set compaction in this document and test set compaction in the literature about testing of digital circuits.

Many automatic test pattern generators, in short ATPGs, concentrate on delivering a set of fault pattern vectors that have a high fault coverage, but they do not consider in the generation the minimization of the number of fault pattern vectors that they will deliver. Compaction of a fault pattern vector set can be executed after the set has been generated by the ATPG, case in which is called static compaction. Or, it can be integrated into the automatic test pattern generation, case in which is called dynamic compaction.

Static compaction can be executed in two ways as presented in [Wang et al., 2006]. The first method is to construct a covering matrix with the fault pattern vectors in the rows and the faults in the columns. That table is commonly filled out by simulating a model of the circuit with the whole generated set of fault pattern vectors and a set of faults using a fault simulator. Then, a subset of fault pattern vectors is selected in the way that all columns, that is to say all faults, are covered. The second method can reduce a set of fault pattern vectors delivered with unspecified values, also called $X$ values or don't care bits. Thereby, similar fault pattern vectors which by means of its unspecified values can be expressed as only one fault pattern vector are searched, i.e. $0X10$ and $01X0$ give 0110.

Static compaction is executed in [Raedtke et al., 1995]. There, it is searched for a reduced set of fault pattern vectors, provided that several fault pattern vectors having nonspecified values for each fault type in a fault set are available. Thus, an optimal set of fault pattern vectors with one fault pattern vector for each fault type is searched in the way of having a high number of equal fault pattern vectors. The searching of such an optimal set has been implemented by means of a genetic algorithm in [Raedtke et al., 1995] as explained below.

A genetic algorithm is a searching heuristic where a population of individuals evolve through a determined number of generations. That, by using crossover and mutation in individuals selected according to a fitness function. In order to find a set of fault pattern vectors with a high number of similar fault pattern vectors, [Raedtke et al., 1995] applies a genetic algorithm in the following way. An individual is a set with fault pattern vectors for each fault type. The individuals having the minimal number of fault pattern vectors, that is to say, the maximum number of equal fault pattern vectors are selected in each generation. Crossover is executed by interchanging fault pattern vectors for the same fault type between two fault pattern vector sets. Mutation is executed by exchanging a fault pattern vector in a fault pattern vector set by another fault pattern vector for that fault type taken from the pool of generated fault pattern vectors for the corresponding fault type. Besides, a hillclimber procedure replaces a fault pattern vector in the fault pattern vector set that present no similar fault pattern vectors inside that fault pattern vector set with another fault pattern vector taken from the

pool of generated fault pattern vectors for the corresponding fault type. At the end, equal fault pattern vectors in the most optimal fault pattern vector set are reduced to only one fault pattern vector. Besides, using the unspecified values, similar fault pattern vectors are reduced using the second method of static compaction presented above. The results typed on table 5.20 show that smaller fault pattern vector sets are obtained in comparison to fault pattern vector sets obtained using other compaction methods.

The clonal selection algorithm developed for optimization problems and explained in section 3.2.2 is proposed in this thesis to be employed for finding a set of fault pattern vectors with a high number of equal fault pattern vectors, resembling the method presented in [Raedtke et al., 1995] and considering the individuals as cells. The advantage of the clonal selection algorithm in comparison to genetic algorithms is that a global optimal solution can be found faster. That is because clone cells of the cell with the best fitness are specifically mutated. Thereby, the number of produced clone cells is proportional to the fitness and the mutation rate applied to the clones is inversely proportional to the same fitness. Hence, the optimal solution can be found faster. Furthermore, the avoidance of a local optimum can be reached through the replacement of the cells that present the worst fitness, i.e. implementing death and birth of cells into the algorithm. That procedure is similar to the hillclimber procedure applied in [Raedtke et al., 1995]. The proposed clonal selection algorithm for finding the optimal set of fault pattern vectors with the highest number of equal fault pattern vectors is presented below.

**Algorithm 5.1:** Fault pattern vector set compaction using the clonal selection algorithm

**Input**: k as the initial number of sets to be created, number of iterations, number of cloning loops, cloning factor, mutation factor, number of worst sets for being replaced
**Output**: best set

```
1:  Computation of the number of available fault pattern vectors per fault in the pool of fault
    pattern vectors
2:  Creation of k sets of fault pattern vectors
3:  foreach iteration do
4:      Computation of the number of different fault pattern vectors in the clone sets
5:      Selecting the set with the minimal number of different fault pattern vectors for being cloned
6:      foreach cloning loop do
7:          Setting the number of clone sets inverse proportional to the number of different fault
            pattern vectors of the set to be cloned
8:          Setting the mutation rate proportional to the number of different fault pattern vectors
            of the set to be cloned
9:          Creation of clone sets of the set with the minimal number of different fault pattern
            vectors
10:         Changing of fault pattern vectors in the clone sets (Mutation)
11:         Computation of the number of different fault pattern vectors in the clone sets
12:         Selecting the clone set with the minimal number of different fault pattern vectors as
            new parent set
13:     end
14:     Replacing sets with the worst number of different fault pattern vectors with a new set
15: end
16: Selecting the set with the minimal number of different fault pattern vectors
17: Maintaining only the different and non-similar fault pattern vectors in the set
```

That algorithm has been implemented in Matlab and in order to evaluate it, different number of fault pattern vectors containing unspecified values for 22 stuck-at faults for the c17 benchmark circuit of the ISCAS 85 set has been generated with the ATPG tool ATALANTA. Thus, each fault pattern vector set contained 22 fault pattern vectors. Executing the implemented program, a set with 14 different fault pattern vectors including unspecified values were found as the best fault pattern vector set. The input variables used by executing the program has been set to: 7 initial number of sets, 3 iterations, 8 cloning loops, mutation factor of 5, clonation factor of 5 and 3 worst sets for being replaced. Additionally, reducing similar vectors that include unspecified values following the second method for static compaction presented in [Wang et al., 2006] and explained above, e.g. $XX111X0$ and $X111XX0$ give $X1111X0$, 8 fault pattern vectors including unspecified values were found. The implemented algorithm can serve on finding a minimal set in situations that require a set of fault pattern vectors with unspecified values such as one of the methods for fault vector dimension reduction presented in next subsection.

In a dynamic fault pattern vector set compaction, the compaction procedure is integrated into the automatic test pattern generator, where usually a fault simulator is used for verifying the fault coverage of every single fault pattern vector. Thereby the ATPG tries to generate fault pattern vectors that detect as much faults as possible in the way of having a set without redundant fault pattern vectors.

One method of dynamic compaction is the reverse order fault simulation, as proposed in [Schulz et al., 1988]. Thereby, fault simulation is applied to a generated set of fault pattern vectors starting from the last generated fault pattern vector. In that procedure, any fault pattern vector which does not detect new faults is removed. That procedure serves for removing fault pattern vectors that appeared to be redundant after its generation. That is why the fault simulation is executed in reverse order. The compactor available in the ATALANTA automatic test pattern generator tool, [Lee and Ha, 1993], executes compaction using that method and additionally the fault pattern vectors in the set are shuffled randomly and fault simulated again several times for removing further fault pattern vectors which do not detect a new fault.

Last method removes fault pattern vectors that do not recognize new faults during fault simulation in reverse order, but it does not care about fault pattern vectors that became redundant in the process of fault pattern vector generation. A method that approached that problem is the Redundant Vector Elimination presented in [Hamzaoglu and Patel, 2000], which by the fault pattern vector generation keeps track of the faults detected by each fault pattern vector, the number of essential faults of each fault pattern vector and the number of times a fault is detected.

When during the generation of fault pattern vectors, a fault pattern vector that detects a fault contains elements which can take any value, either 0 or 1, without affecting the recognition of the fault, those elements can be all left unspecified with $X$s, specified with 1s, specified with 0s, or specified randomly with 1s or 0s. Dynamic fault pattern vector set compaction methods tries to fill those values such as every single fault pattern vector recognizes more faults, [Wang et al., 2006].

The method presented in [Pomeranz et al., 1993] fills the unspecified values of the fault pattern vectors of the set by searching each time for a fault pattern vector that detects the next fault in a list of faults ordered such that, a fault pattern vector for a fault in the list would potentially detect the next fault in the list.

Two other methods of dynamic fault pattern vector set compaction make use of genetic

algorithms as presented in [Rudnick and Patel, 1999] and [Mazumder and Rudnick, 1998]. In both methods, after a fault pattern vector has been generated including unspecified values, a genetic algorithm tries to find a variant of the fault pattern vector such that it detects a higher number of faults than the original one. Fault simulation is used for measuring the fitness of each solution during evolution. By the first method, the unspecified values of the provided fault pattern vector are specified differently for each individual in the population. The population evolve to find the best filling of unspecified values. In the second method, the unspecified values of the provided fault pattern vector are specified differently for each individual in the population, but as the population evolves, specified values of the original fault pattern vector are also changed in order to find a fault pattern vector that detects more faults than the original one. Those methods have been employed for the generation of reduced sets of sequences of fault pattern vectors for sequential circuits without scan design or partial scan design.

Table 5.20 shows the sizes of compacted fault pattern vector sets for the benchmark combinational circuits of ISCAS 85 using the methods explained above and reported in the corresponding literature. We can see that the gap between those results and the lower bounds, taken from [Kajihara et al., 1993], is quite small, specially for the newest results in the table presented by [Hamzaoglu and Patel, 2000]. Although the results by [Raedtke et al., 1995] for the benchmark circuits c1355 and c1908 are smaller than the lower bound and it is impossible to reproduce those results again, the idea of how to implement a static compaction program using genetic algorithms is worth. Note that no results for dynamic fault pattern vector set compaction using genetic algorithms is shown on table 5.20, since the available methods in [Rudnick and Patel, 1999] are applied to sets of sequences of fault pattern vectors for sequential circuits. Even if dynamic compaction techniques are more frequently used by various ATPG tools, static compaction of fault pattern vector sets can be necessary when the ATPG tool does not compact the generated set or when the fault pattern vector set can still be compacted using a compaction method different to the one used by the ATPG tool.

Table 5.20: Summary of results from existing methods of fault pattern vector set compaction for combinational circuits

| Benchmark circuit | Lower bound | [Raedtke et al., 1995] | [Schulz et al., 1988] | [Lee and Ha, 1993] | [Hamzaoglu and Patel, 2000] | [Pomeranz et al., 1993] |
|---|---|---|---|---|---|---|
| c432 | 20 | 36 | 58 | 48 | 27 | - |
| c880 | 12 | 32 | 60 | 52 | 16 | 30 |
| c1355 | 84 | 67 | 88 | 84 | 84 | 86 |
| c1908 | 91 | 65 | 125 | 117 | 106 | 126 |
| c2670 | 39 | 72 | 127 | 115 | 44 | 67 |
| c3540 | 79 | 95 | 171 | 154 | 84 | 111 |
| c5315 | 36 | 74 | 143 | 114 | 37 | 56 |
| c6288 | 6 | 18 | 38 | 28 | 12 | 16 |
| c7552 | 28 | 117 | 231 | 204 | 73 | 87 |

### 5.2.3 Fault vector dimension reduction

Fault vector dimension reduction of fault pattern vectors with binary elements can be made in two ways. The first way is the use or introduction of unspecified values, also called $X$ values or don't care bits, in the fault pattern vectors of a set by the design of the fault recognition module. Although it is not possible to have the unspecified value at the same bit position in all fault pattern vectors of the fault pattern vector set, that procedure helps on reducing the hardware overhead caused by the introduction of a fault recognition module on

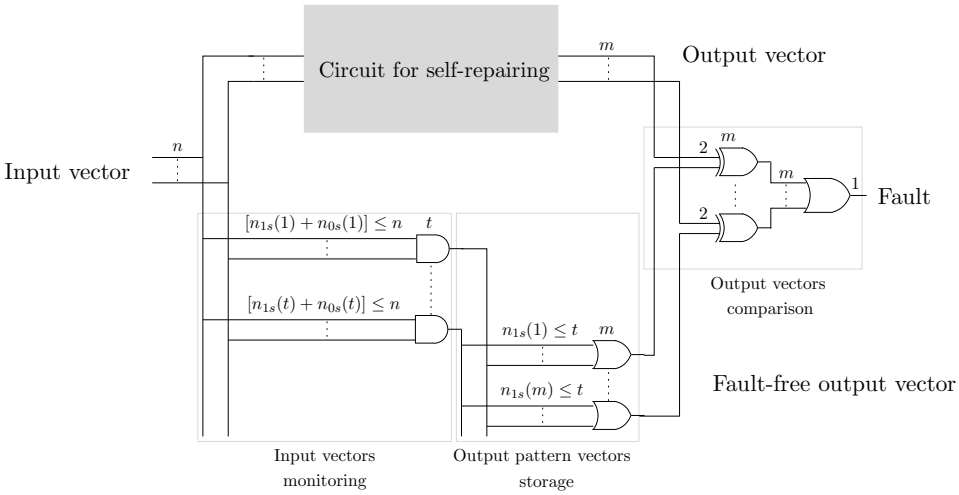



Figure 5.17: Fault recognition module design using unspecified values in the input patterns (abstracted and generalized from [Voyiatzis et al., 2009] ©2009 IEEE)

the circuit for self-repairing and also helps on reducing the mean time to recognize a fault, in short MTTRF, at operation time. The second way can be introduced considering input vectors and output vectors separately. Regarding the input pattern vectors contained in the fault pattern vectors, it consists on the compression of the input pattern vectors at design time and the decompression of the respective compressed vectors at operation time for being applied to the circuit for self-repairing, method that can be only useful for offline testing in a non-concurrent fault recognition scheme. Regarding the output vectors, it consists on the compaction of the output vectors form the circuit for self-repairing and the output pattern vectors for reducing the hardware necessary for the output vectors comparison block and the output pattern vectors storage block, which helps on reducing the whole hardware overhead of the fault recognition module. Those two ways of fault vector dimension reduction are explained in detail in the following subsections.

### Set of fault pattern vectors with unspecified values

Fault vector dimension reduction can be executed using or introducing, when not available, unspecified values $X$ in the fault pattern vectors. The use of unspecified values $X$ in the input pattern vectors contained in the fault pattern vectors allows the design of the input vector monitoring block with AND gates with reduced number of inputs $n_{1s}(i) \leq n$, equal to the number of 1s and 0s in the input pattern vector contained in the respective fault pattern vector $i$, where $1 \leq i \leq t$, as shown in figure 5.17. Then, this procedure helps on reducing the hardware overhead caused by the fault recognition module in the circuit. Besides it helps on decreasing the mean time to recognize a fault because for each unspecified value in an input pattern vector contained in any fault pattern vector, the probability that an input vector occurs and matches that input pattern vector implemented with an AND gate increases by a factor of 2.

The introduction of $X$ values in the output pattern vectors contained in the fault pattern vectors does not help on reducing the hardware overhead, instead it increases the logic needed since a monitor for the $X$ values should be inserted, as implemented in [Voyiatzis et al., 2009]

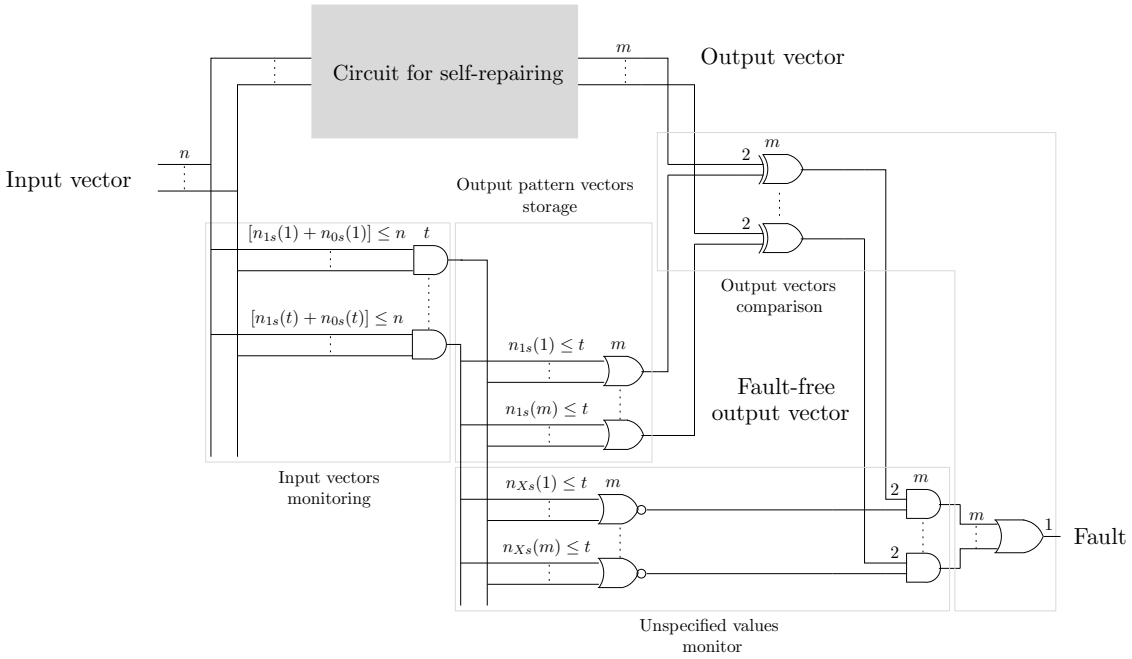



Figure 5.18: Fault recognition module design using an unspecified values monitor (abstracted and generalized from [Voyiatzis et al., 2009] ©2009 IEEE)

and named the MOBEX scheme. That monitor avoids to trigger a fault when any output bit of the circuit and its corresponding output pattern bit $X$, filled randomly with either an 1 or a 0, are different. Being that output bit not necessary for detecting the fault or faults that the fault pattern vector is intended to detect. Figure 5.18 shows the additional block necessary for monitoring the unspecified values in the output pattern vectors. The unspecified values monitor is implemented with NOR gates. Whenever an unspecified value is present in any fault pattern vector, a signal is raised and inverted to cancel by means of an AND gate the signal coming from the respective XOR gate. Most automatic test pattern generators provide fault pattern vectors with unspecified values in both the input and the output pattern vectors. Therefore, when the fault pattern vectors are generated with a conventional ATPG, such an unspecified values monitor is unavoidable.

A way of reducing the hardware overhead introduced by the use of unspecified values in the output pattern vectors contained in the fault pattern vectors is to employ multiplexers for masking that unspecified values, as proposed in [Voyiatzis et al., 2009] and named as the IVEX scheme. With that method, the hardware overhead can be reduced notoriously when the total number of $X$ values of all available output pattern vectors is higher than the total number of 0 values. That, since in the design of the output pattern vectors storage block only the values 0 or 1 of the elements of the output pattern vectors are considered. Table 5.21 shows how that design works. The set of $m$ multiplexers forward either the signals coming from the AND gates that considered the 1s at that bit position or the signals coming from the AND gates that considered the 0s at that bit position, according to the value present at the selector lines which come from the outputs of the circuit. Note that the lines coming from the outputs of the circuit are inverted at the selector inputs of the multiplexers. The case where both inputs to any multiplexer, 1s or 0s, have the value of 1 can not occur. In the case when

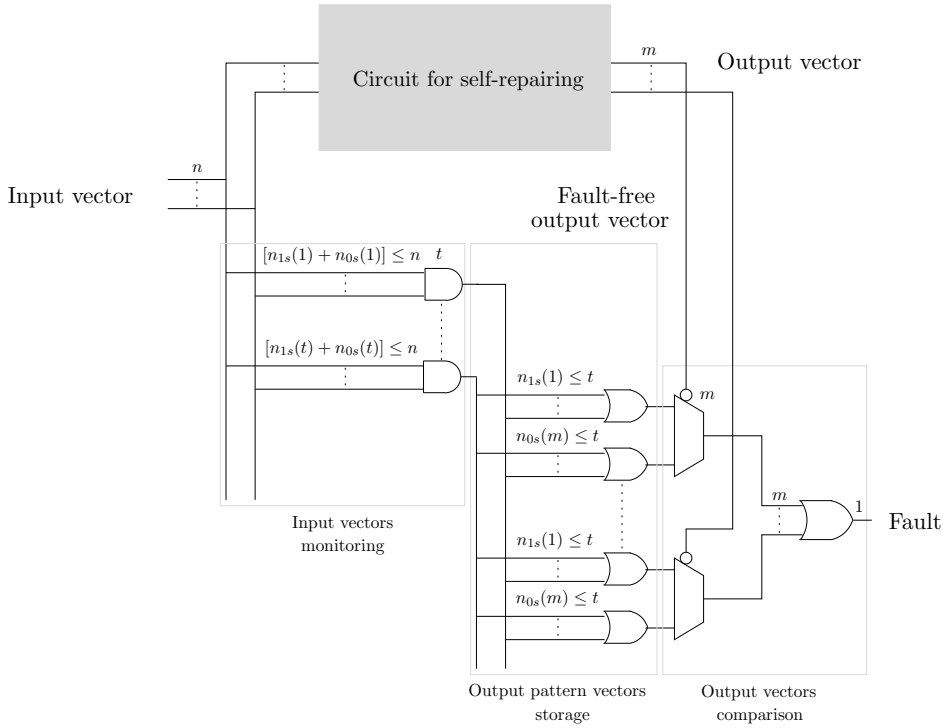



Figure 5.19: Fault recognition module design using multiplexers (abstracted and generalized from [Voyiatzis et al., 2009] ©2009 IEEE)

both inputs 1s or 0s to the same multiplexer have the value of 0, that is the output pattern bit has a $X$ value, it does not matter whether the 1s or the 0s input to the multiplexer is selected, since the output of the multiplexer will always be a 0, which means there is no fault. Then, the $X$ at that bit position has been masked. In case an output bit of the circuit for self-repairing is 0, the respective multiplexer forwards the value of the input 1s, which should have the value of 0 when there is no fault, otherwise it has a value of 1, which means that the output bit from the circuit for self-repairing and the respective output pattern bit are different. In case an output bit of the circuit of self-repairing is 1, the respective multiplexer forwards the input 0s, which should have the value of 0 when there is no fault, otherwise it has a value of 1, which means that the output bit from the circuit for self-repairing and the respective output pattern bit are different.

[Kochte et al., 2009] shows the design of a concurrent fault recognition module using unspecified values in the output pattern vectors. Thereby, when an input pattern vector is present at the input of the circuit for self-repairing, only the output bits that present a value of 0 or 1 in the respective output pattern vector are observed. The design can be understood by means of the truth tables 5.22 and 5.23. When the value of an output pattern bit is 0, the direct value of the respective output bit from the circuit for self-repairing is connected to an AND gate together with the output line coming from the AND gate that detects the respective input pattern vector, signal named as hit. When the hit signal has a value of 1 and the output bit from the circuit for self-repairing has the value of 1, different from the expected value 0, a fault is signalized with a 1 value at the output of the AND gate, otherwise a 0 value is given, which means that no fault occurred at that output bit. Similarly, when the value

Table 5.21: Truth table for the output vectors comparison using multiplexers

| | 1s | 0s | Output bit | MUX selector = $\overline{\text{Output bit}}$ | MUX output = Fault |
|---|---|---|---|---|---|
| X | 0 | 0 | 0 | 1s | 0 |
| | 0 | 0 | 1 | 0s | 0 |
| | 0 | 1 | 0 | 1s | 0 |
| | 0 | 1 | 1 | 0s | 1 |
| | 1 | 0 | 0 | 1s | 1 |
| | 1 | 0 | 1 | 0s | 0 |
| - | 1 | 1 | 0 | 1s | - |
| | 1 | 1 | 1 | 0s | - |

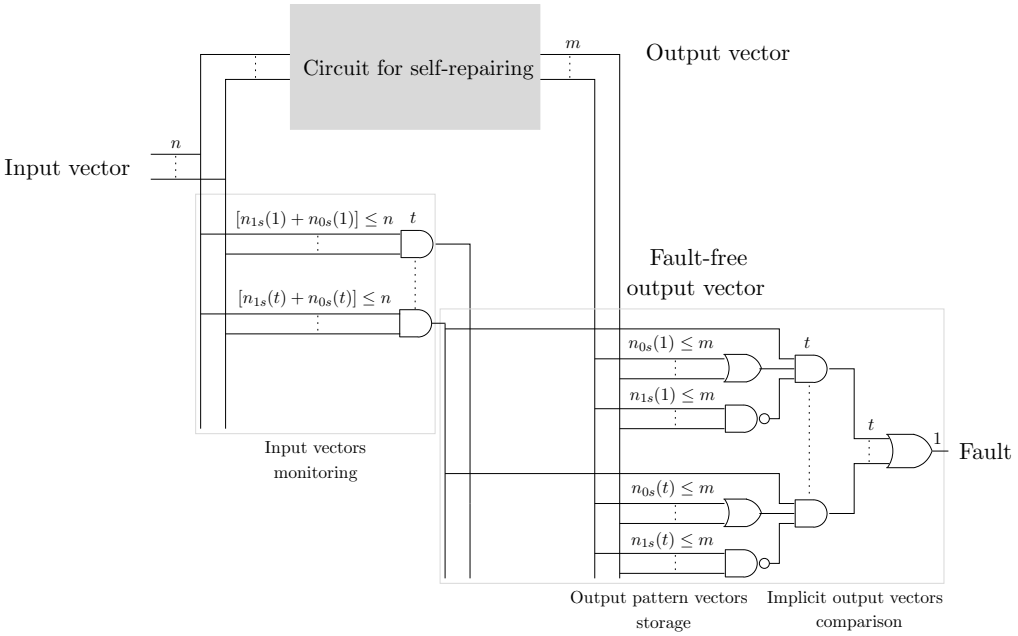



Figure 5.20: Fault recognition module design using implicit output vectors comparison (abstracted and generalized from [Kochte et al., 2009] ©2009 IEEE)

of the output pattern bit is 1, the respective output bit from the circuit for self-repairing is inverted and connected to an AND gate together with the output line coming from the AND gate that detects the respective input pattern vector, signal named as hit. When the hit signal has a value of 1 and the output bit from the circuit for self-repairing has the value of 0, different from the expected value 1, a fault is signalized with a 1 value at the output of the AND gate, otherwise a 0 value is given which means that no fault occurred at that output bit. Since the output signal from the AND gate, that detects the respective input pattern vector, is used for all the output bits to be observed for that output pattern vector, i.e., the outputs having a value 0 or 1, only one AND gate is necessary if all the output signals from the circuit for self-repairing connected directly or inversely are collected by means of an OR gate and a NAND gate respectively, as seen in figure 5.20.

The hardware overhead for all the design alternatives presented above caused by the fault

Table 5.22: Truth table for the implicit output vectors comparison when pattern bit = 0

| Hit | Output bit | Output bit | Fault |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Table 5.23: Truth table for the implicit output vectors comparison when pattern bit = 1

| Hit | Output bit | $\overline{\text{Output bit}}$ | Fault |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

recognition module is expressed symbolically in table 5.26. In order to show the computation of the hardware overhead by means of an example, two minimal sets of fault pattern vectors for the benchmark circuit c17 which has 5 inputs and 2 outputs, one with completely specified values and another containing unspecified values where generated in the following way. A compact set with all specified values has been generated using the ATALANTA automatic test pattern generation tool. Likewise, a compact set containing unspecified values has been generated using ATALANTA and a program that implemented the algorithm 5.1 for compacting the set. So in the first set we have 4 fault pattern vectors with all specified values, which are shown in table 5.24, and in the second set we have 8 fault pattern vectors containing unspecified values in the input and output pattern vectors, which are shown in table 5.25. The following symbols are used in the computation of the hardware overhead of the fault recognition module.

| | |
|---|---|
| $t$ | number of fault pattern vectors in the set |
| $n$ | number of inputs of the circuit |
| $m$ | number of outputs of the circuit |
| $n_{1s\text{IM}}(i)$ | number of 1 values in the line $i$ of the input pattern vectors matrix |
| $n_{0s\text{IM}}(i)$ | number of 0 values in the line $i$ of the input pattern vectors matrix |
| $n_{Xs\text{IM}}(i)$ | number of $X$ values in the line $i$ of the input pattern vectors matrix |
| $n_{1s\text{OM}}(i)$ | number of 1 values in the line $i$ of the outputs pattern vectors matrix |
| $n_{0s\text{OM}}(i)$ | number of 0 values in the line $i$ of the outputs pattern vectors matrix |
| $n_{Xs\text{OM}}(i)$ | number of $X$ values in the line $i$ of the outputs pattern vectors matrix |

The symbolic expression of the hardware overhead for the input vectors monitoring block looks the same for all schemes. However, for the first scheme the number of 1s and 0s in every input pattern vectors is equal to the number of inputs, since all values are specified. Besides the number of input pattern vectors is only 4. The other three schemes present the same hardware overhead for the input vectors monitoring block. The number of 1s and 0s in every input pattern vector is less than number of inputs because of the presence of unspecified

Table 5.24: Fault pattern vectors with all specified bits for the c17 benchmark circuit

| Input pattern vectors | Outputs pattern vectors |
|---|---|
| 10101 | 11 |
| 01010 | 11 |
| 10000 | 00 |
| 01111 | 00 |

Table 5.25: Fault pattern vectors with unspecified bits for the c17 benchmark circuit

| Input pattern vectors | Outputs pattern vectors |
|---|---|
| x00x0 | 00 |
| 10111 | 10 |
| 01100 | 11 |
| 100x1 | 01 |
| 11111 | 10 |
| 00011 | 01 |
| 001xx | 0x |
| 010xx | 11 |

values. That is the reason why a reduction of the hardware overhead can be reached using unspecified values at the input pattern vectors.

Please note that in figures 5.15, 5.18, 5.19 and 5.20, the input vector monitoring block has been drawn only with AND gates for simplifying the figures. However, the inputs to the AND gates are determined by the values in the input pattern vectors. Where, the inputs with a value of 1 should be connected directly and the inputs with a value of 0 should be inverted. Then, the values 1 of an input pattern vector can be collected whit a AND gate and the values 0 of the same input pattern vector can be collected with a NOR gate. The last is possible by considering that $\overline{\text{Input X}} \cdot \overline{\text{Input Y}} = \overline{\text{Input X} + \text{Input Y}}$. Finally, the output of the AND and NOR gates can be connected to a two-input AND gate as seen in figure 5.21. In this way, the hardware overhead of the input vectors monitoring block has been computed more exactly.

In order to compare all the schemes, the necessary gates for the benchmark circuit c17 have been transformed to the NAND logic. Please look for NAND logic at [Wikipedia, 2011] for more details. Were: an AND gate requires 2 NAND gates, an INV gate requires 1 NAND gate, an OR gate requires 3 NAND gates, a XOR gate requires 4 NAND gates, and a Multiplexer unit requires 4 NAND gates considering that a two-input multiplexer has the following Boolean formula: Multiplexer-input-1 $\cdot \overline{\text{Selector}} +$ Multiplexer-input-2 $\cdot$ Selector. The computation of the total number of NAND gates required for the small benchmark circuit c17 is only illustrative. A comparison of all those schemes in terms of hardware overhead with only that example is not possible and it would require to consider bigger circuits with different test pattern vectors sets. Please note that the computation of the required NAND gates for the second, third and fourth schemes has been executed for the set of 8 fault pattern vectors containing unspecified values.

The first scheme that has been introduced by [Sharma and Saluja, 1988] can only be used for the design of the fault recognition module using a fault pattern vector set with only

Table 5.26: Hardware overhead of the fault recognition module

| Block | Number of gates | c17 circuit $n = 5, m = 2$ |
|---|---|---|
| Fault recognition module design using only specified values. Scheme from [Sharma and Saluja, 1988]. $t = 4$ | | |
| Input vectors monitoring | $\sum_{i}^{t} ceil[n_{1sIM}(i) - 1]$ $_{2IN}$AND, $\sum_{i}^{t} ceil[n_{0sIM}(i) - 1]$ $_{2IN}$OR, $t$ $_{2IN}$INV, $t$ $_{2IN}$AND | 6 $_{2IN}$AND, 5 $_{2IN}$OR, 4 $_{2IN}$INV, 4 $_{2IN}$AND |
| Output pattern vectors storage | $\sum_{i}^{m} ceil[n_{1sOM}(i) - 1]$ $_{2IN}$OR | 2 $_{2IN}$OR |
| Output vectors comparison | $m$ $_{2IN}$XOR, $(m-1)$ $_{2IN}$OR | 2 $_{2IN}$XOR, 1 $_{2IN}$OR |
| Total gates | | 10 $_{2IN}$AND, 8 $_{2IN}$OR, 4 $_{2IN}$INV, 2 $_{2IN}$XOR |
| Total $_{2IN}$NAND gates | | 56 $_{2IN}$NAND |
| Fault recognition module design using an unspecified values monitor. Scheme MOBEX from [Voyiatzis et al., 2009]. $t = 8$ | | |
| Input vectors monitoring | $\sum_{i}^{t} ceil[n_{1sIM}(i) - 1]$ $_{2IN}$AND, $\sum_{i}^{t} ceil[n_{0sIM}(i) - 1]$ $_{2IN}$OR, $t$ $_{2IN}$INV, $t$ $_{2IN}$AND | 10 $_{2IN}$AND, 9 $_{2IN}$OR, 8 $_{2IN}$INV, 8 $_{2IN}$AND |
| Output pattern vectors storage | $\sum_{i}^{m} ceil[n_{1sOM}(i) - 1]$ $_{2IN}$OR | 6 $_{2IN}$OR |
| Output vectors comparison | $m$ $_{2IN}$XOR, $(m-1)$ $_{2IN}$OR | 8 $_{2IN}$XOR, 7 $_{2IN}$OR |
| Unspecified values monitor | $\sum_{i}^{m} ceil[n_{XsOM}(i) - 1]$ $_{2IN}$OR, $2m$ $_{2IN}$AND | 0 $_{2IN}$OR, 1 $_{2IN}$AND |
| Total gates | | 19 $_{2IN}$AND, 22 $_{2IN}$OR, 8 $_{2IN}$INV, 8 $_{2IN}$XOR |
| Total $_{2IN}$NAND gates | | 144 $_{2IN}$NAND |
| Fault recognition module design using multiplexers. Scheme IVEX from [Voyiatzis et al., 2009]. $t = 8$ | | |
| Input vectors monitoring | $\sum_{i}^{t} ceil[n_{1sIM}(i) - 1]$ $_{2IN}$AND, $\sum_{i}^{t} ceil[n_{0sIM}(i) - 1]$ $_{2IN}$OR, $t$ $_{2IN}$INV, $t$ $_{2IN}$AND | 10 $_{2IN}$AND, 9 $_{2IN}$OR, 8 $_{2IN}$INV, 8 $_{2IN}$AND |
| Output pattern vectors storage | $\sum_{i}^{m} ceil[n_{1sOM}(i) - 1]$ $_{2IN}$OR, $\sum_{i}^{m} ceil[n_{0sOM}(i) - 1]$ $_{2IN}$OR | 6 $_{2IN}$OR, 5 $_{2IN}$OR |
| Output vectors comparison | $m$ $_{2IN}$MUX, $m$ $_{2IN}$INV, $(m-1)$ $_{2IN}$OR | 8 $_{2IN}$MUX, 8 $_{2IN}$INV, 7 $_{2IN}$OR |
| Total gates | | 18 $_{2IN}$AND, 27 $_{2IN}$OR, 16 $_{2IN}$INV, 8 $_{2IN}$MUX |
| Total $_{2IN}$NAND gates | | 165 $_{2IN}$NAND |
| Fault recognition module design using implicit output vectors comparison. Scheme from [Kochte et al., 2009]. $t = 8$ | | |
| Input vectors monitoring | $\sum_{i}^{t} ceil[n_{1sIM}(i) - 1]$ $_{2IN}$AND, $\sum_{i}^{t} ceil[n_{0sIM}(i) - 1]$ $_{2IN}$OR, $t$ $_{2IN}$INV, $t$ $_{2IN}$AND | 10 $_{2IN}$AND, 9 $_{2IN}$OR, 8 $_{2IN}$INV, 8 $_{2IN}$AND |
| Output pattern vectors storage and implicit output vectors comparison | $\sum_{i}^{t} ceil[n_{1sOM}(i) - 1]$ $_{2IN}$OR, $\sum_{i}^{t} ceil[n_{0sOM}(i) - 1]$ $_{2IN}$OR, $t$ $_{2IN}$INV, $2t$ $_{2IN}$AND, $(t-1)$ $_{2IN}$OR | 2 $_{2IN}$OR, 3 $_{2IN}$OR, 8 $_{2IN}$INV, 16 $_{2IN}$AND, 7 $_{2IN}$OR |
| Total gates | | 34 $_{2IN}$AND, 21 $_{2IN}$OR, 16 $_{2IN}$INV |
| Total $_{2IN}$NAND gates | | 147 $_{2IN}$NAND |

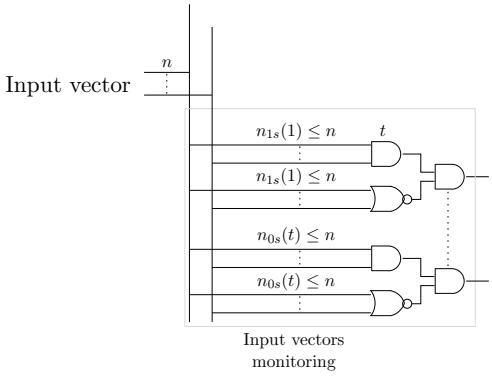



Figure 5.21: Input vector monitoring block (abstracted and generalized from [Voyiatzis et al., 2009] ©2009 IEEE)

specified bits. The second scheme MOBEX introduced by [Voyiatzis et al., 2009] differs from the first scheme in that it presents an unspecified values monitor block for handling with the unspecified values in the output pattern vectors. The third scheme IVEX introduced also by [Voyiatzis et al., 2009], presents a way of masking the unspecified values in the output pattern vectors by using multiplexers instead of XOR gates, only considering the specified values for the design of the output vectors comparison block. Therefore, that scheme is useful when the number of unspecified values $X$ is higher than the number of specified values 0. That scheme has been conceived for handling with unspecified values in the output pattern vectors, however it can also be used when the fault pattern vectors have only specified values. The fourth scheme introduced by [Kochte et al., 2009] can also be used for output pattern vectors with only specified values or output pattern vectors containing unspecified values. That because in the implicit output vectors comparison block, that scheme considers only the specified values in the output pattern vectors. Therefore, a reduction in the hardware overhead can be given when unspecified values are available in the output pattern vectors.

In the case a fault pattern vector set with only specified values 0s and 1s is available, some methods for finding and introducing unspecified values in a given set of fault pattern vectors maintaining the fault coverage are the following. [Miyase and Kajihara, 2004] performs a don't care identification in fault pattern vectors for combinational circuits with the help of a fault simulator. Thereby, for each fault pattern vector, it determines first which faults are covered and then it starts to flip the value of one element on the vector at each time and verify by fault simulation whether the faults are still covered. When the value in that bit position can be changed without affecting the coverage of the fault pattern vector, the value is set to $X$. [Pomeranz and Reddy, 2006] reduces the number of specified values in the given set of fault pattern vectors by increasing the number of fault pattern vectors. That is executed by replacing a fault pattern vector with a subset of other fault pattern vectors which cover all together the same faults covered by the initial fault pattern vector, but have fewer specified values each in comparison with the initial fault pattern vector. [Kochte et al., 2009] reduces the number of specified values in a set of fault pattern vectors by test set stripping and test pattern splitting which are methods that have the same principles as the both methods explained above. Test set stripping takes 0s and 1s from fault pattern vectors away by flipping values and fault simulating for verifying the fault coverage. Test pattern splitting reduces the number of specified values by splitting the faults covered by a single fault pattern vector into

subsets and replacing that fault pattern vector by other fault pattern vectors covering each fault pattern vector only one subset, in consequence increasing the number of pattern vectors in the set. However, [Kochte et al., 2009] limits the number of specified values in each fault pattern to a determined number. So the procedure of test pattern splitting is executed until the desired number of specified values in each fault pattern vector is reached.

#### Input vector compression and output vector compaction

In the area of chip testing, the compression of the input pattern vectors is named test stimulus compression and the compression of the output pattern vectors is named test response compaction. The test stimulus compression should be information lossless in order to maintain the fault coverage of the original test set. On the contrary, the output response compaction can be lossy [Wang et al., 2006], that is why it is named compaction instead of compression.

When a determined digital circuit is provided with built-in self-test, normally the input pattern vectors for being applied in a determined testing phase are generated pseudo-randomly by a test pattern generator circuit based on a linear feedback shift register, and not by an automatic test pattern generator. Only the input pattern vectors that can not be generated by such a pseudo-random pattern generator are generated by an ATPG, which tries to cover the remaining faults named random-pattern resistant faults. Such a set of input pattern vectors contained in the fault pattern vectors generated by the ATPG can be compressed into vectors with less number of elements together with a so called polynomial identifier. Those vectors constitute the seeds of a linear feedback shift register and its computation implies solving a set of linear equations, [Hellebrand et al., 1992]. Only the set of seeds and the respective polynomial identifiers need to be stored in memory. At the time of testing, those seeds can be fed into a linear feedback shift register that decompresses the seed vectors to the original input pattern vectors. That compression scheme is only useful for saving storage space for testing a digital circuit in idle or scheduled times. It is not suitable for the design of a fault recognition module which should perform online concurrent fault recognition, where the input vectors monitoring block looks for a known input pattern vector at the inputs of the fault circuit for self-repairing in order to give a sign for starting the evaluation of the outputs of the circuit for self-repairing.

The output vector from the circuit for self-repairing can be compacted by means of a compactor circuit. Such a compactor circuit can be able to compact the output vector in time or space. A space compactor reduces the elements of the output vector from $m$ to $q$ and a time compactor produces a signature of a sequence of consecutive output vectors. For an online concurrent fault recognition scheme, a space compactor could be of advantage for reducing the hardware overhead of the fault recognition circuit since the output pattern vectors storage and output vectors comparison blocks can be reduced by reducing the elements of the output vectors from $m$ to $q$. The idea of the inclusion of a output vector compactor is given in figure 5.22.

A method for compacting the output vectors is the X-Compact technique presented in [Mitra, 2004] which has been implemented in some commercial test compression tools. The design of the output vector compactor block with the X-Compact technique can allow to work with output pattern vectors with unspecified values in the output pattern vectors storage block. That means no modification in all other blocks is necessary. That because the X-Compact technique is independent of the fault patter vectors, in consequence independent to the ATPG used for generating the fault pattern vectors and correspondingly the covered fault

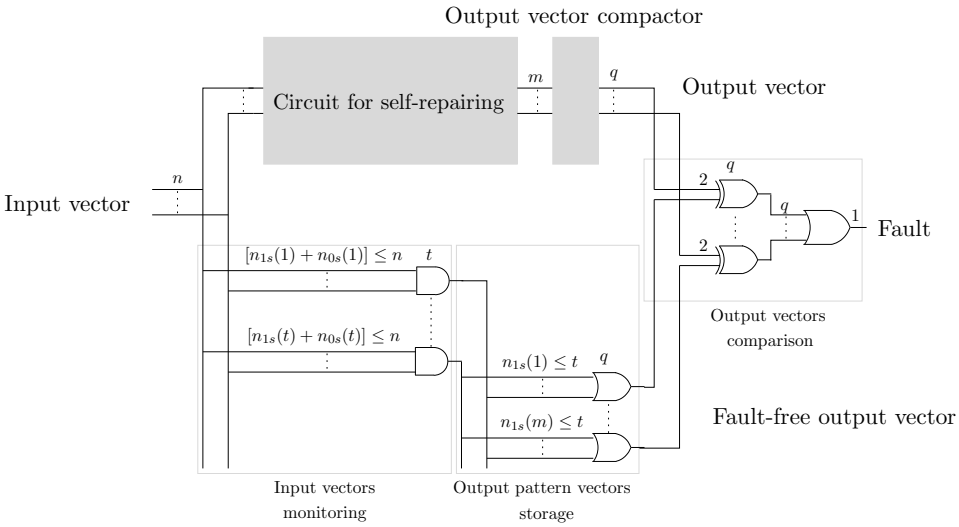



Figure 5.22: Fault recognition module design using an output vector compactor

models. The X-Compact technique give some rules for the filling of the so called X-Matrix with 1s and 0s. That matrix is used for the construction of a circuit based only in XOR gates. XOR gates because of the information loss-less property of that gates, [Mitra, 2004]. Thereby, the number of matrix rows is set with the number of elements of the output vector, in our case $m$. The number of columns of the matrix is set with the number of elements of the compacted output vector $q$. Both numbers should have the following relation: $m \leq 2^q$ and fulfill a set of rules for the detection of faults.

The rules that guarantees to detect one fault are:

- No row contain all 0s
- All rows are different

The additional rule that guarantees to detect two faults is:

- Every row contains an odd number of 1s

The additional rule that guarantees to to detect a fault when an unspecified value is present at the corresponding fault pattern vector is:

- The submatrix obtained by removing a row and a column having 1s in that should not contain a row with all 0s

The additional rule that guarantees to to detect two faults when an unspecified value is present at the corresponding fault pattern vector is:

- The submatrix obtained by removing a row and a column having 1s in that should contain all distinct rows

Then the minimum number of elements for the compacted output vectors that could be achieved, as a function of the number of elements in the output vectors $m$ and the fulfillment of the rules exposed above is shown in table 5.27.

Table 5.27: Possible compaction using the X-Compact technique

| $m$ | $q$ |
|---|---|
| 6-8 | 6 |
| 9-32 | 10 |
| 33-128 | 14 |
| 129-512 | 18 |
| 513-2048 | 22 |

Table 5.28: Compaction using the X-Compact technique for the circuits of ISCAS 85

| Benchmark circuit | $m$ | $q$ |
|---|---|---|
| c17 | 2 | - |
| c432 | 7 | 6 |
| c499 | 32 | 10 |
| c880 | 26 | 10 |
| c1355 | 32 | 10 |
| c1908 | 25 | 10 |
| c2670 | 140 | 18 |
| c3540 | 22 | 10 |
| c5315 | 123 | 14 |
| c6288 | 32 | 10 |
| c7552 | 108 | 14 |

A method for creating such matrix is to choose an odd number $s$ such that $m \leq 2^{2s}$. Then a $\mathbf{A}$ matrix with all $2^s$ combinations should be created and then another matrix $\mathbf{B}$ obtained by bit-wise complementing the first matrix $\mathbf{A}$. The final matrix is the concatenation of the rows of both matrix in the form $\mathbf{A}|\mathbf{B}$. An example of such matrix is given below and its respective circuit is drawn in figure 5.23.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Table 5.28 shows an overview of the possible compaction rate for the benchmark circuits of the set ISCAS 85, if the circuits are provided with such an output vector compactor according to their number of outputs $m$.

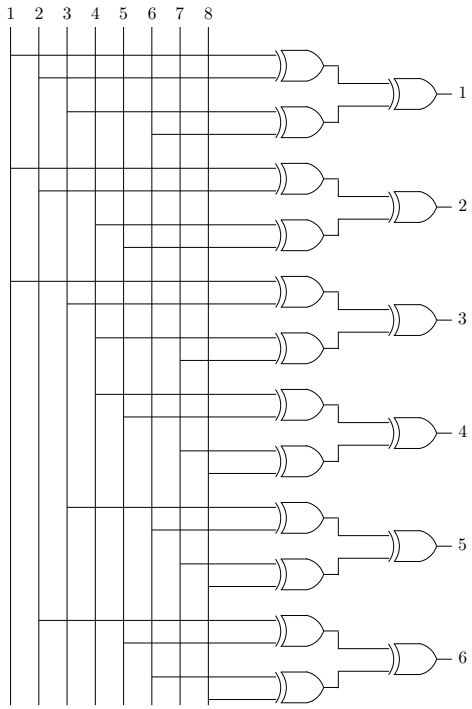



Figure 5.23: X-Compactor circuit for $m = 8$ and $q = 6$ (adapted from [Mitra, 2004] ©2004 IEEE)

## 5.3 Conclusions

This chapter presented the evaluation of some methods for the design of a fault recognition module for a circuit for self-repairing that delivers input and output vectors with real or binary elements.

For the design of such a fault recognition module, a fault vector has been defined as the aggregation of input and output signals. So, the present input and output signals from the present fault vector is compared with a set of fault pattern vectors, in order to determine whether the circuit for self-repairing is faulty or not during its operational phase.

Since the fault recognition process relies on a vector comparison, some distance measurement methods have been evaluated. The Hamming distance measurement method by fault vectors with binary elements allows an exact comparison. Therefore, the determination that there exists a fault in the circuit for self-repairing is straightforward. Under a determined input vector, a mismatch of the output vector from the circuit for self-repairing and a corresponding output pattern vector leads to determine that there exist a fault. In vectors with real elements, an exact comparison is not possible. Therefore, a class has been attached to each fault pattern vector. That class number contains the information of whether the circuit for self-repairing is faulty or not and when faulty, which type of fault is present. For that, some fault distance measurement methods has been evaluated and additionally some class assignation methods.

The fault coverage is dependent on the quality of the set of fault pattern vectors which are generated for covering a set of faults. The fault efficiency is dependent on the fault coverage and can be defined as the quotient between the faults covered and the fault detected. By

the design of the fault recognition module for a circuit for self-repairing that delivers a fault vector with binary elements, the fault efficiency can be in most cases assured to be 1. That because a single fault pattern vector is able to determine whether there exist a fault or not. That is not the case for a circuit with a fault vector with real elements. There, the faults detected are dependent of the distance measurement method, the class assignation method and the set of fault pattern vectors stored into the fault recognition module.

By working with fault pattern vectors with binary elements, the fault recognition module consists in an input vectors monitoring block, an output pattern vectors storage block and an output vectors comparison block. The hardware overhead of those blocks depends on the number of fault pattern vectors and the number of elements of the input pattern vectors and output pattern vectors contained in the fault pattern vectors of the available set. By working with fault pattern vectors with real elements, the recognition module consists in a distance measurement block, a class assigner block and a memory block. The hardware overhead of those blocks depends also on the number of fault pattern vectors and the number of elements of those vectors. That because they should be stored in memory. That is the reason why this chapter evaluated some algorithms and schemes for reducing the number of fault pattern vectors in the available set and the elements, called also dimensions, of the fault vectors.

The mean time to recognize a fault, which is the time taken by a fault recognition module for recognizing that a fault happened in the circuit for self-repairing, is dependent on the quality of the fault pattern vectors set and also on the time that the fault recognition takes to recognize that fault. So, in a circuit for self-repairing that delivers fault pattern vectors with binary elements, the fault recognition module can be designed for working concurrently with the operation of the circuit for self-repairing. In that case, the mean time to recognize a fault depends on the probability that an input vector that matches an input pattern vector appears at the inputs of the circuit for self-repairing.

Regarding fault pattern vectors with binary elements, [Voyiatzis et al., 2009] proposes to design a fault recognition module using all the fault pattern vectors generated by an ATPG. However, the size of a fault pattern vector set for a circuit for self-repairing with many inputs and for a high fault coverage, is big. Therefore, [Kochte et al., 2009] proposes to consider only fault pattern vectors for random pattern resistant faults of for only critical faults. That is feasible, since the self-repairing circuit could use all other random patterns for testing offline the health of the circuit at the start or in scheduled or idle times. However, the advantages of a concurrent fault recognition module is that the circuit does not need to enter in a test mode or to be stopped in order to monitor for critical faults.

Having evaluated some methods for designing a fault recognition module, next chapter deals with the hardware implementation issues for a self-repairing circuit.

## 5.4 Bibliography

Brglez, F., Bryan, D., and Kozminski, K. (1989). *Notes on the ISCAS'89 Benchmarks Circuits*. MCNC.

Bryan, D. (1988). *The ISCAS'85 benchmark circuits and netlist format*. MCNC.

Collaborative Benchmarking and Experimental Algorithmics Laboratory (2007). The Benchmark Archives at CBL (up to 1996).

Hamzaoglu, I. and Patel, J. H. (2000). Test Set Compaction Algorithms for Combinational Circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):283–289. IEEE.

Hellebrand, S., Tarnick, S., Rajski, J., and Courtois, B. (1992). Generation Of Vector Patterns Through Reseeding Of Multiple-Polynomial Linear Feedback Shift Registers. In *International Test Conference - ITC 1992*, pages 120–129.

Kajihara, S., Pomeranz, I., Kinoshita, K., and Reddy, S. M. (1993). Cost-Effective Generation of Minimal Test Sets for Stuck-at Faults in Combinational Logic Circuits. In *30th International Design Automation Conference - DAC 1993*, pages 102–106. ACM.

Kalla, P. and Ciesielski, M. (1998). A Comprehensive Approach to the Partial Scan Problem using Implicit State Enumeration. In *International Test Conference - ITC 1998*, pages 651–657. IEEE Computer Society.

Kirkland, T. and Mercer, M. R. (1988). Algorithms for Automatic Test Pattern Generation. *Design & Test*, 5(3):43–55. IEEE Computer Society.

Kochte, M. A., Zoellin, C. G., and Wunderlich, H.-J. (2009). Concurrent Self-Test with Partially Specified Patterns For Low Test Latency and Overhead. In *14th European Test Symposium*, pages 53–58. IEEE Computer Society.

Lee, H. K. and Ha, D. S. (1991). An Efficient Forward Fault Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation. In *International Test Conference - ITC 1991*, pages 946–955.

Lee, H. K. and Ha, D. S. (1993). On the Generation of Test Patterns for Combinational Circuits. Technical Report 12-93, Department of Electrical Engineering, Virginia Polytechnic Institute and State University.

Li, Y., Makar, S., and Mitra, S. (2008). CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns. In *Conference on Design, Automation and Test in Europe - DATE 2008*, pages 885–890. ACM.

Li, Y., Mutlu, O., Gardner, D. S., and Mitra, S. (2010). Concurrent Autonomous Self-Test for Uncore Components in Systems-on-Chips. In *28th VLSI Test Symposium - VTS 2010*, pages 232–237. IEEE Computer Society.

Mazumder, P. and Rudnick, E. (1998). *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Prentice Hall.

Mitra, S. (2004). X-compact: an efficient response compaction technique. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):421–432. IEEE.

Miyase, K. and Kajihara, S. (2004). XID: Don't care identification of test patterns for combinational circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):321–326. IEEE.

Pomeranz, I., Reddy, L. N., and Reddy, S. M. (1993). COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1040–1049. IEEE.

Pomeranz, I. and Reddy, S. M. (2006). Reducing the number of specified values per test vector by increasing the test set size. *Computers & Digital Techniques*, 153(1):39–46. IEE.

Raedtke, S., Bargfrede, J., and Anheier, W. (1995). Distributed Automatic Test Pattem Generation with a Parallel FAN Algorithm. In *International Conference on Computer Design: VLSI in Computers and Processors - ICCD 1995*, pages 698–702. IEEE.

Reed, I. S. (1973). Boolean Difference Calculus and Fault Finding. *Journal on Applied Mathematics*, 24(1):124–143. Society for Industrial and Applied Mathematics - SIAM.

Rudnick, E. M. and Patel, J. H. (1999). Efficient Techniques for Dynamic Test Sequence Compaction. *Transactions on Computers*, 48(3):323–330. IEEE.

Schulz, M. H., Trischler, E., and Sarfert, T. M. (1988). SOCRATES: A Highly Afficient Automatic Test Pattern Generation System. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1):126–137. IEEE.

Sharma, R. and Saluja, K. K. (1988). An Implementation and Analysis of a Concurrent Built-In Self-Test Technique. In *18th International Symposium on Fault-Tolerant Computing - FTCS 18*, pages 164–169.

Voyiatzis, I., Gizopoulos, D., and Paschalis, A. (2009). An Input Vector Monitoring Concurrent BIST Scheme Exploiting X Values. In *15th On-Line Testing Symposium - IOLTS 2009*, pages 206–207. IEEE.

Wang, L.-T., Cheng, K.-T., and Chang, Y.-W., editors (2009). *Electronic Design Automation: Synthesis, Verification, and Test*. Systems on Silicon. Morgan Kaufmann.

Wang, L.-T., Wu, C.-W., and Wen, X. (2006). *VLSI Test Principles and Architectures: Design for Testability*. Systems on Silicon. Morgan Kaufman.

Wikipedia (2011). Searched words: standard score, standard deviation, normalization, norm, unit vector, variance, Mahalanobis distance, design for test, scan chain, NAND logic.

Williams, M. J. Y. and Angell, J. B. (1973). Enhancing Testability of Large-Scale Integrated Circuits via Test Points and additional Logic. *Transactions on Computers*, 22(1):46–60. IEEE Computer Society.

# 6

# Implementation of a self-repairing unit

A method for making a complex system self-repairing is first to partition that system into simple and independent units which can be implemented as self-repairing units. This chapter presents the architecture, design, simulation and implementation of a self-repairing unit at the Register Transfer Level, in short RTL. The register transfer level of abstraction is used in Hardware Description Languages, in short HDL, for creating circuits at a higher level than the transistor or gate levels. Hardware description languages describe the flow of signals between registers by declaring the registers as variables and describing the flow of signals by using constructs like if-then-else and arithmetic operators. Foremost in this chapter, the architecture of a self-repairing unit gives an overview of the arrangement of its modules, which are implemented as VHDL modules. Since all those modules, excluding the unit itself, look the same for all the units, they all can be taken as templates, in the form of a framework, for the design of any self-repairing unit. The module responsible for the recovery procedure considers redundancy and partial reconfiguration, a feature available by some Field Programmable Gate Arrays, in short FPGAs. For testing the self-repairing unit, a fault injector is considered early in the architecture of the self-repairing unit. A fault injector is helpful for testing the self-repairing unit by the simulation of the design or by its hardware implementation.

Transient faults caused by radiation in SRAM based FPGAs can be detected and repaired by using triple modular redundancy as presented in [Kastensmidt et al., 2006] or by using coding techniques that avoid forbidden state changes in state machines as shown in [Burke and Taft, 2004]. This chapter intends to give a framework for a versatile repairing of permanent faults using active redundancy. This framework uses partial reconfiguration for repairing purposes, although partial reconfiguration can also be used for fitting a large design into a single FPGA by means of temporal partitioning and temporal placement of submodules into the used FPGA, for more information please refer [Purna and Bhatia, 1999], [Christoph Steiger et al., 2003], [Dittmann, 2008] and [Montealegre and Rammig, 2010].

## 6.1 Design of the self-repairing unit

The self-repairing unit can be designed in a modular way describing each module in a separate VHDL file. Each VHDL file has defined the inputs and outputs of the module in the **entity** part and its functionality in the **architecture** part. Each module can contain submodules defining them as **component** in its architecture part before the reserved word **begin**, and defining their signals connection after the reserved word begin.

The next subsection shows the modules required for a self-repairing unit, their signal connections and the functionality of each module. The subsection thereafter enhances the design inserting some modules for fault injection, required for testing the self-repairing unit at design time by its simulation or its hardware implementation. The last subsection shows the modules required for the recovery of the unit by means of partial reconfiguration, if the unit is intended to be implemented in a FPGA. All VHDL modules are explained by means of their VHDL code, pointing out the logic behind the most important lines. It is recommended only to look at the logic of the mentioned code lines and do not stuck reading the whole code. For understanding the format of the VHDL constructs please refer a VHDL reference manual, e.g., [Zwolinski, 2003], [Chu, 2006], [Brown and Vranesic, 2005] or [Pellerin and Taylor, 1996].

### 6.1.1 Initial architecture of the self-repairing unit

In this subsection, the initial architecture for the design of the self-repairing unit is explained in detail, excluding the partial reconfiguration as recovery mean and the testing of the self-repairing unit. Those features will be explained separately in the subsequent subsections.

First of all, the initial architecture of the self-repairing unit is shown in figure 6.1. There, the inputs to the self-repairing unit are: the inputs capable to be disconnected from the unit itself declared as *InputsUser*; the signal *Clock* which allows to synchronize the activity of all the modules in the self-repairing unit; and the signal *Reset* which allows to bring the self-repairing unit to a known initial state. The outputs to the self-repairing unit are: the outputs capable to be disconnected from the unit itself declared as *OutputsUser*; the signal *Ready* that indicates, that the module that executes fault recognition named FaultRecognition is neither searching for faults in the unit, nor the module that executes the recovery procedure RecoveryProcedure is repairing the unit; and the signal *Defect* that indicates that the unit has been encountered faulty and repaired without success. The unit is the module labeled as CircuitForSR. The modules responsible for enabling the inputs and outputs of the unit are EnableInputs and EnableOutputs, which are controlled by the state machine module StateMachine. That module behaves according to signals coming from the modules that execute fault recognition and recovery. Fault recognition is executed using data stored in the memory module SyncMemory. When the recovery of the unit has not been successful, it is attempted to be recovered again a determined number of times registered by the module RecoveryCounter, before giving up by raising up the signal *Defect*. The function of all those enunciated modules are explained in detail below.

#### Circuit for self-repairing module

The unit named as circuit for self-repairing can be described at the register transfer level as a VHDL module. That module has been labeled as CircuitForSR and is shown in code listing 6.1. In the **entity** part of that module, firstly the inputs and outputs of the circuit for

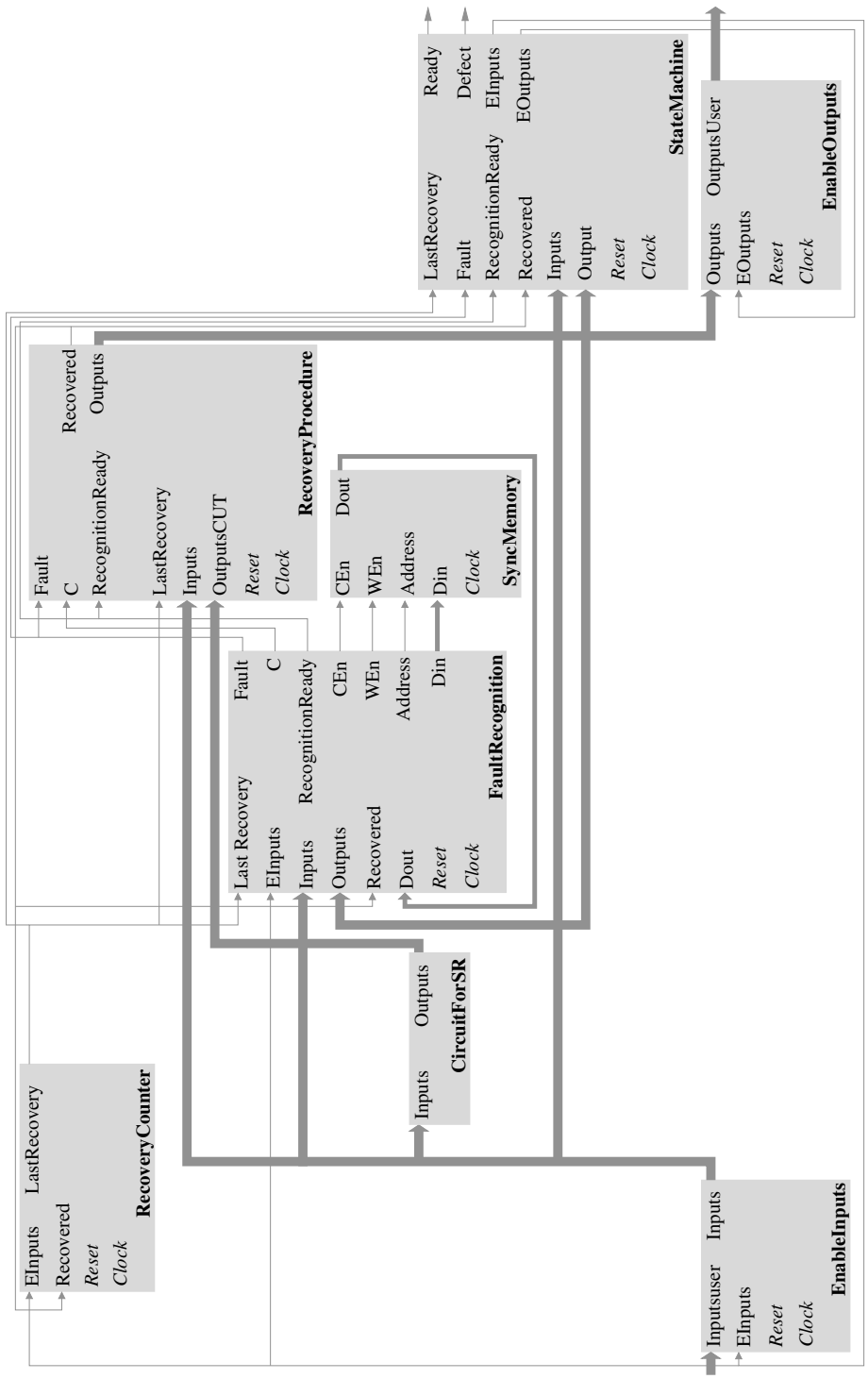



Figure 6.1: Initial architecture of the self-repairing unit

self-repairing can be declared as two vectors of Boolean signals of type `std_logic_vector`. That is a standard logic type defined in the **package** `ieee.std_logic_1164.all` referenced in code line 2, for more details please see [Zwolinski, 2003]. Any definition in a VHDL package can be used referencing the package in the VHDL module with the reserved word **use**, please see code lines 2 and 3. In order to define only once for all the modules that require them, the constants in code lines 7 and 8, number of inputs and number of outputs of the circuit for self-repairing, labeled as size_inputs and size_outputs respectively, they are defined in the user-defined package Constants explained below in their respective subsection. The **architecture** part with the name of Behavioral contains the hardware description of the unit. An example is a tiny combinational circuit with 3 inputs, 1 output, and Boolean formula $(\overline{C} \cdot B) + (C \cdot A)$.

Program Code 6.1: Circuit for self-repairing module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity CircuitForSR is
6      port (
7          Inputs:  in   std_logic_vector(size_inputs-1 downto 0);
8          Outputs: out  std_logic_vector(size_outputs-1 downto 0)
9      );
10 end CircuitForSR;
11
12 architecture Behavioral of CircuitForSR is
13 begin
14
15     Outputs(0) <= ( (not Inputs(2)) and  Inputs(1) ) or\
16                   ( Inputs(2) and Inputs(0) );
17
18 end Behavioral;
```

### Enable inputs module

This module is responsible for connecting the external inputs, labeled as *InputsUser*, to the inputs of the circuit for self-repairing, labeled as *Inputs*. This module is necessary for isolating the unit when the recovery procedure is being executed, or when the unit requires being verified for each *Inputs*/*Outputs* before connecting the *Outputs* to the *OutputsUser*. That is required for hard critical systems when an unverified wrong output can have dramatic consequences. The enable inputs module is implemented by means of a **process**, labeled as EI_PROC. A process allows to describe a circuit by its behavior. A process is evaluated when any of the signals in its sensitivity list changes. So, the process EI_PROC is evaluated when the *Clock* signal changes. Thanks to an **if-then** statement, the process EI_PROC is active only at the rising edge of the clock signal connecting the *InputsUser* signals to the *Inputs* signals if the *EInputs* signal is switched to '1', please see code line 25. In addition, if the *Reset* signal is ever switched to '1', the *InputsUser* signals are also connected to the *User* signals, as shown in code line 22.

Program Code 6.2: Enable inputs module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity EnableInputs is
6      port (
7          Clock:      in std_logic;
8          Reset:      in std_logic;
9          EInputs:    in std_logic;
10         InputsUser: in std_logic_vector(size_inputs-1 downto 0);
11         Inputs:     out std_logic_vector(size_inputs-1 downto 0)
12     );
13 end EnableInputs;
14
15 architecture Behavioral of EnableInputs is
16 begin
17
18     EI_PROC: process (Clock)
19     begin
20         if (Clock'event and Clock = '1') then
21             if Reset = '1' then
22                 Inputs <= InputsUser;
23             else
24                 if (EInputs = '1') then
25                     Inputs <= InputsUser;
26                 end if;
27             end if;
28         end if;
29     end process;
30
31 end Behavioral;
```

#### Enable outputs submodule

The enable outputs module is responsible for connecting the outputs of the circuit for self-repairing, labeled as *Outputs*, to the outputs of the self-repairing unit, labeled as *OutputsUser*. This module, the same as the enable inputs module, is necessary for isolating the unit when the recovery procedure is being executed, or when the circuit for self-repairing requires being verified for each *Inputs*/*Outputs* before connecting the *Outputs* to the *OutputsUser*. The enable outputs module is implemented by means of a **process**, labeled as EO_PROC. The process is active only at the rising edge of the clock signal and connects the *Outputs* signals to the *OutputsUser* signals when the *EOutputs* signal is switched to '1', please see code line 25. If the *Reset* signal is ever switched to '1', the *OutputsUser* signals are left floating, i.e., the signals are neither driven to a logical high '1' nor to a low level '0', they instead present a high impedance which is described in VHDL with the symbol 'Z' as shown in code line 22.

Program Code 6.3: Enable outputs module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity EnableOutputs is
6      port (
7          Clock:       in std_logic;
8          Reset:       in std_logic;
9          EOutputs:    in std_logic;
10         Outputs:     in std_logic_vector(size_outputs-1 downto 0);
11         OutputsUser:out std_logic_vector(size_outputs-1 downto 0)
12     );
13 end EnableOutputs;
14
15 architecture Behavioral of EnableOutputs is
16 begin
17
18     EO_PROC: process (Clock)
19     begin
20         if (Clock'event and Clock = '1') then
21             if Reset = '1' then
22                 OutputsUser <= (others => 'Z');
23             else
24                 if (EOutputs = '1') then
25                     OutputsUser <= Outputs;
26                 end if;
27             end if;
28         end if;
29     end process;
30
31 end Behavioral;
```

### State machine module

The state machine module StateMachine controls the enable inputs and enable outputs modules, through their output signals *EInputs* and *EOutputs*. In addition, it is responsible for the signals *Ready* and *Defect*. The signal *Ready* indicates that the self-repairing unit has no pending fault recognition or recovery tasks. The signal *Defect* indicates that the circuit for self-repairing presents a failure that is unrecoverable. The state machine module receives as input signals from the fault recognition module FaultRecognition the signals *RecognitionReady* and *Fault*, from the recovery procedure module RecoveryProcedure the signal *Recovered*, and from the recovery counter module RecoveryCounter the signal *LastRecovery*, as can be seen in figure 6.1.

Chapter 5 has shown that in case of having a circuit for self-repairing with binary inputs and outputs, it is feasible to implement a fault recognition unit that operates concurrently to the unit. However, for a circuit for self-repairing with real inputs and outputs, a fault recognition unit that operates concurrently is possible only when the fault recognition is fast enough to be ready before the next inputs/outputs are present. In case the fault recognition is not fast enough, the fault recognition unit can miss some inputs/outputs and be only

able to recognize faults in the circuit for self-repairing by just picking the inputs/outputs that it can. Another alternative is an almost-concurrent fault recognition, helpful when the unit requires being verified for each *Inputs*/*Outputs* before connecting the *Outputs* to the *OutputsUser*. That is required for hard critical systems when an unverified wrong output can have dramatic consequences. The state machine for a concurrent fault recognition is shown in figure 6.2 and the state machine for an almost-concurrent fault recognition is shown in figure 6.3.

The state machine for a concurrent fault recognition is by default in the state st_FR. In that state, the signals *EInputs*, *EOutputs* and *Ready* are set to '1' and the signal *Defect* is set to '0'. Under a fault in the circuit for self-repairing, indicated by the input signal *Fault*, and whenever it has not been exceeded the number of possible recoveries, indicated by the signal *LastRecovery* set to '0', the state machine enters into the st_RP state. In that state, the circuit for self-repairing is isolated by disconnecting its *Inputs* and *Outputs* from the external *InputsUser* and *OutputsUser* setting the signals *EInputs* and *EOutputs* to '0'. When the circuit for self-repairing has been recovered, the state machine is informed by the signal *Recovered* and it enters again into the default state sf_FR, where it is verified if the recovered circuit is fault-free. When that is not the case, the state machine enters the st_RP state again. If the repeated recovery procedures have not succeeded in repairing the unit, the state machine enters the st_Defect state. In that state, the *Inputs* and *Outputs* of the circuit are disconnected form the *InputsUser* and *OutputsUser*, and the *Defect* and *Ready* signals are set to '1'.

Unlike the state machine for concurrent fault recognition, the state machine for an almost-concurrent fault recognition has the signals *EInputs*, *EOutputs* and *Ready* set to '0'. Besides, there is a new signal *RecognitionReady* coming for the fault recognition module FaultRecognition. When the fault recognition process is finished and no fault is encountered, the state machine changes state from sf_FR to a new state st_EO, where the *Outputs* of the circuit for self-repairing are connected to the *OutputsUser*. Just after that, the state machine changes to another new state st_EI, where the *InputsUser* are connected to the *Inputs* of the circuit for self-repairing. The state st_EI is the default state, where the signals *Ready* and *EInputs* are set to '1'. From that state, the state machine changes to state st_FR when the *Inputs* or *Outputs* of the circuit for self-repairing change or the *Reset* signal is set to '1'.

The state machine for an almost-concurrent fault recognition is described in VHDL by means of three processes, which has been labeled as SYNC_PROC, NEXT_STATE_DECODE and OUTPUT_DECODE, as show in code lines 34, 59 and 95. The process SYNC_PROC describes the registers in the state machine, that is why that process is evaluated at the rising edge of the *Clock* signal. The NEXT_STATE_DECODE and OUTPUT_DECODE describe the combinational logic named in Mealy and Moore machines as the next state logic and output logic blocks, [Zwolinski, 2003]. The process OUTPUT_DECODE describes the value of the outputs of the state machine for each state under a change only of the state, not of the input signals. Therefore, this state machine is a Moore machine. The process NEXT_STATE_DECODE describes the logic behind the state transitions, coded as next_state <= st_XX, under a change at the inputs of the state machine *LastRecovery*, *Fault*, *RecognitionReady*, *Recovered*, *Inputs* and *Outputs* or the auxiliary signals *Reseted*, *Inputs_i* and *Outputs_i*, which has been declared for saving any change in the respective input signal. If the *Reset* signal is ever set to '1', the output of the state registers are set by default to the state st_EI, the output signals *EInputs*, *EOutputs*, *Ready* and *Defect* are set to the default values that correspond to the default state st_EI, the internal signal *Reseted* is set to '1' and the internal signals *Inputs_i* and *Outputs_i*

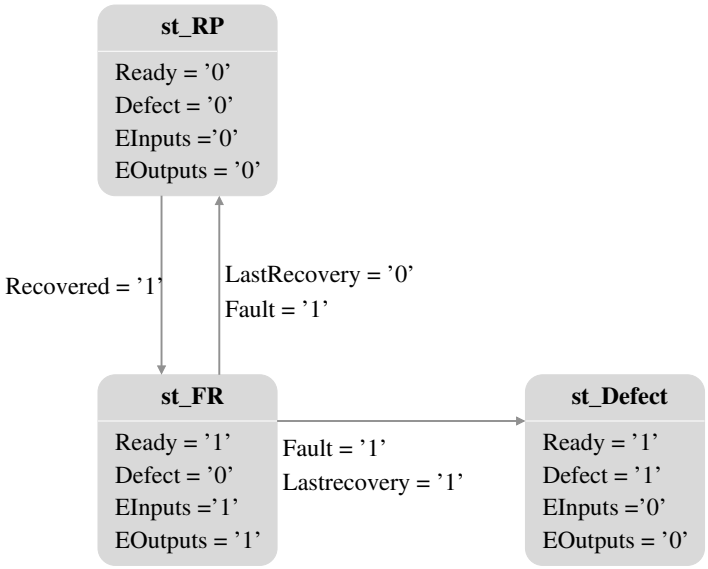



Figure 6.2: State machine for concurrent fault recognition

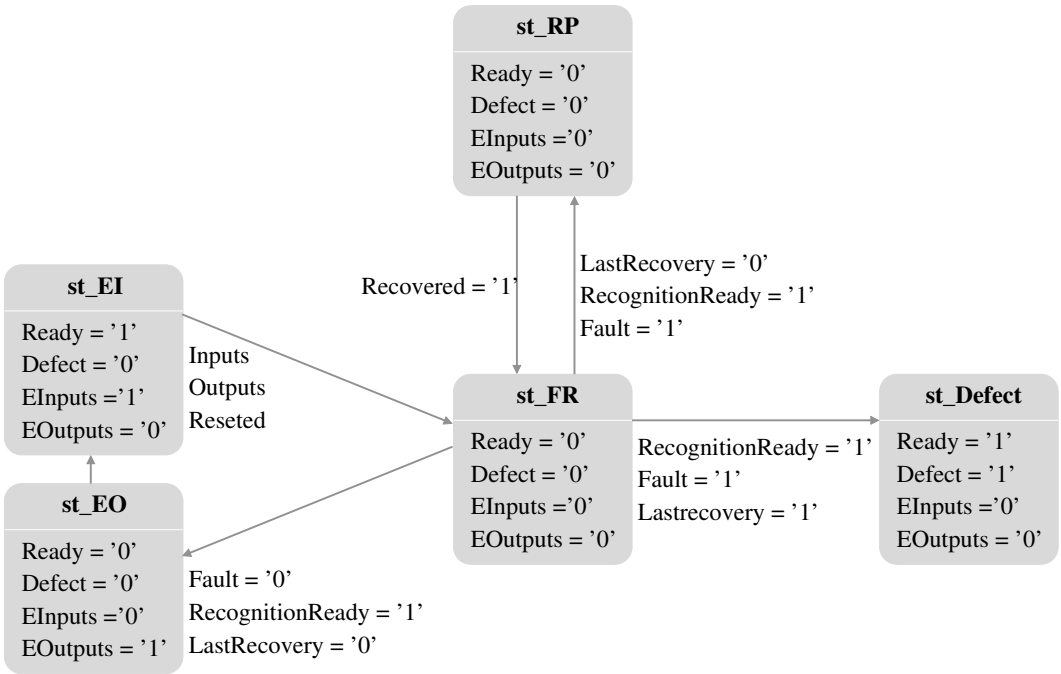



Figure 6.3: State machine for an almost-concurrent fault recognition

are set to the actual values of *Inputs* and *Outputs*. The use of internal signals for the outputs *EInputs_i*, *EOutputs_i*, *Ready_i* and *Defect_i* is based in the look-ahead output buffers scheme for the Moore outputs which guarantees glitch-free output signals and eliminates the propagation delay that the output logic introduces, for more details please refer [Chu, 2006]. All the internal signals and the states has been declared at the head of the architecture part of the module, before the reserved word begin. In the process NEXT_STATE_DECODE, the statement **case** contains a **when others** statement in order that the state machine goes to the know state st_Defect in case an invalid state is given. An invalid state is possible when the number of states is less than the maximum number of possibilities possible with the number of bits taken by the states encoding method. Since each such bit represents a flip-flop by the implementation, and a flip-flop is prone to have a bit-flip at it s output due to environmental noise, the occurrence of an invalid state is possible. The **case** statement in the process OUTPUT_DECODE contains also a **when others** statement in order to have a safe state when the flip-flop for the implementation of the case statement has a bit-flip.

Program Code 6.4: State machine module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity StateMachine is
6      Port (
7          Clock:          in std_logic;
8          Reset:          in std_logic;
9          Inputs:         in std_logic_vector(size_inputs-1 downto 0);
10         Outputs:        in std_logic_vector(size_outputs-1 downto 0);
11         Fault:          in std_logic;
12         LastRecovery:   in std_logic;
13         RecognitionReady:in std_logic;
14         Recovered:      in std_logic;
15         Ready:          out std_logic;
16         Defect:         out std_logic;
17         EInputs:        out std_logic;
18         EOutputs:       out std_logic
19     );
20 end StateMachine;
21
22 architecture Behavioral of StateMachine is
23     type state_type is (st_EI, st_EO, st_FR, st_RP, st_Defect);
24     signal state, next_state: state_type;
25     signal Ready_i: std_logic;
26     signal Defect_i: std_logic;
27     signal EInputs_i: std_logic;
28     signal EOutputs_i: std_logic;
29     signal Inputs_i: std_logic_vector(size_inputs-1 downto 0);
30     signal Outputs_i: std_logic_vector(size_outputs- 1 downto 0);
31     signal Reseted: std_logic;
32 begin
33
34     SYNC_PROC: process (Clock)
```

```
35      begin
36          if (Clock'event and Clock = '1') then
37              if Reset = '1' then
38                  state <= st_EI;
39                  Defect <= '0';
40                  Ready <= '1';
41                  EInputs <= '1';
42                  EOutputs <= '0';
43                  Inputs_i <= Inputs;
44                  Outputs_i <= Outputs;
45                  Reseted <= '1';
46              else
47                  state <= next_state;
48                  Ready <= Ready_i;
49                  Defect <= Defect_i;
50                  EInputs <= EInputs_i;
51                  EOutputs <= EOutputs_i;
52                  Inputs_i <= Inputs;
53                  Outputs_i <= Outputs;
54                  Reseted <= '0';
55              end if;
56          end if;
57      end process;
58
59      OUTPUT_DECODE: process (state)
60      begin
61          case (state) is
62              when st_EI =>
63                  Defect_i <= '0';
64                  Ready_i <= '1';
65                  EInputs_i <= '1';
66                  EOutputs_i <= '0';
67              when st_EO =>
68                  Defect_i <= '0';
69                  Ready_i <= '0';
70                  EInputs_i <= '0';
71                  EOutputs_i <= '1';
72              when st_FR =>
73                  Defect_i <= '0';
74                  Ready_i <= '0';
75                  EInputs_i <= '0';
76                  EOutputs_i <= '0';
77              when st_RP =>
78                  Defect_i <= '0';
79                  Ready_i <= '0';
80                  EInputs_i <= '0';
81                  EOutputs_i <= '0';
82              when st_Defect =>
83                  Defect_i <= '1';
84                  Ready_i <= '1';
85                  EInputs_i <= '0';
86                  EOutputs_i <= '0';
87              when others =>
```

```
88                  Defect_i <= '0';
89                  Ready_i <= '0';
90                  EInputs_i <= '0';
91                  EOutputs_i <= '0';
92          end case;
93      end process;
94
95      NEXT_STATE_DECODE: process (state, Fault, LastRecovery,\
96                                  RecognitionReady, Recovered,\
97                                  Inputs, Inputs_i, Reseted,\
98                                  Outputs, Outputs_i)
99      begin
100         case (state) is
101             when st_EI =>
102                 if ( (Inputs /= Inputs_i) or (Reseted = '1') or\
103                      ((Inputs = Inputs_i) and (Outputs /= Outputs_i))\
104                    )then
105                     next_state <= st_FR;
106                 else
107                     next_state <= st_EI;
108                 end if;
109             when st_EO =>
110                 next_state <= st_EI;
111             when st_FR =>
112                 if RecognitionReady = '1' then
113                     if Fault = '1' then
114                         if LastRecovery = '1' then
115                             next_state <= st_Defect;
116                         else
117                             next_state <= st_RP;
118                         end if;
119                     else
120                         next_state <= st_EO;
121                     end if;
122                 else
123                     next_state <= st_FR;
124                 end if;
125             when st_RP =>
126                 if Recovered = '1' then
127                     next_state <= st_FR;
128                 else
129                     next_state <= st_RP;
130                 end if;
131             when others =>
132                 next_state <= st_Defect;
133         end case;
134     end process;
135
136 end Behavioral;
```

### Fault recognition module

The fault recognition module for a circuit for self-repairing with input and output vectors with binary elements has been exposed graphically at the gate-level in section 5.2. As shown in figure 6.4, in its more general form it consists of: an input vectors monitoring block, an output pattern vectors storage block, an output vectors comparison block, and an output vector compactor. The exposed blocks can be implemented in VHDL using concurrent signal assignment statements, used in a data flow VHDL description, and logical operators, please see [Zwolinski, 2003]. In figure 6.4 there is an additional block named as repairing mechanism assignation block, which is a block dependent on the circuit for self-repairing and on the types of failures it could present.

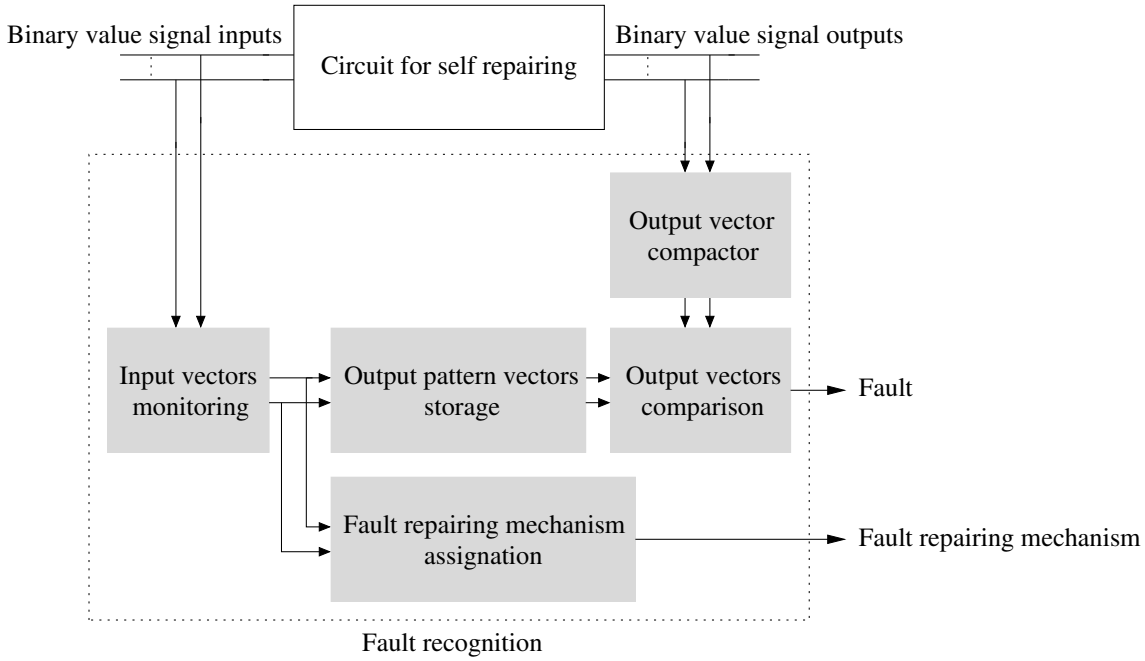



Figure 6.4: A fault recognition module for a unit with binary inputs and outputs

Differently, the fault recognition module for a circuit for self-repairing with input and output vectors with real elements, as shown in figure 6.5, consists of: a dimension reduction block, a vector distances measurement block, a class assignation block and a memory block. The dimension reduction block reduces the dimensions of the fault vector formed with the inputs and the outputs coming from the circuit for self-repairing. By a circuit for fault-repairing with binary inputs and outputs, the inputs from the circuit for self-repairing were monitored by the inputs monitoring block, therefore only the outputs could be compacted through the output vector compactor block. Since an exact comparison of vectors with real elements is not possible, the distances of the fault vector coming from the circuit for self-repairing with each of the fault pattern vectors stored in memory is computed by any of the distance measurement methods presented in chapters 4 and 5. Those distances are used for determining to which class the fault vector coming from the circuit for self-repairing belongs to. For that, it should be planned that one of the classes represents no fault and the remaining classes represent different kinds of faults. If a kind of fault is assigned with a fault repairing mechanism,

the class information gives indirectly the fault repairing mechanism that the recovery procedure should execute for recovering the circuit for self-repairing, as explained in section 4.3. The memory block contains a minimal set of fault pattern vectors with reduced dimensions obtained by evaluating the different distance measurement and class assignation methods, as done in chapter 5. The same distance measurement and class assignation methods, used for finding the minimal fault pattern vector set with reduced dimensions, is recommended to be used in the implementation of the vector distances measurement block and the class assignation block.

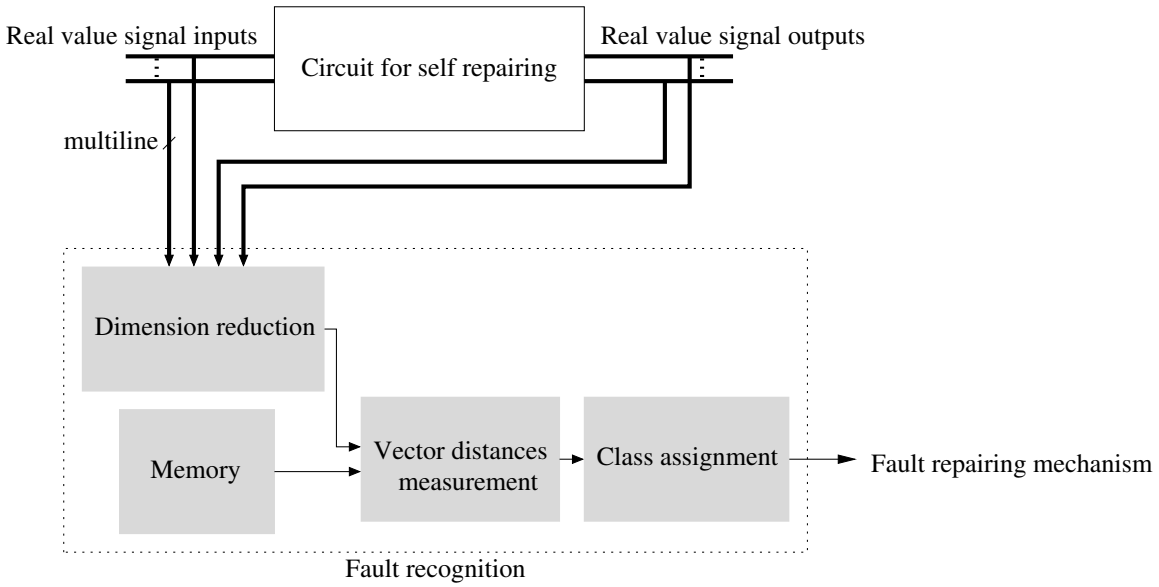



Figure 6.5: A fault recognition module for a unit with real inputs and outputs

The dimensions reduction block requires some data stored in memory. If the block is implemented using the principal component analysis or the singular value decomposition methods, the linear transformation weight matrix $\mathbf{W}_{n\times t}$ should be computed and stored in memory. So, the dimensions reduction block can perform the linear transformation $R_{t\times 1} = \mathbf{W}^T_{n\times t} \times X_{n\times 1}$, where the vector $X_{n\times 1}$ is reduced into the vector $R_{t\times 1}$. For the computation of the matrix $\mathbf{W}_{n\times t}$, please see sections 4.6.1 and 4.6.2. If the block is implemented using the formal immune networks method, $t$ singular values $s_i$ and $t$ right singular vectors $V_i$, obtained by the singular value decomposition of the matrix formed with the set of fault pattern vectors, should be stored in the memory block. So, the dimensions reduction block can compute the elements of the fault vector with reduced dimensions by the formula $r_i = \frac{1}{s_i} X^T V_i$ with $i = 1, ..., t$. For more details please see section 4.8.

The dimension reduction block requires vector and matrix multiplications. The vector distances measurement and class assignation blocks require arithmetic operations such as subtraction, addition, division, multiplication, n-root, n-power, absolute value, etc. The implementation of those operations requires a processing unit which can be embedded in the same chip as the circuit for self-repairing, or placed as an external chip, as proposed in [Montealegre and Rammig, 2008].

Some experiments have been made for implementing all blocks of the fault recognition module, for a circuit for self-repairing with input and output vectors with real elements, as

an application specific circuit. The objective was to have the fault recognition module in the same FPGA as the circuit for self-repairing. The results have been reported in [Dibaj, 2010]. For that, both tool chains were used and compared: Matlab-Simulink-System Generator-ISE and Matlab-AccelDSP-ISE.

Regarding the first tool chain Matlab-Simulink-System Generator-ISE, the fault recognition module has been implemented as a Simulink model using the Xilinx Blockset, available in the System Generator tool. From that Simulink model with Xilinx blocks, it is possible to generate VHDL code automatically. That VHDL code can be synthesized for programming a Xilinx FPGA supported by the chain of tools. One of the available Xilinx blocks is the MCode block. The MCode block allows to embed a Matlab function into a Simulink model, making the inputs and outputs of the function, the inputs and outputs of the MCode block. For example, it is possible to implement the fault recognition module as a Matlab function in the form **function** [ classFaultVector ] = faultrecognitionmanhattan(V,diagS,vcr,mRr,FaultVector), and then embed that function into an MCode block. The output of that function faultrecognitionmanhattan is the class of the given fault vector FaultVector formed with the inputs and outputs of the circuit for self-repairing. The inputs of the function faultrecognitionmanhattan constitute the data necessary for reducing the dimensions of the fault vector, which in this example are the matrix of right singular vectors V and the vector of singular values diagS; the data necessary for computing the vector of distances, which in this example is the matrix with a reduced number of fault pattern vectors with reduced dimensions mRr; and the data for class assignation, which in this example is the vector vcr with the classes of the fault pattern vectors with reduced dimensions of matrix mRr. That data can be accessed from the Matlab workspace through the interface of the MCode block, which is a window where the corresponding variable names should be entered. That is practical, since the required data has also been computed in Matlab, please see code listings 5.9 and 5.16. In addition, that data stored into xl_state variables, which is a Xilinx data type, implements the memory block drawn in figure 6.5. Internal variables in the function have to be also declared as xl_state variables and be defined as persistent for being remembered during simulation and implementation, i.e., **persistent** singularvaluesvar . xl_state variables can have the following types: Boolean, signed and unsigned. As we need variables with real numbers, most xl_state variables were defined as 32 bit signed numbers with 20 bits for the fractional part, i.e., singularvaluesvar = **xl_state**( diagS , {**xlSigned** , 32 , 20} );. However, the MCode block supports a reduced number of Matlab constructs and functions. For example, the MCode block can operate with scalar numbers and vectors, but not with matrices. Nevertheless, one trick is to execute operations between matrices through vector operations.

The fault recognition function for being embedded in the MCode has been programmed using the formal immune network as dimension reduction method, all three Chebyshev, Euclidean and Manhattan vector distances measurement methods, and the nearest neighbor class assignation method. In the case of the function using the Euclidean distance measurement method, it has been necessary to use a Cordic Xilinx module for implementing the root square of a number, since the Matlab implementation of that function is not supported by the MCode block. A Cordic block is a Coordinate Rotation Digital Computer based on the phase shifting method. That method uses addition and bit shifting operations instead of the more resource and time consuming hardware multipliers.

Only the fault recognition module has been implemented and tested in [Dibaj, 2010]. The Simulink models using the Manhattan and Chebyshev vector distances measurement methods, has been synthesized and implemented into a Spartan-3 XC3S200 FPGA kit using the ISE

design software from Xilinx. Those circuits required about 1000 slices and 2000 look-up tables, which made 50% of the available resources of that FPGA. The Simulink model using the Euclidean vector distances measurement method had to be implemented in a bigger FPGA, the Virtex-4 inside the ML403 development board, were it required about 6000 slices and 11000 look-up tables. Furthermore, it required all the available DSPs of that FPGA. The circuit that used the Manhattan distance measurement method presented the best runtime and resources consumption. For further details please take a look at the main source [Dibaj, 2010].

A similar implementation has been made with the tools chain Matlab-AccelDSP-ISE. However, the resource utilization has been even higher. That is because, the Matlab programs or Simulink models are first compiled into C and then into VHDL. Therefore those tools are not able to deliver the most optimal VHDL code. That is the reason why VHDL has been selected for designing the prototype of the self-repairing unit, which is going to be described below in this section. However, when implementing systems at the model level in Simulink, or when algorithms are programmed in Matlab, and when the available FPGA is big enough, the use of those tool chains is a good alternative.

A VHDL implementation of matrix multiplications and arithmetic operations with real numbers expressed in fixed or floating point format for the implementation of the fault recognition module is a complex task. That task can be managed using the Xilinx CORE Generator tool which is available in the Xilinx ISE Design Suite. That tool generates parametrized Intellectual Property cores, in short IP, optimized for Xilinx FPGAs. With that tool it is possible to generate specific cores for addition, subtraction, multiplication, etc. The implementation of the fault recognition module using those cores could be an alternative that can profit of parallelism but it is not a solution for an efficient design, as could be experimented in the implementation of a singular value decomposition module in an FPGA using those modules, please refer to [Hosseinimehr, 2010].

For this first prototype of a self-repairing unit, conceived as a VHDL framework for small circuits, a sequential VHDL description of the fault recognition module instead of a data flow VHDL description has been selected. That, since the other modules have also a sequential VHDL description due to the use of a clock for synchronizing the whole self-repairing unit. For simplifying, a circuit for self-repairing with input and output vectors with binary elements is considered. Then the VHDL code of the fault repairing module considers the blocks shown in figure 6.4 including the fault repairing mechanism assignation, but excluding the output vector compactor, because the used example of circuit for self-repairing has only one output.

The sequential VHDL description of the fault recognition module presented in code listing 6.5 contains only one process labeled as READ_MATCH, as can be seen in code line 30. The fault recognition module is responsible of determining whether the present input pattern at the signal *Inputs* corresponds to the output pattern at the signal *Outputs* coming from the circuit for self-repairing. That happens when the state machine changes from the state st_EI to the state st_FR, under any change in the signals *Inputs* or *Outputs* coming from the circuit for self-repairing, or a *Reset* signal set to '1', as shown in figure 6.3 and implemented in the code line 102 of code listing 6.4. The fault recognition module is triggered also when the state machine changes from the state st_RP to the state st_FR, under a rising edge of the *Recovered* signal, as shown in figure 6.3 and implemented in code line 126 of code listing 6.4. Since in the st_FR state, the state machine sets the value of the signal *EInputs* to '0', then, the fault recognition module starts working under a falling edge of the signal *EInputs* or a rising edge of the signal *Recovered*, if and only if the signal *LastRecovery* has the value of '0', as can be

seen in code line 50 of code listing 6.5. When those conditions are met, the process variable *Initialized* is set to '1'. A process variable can be used only inside a process and can not be seen by other process, it is therefore a local variable. In order to detect the falling and rising edges of the signals *EInputs* and *Recovered*, the internal signals *EInputsold* and *Recoveredold* are defined and then updated in code lines 103 and 104 of code listing 6.5. Those signals allow to compare the present value from the value in the last falling edge *Clock*. Please note that the process READ_MATCH is sensitive to the *Clock* signal in the falling edge of the *Clock* signal, unlike the SYNC_PROC process of the state machine, as shown in code line 41. On the next falling edge of the signal *Clock*, it is searched for a vector stored in memory with the format [self|recovery_method|inputs|outputs] that has in the field inputs the same pattern as the signal *Inputs*, please see code lines from code line 58 forwards. If a vector with such an input pattern is found, the stored outputs are compared with the signal *Outputs*, if a fault is detected, the corresponding recovery method is copied into the variable *C_i*. The field of the fault pattern vector stored in memory labeled as self, flags whether the stored outputs are the expected outputs or an output pattern representing a determined fault that should exclusively be warned. If the Inputs signal is not equal to the variable *ICompare*, where the value has been copied in the inputs field of the fault pattern vector, in the next falling edge clock, another fault pattern vector is read from the memory increasing the memory address, as shown in code line 96. If no fault pattern vector with an input field equal to the signal *Inputs* is stored in memory, the *RecognitionReady_i* variable should be anyway set to '1', for informing that the searching is done. However, that feature has not been implemented in this first prototype, since it has been assumed that the complete set of possibilities of input patterns is stored in memory. Additionally, the *Inputs* signal value for which no fault pattern vector was stored in memory, together with the present value at the signal *Outputs*, can be stored in memory using the assignment statement Din_i <= '0' & "000" & Inputs & Outputs and enabling writing into the memory through WEn_i <= '0'. In the first statement, the variable *Din_i* represents the fault pattern vector to be written into memory, and the variable *WEn_i* represents the write enable signal required to be set to '1' for writing into memory. Writing new fault vectors into memory would allow self-learning. At the end, it is necessary to update all the output signals from the fault recognition module, please see code lines from code line 105 forwards. And finally, under a *Reset* signal set to '1', the output signal *Fault* is set to '0', the recovery mechanism signal *C* is set to '0', which means that no repairing should be executed, and the signals *Address*, *WEn* and *CEn* for the memory are set to their default values, please see code lines from code line 42 forwards.

Program Code 6.5: Fault recognition module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity FaultRecognition is
6      Port (
7          Clock:     in std_logic;
8          Reset:     in std_logic;
9          Inputs:    in std_logic_vector(size_inputs-1 downto 0);
10         Outputs:   in std_logic_vector(size_outputs-1 downto 0);
11         Recovered: in std_logic;
12         LastRecovery: in std_logic;
```

```
13          EInputs:  in std_logic;
14          Dout:     in std_logic_vector(size_fp_vector-1 downto 0);
15          Fault:    out std_logic;
16          C:        out std_logic_vector(size_recovery_method-1\
17                                         downto 0);
18          RecognitionReady:out std_logic;
19          Address:  out integer range 0 to (size_memory-1);
20          Din:      out std_logic_vector(size_fp_vector-1 downto 0);
21          WEn, CEn: out std_logic
22      );
23  end FaultRecognition;
24
25  architecture Behavioral of FaultRecognition is
26      signal EInputsold: std_logic;
27      signal Recoveredold: std_logic;
28  begin
29
30      READ_MATCH: process (Clock)
31          variable Fault_i: std_logic :='0';
32          variable C_i: std_logic_vector(size_recovery_method-1\
33                                         downto 0):=(others => '0');
34          variable RecognitionReady_i:  std_logic := '0';
35          variable Address_i: integer range 0 to (size_memory-1) :=0;
36          variable Initialized: std_logic := '0';
37          variable ICompare: std_logic_vector(size_inputs-1 downto 0);
38          variable OCompare: std_logic_vector(size_outputs-1 downto 0);
39          variable Self: std_logic_vector(size_self-1 downto 0);
40      begin
41          if (Clock'event and Clock = '0') then
42              if Reset = '1' then
43                  Fault <= '0';
44                  Address <= 0;
45                  WEn <= '1';
46                  CEn <= '0';
47                  C <= (others => '0');
48                  C_i := (others => '0');
49              else
50                  if ( EInputsold /= EInputs and EInputs = '0' and\
51                        LastRecovery = '0' ) or\
52                        ( Recoveredold /= Recovered and Recovered =\
53                        '1' and LastRecovery = '0' ) then
54                      Initialized := '1';
55                      RecognitionReady_i := '0';
56                      Address_i := 0;
57                      Fault_i := '0';
58                  elsif ( Initialized =\
59                            '1' and Address_i < size_memory ) then
60                      ICompare := Dout(size_fp_vector-1-size_self-\
61                        size_recovery_method downto size_outputs);
62                      if ICompare = Inputs then
63                          OCompare := Dout(size_outputs-1 downto 0);
64                          Self := Dout(size_fp_vector-1 downto\
65                            size_fp_vector-size_self);
```

```
66                          C_i := Dout(size_fp_vector-1-size_self\
67                            downto size_fp_vector-\
68                            size_recovery_method-size_self);
69                          if self = "0" then
70                              if OCompare = Outputs then
71                                  Fault_i := '1';
72                                  C_i := Dout(size_fp_vector-1-\
73                                    size_self downto size_fp_vector-\
74                                    size_recovery_method-size_self);
75                              elsif OCompare /= Outputs then
76                                  Fault_i := '0';
77                                  C_i := (others => '0');
78                              end if;
79                          elsif self = "1" then
80                              if OCompare = Outputs then
81                                  Fault_i := '0';
82                                  C_i := (others => '0');
83                              elsif OCompare /= Outputs then
84                                  Fault_i := '1';
85                                  C_i := Dout(size_fp_vector-1-\
86                                    size_self downto size_fp_vector-\
87                                    size_recovery_method-size_self);
88                              end if;
89                          end if;
90                          RecognitionReady_i := '1';
91                          Initialized := '0';
92                          Address_i := 0;
93                      elsif ICompare /= Inputs then
94                          Fault_i := '0';
95                          if Address_i < (size_memory-1) then
96                              Address_i := Address_i + 1;
97                          end if;
98                      end if;
99                  else
100                     RecognitionReady_i := '0';
101                     Fault_i:= '0';
102                 end if;
103                 EInputsold <= EInputs;
104                 Recoveredold <= Recovered;
105                 Fault <= Fault_i;
106                 C <= C_i;
107                 RecognitionReady <= RecognitionReady_i;
108                 Address <= Address_i;
109                 Din <= (others => '0');
110                 WEn <= '1';
111                 CEn <= '0';
112             end if;
113         end if;
114     end process;
115
116 end Behavioral;
```

### Memory module

The memory module is responsible of the read process of the stored fault pattern vectors and the write process of new fault pattern vectors, when necessary. The size of the memory is declared generally in the **generic** part of the **entity** part of the memory module, please see code lines 7 and 8 of code listing 6.6, using the variables M and N, which will be defined at the time of the instantiation of the memory module using the reserved words **generic map** with the constants `size_memory` and `size_test_vector`, which are defined numerically in the Constants package to be presented in a subsection later. The inputs to the memory module are: *Address*, *Din* for data in, *WEn* for write enable active '0', *CEn* for chip enable active '0', and *Clock*. The output of the memory module is *Dout* for data out. The input and the outputs of the memory module are defined in the **port** part of the memory module. The array of vectors in the memory is defined as a signal labeled as *mem*, of type `ram_array` and initial data `mem_start`, both defined also in the Constants package presented in a subsection later, as can be seen in code line 19. The memory module has been described as a synchronous memory sensitive to the rising edge of the *Clock* signal. A synchronous memory can be implemented allocating part of the FPGA as RAM, please see [Zwolinski, 2003]. The memory module has only one process named as READ_WRITE_MEM, which is sensitive to the *Clock* signal. If the chip enable signal is set to '0', and the write enable signal is set to '1', the memory module writes into their output signal *Dout* the value in the memory at the position given by the signal *Address*, as can be seen in code line 29. If the chip enable signal is set to '0', and the write enable signal is set to '0', the memory module writes the value of the input signal *Din* into the memory array at the position given by the signal *Address*, as can be seen in code line 27.

Program Code 6.6: Memory module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity SyncMemory is
6      generic (
7          M: integer;
8          N: integer
9          );
10     port (
11         Address: in integer range 0 to (M-1);
12         Din: in std_logic_vector(N-1 downto 0);
13         WEn, CEn, Clock: in std_logic;
14         Dout: out std_logic_vector(N-1 downto 0)
15     );
16 end SyncMemory;
17
18 architecture Behavioral of SyncMemory is
19     signal mem: ram_array := mem_start;
20 begin
21
22     READ_WRITE_MEM: process (Clock) is
23     begin
24         if (Clock'event and Clock = '1') then
```

```
25              if CEn = '0' then
26                  if WEn = '0' then
27                      mem(Address) <= Din;
28                  else
29                      Dout <= mem(Address);
30                  end if;
31              end if;
32          end if;
33      end process;
34
35  end Behavioral;
```

### Recovery procedure module

The recovery procedure module is responsible of executing the recovery mechanism under a recognized fault. It supports recovery using redundancy and partial reconfiguration. Applying redundancy means having other versions of the circuit to switch to them whenever a fault is encountered. Partial reconfiguration of the circuit for self-pairing is only possible when the system is implemented in a FPGA and the development platform allows to perform partial reconfiguration. The RecoveryProcedure module starts working when the input signals *RecognitionReady* and *Fault* are set to '1', and the signal *LastRecovery* is set to '0', as shown in code line 50. Its most important input is the signal *C*, which is given by the fault recognition module and contains the recovery procedure to be executed for the recognized fault. When the recovery procedure indicates to change the circuit for self-repairing using partial reconfiguration, the *StartReconfiguration* signal is set to '1'. Then, the module waits for the rising edge of the input signal *Reconfigured*. When the input signal *Reconfigured* is '1', the recovery procedure module sets the output *Recovered* to '1' and resets the output *StartReconfiguration* to '0', as can be seen in code lines 11, 12 and 61 to 64. When the recovery procedure indicates to use redundant circuits, the recovery procedure is copied into the internal signal *C_i*, the signal *StartReconfiguration* is left inactive, and the *Recovered* signal is set to '1'. Initially, when the recovery procedure has the default value '000', the outputs coming from the circuit for self-repairing, labeled as *OutputsCFSR*, are passed through and given in *Outputs*, please see figure 6.6. But when the recovery procedure has the value of '011', which indicates to use the redundant circuit RC1 defined as a **component** in code line 25 and instantiated in code line 37, the outputs *Outputs1* are passed through, as can be seen in the concurrent statement in code line 39. In the same way other redundant circuit can be defined, instantiated and concurrently assigned in this module. For this prototype, under other values of *C*, that means other recovery procedures, the outputs of the same redundant circuit are passed through, as seen in code line 41.

Program Code 6.7: Recovery procedure module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity RecoveryProcedure is
6      port (
7          Clock:          in std_logic;
```

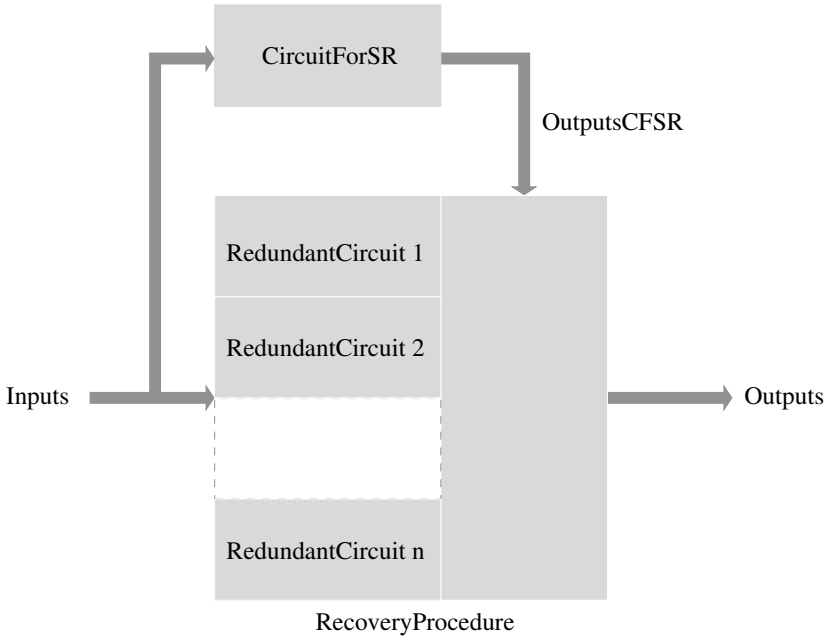



Figure 6.6: Recovery with redundant circuits

```
8           Reset:              in std_logic;
9           C:                  in std_logic_vector(size_recovery_method-\
10                                                  1 downto 0);
11          StartReconfiguration:out std_logic;
12          Reconfigured:       in std_logic;
13          Inputs:             in std_logic_vector(size_inputs-1 downto 0);
14          OutputsCFSR:        in std_logic_vector(size_outputs-1 downto 0);
15          Outputs:            out std_logic_vector(size_outputs-1\
16                                                   downto 0);
17          Fault:              in std_logic;
18          RecognitionReady:   in std_logic;
19          LastRecovery:       in std_logic;
20          Recovered:          out std_logic
21      );
22  end RecoveryProcedure;
23
24  architecture Behavioral of RecoveryProcedure is
25      component RedundantCircuit1 is
26          port (
27              Inputs: in  std_logic_vector(size_inputs-1 downto 0);
28              Outputs: out  std_logic_vector(size_outputs-1 downto 0)
29          );
30      end component;
31      signal Outputs1:std_logic_vector(size_outputs-1 downto 0);
32      signal C_i:std_logic_vector(size_recovery_method-1 downto 0)\
33                                      :="000";
34      signal ReconfiguredOld:std_logic:='0';
35  begin
36
```

```
37      RC1: RedundantCircuit1 port map (Inputs,Outputs1);
38
39      Outputs <=  OutputsCFSR when C_i = "000" else
40                  Outputs1 when C_i = "011" else
41                  Outputs1;
42
43      REC_PROC: process (Clock)
44      begin
45          if (Clock'event and Clock = '1') then
46              if Reset = '1' then
47                  C_i <= (others => '0');
48                  StartReconfiguration <= '0';
49              else
50                  if ( (RecognitionReady = '1') and (Fault = '1') and\
51                        (LastRecovery /= '1') ) then
52                      case (C) is
53                          when "101" =>
54                              StartReconfiguration <= '1';
55                              C_i <= (others => '0');
56                          when others =>
57                              StartReconfiguration <= '0';
58                              C_i <= C;
59                              Recovered <= '1';
60                      end case;
61                  elsif Reconfigured = '1' and\
62                        (Reconfigured /= ReconfiguredOld) then
63                      Recovered <= '1';
64                      StartReconfiguration <= '0';
65                  else
66                      Recovered <= '0';
67                  end if;
68              end if;
69                  ReconfiguredOld <= Reconfigured;
70          end if;
71      end process;
72
73  end Behavioral;
```

### Recovery counter module

The recovery counter module is responsible of counting how many times the circuit has been recovered for the present recognized fault. If the number exceeds the maximum limit given in `max_recoveries` defined in the Constants package, the recovery counter module rises the signal *LastRecovery*. If the number of recoveries does not reach `max_recoveries` for the present fault, for faults recognized later the counter starts counting again. The counter module has been implemented with two processes named as LR_PROC and COUNTER_PROC. In the process COUNTER_PROC, the counter is set to '0' by the rising edge of *EInputs*. Afterwards, in every rising edge of the *Recovered* signal, the internal signal *Counter* is increased by one. If the value in the internal signal *Counter* reaches the `max_recoveries` value in process LR_PROC, the signal *LastRecovery* is raised to '1', as can be seen in code line 25.

Program Code 6.8: Recovery counter module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity RecoveryCounter is
6      port (
7          Clock:        in std_logic;
8          Reset:        in std_logic;
9          EInputs:      in std_logic;
10         Recovered:    in std_logic;
11         LastRecovery:out std_logic);
12 end RecoveryCounter;
13
14 architecture Behavioral of RecoveryCounter is
15     signal Counter: integer;
16 begin
17
18     LR_PROC: process (Clock)
19     begin
20         if (Clock'event and Clock = '0') then
21             if Reset = '1' then
22                 LastRecovery <= '0';
23             else
24                 If Counter >= max_recoveries then
25                     LastRecovery <= '1';
26                 else
27                     LastRecovery <= '0';
28                 end if;
29             end if;
30         end if;
31     end process;
32
33     COUNTER_PROC: process (Recovered, EInputs)
34     begin
35         if EInputs = '1' then
36             Counter <= 0;
37         elsif (Recovered'event and Recovered = '1') then
38             Counter <= Counter + 1;
39         end if;
40     end process;
41
42 end Behavioral;
```

### Constants package

The constants package has been created for gathering in a single place the definitions of constants of all the modules. This allows to change only one file when any change has to be made. In this file are defined: the number of bits of the inputs and outputs of the circuit for self-repairing `size_inputs` and `size_outputs`, the number of bits for the recovery method `size_recovery_method`, the self flag `size_self`, and the total number of bits of the fault

pattern vector `size_fp_vector`. Further are also defined: the number of maximum number of vectors in memory in `size_memory`, the corresponding number of bits for addressing those vectors in `size_memory_in_bits`, and the fault pattern vectors in `mem_start`. The constant `mem_start` has the type `ram_array`, which has also been defined in this package. The user defined type `ram_array` is an **array** with a collection of vectors of the type `std_logic_vector`. Finally, the maximum number of recoveries is defined in `max_recoveries`.

Program Code 6.9: Constants package

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  package Constants is
5
6      constant size_memory:             integer := 16;
7      constant size_memory_in_bits:     integer := 4;
8
9      constant size_inputs:             integer := 3;
10     constant size_outputs:            integer := 1;
11     constant size_recovery_method:    integer := 3;
12     constant size_self:               integer := 1;
13     constant size_fp_vector:        integer := size_self +\
14               size_recovery_method + size_inputs + size_outputs;
15
16     constant max_recoveries:          integer := 3;
17
18     type ram_array is array (0 to 2**size_memory_in_bits-1) of\
19           std_logic_vector(size_fp_vector-1 downto 0);
20
21     constant mem_start: ram_array := (  "01100001", -- 61h
22                                         "01100011", -- 63h
23                                         "01100100", -- 64h
24                                         "01010110", -- 66h
25                                         "00111001", -- 39h
26                                         "00111010", -- 3ah
27                                         "00111101", -- 3dh
28                                         "00111110", -- 3eh
29                                         "11100000", -- e0h
30                                         "11100010", -- e2h
31                                         "11100101", -- e5h
32                                         "11010111", -- e7h
33                                         "10111000", -- b8h
34                                         "10111011", -- bbh
35                                         "10111100", -- bch
36                                         "10111111"  -- bfh
37     );
38 end Constants;
39
40 package body Constants is
41
42 end Constants;
```

### 6.1.2 Partial reconfiguration for recovering the unit

When the circuit for self-repairing is planned to be implemented by means of an FPGA, one recovery mechanism can be to use partial reconfiguration. Partial reconfiguration is the process that allows to swap a partial reconfigurable module with another version of it, while the rest of the static modules are operating. In our case, the circuit for self-repairing can be defined as a partial reconfigurable module, an the rest of the modules in the architecture can be defined as static modules.

The use of partial reconfiguration demands some additional modules in the architecture of the self-repairing unit. Firstly, the partial reconfiguration process requires a partial reconfiguration controller which is proposed to be placed as an static module in the same FPGA, as can be seen in figure 6.7. Secondly, the region where a partial reconfigurable module is placed in the FPGA is a dynamic region, and similarly, the region where the static modules are placed in the FPGA is the static region. In the partial reconfiguration design flow, partial reconfigurable modules and static modules are implemented separately and they are merged. In order to ease that implementation process, it is desirable to have only one static module. Therefore, all the modules of the architecture of the self-repairing unit, excluding the circuit for self-repairing, are declared as components and instantiated connecting them together in only one upper module named RecognizerRepairer. That upper module is shown in figure 6.8. Thirdly, the connection signals of a partial reconfigurable module with the static upper module, should pass through the so called bus macros as can be seen in figures 6.8 and 6.7.

The partial reconfiguration controller implemented in the self-partial reconfigurator module and the bus macros are explained in detail below. Since in next subsection for testing purposes some modules are added to the architecture, the ReconfiguratorRepairer module is explained in that subsection.

#### Self-partial reconfigurator module

The behavior of the partial reconfigurable controller depends on the FPGA chip and development platform to be used. In this section, as an example, the implementation of a partial reconfigurable controller for the ML403 Virtex-4 FX evaluation board having a XC4VFX12-FF668-10 chip is presented. Although at the end the triggering of the partial reconfiguration by the same FPGA could not get it to work due to hardware bugs in the evaluation board ML403 Virtex-4 FX, the implementation issues and the obstacles are exposed in order to show how far it has been possible to go in the implementation with the available hardware.

A Xilinx FPGA can reconfigure itself when an embedded microprocessor or an state machine reads the bitstream out of a storage device and send the data to the internal configuration port, named ICAP. Or, it can be reconfigured by an external microprocessor that reads the bitstream and send the data to any standard configuration port of the FPGA, please see [Dye, 2012]. The FPGA chip on the available evaluation platform can be reconfigured by any available external device using the configuration ports: SelectMap, Serial, and JTAG, please see [Xilinx, 2009e] for more information. The bitstreams can be retrieved from: the platform flash, the linear flash, a compact flash card, or from the computer using a parallel cable IV from Xilinx, for more information please see [Xilinx, 2006c]. For retrieving bitstreams from the linear flash a Complex Programmable Logic Device, in short CPLD, is available, and for retrieving bitstreams from a Compact Flash card a System Advanced Configuration Environment controller chip, in short System ACE controller chip, designed by Xilinx is

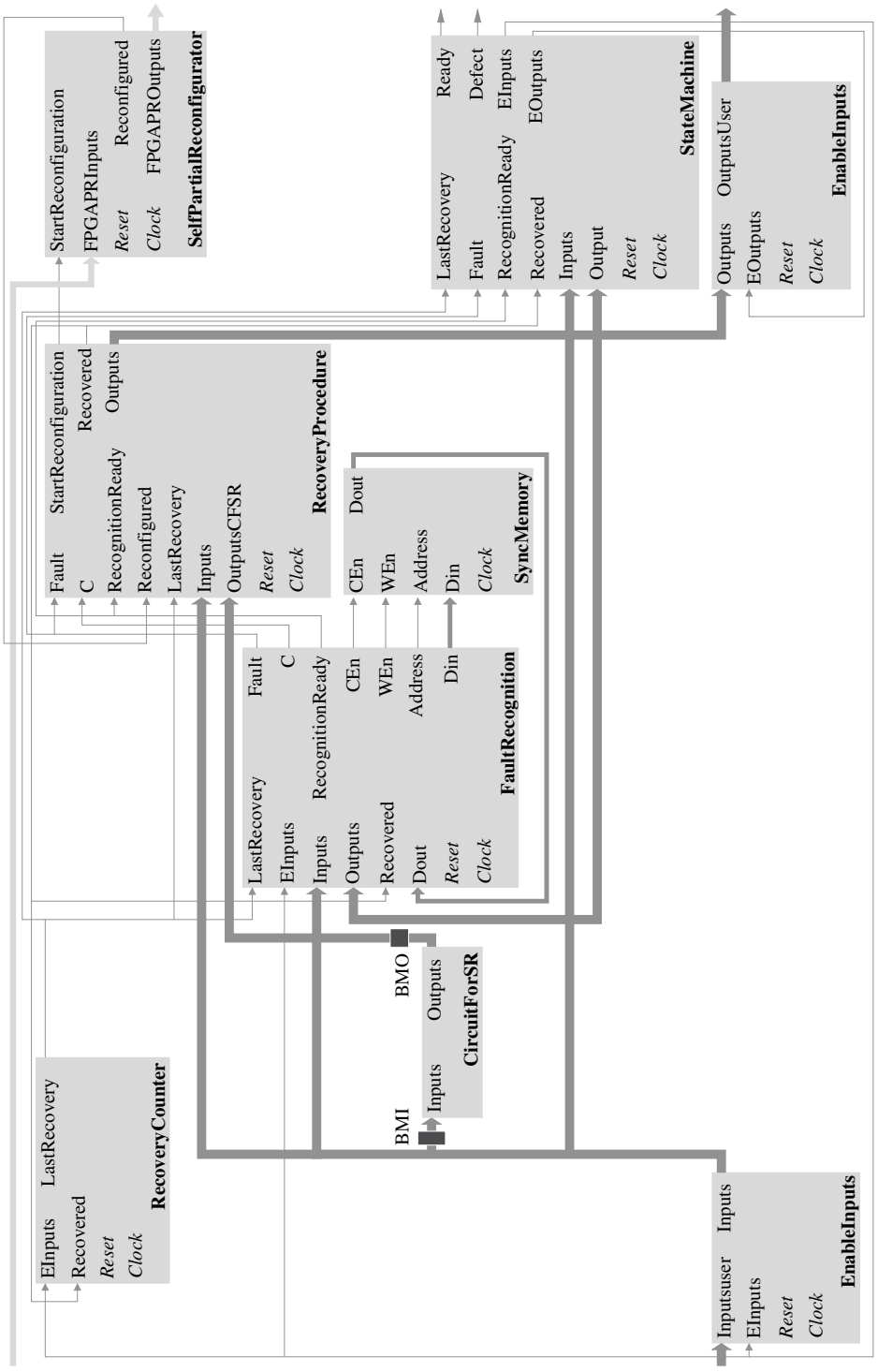



Figure 6.7: Architecture of the self-repairing system

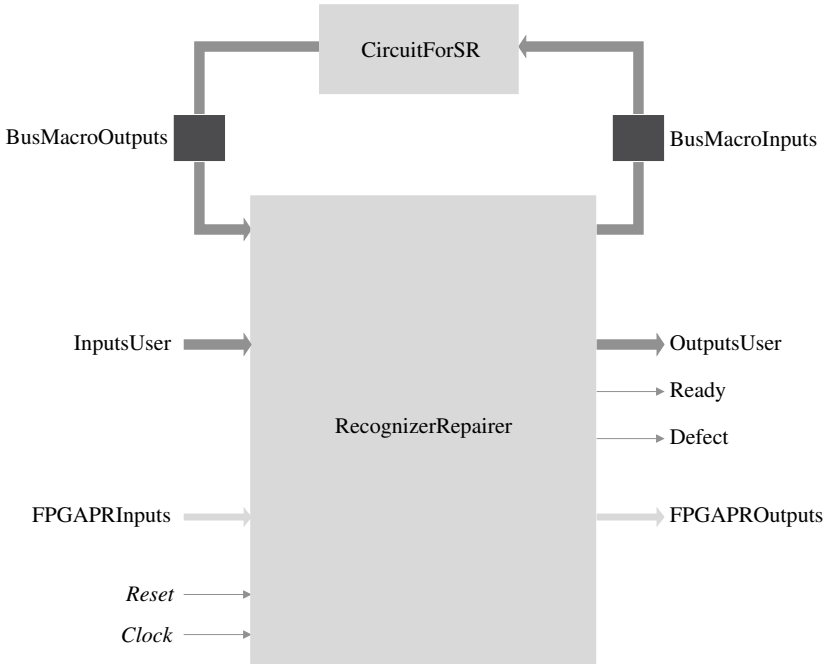



Figure 6.8: Fault recognizer and repairer module for the circuit for self-repairing

available. The FPGA, platform flash, CPLD, and System ACE controller chips are connected together in a JTAG chain. JTAG comes from Joint Test Action Group, which is the name given to the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture. That JTAG chain can be used to program the FPGA, the platform flash, and the CPLD. And also for downloading bitstreams to the FPGA using the parallel cable IV and the iMPACT software tool. The System ACE controller chip is physically connected with the FPGA chip through a port named microprocessor unit port, in short MPU port. The 16 line data bus and some lines of the 7 line address bus of that MPU port are shared with an USB controller, please see [Xilinx, 2004]. The System ACE MPU port allows the FPGA to instruct the System ACE controller to load one on the eight bitstreams stored in a flash card for reconfiguring itself, process that can be named as self-partial reconfiguration, for more information please see [Xilinx, 2006c] and [Xilinx, 2009d].

Once the FPGA has been configured loading a full bitstream, that is to say, a bitstream with the configuration of the whole circuit, partial bitstreams, that is to say, bitstreams with the configuration of only a dynamic region of the FPGA, can be loaded for partial reconfiguring the FPGA. A bitstream contains information about the physical place of the circuit. Therefore, in partial reconfiguration it is necessary only to load the partial bitstream that contains the partial reconfigurable module, without worrying about anything else, please see [Xilinx, 2006b]. The job of retrieving a bitstream from a compact flash card and downloading it into the FPGA is done by the System ACE controller in the ML403 Virtex-4 FX evaluation board using the JTAG chain. But, for implementing self-partial reconfiguration, it is necessary to have a circuit in the FPGA that instructs the System ACE controller to load the partial bitstream into that FPGA. That circuit has been implemented in the SelfPartialReconfigurator module in the form of a state machine shown in code listing 6.10. Please note that if the procedure of retrieving a partial bitstream from any storage device and its

downloading into the FPGA is not supported by the development platform to be used, it should be implemented in the SelfPartialReconfigurator module.

The SelfPartialReconfigurator module has the *StartReconfiguration* signal as main input and the *Reconfigured* as main output. Firstly, it is theoretically possible to start reconfiguration of the FPGA setting the bit *CFGSTART* of the *CONTROLREG* register of the System ACE controller. Additionally, in the *CONTROLREG* register the bits *CFGMODE*, *CFGSEL*, *CFGRESET*, *CFGADDRBIT0-2* should be set following the description given in the datasheet [Xilinx, 2009d] for using the *CFGSTART* signal adequately. Secondly, it is possible to know that the reconfiguration is finished by an interrupt given at the MPU interface port signal *MPIRQ*. But, that interrupt signal should be firstly enabled setting the bit *CFGDONEIRQ* in the *CONTROLREG* register, and be acknowledged as received by setting the bit RESETIRQ in the *CONTROLREG* register.

For writing into the *CONTROLREG* register of the System ACE controller, it is necessary to reproduce the single register write cycle shown in the System ACE controller datasheet [Xilinx, 2009d]. The single register write cycle indicates to give the address of the register to be written at the 7 line MPU register address bus, set to '0' the MPU chip enable interface port signal $\overline{MPCE}$, set to '1' the MPU output enable interface port signal $\overline{MPOE}$, in the first clock cycle. In the second clock cycle the MPU write enable interface port signal $\overline{MPWE}$ should be set to '0' and the address of the register should be given at the 16 line MPU register data bus. Finally during the third clock cycle, the MPU write enable interface port signal $\overline{MPWE}$ should be set back to '1' for completing the writing process. In the *BUSMODEREG* register it is possible to configure the use of a 16 or 8 line MPU register data bus. Since the *CONTROLREG* register has only 8 bits, the *BUSMODEREG* register should be set accordingly.

Code listing 6.10 presents two processes for helping on generating the single register write cycle, Generation_SYSACE_CLK_halb and Generation_SYSACE_CLK_quarter, assuring the minimum timing parameters of the signals. The processes SM_CombProcess, SM_NextStateProcess and SM_OutputsProcess, implement the state machine that starts the reconfiguration of the FPGA and waits for the interrupt to inform setting the output signal *Reconfigured* that the reconfiguration is finished. The outputs to be connected to the System ACE controller are assigned with concurrent signal assignment statements as shown in code line 68 to 73. This code could not be debugged completely due to hardware bugs in the evaluation board ML403 Virtex-4 FX, specially in the System ACE controller chip.

Although the implementation of the self-repairing unit is explained in a section hereafter, some implementation details related to the module SelfPartialReconfigurator are given here in order to support better the arguments of why automatic self-partial reconfiguration did not worked out. In the compact flash card, only one partial reconfiguration bitstream file with the extension .ace file has been stored following the recommended file structure. To produce that partial bitstream, Windows XP and the Partial Reconfiguration Early Access Software Tools for ISE 9.2i SP4 have been used and found to be the minimum requirement. For the configuration of the FPGA through the System ACE controller, the configuration selector switch SW12 of the evaluation board is to be set to the SYS ACE position, please see the placement of that switch in the user guide of the evaluation board [Xilinx, 2006c].

The partial bitstream has been successfully downloaded in the dynamic region of the FPGA pressing the System ACE reset button RST in the evaluation board that re-programs the FPGA. However, neither the reconfiguration of the dynamic region of the FPGA setting the bit *CFGSTART* of the *CONTROLREG* register, nor the interruption at *MPIRQ* after a

reconfiguration works. The support personal of Xilinx indicated on April 2010, that triggering configuration through the System ACE controller using its MPU interface did not work and this is why it is not supported and not documented completely in the datasheet. A further try with the soft reset using the *CFGRESET* bit of the *CONTROLREG* register did not work too. The Xilinx team suggested to work around driving directly the RESET pin of the System ACE controller chip using an I/O pin of the FPGA, but that is not possible with the evaluation board ML403 Virtex-4 FX, since that RESET pin is already hardwired, but with an own designed board it could work.

Since partial reconfiguration as a repairing procedure could not be fully tested and no solution with the available hardware was possible, partial reconfiguration has been just tested using the iMPACT software tool for downloading the partial bitstreams into the FPGA using the JTAG chain and the parallel cable from Xilinx. That has been done when the *StartReconfiguration* signal was raised being visualized by a led. After reconfiguration, the *Reconfigured* signal has been manually set using an externally hardwired switch. Further experiments can be done with newer development boards or FPGA chips, if hopefully dynamic partial reconfiguration by self reconfiguration is already fully supported as promised in [Dye, 2012].

Program Code 6.10: Self-partial reconfigurator module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.Constants.all;
5
6  entity SelfPartialReconfigurator is
7      port (
8          SYSACE_CLK:           in std_logic;
9          Reset:                in std_logic;
10         StartReconfiguration: in std_logic;
11         Reconfigured:         out std_logic;
12         SYSACE_MPCE:          out std_logic;
13         SYSACE_MPWE_USB_WR_N: out std_logic;
14         SYSACE_MPOE_USB_RD_N: out std_logic;
15         SYSACE_Address:       out std_logic_vector(6 downto 1);
16         SYSACE_Data:          out std_logic_vector(15 downto 0);
17         SYSACE_MPIRQ:         in std_logic
18     );
19 end SelfPartialReconfigurator;
20
21 architecture Behavioral of SelfPartialReconfigurator is
22     subtype RegDataType is std_logic_vector(15 downto 0);
23     subtype RegAddressType is std_logic_vector(6 downto 1);
24     signal DataToWrite: RegDataType;
25     signal RegisterAddress: RegAddressType;
26     constant Register_Data_Zeros: RegDataType:=(others=>'0');
27
28     type state_type is (st_Init_0, st_Init_1, st_Init_2,\
29        st_WaitStartRec, st_Reconf_1, st_Reconf_2, st_Reconf_3,\
30        st_WaitInt, st_MarkReconf, st_ResetInt_1, st_ResetInt_2,\
31        st_ResetInt_3);
32     signal state, next_state: state_type;
```

```
33
34      signal SYSACE_CLK_halb: std_logic:='0';
35      signal SYSACE_CLK_quarter: std_logic:='0';
36      signal Reconfigured_i: std_logic:='0';
37      signal WriteRegister: std_logic:='0';
38      signal SYSACE_MPIRQOld: std_logic:='0';
39      signal StartReconfigurationOld: std_logic:='0';
40
41      --System ACE Chip Register Addresses
42      constant BUSMODEREG_Address: RegAddressType:= "00" & x"0";
43      constant CONTROLREG_LSB_Address: RegAddressType:= "00" & x"C";
44
45      --BUSMODEREG Bits
46      constant BUSMODE0: RegDataType:= x"0001";
47
48      --CONTROLREG Bits
49      constant FORCELOCKREQ: RegDataType:= x"0001";
50      constant LOCKREQ: RegDataType:= x"0002";
51      constant FORCECFGADDR: RegDataType:= x"0004";
52      constant FORCECFGMODE: RegDataType:= x"0008";
53      constant CFGMODE: RegDataType:= x"0010";
54      constant CFGSTART: RegDataType:= x"0020"; --* StartReconfiguration
55      constant CFGSEL: RegDataType:= x"0040";
56      constant CFGRESET: RegDataType:= x"0080";
57      constant DATABUFRDYIRQ: RegDataType:= x"0100";
58      constant ERRORIRQ: RegDataType:= x"0200";
59      constant CFGDONEIRQ: RegDataType:= x"0400"; --* Reconfigured
60                                                  --  Interrup. trigger
61      constant RESETIRQ: RegDataType:= x"0800"; --* Reconfigured
62                                                --  Interruption reset
63      constant CFGPROG: RegDataType:= x"1000";
64      constant CFGADDRBIT0: RegDataType:= x"2000"; --* Configuration
65      constant CFGADDRBIT1: RegDataType:= x"4000"; --  file number 0-7
66      constant CFGADDRBIT2: RegDataType:= x"8000"; --
67  begin
68      SYSACE_MPCE <= not '1';
69      SYSACE_MPOE_USB_RD_N <= not '0';
70      SYSACE_MPWE_USB_WR_N <= SYSACE_CLK_quarter when\
71                              WriteRegister = '1' else '1';
72      SYSACE_Address <= RegisterAddress;
73      SYSACE_Data <= DataToWrite;
74
75      Generation_SYSACE_CLK_halb: process(SYSACE_CLK)
76      begin
77          if rising_edge(SYSACE_CLK) then
78              SYSACE_CLK_halb <= not SYSACE_CLK_halb;
79          end if;
80      end process;
81
82      Generation_SYSACE_CLK_quarter: process(SYSACE_CLK_halb)
83      begin
84          if rising_edge(SYSACE_CLK_halb) then
85              SYSACE_CLK_quarter <= not SYSACE_CLK_quarter;
```

```
86          end if;
87      end process;
88
89      SM_CombProcess: process (SYSACE_CLK_quarter)
90      begin
91           if rising_edge(SYSACE_CLK_quarter) then
92               if Reset = '1' then
93                   state <= st_WaitStartRec;
94                   Reconfigured <= '0';
95               else
96                   state <= next_state;
97                   Reconfigured <= Reconfigured_i;
98                   SYSACE_MPIRQOld <= SYSACE_MPIRQ;
99                   StartReconfigurationOld <= StartReconfiguration;
100              end if;
101          end if;
102     end process;
103
104     SM_NextStateProcess: process (state, StartReconfiguration, \
105                                   SYSACE_MPIRQ)
106     begin
107         case (state) is
108             when st_Init_0 =>
109                 next_state <= st_Init_1;
110             when st_Init_1 =>
111                 next_state <= st_Init_2;
112             when st_Init_2 =>
113                 next_state <= st_WaitStartRec;
114             when st_WaitStartRec =>
115                 if ( StartReconfigurationOld /= \
116                      StartReconfiguration ) and \
117                      StartReconfiguration = '1' then
118                     next_state <= st_Reconf_1;
119                 else
120                     next_state <= st_WaitStartRec;
121                 end if;
122             when st_Reconf_1 =>
123                 next_state <= st_Reconf_2;
124             when st_Reconf_2 =>
125                 next_state <= st_Reconf_3;
126             when st_Reconf_3 =>
127                 next_state <= st_WaitInt;
128             when st_WaitInt =>
129                 if ( SYSACE_MPIRQOld /= SYSACE_MPIRQ ) and\
130                      SYSACE_MPIRQ = '1' then
131                     next_state <= st_MarkReconf;
132                 else
133                     next_state <= st_WaitInt;
134                 end if;
135             when st_MarkReconf =>
136                 next_state <= st_ResetInt_1;
137             when st_ResetInt_1 =>
138                 next_state <= st_ResetInt_2;
```

```
139             when st_ResetInt_2 =>
140                 next_state <= st_ResetInt_3;
141             when st_ResetInt_3 =>
142                 next_state <= st_WaitStartRec;
143             when others =>
144                 next_state <= st_WaitStartRec;
145         end case;
146     end process;
147
148     SM_OutputsProcess: process (state)
149     begin
150         case (state) is
151             when st_Init_0 =>
152                 Reconfigured_i <= '0';
153                 WriteRegister <= '0';
154                 RegisterAddress <= BUSMODEREG_Address;
155                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
156             when st_Init_1 =>
157                 Reconfigured_i <= '0';
158                 WriteRegister <= '1';
159                 RegisterAddress <= BUSMODEREG_Address;
160                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
161             when st_Init_2 =>
162                 Reconfigured_i <= '0';
163                 WriteRegister <= '1';
164                 RegisterAddress <= CONTROLREG_LSB_Address;
165                 DataToWrite <= Register_Data_Zeros or CFGDONEIRQ;
166             when st_WaitStartRec =>
167                 Reconfigured_i <= '0';
168                 WriteRegister <= '0';
169                 RegisterAddress <= BUSMODEREG_Address;
170                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
171             when st_Reconf_1 =>
172                 Reconfigured_i <= '0';
173                 WriteRegister <= '1';
174                 RegisterAddress <= CONTROLREG_LSB_Address;
175                 DataToWrite <= Register_Data_Zeros or CFGRESET\
176                                or CFGDONEIRQ;
177             when st_Reconf_2 =>
178                 Reconfigured_i <= '0';
179                 WriteRegister <= '1';
180                 RegisterAddress <= CONTROLREG_LSB_Address;
181                 DataToWrite <= Register_Data_Zeros or CFGSTART\
182                                or CFGRESET or CFGDONEIRQ;
183             when st_Reconf_3 =>
184                 Reconfigured_i <= '0';
185                 WriteRegister <= '1';
186                 RegisterAddress <= CONTROLREG_LSB_Address;
187                 DataToWrite <= ( ( Register_Data_Zeros or CFGSTART )\
188                                  and not CFGRESET ) or CFGDONEIRQ;
189             when st_WaitInt =>
190                 Reconfigured_i <= '0';
191                 WriteRegister <= '0';
```

```
192                 RegisterAddress <= BUSMODEREG_Address;
193                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
194             when st_MarkReconf =>
195                 Reconfigured_i <= '1';
196                 WriteRegister <= '0';
197                 RegisterAddress <= BUSMODEREG_Address;
198                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
199             when st_ResetInt_1 =>
200                 Reconfigured_i <= '0';
201                 WriteRegister <= '1';
202                 RegisterAddress <= CONTROLREG_LSB_Address;
203                 DataToWrite <= ( Register_Data_Zeros and \
204                                  not CFGSTART ) or CFGDONEIRQ;
205             when st_ResetInt_2 =>
206                 Reconfigured_i <= '0';
207                 WriteRegister <= '1';
208                 RegisterAddress <= CONTROLREG_LSB_Address;
209                 DataToWrite <= Register_Data_Zeros or RESETIRQ\
210                                or CFGDONEIRQ;
211             when st_ResetInt_3 =>
212                 Reconfigured_i <= '0';
213                 WriteRegister <= '1';
214                 RegisterAddress <= CONTROLREG_LSB_Address;
215                 DataToWrite <= ( Register_Data_Zeros and \
216                                  not RESETIRQ ) or CFGDONEIRQ;
217             when others =>
218                 Reconfigured_i <= '0';
219                 WriteRegister <= '0';
220                 RegisterAddress <= BUSMODEREG_Address;
221                 DataToWrite <= Register_Data_Zeros or BUSMODE0;
222         end case;
223     end process;
224
225 end Behavioral;
```

### Bus macros

A bus macro is a sort of module with fixed connection points that serves for interfacing the dynamic and the static regions. A bus macro is not required to be programmed because it is provided together with the software tools for partial reconfiguration as a file with `.nmc` extension. It has a naming convention that contains: the FPGA device it is prepared for; the physical direction of its unidirectional connection points, i.e., right to left, left to right, top to bottom, bottom to top; its synchronicity, where a synchronous macro has a clock line and registers for capturing the data at the connection points; and its width, which is the number of configurable logic blocks it is made of, having a narrow bus macro with two CLBs for eight connection points and a wide bus macro with four CLBs for sixteen connection points.

Since the example of circuit for self-repairing has only 3 inputs and one output, the bus macros `busmacro_xc4v_r2l_async_narrow.nmc` for the inputs and for the output the bus macro `busmacro_xc4v_l2r_async_narrow.nmc`, have been selected. They should be defined as components and instantiated mapping its connection points according to the circuit design

in the top module, to be presented further in this chapter. Besides, the physical placement of the bus macros should be given in the constraints file of the top module, to be presented also further in this chapter. For more information about bus macros and advising of how to physically place them, please refer to the manual of the software tool for partial reconfiguration used, in this case [Xilinx, 2008].

### 6.1.3 Fault injection for testing the self-repairing unit

In order to confirm that the self-repairing unit is capable of repairing itself, it is necessary to inject some faults into the circuit for self-repairing. In the literature, there exist some techniques for inserting faults which are for example: the use of partial reconfiguration for downloading bitstreams in the FPGA containing a faulty circuit for self-repairing, presented in [Sterpone and Violante, 2007]; the use of simulator commands during the simulation of the design for altering some signals or variables in the circuit for self-repairing, technique presented in [Jenn et al., 1994]; the use of the existing boundary scan architecture of the FPGA for injecting faults at any input or output signal of the circuit for self-repairing which is connected externally to a I/O pin of the FPGA having a boundary scan cell that can override its value, as discussed in [Chakraborty and Chiang, 2002]; or the modification of the VHDL code by: adding saboteur modules that when active modify the value or timing characteristics of signals in the circuit for self-repairing, or adding mutant modules that when active overlap the circuit for self-repairing with any faulty one, [Jenn et al., 1994]. The VHDL mutants or mutants of the binary partial bitstream of the circuit for self-repairing could be produced using the method exposed in [Xie et al., 2011] and [Becker et al., 2012].

Those methods can be classified in methods of fault injection during the simulation of the self-repairing unit and methods of fault injection while the self-repairing system is operating. The use of simulator commands during simulation can help on debugging the design of the self-repairing unit. Furthermore the insertion of saboteur and mutant modules into the self-repairing unit can also help on debugging the VHDL design. The use of partial reconfiguration requires to have an additional fault injection controller that triggers partial reconfiguration. The use of the existing boundary scan in the hardware platform can be helpful whenever the circuit for self-repairing has its inputs and outputs connected to the external I/O pins of the FPGA or implementing chip. The insertion of fault injection modules early in the design can be helpful for debugging the self-repairing unit or for testing it during its operational life. Besides, working at the VHDL level allows to insert faults for different fault models, i.e. faults at the gate level, please see [Misera and Sieber, 2007] or could allow to insert fault at different points in the design automatically, please see [Baraza et al., 2005].

Some experiments for inserting stuck-at faults in a circuit at the VHDL level were executed and presented in the Bachelor thesis [Traut, 2010]. From the results, it could be concluded that stuck-at faults can be inserted at the inputs and outputs of the circuit for self-repairing using saboteur modules, and stuck-at faults at any internal signal inside the circuit for self-repairing can be inserted using mutant modules of the circuit for self-repairing.

Therefore, the proposed architecture of the self-repairing unit has been enhanced with three further VHDL modules SaboteursInModule, SaboteursOutModule and MutantsModule, as can be seen in figure 6.9. The connection of those modules with the circuit for self-repairing is shown in figure 6.10. Since we also use redundancy as a repairing mechanism, the connection with redundant circuits drawn in figure 6.11 considered the possibility of injecting a fault even after the circuit for self-repairing has been recovered. In order to be able to control fault

injection from the outside, some control inputs for fault injection have been added. Those can be seen in the RecognizerRepairer module in figure 6.12. The input signals *SaboteurInputs*, *SaboteurOutputs* and *Mutant* serve for activating the respective fault injection module. The input signal *StuckAt* serves for selecting between stuck-at-1 or stuck-at-0 faults. The input signal *Injection* serves for injecting the elected fault. It could be possible, of course, to inject two faults or more simultaneously, but analyzing the behavior of the circuit under more than two faults is not straightforward, thus, this situation has been avoided. And finally, the input signal *FaultRemoval* removes the inserted fault, but not the failure or error generated by the presence of that fault. The functionality and design of the fault injection modules is explained below, followed by the description of the RecognizerRepairer and Top modules of the final self-repairing unit.

### Saboteurs at the inputs module

A saboteur is an element that can be placed breaking a signal line for altering its value or timing characteristics. It is intended to place saboteurs at the inputs of the circuit for self-repairing for inserting stuck-at faults at those lines. For that, a VHDL module named SaboteursInModule has been implemented and is shown in code listing 6.11.

This module has an input for enabling the saboteurs at the inputs named *SaboteurInputs*, an input for determining if an stuck-at-1 or an stuck-at-0 is inserted in the signal line named *StuckAt1*, an input for injecting a fault named as *Injection*, and an input for removing the fault but not the failure or fault produced *FaultRemoval*. The signal line to be altered enters through the input vector *Inputs* and the altered signal line gets out through the output vector *InputsFI*. The module is synchronized by the *Clock* signal and set to its default state through the signal *Reset*.

The module has only one process named SI_PROC. At a rising edge of the signal *Injection*, if and only if the module is enabled by setting to '1' the signal *SaboteurInputs*, the stuck-at fault selected is inserted to a random input line of the input vector Inputs, as can be seen in code lines 35 to 49. Thereby, the signal *InjectedFaults* is set to '1', which allows the concurrent statement in code line 25 to pass the inputs with the injected fault out. The input in which the stuck-at fault will be inserted is computed randomly in code lines 50 to 56. At a rising edge of the input *FaultRemoval*, the signal *InjectedFaults* is set to '0', which prevents to pass the inputs with the inserted fault out, as can be seen in code lines 57 to 57 and 25.

Program Code 6.11: Saboteurs at the inputs module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity SaboteursInModule is
6      port (
7          Clock:          in std_logic;
8          Reset:          in std_logic;
9          SaboteurInputs: in std_logic;
10         StuckAt1:       in std_logic;
11         Injection:      in std_logic;
12         FaultRemoval:   in std_logic;
13         Inputs:         in std_logic_vector(size_inputs-1 downto 0);
```

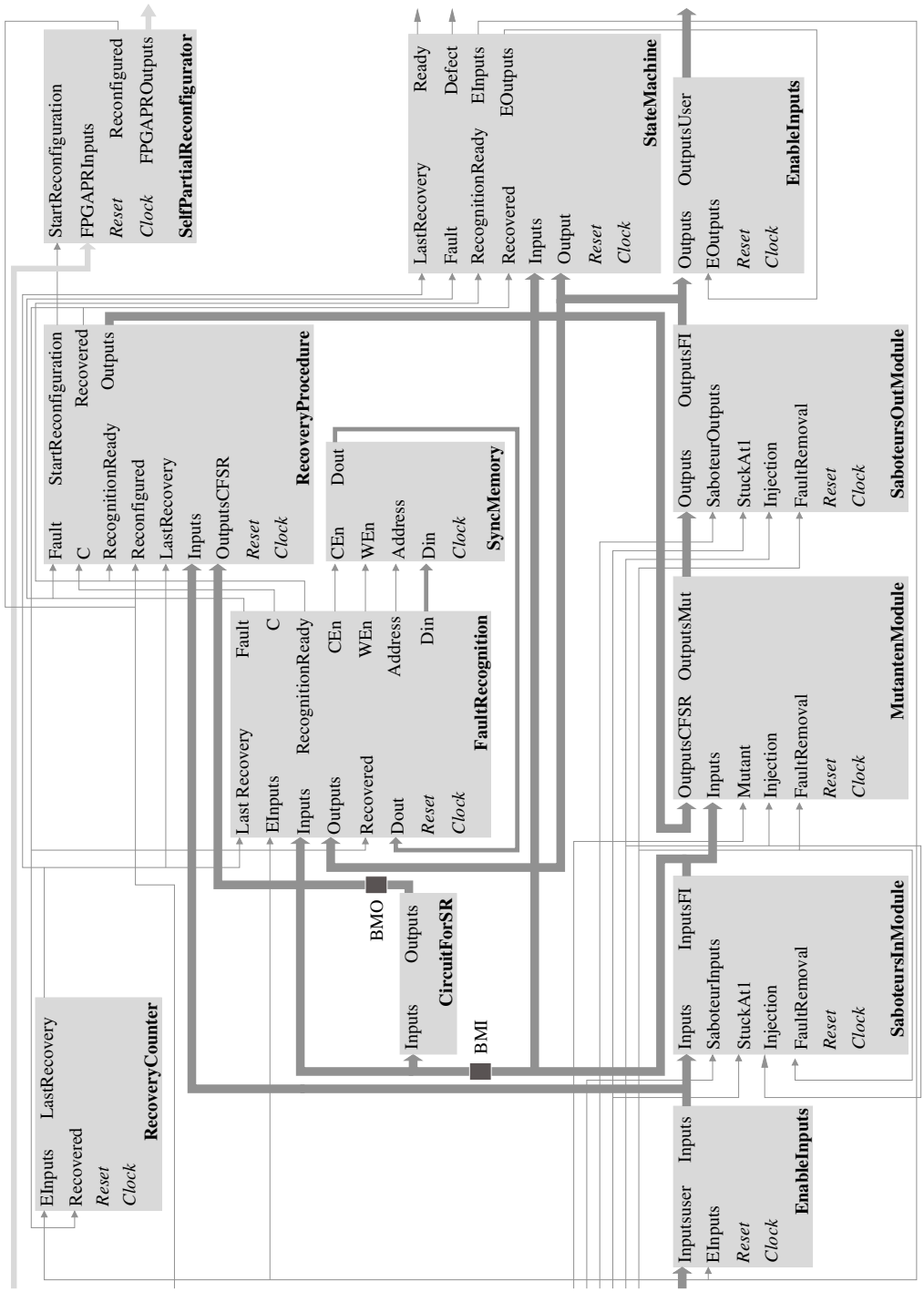



Figure 6.9: Architecture for testing the self-repairing system

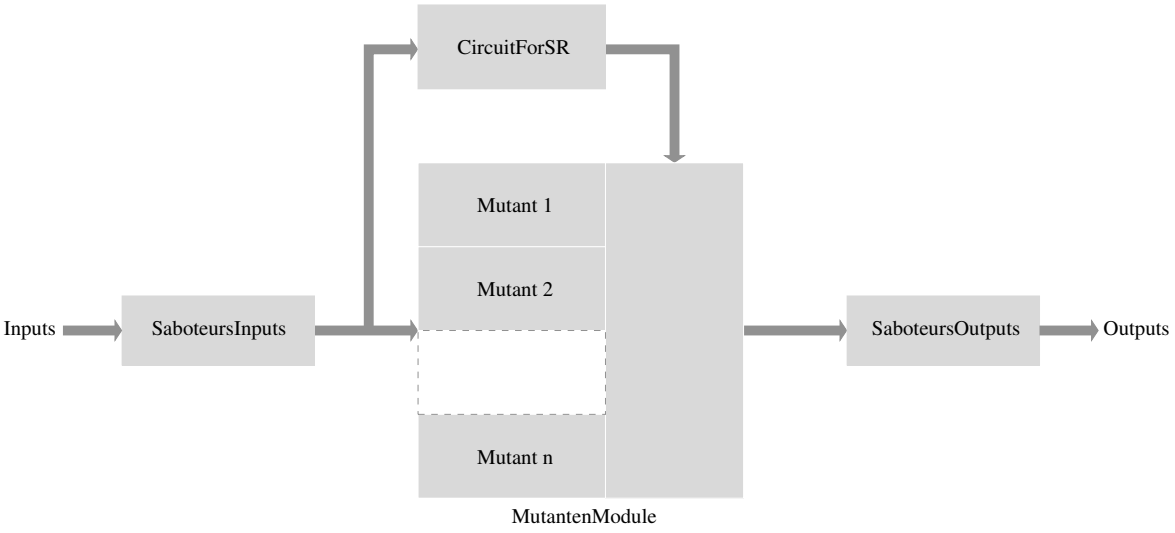



Figure 6.10: Fault injector

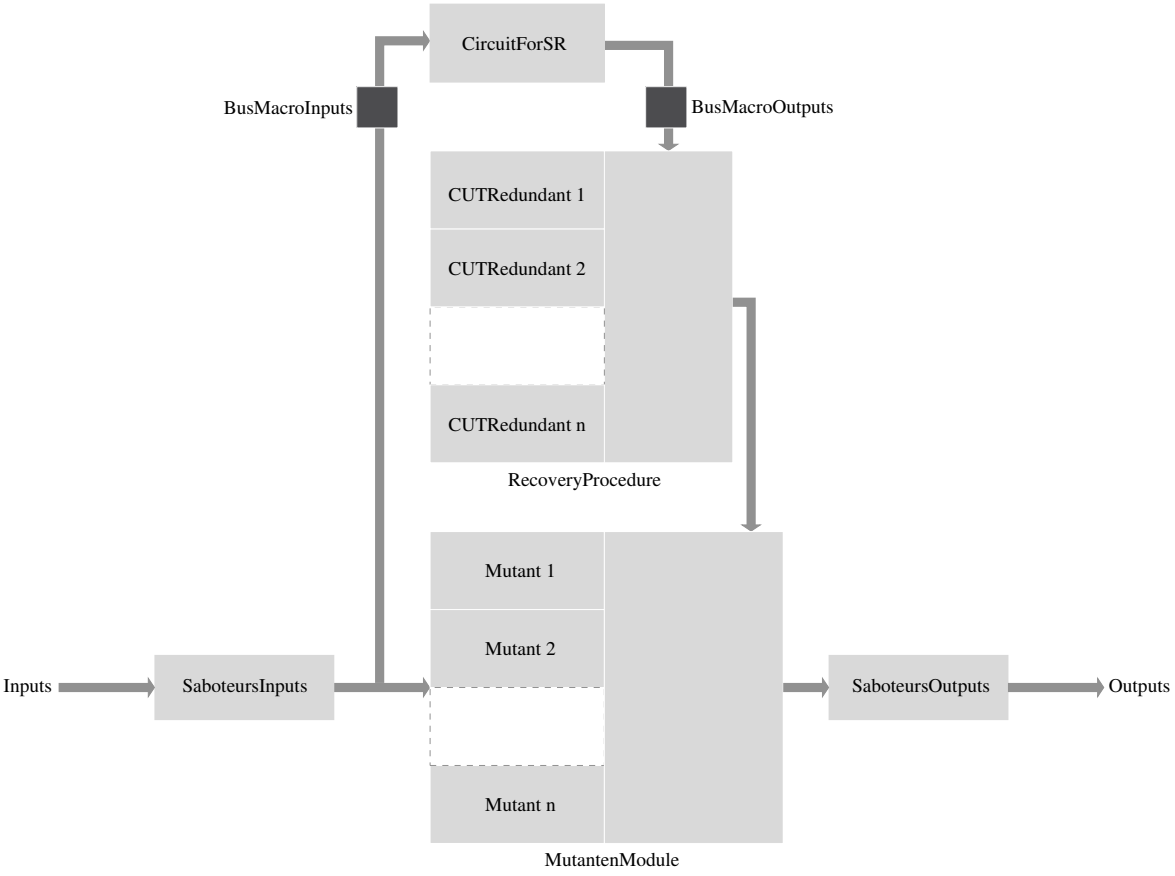



Figure 6.11: Fault injector and fault repairer connection

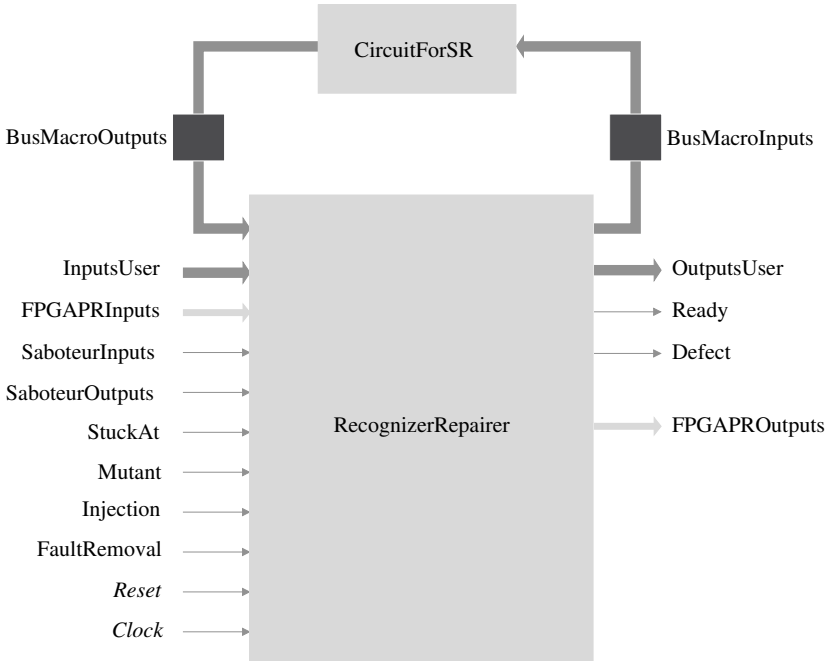



Figure 6.12: Fault recognizer and repairer module with an embedded fault injector

```
14          InputsFI:       out std_logic_vector(size_inputs-1 downto 0)
15      );
16  end SaboteursInModule;
17
18  architecture Behavioral of SaboteursInModule is
19      signal InjectionOld: std_logic;
20      signal RandomNumber: integer:= 0;
21      signal InputsTemp: std_logic_vector(size_inputs-1 downto 0);
22      signal InjectedFaults: std_logic:='0';
23      signal FaultRemovalOld: std_logic:='0';
24  begin
25      InputsFI <= InputsTemp when (InjectedFaults = '1') else Inputs;
26
27      SI_PROC: process (Clock)
28      begin
29          if (Clock'event and Clock = '1') then
30              if Reset = '1' then
31                  FaultRemovalOld <= FaultRemoval;
32                  InjectedFaults <= '0';
33                  RandomNumber <= 0;
34              else
35                  if ((Injection /= InjectionOld) and\
36                          (Injection = '1') and (SaboteurInputs = '1')\
37                          and (StuckAt1 = '1') and\
38                          (InjectedFaults = '0')) then
39                      InputsTemp <= Inputs;
40                      InputsTemp(RandomNumber) <= '1';
41                      InjectedFaults <= '1';
```

```
42                  elsif ((Injection /= InjectionOld) and\
43                          (Injection = '1') and (SaboteurInputs = '1')\
44                          and (StuckAt1 = '0')\
45                          and (InjectedFaults = '0')) then
46                      InputsTemp <= Inputs;
47                      InputsTemp(RandomNumber) <= '0';
48                      InjectedFaults <= '1';
49                  end if;
50                  if ((RandomNumber = (size_inputs-1)) and\
51                        (InjectedFaults = '0')) then
52                      RandomNumber <= 0;
53                  elsif ((RandomNumber < (size_inputs-1)) and\
54                          (InjectedFaults = '0')) then
55                      RandomNumber <= RandomNumber + 1;
56                  end if;
57                  if ((FaultRemoval = '1') and\
58                        (FaultRemovalOld /= FaultRemoval)) then
59                      InjectedFaults <= '0';
60                  end if;
61                  InjectionOld <= Injection;
62                  FaultRemovalOld <= FaultRemoval;
63              end if;
64          end if;
65      end process;
66
67  end Behavioral;
```

### Saboteurs at the outputs module

This module is responsible of inserting stuck-out faults at the output lines of the circuit for self-repairing. It has been named SaboteursOutModule and works exactly the same as the SaboteursInModule, please see last subsection, except of it is enabled when the external signal *SaboteurOutputs* is set to '1'. Since the example circuit for self-repairing has only one output, the stuck-at fault is always inserted at that output. The lines for generating a fault at a random output in code listing 6.12 make sense for a circuit for self-repairing with many output lines.

Program Code 6.12: Saboteurs at the outputs module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity SaboteursOutModule is
6      port (
7          Clock:           in std_logic;
8          Reset:           in std_logic;
9          SaboteurOutputs: in std_logic;
10         StuckAt1:        in std_logic;
11         Injection:       in std_logic;
12         FaultRemoval:    in std_logic;
13         Outputs:         in std_logic_vector(size_outputs-1 downto 0);
```

```
14          OutputsFI:      out std_logic_vector(size_outputs-1 downto 0)
15      );
16  end SaboteursOutModule;
17
18  architecture Behavioral of SaboteursOutModule is
19      signal InjectionOld: std_logic:='0';
20      signal RandomNumber: integer:= 0;
21      signal OutputsTemp: std_logic_vector(size_outputs-1 downto 0);
22      signal InjectedFaults: std_logic:='0';
23      signal FaultRemovalOld: std_logic:='0';
24
25  begin
26      OutputsFI <=  OutputsTemp when (InjectedFaults = '1')\
27                    else Outputs;
28      SO_PROC: process (Clock)
29      begin
30          if (Clock'event and Clock = '1') then
31              if Reset = '1' then
32                  FaultRemovalOld <= FaultRemoval;
33                  InjectedFaults <= '0';
34                  RandomNumber <= 0;
35              else
36                  if ((Injection /= InjectionOld) and (Injection = '1')\
37                        and (SaboteurOutputs = '1')\
38                        and (StuckAt1 = '1')\
39                        and (InjectedFaults = '0')) then
40                      OutputsTemp <= Outputs;
41                      OutputsTemp(RandomNumber) <= '1';
42                      InjectedFaults <= '1';
43                  elsif ((Injection /= InjectionOld)\
44                           and (Injection = '1')\
45                           and (SaboteurOutputs = '1')\
46                           and (StuckAt1 = '0')\
47                           and (InjectedFaults = '0')) then
48                      OutputsTemp <= Outputs;
49                      OutputsTemp(RandomNumber) <= '0';
50                      InjectedFaults <= '1';
51                  end if;
52                  if ((RandomNumber = (size_outputs-1)) and\
53                        (InjectedFaults = '0')) then
54                      RandomNumber <= 0;
55                  elsif ((RandomNumber < (size_outputs-1)) and\
56                           (InjectedFaults = '0')) then
57                      RandomNumber <= RandomNumber + 1;
58                  end if;
59                  if ((FaultRemoval = '1') and\
60                        (FaultRemovalOld /= FaultRemoval)) then
61                      InjectedFaults <= '0';
62                  end if;
63                  InjectionOld <= Injection;
64                  FaultRemovalOld <= FaultRemoval;
65              end if;
66          end if;
```

```
67      end process;
68
69  end Behavioral;
```

### Mutants module

A mutant is an element that has undergone a change. Resembling that, a faulty circuit for self-repairing can be seen as a mutant. A faulty circuit for self-repairing can be described in a VHDL module, similarly as the original circuit for self-repairing has been described. In the description of that VHDL module, the value of signals can be altered in the same way as a saboteur does; components, operators or variables can be exchanged, i.e., an AND gate by an OR gate; or the behavior of a determined fault model can be described affecting the original module. A variety of mutants can be designed and connected in the place of the circuit for self-repairing on demand. Thus, the module MutantsModule injects a fault replacing the original circuit for self-repairing by a mutant of it named here CircuitForSRX. For that, the mutant module CircuitForSRX should be declared as a component, instantiated, and its input and output lines should be connected in the way that it replaces the original circuit for self-repairing. That happens at the rising edge of the external signal *Injection*, if and only if the external signal *Mutant* is set to '1', as can be seen in code lines 46 to 50. Similarly to the saboteurs, the mutant to be connected instead of the circuit for self-repairing is selected randomly in code lines 51 to 57. The faulty circuit for self-repairing, named here mutant, is replaced by the original circuit under a rising edge of the external signal *FaultRemoval*, as show in code lines 58 to 61. In code listing 6.13 only one mutant is shown, but similarly more mutants can be added.

Code listing 6.14 shows the example of a mutant having the circuit for self-repairing with an stuck-at-0 fault at the output of the first AND gate, as shown in code line 14.

Program Code 6.13: Mutants module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity MutantsModule is
6          port (
7              Clock:        in std_logic;
8              Reset:        in std_logic;
9              Mutant:       in std_logic;
10             Injection:    in std_logic;
11             FaultRemoval: in std_logic;
12             Inputs:       in std_logic_vector(size_inputs-1 downto 0);
13             OutputsCFSR:  in std_logic_vector(size_outputs-1 downto 0);
14             OutputsMut:   out std_logic_vector(size_outputs-1 downto 0)
15         );
16 end MutantsModule;
17
18 architecture Behavioral of MutantsModule is
19         component CircuitForSRX
20             port (
21             Inputs: in std_logic_vector( size_inputs - 1 downto 0 );
```

```
22              Outputs:out std_logic_vector( size_outputs - 1 downto 0 )
23          );
24          end component;
25
26          signal InjectionOld: std_logic;
27          signal RandomNumber: integer:= 0;
28          signal InjectedFaults: std_logic:='0';
29          signal FaultRemovalOld: std_logic:='0';
30          constant MutantsNumber: integer := 1;
31          signal OutputsX: std_logic_vector(size_outputs - 1 downto 0);
32
33  begin
34          CFSRX: CircuitForSRX port map (Inputs,OutputsX);
35
36          OutputsMut <= OutputsX when ((RandomNumber = 1) and\
37                    (InjectedFaults = '1')) else OutputsCFSR;
38
39          MUT_PROC: process (Clock)
40          begin
41              if (Clock'event and Clock = '1') then
42                  if Reset = '1' then
43                      InjectedFaults <= '0';
44                      RandomNumber <= 0;
45                  else
46                      if ((Injection /= InjectionOld) and\
47                        (Injection = '1') and (Mutant = '1')\
48                        and (InjectedFaults = '0')) then
49                          InjectedFaults <= '1';
50                      end if;
51                      if ((RandomNumber = MutantsNumber) and\
52                        (InjectedFaults <= '0')) then
53                          RandomNumber <= 1;
54                      elsif ((RandomNumber < MutantsNumber) and\
55                         (InjectedFaults <= '0')) then
56                          RandomNumber <= RandomNumber + 1;
57                      end if;
58                      if ((FaultRemoval = '1') and\
59                        (FaultRemovalOld /= FaultRemoval)) then
60                          InjectedFaults <= '0';
61                      end if;
62                      InjectionOld <= Injection;
63                      FaultRemovalOld <= FaultRemoval;
64                  end if;
65              end if;
66          end process;
67
68  end Behavioral;
```

Program Code 6.14: Mutant module with an stuck-at-0 fault

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
```

```
4
5  entity CircuitUnderTestX is
6      Port ( Inputs:  in  std_logic_vector( size_inputs-1 downto 0 );
7             Outputs: out std_logic_vector( size_outputs-1 downto 0 ) );
8  end CircuitUnderTestX;
9
10 architecture Behavioral of CircuitUnderTestX is
11
12 begin
13
14 Outputs(0) <= '0' or ( Inputs(2) and Inputs(0) );
15
16 end Behavioral;
```

### Recognizer-repairer module

The RecognizerRepairer shown in figure 6.12 is an upper module that groups together all modules of the self-repairing circuit with the exception of the circuit for self-repairing and the bus macros. It has been created for grouping all static modules into only one to ease the partial reconfiguration design flow. This module has the input and output signals shown in figure 6.12. All modules SelfPartialReconfigurator, MutantsModule, SaboteursOutModule, SaboteursInModule, EnableInputs, EnableOutputs, FaultRecognition, SyncMemory, RecoveryCounter, RecoveryProcedure and StateMachine are declared as components and then instantiated with the names SPR, MUT, SO, SI, EI, EO, FR, MEM, RC, RP and SM respectively in code listing 6.15. The signals in code lines 189 to 207 had to be declared in order to connect all those modules using positional association, that is to say, mapping the signals respecting the order in which the inputs and output ports on the interface of the modules, in the **entity** part of their code, are declared.

Program Code 6.15: Recognizer-repairer module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity RecognizerRepairer is
6      port (
7          Clock:           in std_logic;
8          Reset:           in std_logic;
9          InputsUser:      in std_logic_vector(size_inputs-1 downto 0);
10         OutputsUser:     out std_logic_vector(size_outputs-1 downto 0);
11         Defect:          out std_logic;
12         Ready:           out std_logic;
13         InputsCircuit:   out std_logic_vector(size_inputs-1 downto 0);
14         OutputsCircuit:  in std_logic_vector(size_outputs-1 downto 0);
15         Mutant:          in std_logic;
16         SaboteurInputs:  in std_logic;
17         SaboteurOutputs:in std_logic;
18         StuckAt1:        in std_logic;
19         Injection:       in std_logic;
20         FaultRemoval:    in std_logic;
```

```
21          SYSACE_CLK:       in std_logic;
22          SYSACE_MPCE:      out std_logic;
23          SYSACE_MPWE_USB_WR_N:out std_logic;
24          SYSACE_MPOE_USB_RD_N:out std_logic;
25          SYSACE_Address:   out std_logic_vector(6 downto 1);
26          SYSACE_Data:      out std_logic_vector(15 downto 0);
27          SYSACE_MPIRQ:     in std_logic
28      );
29  end RecognizerRepairer;
30
31  architecture Behavioral of RecognizerRepairer is
32      component SelfPartialReconfigurator is
33          port (
34              SYSACE_CLK:           in std_logic;
35              Reset:                in std_logic;
36              StartReconfiguration:in std_logic;
37              Reconfigured:         out std_logic;
38              SYSACE_MPCE:          out std_logic;
39              SYSACE_MPWE_USB_WR_N:out std_logic;
40              SYSACE_MPOE_USB_RD_N:out std_logic;
41              SYSACE_Address:       out std_logic_vector(6 downto 1);
42              SYSACE_Data:          out std_logic_vector(15 downto 0);
43              SYSACE_MPIRQ:         in std_logic
44          );
45      end component;
46
47      component MutantsModule is
48          port (
49              Clock:        in std_logic;
50              Reset:        in std_logic;
51              Mutant:       in std_logic;
52              Injection:    in std_logic;
53              FaultRemoval:in std_logic;
54              Inputs:       in std_logic_vector(size_inputs-1 downto 0);
55              OutputsCFSR: in std_logic_vector(size_outputs-1 downto 0);
56              OutputsMut:  out std_logic_vector(size_outputs-1 downto 0)
57          );
58      end component;
59
60      component SaboteursOutModule is
61          port (
62              Clock:        in std_logic;
63              Reset:        in std_logic;
64              SaboteurOutputs:in std_logic;
65              StuckAt1:     in std_logic;
66              Injection:    in std_logic;
67              FaultRemoval:in std_logic;
68              Outputs:      in std_logic_vector(size_outputs-1 downto 0);
69              OutputsFI:    out std_logic_vector(size_outputs-1 downto 0)
70          );
71      end component;
72
73      component SaboteursInModule is
```

```
74          port (
75              Clock:        in std_logic;
76              Reset:        in std_logic;
77              SaboteurInputs:in std_logic;
78              StuckAt1:     in std_logic;
79              Injection:    in std_logic;
80              FaultRemoval:in std_logic;
81              Inputs:       in std_logic_vector(size_inputs-1 downto 0);
82              InputsFI:     out std_logic_vector(size_inputs-1 downto 0)
83          );
84      end component;
85
86      component EnableInputs
87          port (
88              Clock:        in std_logic;
89              Reset:        in std_logic;
90              EInputs:      in std_logic;
91              InputsUser:   in std_logic_vector(size_inputs-1 downto 0);
92              Inputs:       out std_logic_vector(size_inputs-1 downto 0)
93          );
94      end component;
95
96      component EnableOutputs
97          port (
98              Clock:        in std_logic;
99              Reset:        in std_logic;
100             EOutputs:     in std_logic;
101             Outputs:      in std_logic_vector(size_outputs-1 downto 0);
102             OutputsUser: out std_logic_vector(size_outputs-1 downto 0)
103         );
104     end component;
105
106     component FaultRecognition
107         port (
108             Clock:     in std_logic;
109             Reset:     in std_logic;
110             Inputs:    in std_logic_vector( size_inputs - 1 downto 0 );
111             Outputs:   in std_logic_vector( size_outputs - \
112                                             1 downto 0 );
113             Recovered:in std_logic;
114             LastRecovery:in std_logic;
115             EInputs:   in std_logic;
116             Fault:     out std_logic;
117             C:         out std_logic_vector( size_recovery_method - \
118                                              1 downto 0 );
119             RecognitionReady:out std_logic;
120             Address:   out integer range 0 to ( size_memory-1 );
121             Din:       out std_logic_vector( size_fp_vector - \
122                                              1 downto 0 );
123             WEn, CEn: out std_logic;
124             Dout:      in std_logic_vector( size_fp_vector - \
125                                             1 downto 0 )
126         );
```

```
127     end component;
128
129     component SyncMemory
130         generic (
131             M: integer;
132             N: integer
133         );
134         port (
135             Address:in integer range 0 to ( size_memory-1 );
136             Din:    in std_logic_vector(size_fp_vector-1 downto 0);
137             WEn, CEn, Clock:in std_logic;
138             Dout:   out std_logic_vector(size_fp_vector-1 downto 0)
139         );
140     end component;
141
142     component RecoveryCounter
143         port (
144             Clock:        in std_logic;
145             Reset:        in std_logic;
146             EInputs:      in std_logic;
147             Recovered:    in std_logic;
148             LastRecovery:out std_logic
149         );
150     end component;
151
152     component RecoveryProcedure
153         port (
154             Clock:        in std_logic;
155             Reset:        in std_logic;
156             C:            in std_logic_vector(size_recovery_method- \
157                                               1 downto 0);
158             StartReconfiguration:out std_logic;
159             Reconfigured:in std_logic;
160             Inputs:       in std_logic_vector(size_inputs-1 downto 0);
161             OutputsCFSR: in std_logic_vector(size_outputs-1 downto 0);
162             Outputs:      out std_logic_vector(size_outputs\
163                                                -1 downto 0);
164             Fault:        in std_logic;
165             RecognitionReady:in std_logic;
166             LastRecovery:in std_logic;
167             Recovered:    out std_logic
168         );
169     end component;
170
171     component StateMachine
172         port  (
173             Clock:     in std_logic;
174             Reset:     in std_logic;
175             Inputs:    in std_logic_vector( size_inputs - 1 downto 0 );
176             Outputs:   in std_logic_vector( size_outputs - \
177                                             1 downto 0 );
178             Fault:     in std_logic;
179             LastRecovery:in std_logic;
```

```
180             RecognitionReady:in std_logic;
181             Recovered:in std_logic;
182             Ready:    out std_logic;
183             Defect:   out std_logic;
184             EInputs:  out std_logic;
185             EOutputs: out std_logic
186         );
187     end component;
188
189     signal StartReconfiguration: std_logic;
190     signal Reconfigured: std_logic;
191     signal Inputs: std_logic_vector( size_inputs - 1 downto 0 );
192     signal InputsFI: std_logic_vector( size_inputs - 1 downto 0 );
193     signal OutputsCFSR: std_logic_vector( size_outputs - 1 downto 0 );
194     signal OutputsRec: std_logic_vector( size_outputs - 1 downto 0 );
195     signal OutputsMut: std_logic_vector( size_outputs - 1 downto 0 );
196     signal OutputsFI: std_logic_vector( size_outputs - 1 downto 0 );
197     signal EInputs: std_logic;
198     signal EOutputs: std_logic;
199     signal Recovered: std_logic;
200     signal LastRecovery: std_logic;
201     signal Fault: std_logic;
202     signal C: std_logic_vector( size_recovery_method - 1 downto 0 );
203     signal RecognitionReady: std_logic;
204     signal Address: integer range 0 to ( size_memory-1 );
205     signal Din: std_logic_vector(size_fp_vector-1 downto 0);
206     signal WEn, CEn: std_logic;
207     signal Dout: std_logic_vector(size_fp_vector-1 downto 0);
208
209 begin
210     InputsCircuit <= InputsFI;
211     OutputsCFSR <= OutputsCircuit;
212
213     SPR: SelfPartialReconfigurator port map (SYSACE_CLK,Reset,\
214             StartReconfiguration,Reconfigured,SYSACE_MPCE,\
215             SYSACE_MPWE_USB_WR_N,\
216             SYSACE_MPOE_USB_RD_N,SYSACE_Address,\
217             SYSACE_Data,SYSACE_MPIRQ);
218     MUT: MutantsModule port map (Clock,Reset,Mutant,Injection,\
219             FaultRemoval,InputsFI,OutputsRec,OutputsMut);
220     SO: SaboteursOutModule port map (Clock,Reset,SaboteurOutputs,\
221            StuckAt1,Injection,FaultRemoval,OutputsMut,OutputsFI);
222     SI: SaboteursInModule port map (Clock,Reset,SaboteurInputs,\
223            StuckAt1,Injection,FaultRemoval,Inputs,InputsFI);
224
225     EI: EnableInputs port map (Clock,Reset,EInputs,InputsUser,Inputs);
226     EO: EnableOutputs port map (Clock,Reset,EOutputs,OutputsFI,\
227            OutputsUser);
228     FR: FaultRecognition port map (Clock,Reset,Inputs,OutputsFI,\
229            Recovered,LastRecovery,EInputs,Fault,C,RecognitionReady,\
230            Address,Din,WEn,CEn,Dout);
231     MEM: SyncMemory generic map (size_memory,size_fp_vector)\
232             port map (Address,Din,WEn,CEn,Clock,Dout);
```

```
233     RC: RecoveryCounter port map(Clock,Reset,EInputs,Recovered,\
234             LastRecovery);
235     RP: RecoveryProcedure port map (Clock,Reset,C,\
236             StartReconfiguration,Reconfigured,InputsFI,OutputsCFSR,\
237             OutputsRec,Fault,RecognitionReady,LastRecovery,Recovered);
238     SM: StateMachine port map (Clock,Reset,Inputs,OutputsFI,Fault,\
239             LastRecovery,RecognitionReady,Recovered,Ready,Defect,\
240             EInputs,EOutputs);
241
242 end Behavioral;
```

### Top architecture module

Finally, in the TopArchitecture module shown in code listing 6.16, the partial reconfigurable module CircuitForSR to be placed in a dynamic region, the module RecognizerRepairer to be placed on a static region, and the bus macros busmacro_xc4v_l2r_async_narrow and busmacro_xc4v_r2l_async_narrow are first declared and then instantiated with the names CFSR, RR, BMO and BMI respectively. The signals in code lines 107 to 110 connect the instantiated modules using positional association. Please note that the input ports which are not used in the BMI and BMI instances have the value of '1' and the outputs which are not used are left 'open'.

Program Code 6.16: Top architecture module

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity TopArchitecture is
6      port (
7          Clock:           in std_logic;
8          Reset:           in std_logic;
9          InputsUser:      in std_logic_vector(size_inputs-1 downto 0);
10         OutputsUser:     out std_logic_vector(size_outputs-1 downto 0);
11         Defect:          out std_logic;
12         Ready:           out std_logic;
13         Mutant:          in std_logic;
14         SaboteurInputs:  in std_logic;
15         SaboteurOutputs: in std_logic;
16         StuckAt1:        in std_logic;
17         Injection:       in std_logic;
18         FaultRemoval:    in std_logic;
19         SYSACE_CLK:      in std_logic;
20         SYSACE_MPCE:     out std_logic;
21         SYSACE_MPWE_USB_WR_N: out std_logic;
22         SYSACE_MPOE_USB_RD_N: out std_logic;
23         SYSACE_Address:  out std_logic_vector(6 downto 1);
24         SYSACE_Data:     out std_logic_vector(15 downto 0);
25         SYSACE_MPIRQ:    in std_logic
26     );
27 end TopArchitecture;
```

```
28
29 architecture Behavioral of TopArchitecture is
30     component busmacro_xc4v_l2r_async_narrow is
31         port (
32             input0: in std_logic;
33             input1: in std_logic;
34             input2: in std_logic;
35             input3: in std_logic;
36             input4: in std_logic;
37             input5: in std_logic;
38             input6: in std_logic;
39             input7: in std_logic;
40             output0: out std_logic;
41             output1: out std_logic;
42             output2: out std_logic;
43             output3: out std_logic;
44             output4: out std_logic;
45             output5: out std_logic;
46             output6: out std_logic;
47             output7: out std_logic
48         );
49     end component;
50
51     component busmacro_xc4v_r2l_async_narrow is
52         port (
53             input0: in std_logic;
54             input1: in std_logic;
55             input2: in std_logic;
56             input3: in std_logic;
57             input4: in std_logic;
58             input5: in std_logic;
59             input6: in std_logic;
60             input7: in std_logic;
61             output0: out std_logic;
62             output1: out std_logic;
63             output2: out std_logic;
64             output3: out std_logic;
65             output4: out std_logic;
66             output5: out std_logic;
67             output6: out std_logic;
68             output7: out std_logic
69         );
70     end component;
71
72     component CircuitForSR is
73         port (
74             Inputs: in std_logic_vector(size_inputs-1 downto 0);
75             Outputs:out std_logic_vector(size_outputs-1 downto 0)
76         );
77     end component;
78
79     component RecognizerRepairer is
80         port (
```

```
81              Clock:       in std_logic;
82              Reset:       in std_logic;
83              InputsUser:  in std_logic_vector(size_inputs-1 downto 0);
84              OutputsUser:out std_logic_vector(size_outputs-1 downto 0);
85              Defect:      out std_logic;
86              Ready:       out std_logic;
87              InputsCircuit:out std_logic_vector(size_inputs-1\
88                                                 downto 0);
89              OutputsCircuit:in std_logic_vector(size_outputs-1\
90                                                 downto 0);
91              Mutant:      in std_logic;
92              SaboteurInputs: in std_logic;
93              SaboteurOutputs:in std_logic;
94              StuckAt1:    in std_logic;
95              Injection:   in std_logic;
96              FaultRemoval:in std_logic;
97              SYSACE_CLK:  in std_logic;
98              SYSACE_MPCE:out std_logic;
99              SYSACE_MPWE_USB_WR_N:out std_logic;
100             SYSACE_MPOE_USB_RD_N:out std_logic;
101             SYSACE_Address: out std_logic_vector(6 downto 1);
102             SYSACE_Data: out std_logic_vector(15 downto 0);
103             SYSACE_MPIRQ:in std_logic
104         );
105     end component;
106
107     signal InputsRR: std_logic_vector(size_inputs-1 downto 0);
108     signal InputsFromBMI: std_logic_vector(size_inputs-1 downto 0);
109     signal OutputsToBMO: std_logic_vector(size_outputs-1 downto 0);
110     signal OutputsBMO: std_logic_vector(size_outputs-1 downto 0);
111
112 begin
113     CFSR: CircuitForSR port map (InputsFromBMI,OutputsToBMO);
114     RR: RecognizerRepairer port map (Clock,Reset,InputsUser,\
115             OutputsUser,Defect,Ready,\
116             InputsRR,OutputsBMO,\
117             Mutant,SaboteurInputs,SaboteurOutputs,StuckAt1,\
118             Injection,FaultRemoval,\
119             SYSACE_CLK,SYSACE_MPCE,SYSACE_MPWE_USB_WR_N,\
120             SYSACE_MPOE_USB_RD_N,SYSACE_Address,SYSACE_Data,\
121             SYSACE_MPIRQ);
122     BMI: busmacro_xc4v_l2r_async_narrow port map (InputsRR(0),\
123             InputsRR(1),InputsRR(2),'1','1','1','1','1',\
124             InputsFromBMI(0),InputsFromBMI(1),InputsFromBMI(2),\
125             open,open,open,open,open);
126     BMO: busmacro_xc4v_r2l_async_narrow port map (OutputsToBMO(0),\
127             '1','1','1','1','1','1','1',OutputsBMO(0),open,open,open,\
128             open,open,open,open);
129
130 end Behavioral;
```

## 6.2 Simulation of the self-repairing unit

The module at the top of the hierarchy for describing the self-repairing unit is the TopArchitecture module. Its inputs and outputs constitute the interface of the self-repairing unit outwards. For simulating the behavior of the self-repairing unit, the inputs of the TopArchitecture module can be stimulated by means of a test bench and executed by a simulator.

A test bench is also a VHDL module which has neither inputs nor outputs declared at its interface, but the module to be simulated declared and then instantiated. A test bench module produces stimuli for all input signals of the module to be simulated. A VHDL test bench is very useful for debugging any VHDL module. Test benches for all modules presented in this chapter have been created at the design time in order to debug them, however they will not be presented here. A test bench works together with a simulator that imitates the behavior of the circuit described in the module to be simulated. A simulator allows to observe in a command-line interface and/or a graphical user interface the output signals of the module to be simulated. ISIM, a simulator from Xilinx, allows to see the input and output signals at the interface, the internal signals and the variables of the module to be simulated.

The module named TopArchitectureTB, shown in code listing 6.17, has been created as test bench of the module TopArchitecture in order to verify the behavior of the self-repairing unit, the fault injector and the partial reconfiguration procedure. In the test bench TopArchitectureTB, the module TopArchitecture has been firstly declared as a component an then it has been instantiated, as can be seen in the code lines 9 to 29 and 53 to 59. Please note that all its inputs and outputs had also to be declared as signals, as can be seen in code lines 31 to 48. In code lines 50 and 61, the signal *Reconfigured* has been declared and then its value has been assigned to the signal *SYSACE_MPIRQ* in order to handle with a more clear term and to deploy a simulation independent of the development platform being used.

Program Code 6.17: Top architecture test bench

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.Constants.all;
4
5  entity TopArchitectureTB is
6  end TopArchitectureTB;
7
8  architecture Behavioral of TopArchitectureTB is
9      component TopArchitecture is
10     port (
11         Clock:           in std_logic;
12         Reset:           in std_logic;
13         InputsUser:      in std_logic_vector(size_inputs-1 downto 0);
14         OutputsUser:     out std_logic_vector(size_outputs-1 downto 0);
15         Defect:          out std_logic;
16         Ready:           out std_logic;
17         Mutant:          in std_logic;
18         SaboteurInputs:  in std_logic;
19         SaboteurOutputs: in std_logic;
20         StuckAt1:        in std_logic;
21         Injection:       in std_logic;
22         FaultRemoval:    in std_logic;
```

```
23          SYSACE_MPCE:     out std_logic;
24          SYSACE_MPWE_USB_WR_N:out std_logic;
25          SYSACE_MPOE_USB_RD_N:out std_logic;
26          SYSACE_Address: out std_logic_vector(6 downto 1);
27          SYSACE_Data:     out std_logic_vector(15 downto 0);
28          SYSACE_MPIRQ:    in std_logic
29          );
30      end component;
31      signal Clock: std_logic:='0';
32      signal Reset: std_logic;
33      signal InputsUser: std_logic_vector(size_inputs-1 downto 0);
34      signal OutputsUser: std_logic_vector(size_outputs-1 downto 0);
35      signal Defect: std_logic;
36      signal Ready: std_logic;
37      signal Mutant: std_logic;
38      signal SaboteurInputs: std_logic;
39      signal SaboteurOutputs: std_logic;
40      signal StuckAt1: std_logic;
41      signal Injection: std_logic;
42      signal FaultRemoval: std_logic;
43      signal SYSACE_MPCE: std_logic;
44      signal SYSACE_MPWE_USB_WR_N: std_logic;
45      signal SYSACE_MPOE_USB_RD_N: std_logic;
46      signal SYSACE_Address: std_logic_vector(6 downto 1);
47      signal SYSACE_Data: std_logic_vector(15 downto 0);
48      signal SYSACE_MPIRQ: std_logic;
49
50      signal Reconfigured: std_logic;
51
52  begin
53      TA: TopArchitecture port map (Clock,Reset,\
54              InputsUser,OutputsUser,\
55              Defect,Ready,\
56              Mutant,SaboteurInputs,SaboteurOutputs,StuckAt1,\
57              Injection,FaultRemoval,\
58              SYSACE_MPCE,SYSACE_MPWE_USB_WR_N,SYSACE_MPOE_USB_RD_N,\
59              SYSACE_Address,SYSACE_Data,SYSACE_MPIRQ);
60
61      SYSACE_MPIRQ <= Reconfigured;
62
63      Clock <= not Clock after 0.1ns;
64
65      STIMULUS: process
66      begin
67
68  -- initial values (faulty circuit)
69        Reset <= '0';
70        InputsUser <= "010";
71        Mutant <= '0';
72        SaboteurInputs <= '0';
73        SaboteurOutputs <= '0';
74        StuckAt1 <= '0';
75        Injection <= '0';
```

```
76          FaultRemoval <= '0';
77          Reconfigured <= '0';
78          wait for 16ns;
79
80      -- reset
81          Reset <= '1';
82          wait for 2ns;
83          Reset <= '0';
84          wait for 16ns;
85
86      -- change of inputs (first recovery)
87          InputsUser <= "111";
88          wait for 33ns;
89
90      -- saboteur inputs
91          SaboteurInputs <= '1';
92          StuckAt1 <= '0';
93          wait for 3ns;
94          Injection <= '1';
95          wait for 2ns;
96          Injection <= '0';
97          wait for 3ns;
98          FaultRemoval <= '1';
99          wait for 2ns;
100         FaultRemoval <= '0';
101         wait for 3ns;
102         SaboteurInputs <= '0';
103         wait for 10ns;
104         Reset <= '1';
105         wait for 2ns;
106         Reset <= '0';
107         wait for 20ns;
108
109     -- saboteur outputs
110         SaboteurOutputs <= '1';
111         StuckAt1 <= '0';
112         wait for 3ns;
113         Injection <= '1';
114         wait for 2ns;
115         Injection <= '0';
116         wait for 13ns;
117         SaboteurOutputs <= '0';
118         wait for 10ns;
119         Reset <= '1';
120         wait for 2ns;
121         Reset <= '0';
122         wait for 18ns;
123
124     -- mutant
125         Mutant <= '1';
126         wait for 3ns;
127         Injection <= '1';
128         wait for 2ns;
```

```
129         Injection <= '0';
130         wait for 10ns;
131         FaultRemoval <= '1';
132         wait for 2ns;
133         FaultRemoval <= '0';
134         wait for 3ns;
135         Mutant <= '0';
136         wait for 10ns;
137         Reset <= '1';
138         wait for 2ns;
139         Reset <= '0';
140         wait for 12ns;
141
142   -- partial reconfiguration
143         InputsUser <= "011";
144         wait for 5ns;
145         SaboteurOutputs <= '1';
146         StuckAt1 <= '0';
147         wait for 3ns;
148         Injection <= '1';
149         wait for 2ns;
150         Injection <= '0';
151         wait for 5ns;
152         Reconfigured <= '1';
153         wait for 2ns;
154         Reconfigured <= '0';
155         wait for 10ns;
156         FaultRemoval <= '1';
157         wait for 2ns;
158         FaultRemoval <= '0';
159         wait for 10ns;
160         Reconfigured <= '1';
161         wait for 2ns;
162         Reconfigured <= '0';
163         wait for 3ns;
164         SaboteurOutputs <= '0';
165         wait for 8ns;
166
167         wait;
168       end process;
169
170   end;
```

In code line 63 of code listing 6.17 a clock signal with a frequency of 50 MHz is produced. That frequency is obtained because that signal switches from '0' to '1' and vice versa each nanosecond, producing a signal with period of 2 nanoseconds. Then, inverting the period, a frequency of 50 MHz is obtained as follows: Frequency $= \frac{1}{\text{Period}} = \frac{1}{2\ \text{ns}} = 50$ MHz.

The process STIMULUS shown in code line 65 contains the sequential assertion of the input signals to desired values. Once the input signals are asserted to initial values, as can be seen in code lines 69 to 77, the VHDL construct wait for, shown in code line 78, serves for having asserted that inputs to a determined value a time length, in this case 3 nanoseconds, before the inputs are asserted to a new value.

Table 6.1: Circuit for self-repairing truth table

| C | B | A | $\overline{C}$ | $\overline{C} \cdot B$ | $C \cdot A$ | $(\overline{C} \cdot B) + (C \cdot A)$ | $(C \cdot A) + 0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

The assertions in code lines 68 to 165 have been done in the way to verify the operation of the self-repairing circuit, the operation of the fault injector and the recovery mechanism using reconfiguration. Running with the ISIM simulator the test bench presented in code listing 6.17, the simulation shown in figure 6.13 is obtained. Please take in mind that by running the ISIM simulator under Linux, it has been necessary to bypass the bus macros in the TopArchitecture module, since the bus macro files could not be interpreted by the ISIM simulator correctly giving out undefined signal values by the simulation.

The circuit for self-repairing has the Boolean formula $(\overline{C} \cdot B) + (C \cdot A)$, whose gate diagram and truth table is shown in figure 6.14 and table 6.1 respectively. In code line 70 of code listing 6.17 the *InputsUser* are asserted to the initial value of '010'. A faulty circuit for self-repairing with an stuck-at-0 at the output of the AND gate $C \cdot A$, is present at the very beginning. In the simulation shown in figure 6.13 the output *OutputsUser* has a correct value of '1' since for a non-faulty as for a faulty circuit for self-repairing the output is the same, as can be seen in the truth table 6.1. Therefore, the stuck-at-0 fault at the output of the AND gate $C \cdot A$ is not an active fault and can be called a dormant fault, because it produces a latent error which can not be detected, those terms has been taken from [Avižienis et al., 1992]. Please note that the signal *Ready* goes down at the beginning. That signal goes down when the module FaultRecognition checks whether the outputs of the circuit for self-repairing are correct.

In code line 81 of code listing 6.17, the *Reset* signal is asserted to '1'. A reset triggers also a fault recognition which happens when the signal *Ready* goes down. After the signal *Reset* is set to '1', fault recognition is triggered. During that time, the signals at *OutputsUser* are neither driven to a logical '1' not to a logical '0', state named as floating or of high impedance because the signals have high voltage and allow to pass very small amount of current through, please see [Wikipedia, 2012]. That state is marked by a 'Z' at the simulation shown in figure 6.13.

In code line 87 of code listing 6.17, the signal *InputsUser* are changed to the binary value '111'. A change at the inputs of the circuit for self-repairing triggers fault recognition shown by a low signal at Ready. Since for that inputs the dormant stuck-at-0 fault at the output of the AND gate $(C \cdot A) + 0$ becomes active, a fault is found by the FaultRecognition module as can be seen by the signal *Fault* in figure 6.13. Then, recovery is performed by the module which description has been shown in code listing 6.7 replacing the circuit for self-repairing by the redundant circuit CircuitForSRX, which is fault free. The signal *Recovered*, shows that the recovery is finished. The signal *Ready* goes again up showing that the recovery has been

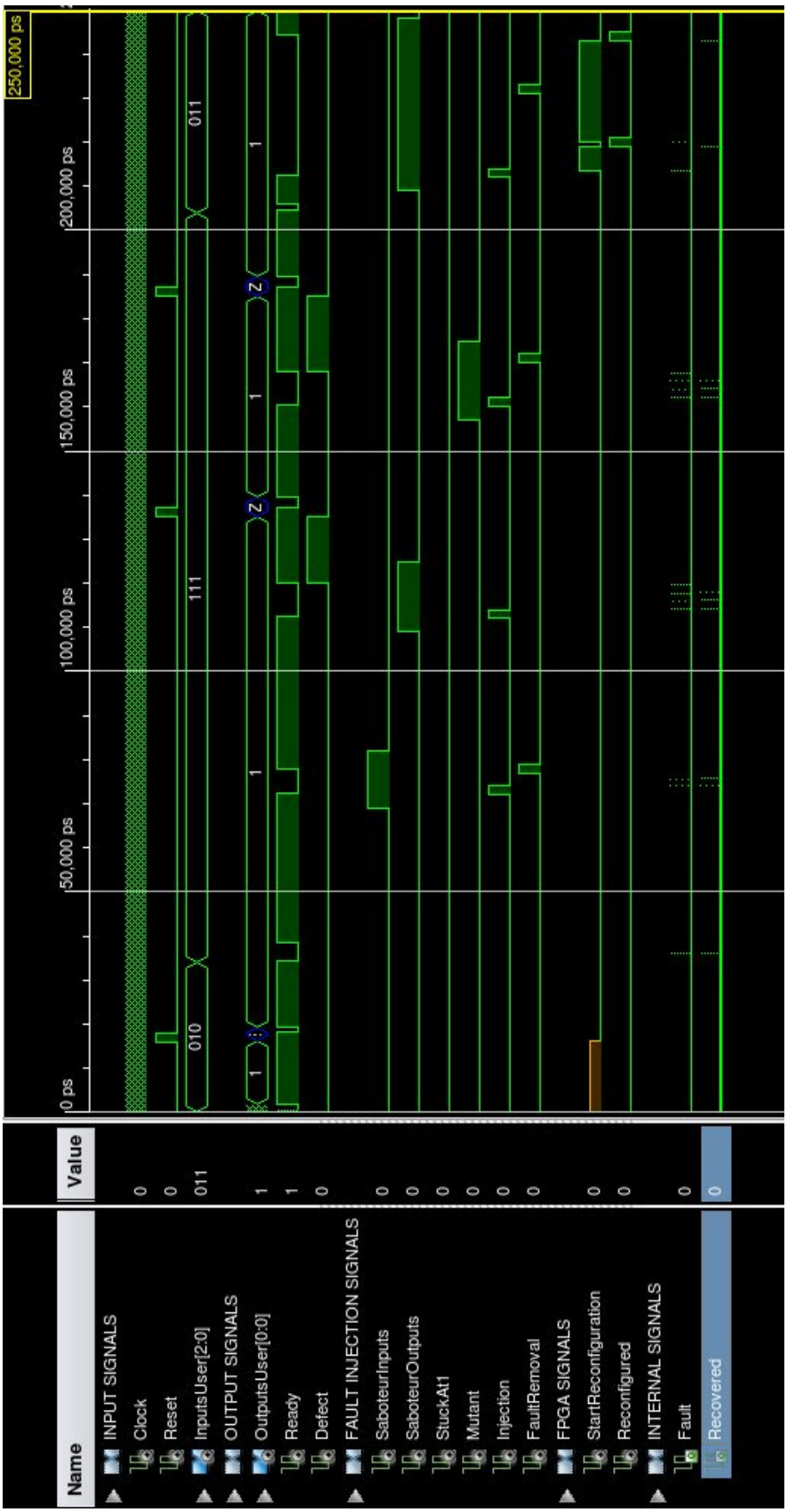



Figure 6.13: Simulation of the self-repairing circuit

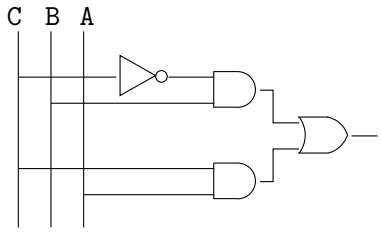



Figure 6.14: Circuit for self-repairing

Table 6.2: Fault vectors with recovery mechanisms

| Self | Recovery procedure | Inputs | Outputs |
|---|---|---|---|
| 0 | 110 | 000 | 1 |
| 0 | 110 | 001 | 1 |
| 0 | 110 | 010 | 0 |
| 0 | 101 | 011 | 0 |
| 0 | 011 | 100 | 1 |
| 0 | 011 | 101 | 0 |
| 0 | 011 | 110 | 1 |
| 0 | 011 | 111 | 0 |
| 1 | 110 | 000 | 0 |
| 1 | 110 | 001 | 0 |
| 1 | 110 | 010 | 1 |
| 1 | 101 | 011 | 1 |
| 1 | 011 | 100 | 0 |
| 1 | 011 | 101 | 1 |
| 1 | 011 | 110 | 0 |
| 1 | 011 | 111 | 1 |

Recovery procedure '101' means partial reconfiguration.
Recovery procedure '110' and '011' mean redundancy.

successful since a new fault recognition finds no faults anymore. The recovery procedure using a redundant module has been selected by the RecoveryProcedure module based on the field corresponding to the recovery procedure saved on the fault vectors in the constants file Constants presented in subsection 6.1.1 and shown in table 6.2.

The simulation from 50 000 ps to 100 000 ps, shown in figure 6.13, intends to show the behavior of the self-repairing circuit by inserting a stuck-at-0 fault at one of its inputs using a saboteur as fault injector. For that, the signal *SaboteurInputs* is set to '1' and the fault is injected by rising the signal *Injection*. That fault is detected as can be seen at the internal signal *Fault* and then recovered, as can be seen at the internal signal *Recovered*. The fault is removed after the second recovery rising the signal *FaultRemoval*. Therefore, a third recovery is not executed and the signal *Ready* becomes the value of '1' again.

The simulation from 100 000 ps to 150 000 ps shown in figure 6.13, shows the behavior of the self-repairing circuit by inserting a stuck-at-0 fault at one of its outputs using also a saboteur as fault injector. Here, the signal *SaboteurOutputs* is set to '1' and the fault is

injected by rising the signal *Injection*. That fault is detected as can be seen at the internal signal *Fault* and then recovered, as can be seen at the internal signal *Recovered*. The fault has been recovered three times, but since a fault is found a fourth time, the signal *Defect* is raised to '1'. The injected fault can be removed in the simulation, only by rising to '1' the signal *Reset*.

The simulation from 150 000 ps to 200 000 ps shown in figure 6.13, shows the behavior of the self-repairing circuit by inserting a fault into the circuit using as fault injector a mutant of that circuit. The mutant delivers an output of '0' in any case. For that, the signal *Mutant* is set to '1' and the fault is injected by rising the signal *Injection*. That fault is detected as can be seen at the internal signal *Fault* and then recovered, as can be seen at the internal signal *Recovered*. The fault is recovered three times, but since a fault is found a fourth time, the signal *Defect* is raised to '1'. The injected fault can be removed in this case, only by rising to '1' the signal *Reset*.

The simulation from 200 000 ps to 250 000 ps shown in figure 6.13, shows how the circuit can be recovered by using partial reconfiguration. Partial reconfiguration is the recovery mechanism when finding a fault under inputs '011', and has been encoded to the binary value of '101' and stored in memory as shown in table 6.2 and code listings 6.9 and 6.6. The recovery procedure '101' triggers partial reconfiguration in the module RecoveryProcedure setting to '1' the signal *StartReconfiguration*. For the simulation, a fault at the outputs is introduced by means of a saboteur. The fault is recognized and then the recovery mechanism of partial reconfiguration is triggered. The module RecoveryProcedure waits for the signal *Reconfigured*, which is set to '1' thereafter. However, since the fault is not removed, the module FaultRecognition finds the fault again, and the recovery by means of partial reconfiguration is triggered again. Before the signal *Reconfigured* is raised, the fault is removed by setting to '1' the signal *FaultRemoval*, as can be seen in code line 156 of code listing 6.17. Therefore, after the signal *Reconfigured* is raised again, the signal *Ready* is raised indicating that the system has been successfully recovered and the outputs for the given inputs are correct.

## 6.3 Implementation of the self-repairing unit

Once a successful simulation of the self-repairing circuit has been executed, its implementation in hardware can be started. The implementation of a VHDL design on a Xilinx FPGA requires: to synthesize the VHDL top module using the **XST** tool for creating a Xilinx-specific netlist file with the extension `.ngc`, taken from the term Native Generic Circuit; to create a constraints file of the top module with the extension `.ucf`, taken from the term User Constraints File, using a text editor or the tools **Floorplanner** or **PlanAhead**; to translate the netlist file to a file with extension `.ngd`, taken from the term Native Generic Database, that contains a logical description of the design using AND/OR gates, decoders, flip-flops and RAMs, using the **NGDBuild** program that requires as inputs the busmacro files with extension `.nmc` and the created constraints file with the extension `.ucf`; to map the logical description of the design to the FPGA hardware using the **MAP** program, which gives as output a file with extension `.ncd`, taken from the term Native Circuit Description; to place and route the mapped design using the **PAR** program getting a file with the same extension `.ncd`; to generate a bitstream for the Xilinx FPGA configuration using the **BITGen** program getting a binary file with the extension `.bit`; and to download the binary file into the FPGA memory cells using the **iMPACT** configuration tool. For more information please see the

Xilinx Development System Reference Guide [Xilinx, 2005] or its updated Command Line Tools User Guide [Xilinx, 2009a] and the XST User Guide [Xilinx, 2009f].

The implementation of a VHDL design considering partial reconfiguration requires to execute a different implementation flow that entails to synthesize, translate, map, place and route the static, reconfigurable and top modules separately and then merge them together for generating the binary files for being downloaded into the FPGA. For executing that implementation flow, the program **Early-Access** form Xilinx is required. Then it is necessary to prepare a by Xilinx recommended project directory structure containing the folders: non_pr with subdirectories for implementing non-partial reconfigurable versions of the design for testing its functioning before inserting partial reconfiguration; synth with subdirectories for synthesizing separately the top, static and reconfigurable modules; prm with subdirectories for the translation, map, place and route of each partial reconfigurable module; static for the translation, map, place and route of the static modules; top for the translation of the top module; merges with subdirectories for the verification and assemble of the static modules with each of the partial reconfigurable modules, step which gives the binary files out for downloading into the FPGA; and finally and option folder data containing the VHDL files of all modules and the busmacro files. For more information please see the Early Access Partial Reconfiguration User Guide [Xilinx, 2008], the Command Line PR Implementation document [Xilinx, 2006a], the Partial reconfiguration Design with PlanAhead manual [Xilinx, 2007] or the newer updated partial Reconfiguration User Guide [Xilinx, 2010].

For simplifying the whole implementation process, the architecture for self-repairing has been conceived to contain a single static module, the RecognizerRepairer module, and a single partial reconfigurable module, the CircuitForSR module. Then, the top module has those two modules instantiated with the names RR and CFSR, and the bus macros with the names BMI and BMO. In the user constraints file shown in code listing 6.18, which has been created with a text editor, location constraints have been declared. First of all the signal *Clock* has been connected to the SYSCLK having the label AE14, as can be seen in the code line 1, which is the output of a crystal oscillator available in the development board and gives a signal of 100 MHz, please see the documents [Xilinx, 2006c] and [Xilinx, 2004] for more information. The inputs and outputs of the circuit for self-repairing, the *Reset*, *Defect* and *Ready* signals, and the signals for fault injection have been located in the way of having them connected to the available I/O pins of the expansion I/O connector J6 of the development board documented in [Xilinx, 2006c], please see code lines 3 to 15 of code listing 6.18. The three line *Inputs*, *StuckAt1*, *SaboteursInputs*, *SaboteursOutputs*, and *Mutant* signals, have been connected externally to toggle switches in a breadboard, please see figure 6.15; the *Reset*, *FaultRemoval* and *Injection* signals, have been connected externally to push-button switches; and the one line *Outputs*, *Defect* and *Ready* signals have been connected externally to leds lying on the breadboard. The inputs and outputs serving for the execution of partial reconfiguration are located to the corresponding FPGA I/O pins which connect to the System ACE controller in the development board, shown in the schematics of the development board [Xilinx, 2004] and declared in code lines 17 to 43 of code listing 6.18.

In order to separate the FPGA resources used for the static region from the resources used for the dynamic region, physical constraints for mapping, placement and routing are necessary. That is possible using the AREA_GROUP constraint which allows to partition the design into physical regions by declaring labels for each region, for more details please refer to [Xilinx, 2009b] and [Xilinx, 2008]. It is demanding to declare AREA_GROUP constraints for each reconfigurable region. Therefore, the instance CFSR of the circuit for self-repairing

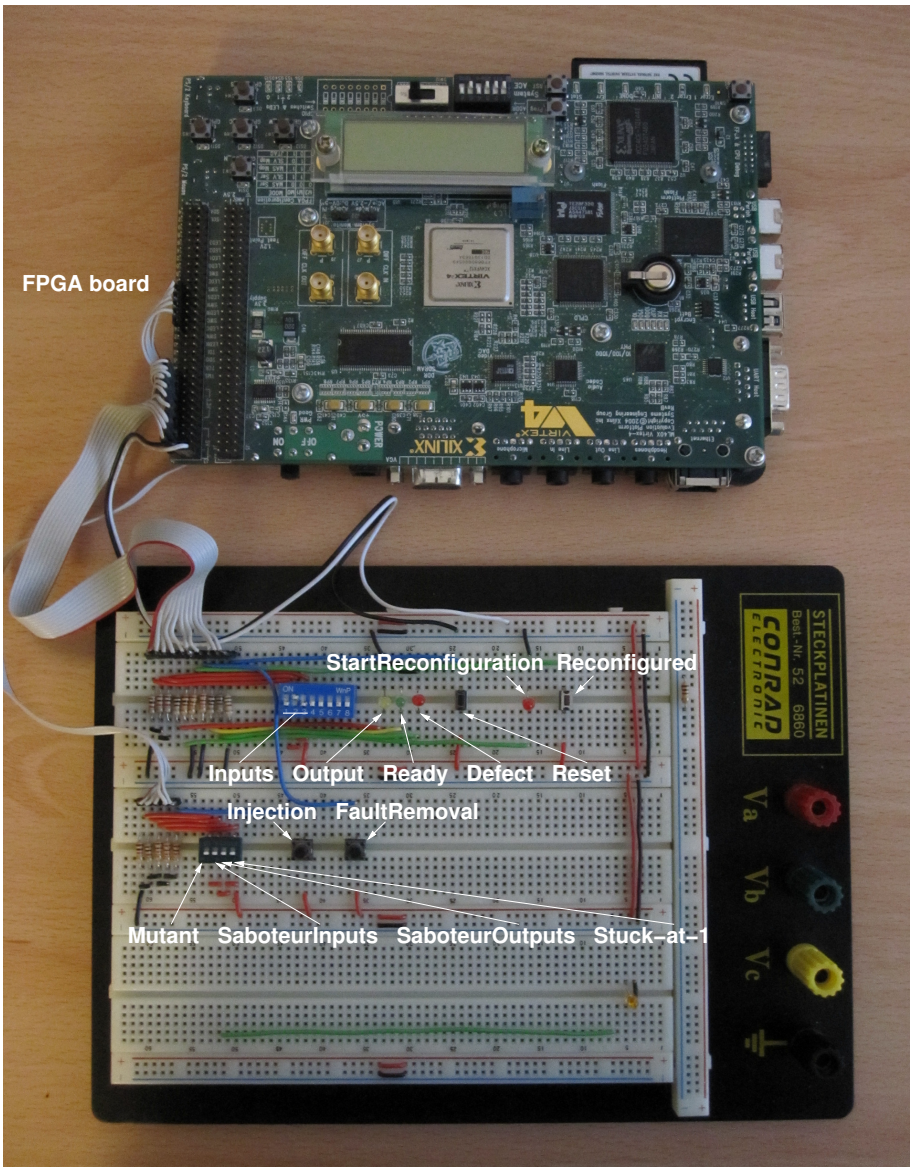



Figure 6.15: FPGA board and breadboard for the hardware implementation

has been declared as the area group AG_CFSR, for which the physical range of slices with the lower left corner X26Y0 and upper right corner X45Y125 has been reserved, as can be seen in code line 49. That has been possible thanks to the a graph of the FPGA which can be produced by the program **PlanAhead**. Slices contain look-up tables and flip-flops, if RAM memory, multipliers or other kind of logic is required in the reconfigurable region, constraints for that logic are also necessary. Since the circuit for self-repairing is a simple combinational circuit, the RANGE declaration for slices is enough. Now, the instances of the bus macros BMI and BMO which connect the inputs and outputs of the circuit for self-repairing with the static part of the design are located physically in the FPGA to the slices with coordinates X24Y122 and X24Y120 respectively, as can be see in code lines 52 and 53. The mode of the area group AG_CFSR is declared with RECONFIG for preventing that the **NGDBuild** reports errors at the time of translating the design [Xilinx, 2008], as can be seen in code line 50. The instance RR of the module ReconfiguratorRepairer to be placed in a static region has been declared as the area group AG_RR in code line 55. However, no information about a defined placement into the FPGA for that area group is provided in this constraints file since it is not necessary for a static region, [Xilinx, 2008].

Program Code 6.18: Constrains file

```
1  NET "Clock"  LOC = "AE14";
2
3  NET "Reset" LOC = "AA24";
4  NET "Defect" LOC = "V20";
5  NET "Ready" LOC = "AC25";
6  NET "FaultRemoval" LOC = "AC24";
7  NET "OutputsUser(0)" LOC = "Y24";
8  NET "InputsUser(0)" LOC = "Y26";
9  NET "InputsUser(1)" LOC = "W26";
10 NET "InputsUser(2)" LOC = "AB23";
11 NET "Injection" LOC = "AB25";
12 NET "StuckAt1" LOC = "AD23";
13 NET "SaboteurOutputs" LOC = "AC26";
14 NET "SaboteurInputs" LOC = "AD26";
15 NET "Mutant" LOC = "AC22";
16
17 NET "SYSACE_CLK" LOC = "AF11";
18 NET "SYSACE_MPCE" LOC = "AD5";
19 NET "SYSACE_MPWE_USB_WR_N" LOC = "Y8";
20 NET "SYSACE_MPOE_USB_RD_N" LOC = "AA8";
21 NET "SYSACE_MPIRQ" LOC = "AD4";
22 NET "SYSACE_Address(1)" LOC = "Y10";
23 NET "SYSACE_Address(2)" LOC = "AA10";
24 NET "SYSACE_Address(3)" LOC = "AC7";
25 NET "SYSACE_Address(4)" LOC = "Y7";
26 NET "SYSACE_Address(5)" LOC = "AA9";
27 NET "SYSACE_Address(6)" LOC = "Y9";
28 NET "SYSACE_Data(0)" LOC = "AB7";
29 NET "SYSACE_Data(1)" LOC = "AC9";
30 NET "SYSACE_Data(2)" LOC = "AB9";
31 NET "SYSACE_Data(3)" LOC = "AE6";
32 NET "SYSACE_Data(4)" LOC = "AD6";
```

```
33 NET "SYSACE_Data(5)" LOC = "AF9";
34 NET "SYSACE_Data(6)" LOC = "AE9";
35 NET "SYSACE_Data(7)" LOC = "AD8";
36 NET "SYSACE_Data(8)" LOC = "AC8";
37 NET "SYSACE_Data(9)" LOC = "AF4";
38 NET "SYSACE_Data(10)" LOC = "AE4";
39 NET "SYSACE_Data(11)" LOC = "AD3";
40 NET "SYSACE_Data(12)" LOC = "AC3";
41 NET "SYSACE_Data(13)" LOC = "AF6";
42 NET "SYSACE_Data(14)" LOC = "AF5";
43 NET "SYSACE_Data(15)" LOC = "AA7";
44 % NET "FPGA_DONE" LOC = "H14";
45 % NET "FPGA_PROG_B" LOC = "H15";
46 % NET "FPGA_INIT" LOC = "G15";
47
48 INST "CFSR" AREA_GROUP = "AG_CFSR";
49 AREA_GROUP "AG_CFSR" RANGE = SLICE_X26Y0:SLICE_X45Y125;
50 AREA_GROUP "AG_CFSR" MODE = RECONFIG;
51
52 INST "BMI" LOC = "SLICE_X24Y122";
53 INST "BMO" LOC = "SLICE_X24Y120";
54
55 INST "RR" AREA_GROUP = "AG_RR";
```

During synthesis it is advisable to select a Hamming-3 encoding for the states of state machines. This feature is available in the Synopsis tools. It allows the automatic detection and correction of a single bit in the states of the state machine. The error detection and correction circuitry is automatically added by the tool. For more information please see [Sutton, 2012]. The XST synthesis tool from Xilinx does not include such a feature. Instead, it has an option for selecting the type of state machine encoding that the synthesis tool should use. Among others, the Gray encoding is available, which allows to have glitchless state machine output signals, or the one-hot encoding which allows to get a faster circuit using a flip-flop per state. Using a single flip-flop per state in a state machine sometimes reduces the circuitry of the next state and output logic, reason why the circuit delivers the outputs faster, please see [Brown and Vranesic, 2005]. The encoding of the states of the state machine can be also specified explicitly in the VHDL design declaring the states as constants with binary values reflecting the desired encoding, for more information please refer to [Pellerin and Taylor, 1996]. As an alternative, the synthesis program **XST** from Xilinx offers an option named safe_implementation, which makes the synthesis program to implement the state machines in safe mode. Safe mode means that if the state machine enters in an invalid state, additional logic forces the state machine to go to a safe recovery state which can be defined previously in the VHDL design or can be the reset state taken by the synthesis program as the recovery state, for more information please see [Xilinx, 2009f]. The design has been synthesized with the safe implementation mode enabled and with the Gray encoding for the states of the state machine.

The synthesis of the RecognizerRepairer module reported a resource consumption of 154 slices, 158 slice flip flops and 260 4-inputs LUTs. The modules CircuitForSR and TopArchitecture has been also synthesized. And then all modules have been translated, mapped, placed and routed separately and then merged, following the implementation of a VHDL design considering partial reconfiguration described above. The binary files of the static region and the

partial reconfigurable modules for the dynamic region are created during the merging process that uses the programs **PR_verifydesign** and **PR_assemble** of the Early Access suite. Those programs are run using the static module with each one of the partial reconfigurable modules. The obtained binary files can be downloaded into the FPGA by using the **iMPACT** program. The self-partial reconfiguration has been planned so that the FPGA gets the partial bit files of the circuit to be partially reconfigured from a CompactFlash card inserted in the development board. In that case, the partial bit files have to be converted to .ace files by means of the **GenACE** program which usage is described in the Embedded System Tools Reference Manual from Xilinx [Xilinx, 2009c]. Since the self-partial reconfiguration could not be tested due to bugs in hardware of the development platform, the partial reconfiguration has been executed using the **iMPACT** program each time the *StartReconfiguration* signal was raised and visualized through a led wired in a breadboard. After the partial reconfiguring of the circuit, the *Reconfigured* signal has been raised using an externally wired push-button. The self-repairing feature of the self-repairing circuit could be successfully tested completely using redundancy as verified in the simulation.

## 6.4 Performance of the self-repairing unit

The performance of the self-repairing circuit can be measured by the Mean Time To Recognize a Fault plus the Mean Time To Recovery as shown graphically in figure 6.16. The MTTRF is the time since a fault is present in the circuit until it is recognized. It is dependent on the place on which a fault pattern vector that recognizes that fault is present in the fault pattern vector set in memory. When many fault pattern vectors recognize that fault, the place of the first fault pattern vector that recognizes that fault determines the time when the fault is going to be recognized. The worst case is when the first fault pattern vector that recognizes a fault is placed at the end of the fault pattern vector set in memory. Therefore, the number of fault pattern vectors in the fault pattern vector set gives the worst time to recognize a fault. If the fault pattern vector set is compact and has a good fault coverage, the fault can be recognized faster and the fault recovery can take place soon thereafter. It would be advisable to place the fault pattern vectors that recognize critical faults at the beginning of the set in memory in the way the respective critical faults are recognized earlier. For example the recovery mechanism for a stuck-at-0 fault at the output of the AND gate $\mathtt{C} \cdot \mathtt{A}$ of the tiny circuit taken as example is placed in third place in memory, as can be seen in table 6.2. Fact that reduces the MTTRF for that fault.

Another approach can be to use a Content-Addressable Memory, in short CAM, for storing the fault pattern vectors. By such a memory, it is not given a memory address for reading a data word at that position in the memory, instead a data word is provided for being matched with the data stored in the memory and then the address of the data that matches the given data word is given as output. That kind of memory is expensive since comparators are necessary, the more comparators are used in parallel the faster the memory gives the address of the matching word. Content-Addressable Memory can be designed in FPGAs as described in the application notes of Xilinx [Xilinx, 2000] and Actel [ACTEL, 2003]. The programmable logic devices APEX from Altera for Systems-on-a-Programmable-Chip provide dedicated circuitry for CAMs [Altera, 1999]. However, the size of the memory is limited by the resources of the FPGA or the available dedicated circuitry. Fact that justify the reduction of the number of fault pattern vectors and their dimension that are going to be stored in

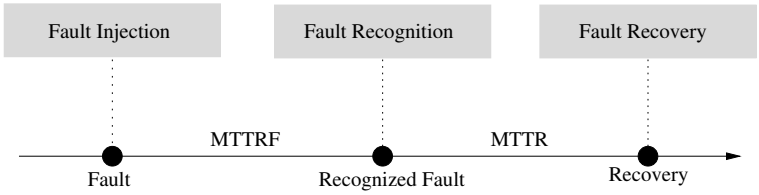



Figure 6.16: Performance measurement

memory for fault recognition.

Such a parallel comparison of the input vector with the input pattern vectors has been taken place in the input vectors monitoring block shown in figure 5.22 for a combinational circuit for self-repairing. In that case, the fault recognition is executed concurrently and the MTTRF is equal to the time the combinational logic for the output vector compaction, if present, and the output vectors comparison takes for reporting a fault. It can be assumed that the fault recognition designed in that way is the fastest and the most resource efficient. However such a design is limited to fault vectors with binary elements and does not comprise fault recovery assignment. Fault recognition by system on chips controlled by microcontrollers or microprocessors could apply associative arrays or hashing techniques implemented in software, please see [Wikipedia, 2012], which are usually supported again by a hardware Content-Addressable Memory.

In [Sharma and Saluja, 1988] the computation of the time required for test completion is presented. That time is the time required for testing a circuit with the whole set of available testing vectors in memory. Computing that time for a self-repairing circuit is not the main objective as it is by self-testing circuits. In a self-repairing circuit, important is to perform fault recovery when a single fault is recognized by means of the stored set of fault pattern vectors. Reason why only active faults can be recognized and dormant ones not.

The Mean Time to Recovery shown in figure 6.16 depends on the available recovery methods. The time partial reconfiguration takes depends on the type of FPGA, e.g. Virtex 4, the configuration mode and interface used for reconfiguring that FPGA [Xilinx, 2009e], e.g. JTAG, the type of memory from where the partial bitstream is taken, e.g. CompactFlash, and the size of the partial bitstream which is proportional to the size of the circuit to be reconfigured. By using redundancy for recovery, switching to a redundant circuit requires less time than partial reconfiguration, however, it demands resources of the FPGA, increasing in that way the resource consumption of the self-repairing circuit.

## 6.5 Conclusions

This chapter presented in detail the design and implementation of the framework for self-repairing a unit having taken as an example a tiny combinational circuit. The resource consumption of the RecognizerRepairer module is a parameter which should not vary much when being used for larger circuits. The self-repairing design, presented in this chapter can be used as a template for other circuits, which has been the intention of this thesis. Due to bugs in the available development board, partial reconfiguration has been proved to work but self-partial reconfiguration could not completely be debugged and demonstrated functioning. The self-repairing functionality of the framework using redundancy has been fully proved to

work.

## 6.6 Bibliography

ACTEL (2003). *Content-Addressable Memory (CAM) in Actel Devices*. Application Note AC194.

Altera (1999). *CAM Comparison: APEX 20KE vs. Virtex-E Devices*. Technical Brief 61.

Avižienis, A., Kopetz, H., and Laprie, J. C., editors (1992). *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer.

Baraza, J. C., Gracia, J., Gil, D., and Gil, P. J. (2005). Improvement of Fault Injection Techniques Based on VHDL Code Modification. In *10th International Conference on High-Level Design Validation and Test Workshop*, pages 19–26. IEEE.

Becker, M., Kuznik, C., Joy, M. M., Xie, T., and Mueller, W. (2012). Binary Mutation Testing Through Dynamic Translation. In *42nd International Conference on Dependable Systems and Networks - DSN 2012*, pages 1–12. IEEE.

Brown, S. and Vranesic, Z. (2005). *Fundamentals of Digital Logic with VHDL Design*. Mcgraw-Hill, second edition.

Burke, G. and Taft, S. (2004). Fault Tolerant State Machines. In *MALPD 2004*.

Chakraborty, T. J. and Chiang, C.-H. (2002). A novel fault injection method for system verification based on FPGA boundary scan architecture. In *International Test Conference - ITC 2002*, pages 923–929.

Christoph Steiger, H. W., Platzner, M., and Thiele, L. (2003). Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *24th Real-Time Systems Symposium - RTSS 2003*, pages 224–225. IEEE.

Chu, P. P. (2006). *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. John Wiley & Sons.

Dibaj, P. (2010). Hardware Fault Recognition Unit. Bachelor's thesis, University of Paderborn.

Dittmann, F. (2008). *Methods to Exploit Reconfigurable Fabrics*. PhD thesis, University of Paderborn.

Dye, D. (2012). Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite. Xilinx White Paper: Virtex-4, Virtex-5, Virtex-6, and 7 Series FPGAs.

Hosseinimehr, M. (2010). Implementation of a Singular Value Decomposition Module on an FPGA. Master's thesis, University of Paderborn.

Jenn, E., Arlat, J., Rimbn, M., Ohlsson, J., and Karlsson, J. (1994). Fault Injection into VHDL Models: The MEFISTO Tool. In *24th International Symposium on Fault-Tolerant Computing - FTCS 24*, pages 66–75.

Kastensmidt, F. L., Carro, L., and Reis, R. (2006). *Fault-Tolerance Techniques for SRAM-Based FPGAs*. Frontiers in Electronic Testing. Springer.

Misera, S. A. and Sieber, A. (2007). Fehlerinjektionstechniken in SystemC-Beschreibungen mit Gate- und Switch-Level-Verhalten. In *Dresdner Arbeitstagung für Schaltungs- und Systementwurf - DASS 2007*. TUD Press.

Montealegre, N. and Rammig, F. J. (2008). Immuno-repairing of FPGA designs. In Hinchey, M., Pagnoni, A., Rammig, F. J., and Schmeck, H., editors, *20th World Computer Congress, 2nd International Conference on Biologically-Inspired Collaborative Computing*, volume 268, pages 137–149. Springer.

Montealegre, N. and Rammig, F. J. (2010). Dynamic Partial Reconfiguration by Means of Algorithmic Skeletons - A Case Study. In Platzner, M., Teich, J., and Wehn, N., editors, *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*, pages 183–198. Springer.

Pellerin, D. and Taylor, D. (1996). *VHDL Made Easy*. Prentice Hall.

Purna, K. M. G. and Bhatia, D. (1999). Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *Transactions on Computers*, 48(6):579–590. IEEE.

Sharma, R. and Saluja, K. K. (1988). An Implementation and Analysis of a Concurrent Built-In Self-Test Technique. In *18th International Symposium on Fault-Tolerant Computing - FTCS 18*, pages 164–169.

Sterpone, L. and Violante, M. (2007). A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs. *Transactions on Nuclear Science*, 54(4):965–970. IEEE.

Sutton, A. (2012). FPGA Design Solutions for Military and Aerospace Applications. White paper. Synopsis.

Traut, A. (2010). Fehler-Injektor fur digitale Schaltungen. Master's thesis, University of Paderborn.

Wikipedia (2012). Searched words: register-transfer level, clock signal, reset, high impedance, arithmetic logic unit, associative array, content-addressable memory, hash function, hash table.

Xie, T., Mueller, W., and Letombe, F. (2011). HDL-Mutation Based Simulation Data Generation by Propagation Guided Search. In *14th Euromicro Conference on Digital System Design*, pages 608–615.

Xilinx (2000). *Using Block RAM for High Performance Read/Write CAMs*. XAPP204 (v1.2).

Xilinx (2004). *ML401/2/3 Block Diagram*.

Xilinx (2005). *Development System Reference Guide*.

Xilinx (2006a). *Command Line PR Implementation*.

Xilinx (2006b). *Early Access Partial Reconfiguration User Guide. For ISE 8.1.01i.* UG208 (v1.1).

Xilinx (2006c). *ML401/ML402/ML403 Evaluation Platform User Guide.* UG080 (v2.5).

Xilinx (2007). *Partial reconfiguration Design with PlanAhead.*

Xilinx (2008). *Early Access Partial Reconfiguration User Guide. For ISE 9.2.04i.* UG208 (v1.2).

Xilinx (2009a). *Command Line Tools User Guide.* UG628 (v11.4).

Xilinx (2009b). *Constraints Guide.* UG625 (v11.4).

Xilinx (2009c). *Embedded System Tools Reference Manual. EDK 11.3.1.* UG111.

Xilinx (2009d). *System ACE CompactFlash Solution.* DS080 (v1.4).

Xilinx (2009e). *Virtex-4 FPGA Configuration User Guide.* UG071 (v1.11).

Xilinx (2009f). *XST User Guide.* UG627 (v11.3).

Xilinx (2010). *Partial Reconfiguration User Guide.* UG702 (v12.3).

Zwolinski, M. (2003). *Digital System Design with VHDL.* Prentice Hall, second edition.

# 7

# Major contributions and further work

## 7.1 Major contributions

Chapter 2 presented a careful review of the literature available in the field of self-repairing hardware systems. That review allowed to see the focus, strengths and weaknesses of each approach. Thereby, it could be realized that the design of the fault recognition module, which is a key module in a self-repairing hardware system, has not been presented in detail in most of the approaches. Then, regarding the design of the fault recognition module for a self-repairing circuit, the first major contribution of this thesis is the design methodology for an online concurrent fault recognition module, which focuses on the reduction of the fault recognition latency and the hardware overhead, explained in chapters 4 and 5.

Chapter 1 presented the outline of an architecture for a self-repairing system that assures a fail-safe state of the system in case of having an unrecoverable defect during system recovery, and during fault recognition when required. That architecture has been designed in a modular way with the sight of having modules which can be added to a circuit described also as a hardware module. Thus, the second major contribution of this thesis is the architecture described at the RTL level of abstraction in the VHDL hardware description language for designing self-repairing hardware systems. That self-repairing architecture has been used as a framework for designing a self-repairing tiny circuit, which has been simulated and implemented in an FPGA successfully. The delivered modular architecture constitutes a further step towards the automatic insertion of hardware modules described at the RTL level for building a self-repairing system. That procedure is similar to the existing insertion of built-in self-test circuitry by RTL logic BIST tools, such as Tessent LogicBIST from Mentor Graphics or TurboBIST from Syntest, or also similar to the automatic insertion of error detection structures in RTL code presented in [Entrena et al., 2001] and [Mohanram et al., 2002]. The description of the architecture in VHDL can be synthesized and implemented for any hardware or FPGA platform. Fact which makes the architecture hardware platform independent.

Furthermore, the addition of a fault injection module to the architecture, based on saboteurs and mutants of the circuit for self-repairing, provides with a tool for simulating, debugging and evaluating the designed self-repairing circuit with the implemented fault recognition and recovery procedure modules.

For the design of the fault recognition module, helpful methods inspired by the immune system have been investigated. In chapter 3, the basic idea of the most important algorithms in the field of artificial immune systems and their pseudocodes have been presented. In the literature, those algorithms are explained mixing biological terms with computing terms, making hard to understand the way in which biological concepts have been transferred to the algorithms and how to implement and improve them. Therefore, the explanation starts with the most basic biological concepts required for understanding the algorithms. It is important to stress that the listed pseudocodes of the presented algorithms have been verified by coding them on functioning programs, which have not been presented in this thesis. Furthermore, a comparison of all algorithms specifying their inputs, results and their main application, resumes the work done to date in the field of artificial immune systems.

For the design of the fault recognition module, it was assumed that a set of fault pattern vectors for the recognition of faults is available. Since such a set of vectors in many systems is huge, it produces a high recognition latency, and it requires a high amount of memory resources, vectors dimension reduction and reduction of the number of vectors have been identified as possible solution. The algorithm named cytokine Formal Immune Network from the field of artificial immune systems has not been explained in chapter 3, but instead in chapter 4, because it can be better understood in connection with methods for vector dimension reduction such as the Principal Component Analysis. The Formal Immune Network, Principal Component Analysis, and Singular Value Decomposition can be employed for reducing the dimension of vectors. Thereby in this thesis, the mathematical similarities between PCA and FIN has been uncovered. The variable cytokine in a cFIN represents a class, which is associated to every vector in the given set, and is used by the reduction of the number of vectors, eliminating similar vectors that have the same class. That part of the method, inspired by the biological processes Apoptosis and Autoimmunization, can also be employed in combination with other dimension reduction methods such as PCA or SVD. Therefore, in chapter 5 such combinations have been evaluated considering as parameters: the number of reduced dimensions, the threshold for the reduction of the number of fault pattern vectors, the distance measurement method, and the class assignation method. The implementation of the different algorithms presented in Matlab serves for finding the dimension reduction method, number of dimensions, distance measurement method and class assignation method that provide the best recognition for the given set of fault pattern vectors. Then, those parameters can be used for designing and implementing the fault recognition module in hardware. That procedure can be applied in a hardware implementation for systems that have multiple line input and output vectors with real value elements.

The design of a fault recognition module for a digital circuit that has vectors with one-bit binary value elements is different. Hence, a review of methods for designing a concurrent fault recognition module for a combinational circuit are given. Thereby, the hardware overhead of that module has been shown that can be reduced using the unspecified values contained in the given fault pattern vectors. In this thesis, the reduction of the number of fault pattern vectors containing unspecified values using the clonal selection algorithm has been proposed. Then, the reduced set can be used for designing the concurrent fault recognition module. For reducing the hardware overhead of that module, this thesis proposes also to use a compactor

at the outputs of the circuit for self-repairing in order to be able to save just compacted output pattern vectors to be used by the output vectors comparison block. For that, the so called Compact-X technique for the compaction of output vectors has been found suitable.

## 7.2 Further work

The fault recognition module has been implemented for an almost-concurrent fault recognition, which can be applied to hardware systems that have input and output vectors with real value elements. However, in the hardware implementation, a combinational circuit has been used as the circuit for self-repairing. With the methods presented in chapter 5, it is possible to describe in VHDL, a concurrent fault recognition module to be used for the given combinational circuit. It is also possible instead of using a combinational circuit as an example, to use a hardware system that have input and output vectors with real value elements to test the fault recognition module described in VHDL performing minor modifications.

In the implementation of the fault recognition module, a complete fault pattern vector set has been used. However, a reduced or incomplete fault pattern vector set can be used and self-learning can be applied for adding fault pattern vectors online. That requires to declare an oversized memory in order to have enough place for adding fault pattern vectors on demand.

Self-learning, which consists on the enhancement of the fault pattern vector set online on demand, has not been included in the evaluation of methods for the reduction of the size of fault pattern vectors set. The effect of that feature in the recognition rate can be worth to be evaluated.

The recovery mechanism has been associated to each fault pattern vector stored in memory. For fault pattern vectors which have not a recovery mechanism associated, or for fault pattern vectors added online, fault diagnosis can be executed. Fault diagnosis is time expensive, therefore improvement of fault diagnosis techniques are required.

A complex circuit can be partitioned into smaller subcircuits which can be made self-repairing adding a fault recognition and a recovery procedure modules for each subcircuit.

Having a circuit which can be partitioned in smaller subcircuits, distributed fault recognition and centralized fault recovery can be implemented. The management of the self-repairing actions can be modeled and simulated following the paradigm in [Montealegre and Rammig, 2012], that proposes an immune network of agents, or simulated at the transaction level using corresponding tools.

The fault recognition module has been conceived for finding permanent faults in the circuit for self-repairing. Recognition of transient faults could also be added using methods of concurrent checking. An starting point can be the use of modules for the realization of a totally self-checking state machine such as the approaches in [Bolchini et al., 2000] and [Zeng et al., 1999]. Potentially recovery states or spare states can also be considered for the design of self-repairing sequential circuits.

The architecture for the design of self-repairing hardware systems can be implemented at the system level of abstraction using the tool chain Matlab-Simulink-SystemGenerator-ISE. In that case, a toolbox for inserting fault repairing in circuits can be created. Such an implementation can be fruitful for testing fault recognition modules that use different parameters or have different design.

## 7.3 Bibliography

Akoglu, A., Sreeramareddy, A., and Josiah, J. G. (2009). FPGA based distributed self healing architecture for reusable systems. *Cluster Computing*, 12(3):269–284.

Avižienis, A. (2006). An Immune System Paradigm for the Assurance of Dependability of Collaborative Self-organizing Systems. In *19th World Computer Congress, 1st International Conference on Biologically Inspired Computing*, volume 216 of *IFIP International Federation for Information Processing*, pages 1–6. Springer.

Bolchini, C., Montandon, R., Salice, F., and Sciuto, D. (2000). Design of VHDL-Based Totally Self-Checking Finite-State Machine and Data-Path Descriptions. *Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):98–103. IEEE.

Entrena, L., López, C., and Olías, E. (2001). Automatic Insertion of Fault-Tolerant Structures at the RTL Level. *7th International On-Line Testing Workshop*, pages 183–200.

Kochte, M. A., Zoellin, C. G., and Wunderlich, H.-J. (2009). Concurrent Self-Test with Partially Specified Patterns For Low Test Latency and Overhead. In *14th European Test Symposium*, pages 53–58. IEEE Computer Society.

Li, Y., Makar, S., and Mitra, S. (2008). CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns. *Design, Automation and Test in Europe - DATE 2008*, pages 885–890.

Mohanram, K., Krishna, C. V., and Tuba, N. A. (2002). A Methodology for Automated Insertion of Concurrent Error Detection Hardware in Synthesizable Verilog RTL. In *International Symposium on Circuits and Systems - ISCAS 2002*, volume 1, pages 577–580. IEEE.

Montealegre, N. and Rammig, F. J. (2012). Agent-Based Modeling and Simulation of Artificial Immune Systems. In *15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops - ISORCW 2012, Third Workshop on Self-Organizing Real-Time Systems - SORT 2012*, pages 212–219. IEEE.

Zeng, C., Saxena, N., and McCluskey, E. J. (1999). Finite state machine synthesis with concurrent error detection. *Proceedings of the International Test Conference*, pages 672–678.

# List of publications

In the context of this thesis five conference papers and one journal paper were published. The self-repairing unit has been presented as a demo at an international conference. And finally three bachelor theses have been supervised.

## Conference papers

Norma Montealegre and Franz Josef Rammig. *Immuno-Repairing of FPGA Designs*. In Mike Hinchey, Anastasia Pagnoni, Franz Josef Rammig, and Hartmut Schmeck, editors, World Computer Congress (WCC 2008), Biologically-Inspired Collaborative Computing (BICC 2008), volume 268 of the International Federation for Information Processing Series, pages 137-149, Springer, Milano, Italy, September, 7-10 2008.

Norma Montealegre. *Fault Tolerance in FPGA Designs by Means of Immunocomputing*. In René Schüffny, Reiner G Spallek, and Günter Elst, editors, Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2008), pages 131-136, Fraunhofer Institut Integrierte Schaltung, Institutsteil Entwurfsutomatisierung, Springer, Dresden, Germany, May, 15-16 2008.

Masoud Hosseinimehr and Norma Montealegre. *Implementation of a Singular Value Decomposition Module on an FPGA*. In Teofilo Gonzalez, editor, 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011), ACTA Press, Dallas, USA, December, 14-16 2011.

Norma Montealegre and Franz J. Rammig. *Agent-Based Modeling and Simulation of Artificial Immune Systems*. In 3rd Workshop on Self-Organizing Real-Time Systems (SORT 2012) and 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW 2012), pages 212-219, IEEE, Shenzhen, China, April, 11 2012.

Norma Montealegre. *Agent-Based Artificial Immune System Model for the Detection of Faults in a Distributed Satellite System*. In the 1st European conference on Satellite TELecommunications (ESTEL 2012), IEEE-AESS, Rom, Italy, October, 2-5 2012.

## Journal papers

Norma Montealegre and Sebastian Hagenkötter. *Process integrated wire-bond quality control by means of cytokine-Formal Immune Networks*. Journal of Intelligent Manufacturing, volume 23, number 3, pages 699-715, 2012.

## Demonstrations

Norma Montealegre. *Demonstration of the Fault Recognition and Recovery of FPGA Circuits by Means of Cytokine-Formal Immune Networks.* In Neil Bergmann, Oliver Diessel, and Lesley Shannon, editors, International Conference on Field-Programmable Technology (FPT 2009), pages 384-397, IEEE, Sydney, Australia, December, 9-11 2009.

## Theses

Puya Dibaj. *Hardware Fault Recognition Unit.* Bachelor thesis. Universität Paderborn, May, 2010.

Alexander Traut. *Fehler Injektor für digitale Schaltungen.* Bachelor thesis. Universität Paderborn, May, 2010.

Masoud Hosseinimehr. *Implementation of a Singular value Decomposition Module on an FPGA.* Bachelor thesis. Universität Paderborn, May, 2010.

# Bibliography


Abramovici, M., Emmert, J. M., and Stroud, C. E. (2001). Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis and Fault Tolerance for FPGAs in Adaptive Computing Systems. In *3rd NASA/DoD Workshop on Evolvable Hardware*, pages 73–92. IEEE Computer Society.

Abramovici, M., Stroud, C., Hamilton, C., Wijesuriya, S., and Verma, V. (1999). Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications. In *International Test Conference*, pages 973–982.

ACTEL (2003). *Content-Addressable Memory (CAM) in Actel Devices*. Application Note AC194.

AISweb (2010).

Akoglu, A., Sreeramareddy, A., and Josiah, J. G. (2009). FPGA based distributed self healing architecture for reusable systems. *Cluster Computing*, 12(3):269–284. Springer.

Al-Asaad, H. and Shringi, M. (2000). On-line built-in self-test for operational faults. In *AUTOTESTCON 2000*, pages 168–174.

Al-Yamani, A. A. (2004). *Deterministic Built-In Self Test for Digital Circuits*. PhD thesis, Stanford University.

Allen, D., Cumano, A., Dildrop, R., Kocks, C., Rajewsky, K., Tajewsky, N., Roes, J., Sablitzky, F., and Siekevitz, M. (1987). Timing, Genetic Requirements and Functional Consequences of Somatic Hypermutation during B-Cell Development. *Immunological Reviews*, 96(1):5–22. Blackwell Publishing Ltd.

Altera (1999). *CAM Comparison: APEX 20KE vs. Virtex-E Devices*. Technical Brief 61.

Amaral, J. L. M. (2011). Fault Detection in Analog Circuits Using a Fuzzy Dendritic Cell Algorithm. In *10th International Conference on Artificial Immune Systems - ICARIS 2011*. Springer.

Avižienis, A. (2000). A Fault Tolerance Infrastructure for Dependable Computing with High-Performance COTS Components. In *International Conference on Dependable Systems and Networks - DSN 2000*, pages 496–500. IEEE.

Avižienis, A. (2002). An Immune System Paradigm for the Design of Fault Tolerant Systems. In *4th European Dependable Computing Conference on Dependable Computing - EDCC 4*, Lecture Notes in Computer Science, pages 81–83. Springer.

Avižienis, A. (2006). An Immune System Paradigm for the Assurance of Dependability of Collaborative Self-organizing Systems. In *19th World Computer Congress, TC 10: 1st International Conference on Biologically Inspired Computing*, volume 216 of *IFIP International Federation for Information Processing*, pages 1–6. Springer.

Avižienis, A., Kopetz, H., and Laprie, J. C., editors (1992). *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer.

Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *Transactions on Dependable and Secure Computing*, 1(1):11–33. IEEE.

Avizienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., and Rubin, D. K. (1971). The STAR (Self-Testing and Self-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design. *Transactions on Computers*, 20(11):1312–1321. IEEE.

Baraza, J. C., Gracia, J., Gil, D., and Gil, P. J. (2005). Improvement of Fault Injection Techniques Based on VHDL Code Modification. In *10th International Conference on High-Level Design Validation and Test Workshop*, pages 19–26. IEEE.

Barker, W., Halliday, D. M., Thoma, Y., Sanchez, E., Tempesti, G., and Tyrell, A. M. (2007). Fault Tolerance Using Dynamic Reconfiguration on the POEtic Tissue. *Transactions on Evolutionary Computation*, 11(5):666–684. IEEE.

Becker, M., Kuznik, C., Joy, M. M., Xie, T., and Mueller, W. (2012). Binary Mutation Testing Through Dynamic Translation. In *42nd International Conference on Dependable Systems and Networks - DSN 2012*, pages 1–12. IEEE.

Bellato, M., Bernardi, P., Bortolato, D., Candelori, A., Ceschia, M., Paccagnella, A., Rebaudengo, M., Reorda, M. S., Violante, M., and Zambolin, P. (2004). Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA. In *Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 584–589.

Boesen, M. R. and Madsen, J. (2009). eDNA: A Bio-Inspired Reconfigurable Hardware Cell Architecture Supporting Self-organisation and Self-healing. In *NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2009*, pages 147–154.

Boesen, M. R., Madsen, J., and Keymeulen, D. (2011). Autonomous Dynamically Self-organizing and Self-healing Distributed Hardware Architecture the eDNA Concept. In *Aerospace Conference*, pages 1–13. IEEE.

Bolchini, C., Montandon, R., Salice, F., and Sciuto, D. (2000). Design of VHDL-Based Totally Self-Checking Finite-State Machine and Data-Path Descriptions. *Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):98–103. IEEE.

Bolchini, C., Sandionigi, C., Fossati, L., and Codinachs, D. M. (2011). A reliable fault classifier for dependable systems on SRAM-based FPGAs. In *17th International On-Line Testing Symposium - IOLTS 2011*. IEEE.

Bouajila, A., Zeppenfeld, J., Stechele, W., and Herkersdorf, A. (2011). An Architecture and an FPGA Prototype of a Reliable Processor Pipeline Towards Multiple Soft- and Timing Errors. In *14th International Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2011*, pages 225 – 230. IEEE.

Bouajila, A., Zeppenfeld, J., Stechele, W., Herkersdorf, A., Bernauer, A., Bringmann, O., and Rosenstiel, W. (2006). Organic Computing at the System on Chip Level. In *International Conference on Very Large Scale Integration*, pages 338–341. IFIP.

Bradley, D., Ortega-Sanchez, C., and Tyrell, A. (2000). Embryonics + Immunotronics: A Bio-Inspired Approach to Fault Tolerance. In *2nd NASA/DoD Workshop on Evolvable Hardware*, pages 215–223. IEEE Computer Society.

Bradley, D. W. (2002). Immunotronics - Novel Finite-State-Machine architectures with built-in self-test using self-nonself differentiation. *Transactions on Evolutionary Computation*, 6(3):227–238. IEEE.

Bradley, D. W. and Tyrell, A. M. (2001). The Architecture for a Hardware Immune System. In *3rd NASA/DoD Workshop on Evolvable Hardware*, pages 193–200. IEEE Computer Society.

Brglez, F., Bryan, D., and Kozminski, K. (1989). *Notes on the ISCAS'89 Benchmarks Circuits*. MCNC.

Brown, S. and Vranesic, Z. (2005). *Fundamentals of Digital Logic with VHDL Design*. Mcgraw-Hill, second edition.

Bryan, D. (1988). *The ISCAS'85 benchmark circuits and netlist format*. MCNC.

Burke, G. and Taft, S. (2004a). Fault Tolerant State Machines. Technical Report D160/MALPD 2004, Jet Propulsion Laboratory, California Institute of Technology.

Burke, G. and Taft, S. (2004b). Fault Tolerant State Machines. In *MALPD 2004*.

Castro, L. N. and Timmis, J. (2002). *Artificial Immune Systems. A new Computational Intelligence Approach*. Springer.

Chakraborty, T. J. and Chiang, C.-H. (2002). A novel fault injection method for system verification based on FPGA boundary scan architecture. In *International Test Conference - ITC 2002*, pages 923–929.

Chmelař, E. (2004a). Minimizing the Number of Test Configurations for FPGAs. In *International Conference on Computer Aided Design - ICCAD 2004*, pages 899–902. IEEE/ACM.

Chmelař, E. (2004b). *The Test and Diagnosis of FPGAs*. PhD thesis, Stanford University.

Christoph Steiger, H. W., Platzner, M., and Thiele, L. (2003). Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. In *24th Real-Time Systems Symposium - RTSS 2003*, pages 224–225. IEEE.

Chu, P. P. (2006). *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. John Wiley & Sons.

Collaborative Benchmarking and Experimental Algorithmics Laboratory (2007). The Benchmark Archives at CBL (up to 1996).

Dibaj, P. (2010). Hardware Fault Recognition Unit. Bachelor's thesis, University of Paderborn.

Dictionary.com, L. (2012). English dictionary. dictionary.reference.com.

Dittmann, F. (2008). *Methods to Exploit Reconfigurable Fabrics*. PhD thesis, University of Paderborn.

Dye, D. (2012). Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite. Xilinx White Paper: Virtex-4, Virtex-5, Virtex-6, and 7 Series FPGAs.

École Polytechnique Fédérale de Lausanne, University of York, The University of Glasgow, Université de Lausanne, and Universitat Politécnica de Catalunya (1.09.2001 - 31.1.2005). Reconfigurable POEtic Tissue (POETIC) Project. http://cordis.europa.eu.

Emmert, J. M., Stroud, C. E., Skaggs, B., and Abramovici, M. (2000). Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration. In *Symposium on Field-Programmable Custom Computing Machines*, pages 165–174. IEEE.

Entrena, L., López, C., and Olías, E. (2001). Automatic Insertion of Fault-Tolerant Structures at the RT Level. In *7th International On-Line Testing Workshop*, pages 183–200. 48-50.

Fodor, I. K. (2002). A survey of dimension reduction techniques. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.

Forrest, S., Perelson, A. S., Allen, L., and Cherukuri, R. (1994). Self-Nonself Discrimination in a Computer. In *Symposium on Research in Security and Privacy*, pages 202–212. IEEE.

Garvie, M. and Thompson, A. (2004). Scrubbing away transients and Jiggling around the permanent: Long survival of FPGA systems through evolutionary self-repair. In *10th International On-Line Testing Symposium - IOLTS 2004*, pages 155–160. IEEE.

Gössel, M., Ocheretny, V., Sogomonyan, E., and Marienfeld, D. (2008). *New Methods of Concurrent Checking*. Frontiers in Electronic Testing. Springer.

Greensmith, J. (2007). *The Dendritic Cell Algorithm*. PhD thesis, University of Nottingham.

Hamzaoglu, I. and Patel, J. H. (2000). Test Set Compaction Algorithms for Combinational Circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):283–289. IEEE.

Hellebrand, S., Tarnick, S., Rajski, J., and Courtois, B. (1992). Generation Of Vector Patterns Through Reseeding Of Multiple-Polynomial Linear Feedback Shift Registers. In *International Test Conference - ITC 1992*, pages 120–129.

Hosseinimehr, M. (2010). Implementation of a Singular Value Decomposition Module on an FPGA. Master's thesis, University of Paderborn.

Huang, W.-J. and McCluskey, E. J. (2001). A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations. In *International Symposium on Field-Programmable Gate Arrays - FPGA 2001*. ACM/SIGDA.

Huang, W.-J. R. (2001). *Dependable Computing Techniques for Reconfigurable Hardware*. PhD thesis, Stanford University.

Jenn, E., Arlat, J., Rimbn, M., Ohlsson, J., and Karlsson, J. (1994). Fault Injection into VHDL Models: The MEFISTO Tool. In *24th International Symposium on Fault-Tolerant Computing - FTCS 24*, pages 66–75.

Jerne, N. K. (1974). Clonal selection in a lymphocyte network. In Edelman, G. M., editor, *Cellular selection and regulation in the immune response*, pages 39–48. Raven Press, New York.

Jerne, N. K. (1985). The Generative Grammar of the Immune System, Nobel lecture, 8 December 1984. *Bioscience Reports*, 5(6):439–451. Springer.

Kajihara, S., Pomeranz, I., Kinoshita, K., and Reddy, S. M. (1993). Cost-Effective Generation of Minimal Test Sets for Stuck-at Faults in Combinational Logic Circuits. In *30th International Design Automation Conference - DAC 1993*, pages 102–106. ACM.

Kalla, P. and Ciesielski, M. (1998). A Comprehensive Approach to the Partial Scan Problem using Implicit State Enumeration. In *International Test Conference - ITC 1998*, pages 651–657. IEEE Computer Society.

Kastensmidt, F. L., Carro, L., and Reis, R. (2006). *Fault-Tolerance Techniques for SRAM-Based FPGAs*. Frontiers in Electronic Testing. Springer.

Kephard, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36(1):41–50. IEEE Computer Society.

Kimball, J. W. (1994). *Biology*. Addison-Wesley, 6 edition.

Kirkland, T. and Mercer, M. R. (1988). Algorithms for Automatic Test Pattern Generation. *Design & Test*, 5(3):43–55. IEEE Computer Society.

Koal, T., Scheit, D., Schölzel, M., and Vierhaus, H. T. (2011). On the Feasibility of Built-In Self Repair for Logic Circuits. In *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems - DFT 2011*, pages 316–324. IEEE.

Koal, T., Scheit, D., and Vierhaus, H. T. (2009). A Concept for Logic Self Repair. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools - DSD 2009*, pages 621–624.

Koal, T., Ulbricht, M., and Vierhaus, H. T. (2012). Combining On-Line Fault Detection and Logic Self Repair. In *15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2012*, pages 288–293. IEEE.

Koal, T. and Vierhaus, H. T. (2008). Basic Architecture for Logic Self Repair. In *14th International On-Line Testing Symposium - IOLTS 2008*, pages 177–178. IEEE.

Kochte, M. A., Zoellin, C. G., and Wunderlich, H.-J. (2009). Concurrent Self-Test with Partially Specified Patterns For Low Test Latency and Overhead. In *14th European Test Symposium*, pages 53–58. IEEE Computer Society.

Koren, I. and Krishna, C. M. (2007). *Fault Tolerant Systems*. Morgan Kaufmann.

Kothe, R. and Vierhaus, H. T. (2006). Embedded Self Repair by Transistor and Gate Level Reconfiguration. In *Conference on Design and Diagnostics of Electronic Circuits and Systems - DDECS 2006*, pages 208–213. IEEE.

Krishnan, S. and Kerkhoff, H. G. (2012). A Robust Metric for Screening Outliers from Analogue Product Manufacturing Tests Responses. In *6th European Test Symposium - ETS 2011*. IEEE.

Kumar, V. V. and Lach, J. (2003). Fine-Grained Self-Healing Hardware for Large-Scale Autonomic Systems. In *14th International Workshop on Database and Expert Systems Applications*, pages 707–712. IEEE Computer Society.

Lala, P. K. (2000). *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann.

Lee, H. K. and Ha, D. S. (1990). SOPRANO: An Efficient Automatic Test Pattern Generator for Stuck-Open Faults in CMOS Combinational Circuits. In *27th Design Automation Conference*, pages 660–666.

Lee, H. K. and Ha, D. S. (1991). An Efficient Forward Fault Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation. In *International Test Conference - ITC 1991*, pages 946–955.

Lee, H. K. and Ha, D. S. (1993). On the Generation of Test Patterns for Combinational Circuits. Technical Report 12-93, Department of Electrical Engineering, Virginia Polytechnic Institute and State University.

Lee, H. K. and Ha, D. S. (1996). HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1048–1058. IEEE.

Li, Y., Makar, S., and Mitra, S. (2008a). CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns. In *Design, Automation and Test in Europe - DATE 2008*, pages 885–890.

Li, Y., Makar, S., and Mitra, S. (2008b). CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns. In *Conference on Design, Automation and Test in Europe - DATE 2008*, pages 885–890. ACM.

Li, Y., Mutlu, O., Gardner, D. S., and Mitra, S. (2010a). Concurrent Autonomous Self-Test for Uncore Components in System-on-Chips. In *28th VLSI Test Symposium - VTS 2010*, pages 232–237. IEEE.

Li, Y., Mutlu, O., Gardner, D. S., and Mitra, S. (2010b). Concurrent Autonomous Self-Test for Uncore Components in Systems-on-Chips. In *28th VLSI Test Symposium - VTS 2010*, pages 232–237. IEEE Computer Society.

Lipsa, G., Herkersdorf, A., Rosenstiel, W., Bringmann, O., and Stechele, W. (2005). Towards a Framework and a Design Methodology for Autonomic SoC. In *2nd International Conference on Autonomic Computing - ICAC 2005*, pages 391–392.

López-Ongil, C., Entrena, L., García-Valderas, M., and Portela-García, M. (2007). Automatic Tools for Design Hardening. In Velazco, R., Fouillat, P., and Reis, R., editors, *Radiation Effects on Embedded Systems*, pages 183–200. Springer.

Marchal, P., Nussbaum, P., Piguet, C., Durand, S., Mange, D., Sanchez, E., Stauffer, A., and Tempesti, G. (1996). Embryonics: The Birth of Synthetic Life. In Sanchez, E. and Tomassini, M., editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 616–196. Springer.

Mardia, K. V., Kent, J. T., and Bibby, J. M. (1979). *Multivariate Analysis*. Probability and Mathematical Statistics. Academic Press.

Marinos, P. N. (1969). The Organization of a Self-Repairing System from Multifunctional Units. *Proceedings of the IEEE*, 57(7):1320.

Mazumder, P. and Rudnick, E. (1998). *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Prentice Hall.

Misera, S. A. and Sieber, A. (2007). Fehlerinjektionstechniken in SystemC-Beschreibungen mit Gate- und Switch-Level-Verhalten. In *Dresdner Arbeitstagung für Schaltungs- und Systementwurf - DASS 2007*. TUD Press.

Mitra, S. (2000). *Diversity Techniques For Concurrent Error Detection*. PhD thesis, Stanford University.

Mitra, S. (2004). X-compact: an efficient response compaction technique. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):421–432. IEEE.

Mitra, S. (2008). Globally Optimized Robust Systems to Overcome Scaled CMOS Reliability Challenges. In *Design, Automation and Test in Europe - DATE 2008*, pages 941–946.

Mitra, S., Huang, W.-J., Saxena, N. R., Yu, S.-Y., and McCluskey, E. J. (2000). Dependable Adaptive Computing Systems - The Stanford CRC ROAR Project. In *Pacific RIM International Symposium on Dependable Computing - Fast Abstracts*.

Mitra, S., Huang, W.-J., Saxena, N. R., Yu, S.-Y., and McCluskey, E. J. (2004). Reconfigurable Architecture for Autonomous Self-Repair. *Design and Test of Computers*, 21(4):228–240. IEEE.

Miyase, K. and Kajihara, S. (2004). XID: Don't care identification of test patterns for combinational circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):321–326. IEEE.

Mohanram, K., Krishna, C. V., and Tuba, N. A. (2002). A Methodology for Automated Insertion of Concurrent Error Detection Hardware in Synthesizable Verilog RTL. In *International Symposium on Circuits and Systems - ISCAS 2002*, volume 1, pages 577–580. IEEE.

Montealegre, N. and Rammig, F. J. (2008). Immuno-repairing of FPGA designs. In Hinchey, M., Pagnoni, A., Rammig, F. J., and Schmeck, H., editors, *20th World Computer Congress, 2nd International Conference on Biologically-Inspired Collaborative Computing*, volume 268, pages 137–149. Springer.

Montealegre, N. and Rammig, F. J. (2010a). Dynamic Partial Reconfiguration by Means of Algorithmic Skeletons - A Case Study.

Montealegre, N. and Rammig, F. J. (2010b). Dynamic Partial Reconfiguration by Means of Algorithmic Skeletons - A Case Study. In Platzner, M., Teich, J., and Wehn, N., editors, *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*, pages 183–198. Springer.

Montealegre, N. and Rammig, F. J. (2012). Agent-Based Modeling and Simulation of Artificial Immune Systems. In *15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops - ISORCW 2012, Third Workshop on Self-Organizing Real-Time Systems - SORT 2012*, pages 212–219. IEEE.

Moreno, J., Thoma, Y., Sanchez, E., Torrez, O., and Tempesti, G. (2004). Hardware Realization of a Bio-Inspired POEtic Tissue. In *NASA/DoD Conference on Evolvable Hardware*, pages 237–244. IEEE.

Moreno, J. M., Madrenas, J., Faura, J., Cantó, E., Cabestany, J., and Insenser, J. M. (1998). Feasible Evolutionary and Self-Repairing Hardware by Means of the Dynamic Reconfiguration Capabilities of the FIPSOC Devices. In *2nd International Conference on Evolvable Systems: From Biology to Hardware - ICES 1998*, Lecture Notes in Computer Science, pages 345–355. Springer.

Oh, N. (2000). *Software Implemented Hardware Fault Tolerance*. PhD thesis, Stanford University.

Oja, E. (2003). Principal Component Analysis. In Arbib, M. A., editor, *The Handbook of Brain Theory and neural Networks*. The MIT Press.

Ortega-Sanchez, C., Mange, D., Smith, S., and Tyrell, A. (2000). Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties. *Genetic Programming and Evolvable Machines*, 1(3):187–215.

Paulsson, K., Hübner, M., and Becker, J. (2006a). Methods for Run-time Failure Recognition and Recovery in Dynamic and Partial Reconfigurable Systems Based on Xilinx Virtex-II Pro FPGAs. In *Symposium on Emerging VLSI Technologies and Architectures*. IEEE Computer Society.

Paulsson, K., Hübner, M., and Becker, J. (2006b). Strategies to On- Line Failure Recovery in Self- Adaptive Systems based on Dynamic and Partial Reconfiguration. In *1st NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2006*, pages 288–291. IEEE Computer Society.

Pellerin, D. and Taylor, D. (1996). *VHDL Made Easy*. Prentice Hall.

Perelson, A. S. (1989). Immune Network Theory. *Immunologival Reviews*, 110(1):5–36. Munksgaard.

Pomeranz, I., Reddy, L. N., and Reddy, S. M. (1993). COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):1040–1049. IEEE.

Pomeranz, I. and Reddy, S. M. (2006). Reducing the number of specified values per test vector by increasing the test set size. *Computers & Digital Techniques*, 153(1):39–46. IEE.

Purna, K. M. G. and Bhatia, D. (1999). Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers. *Transactions on Computers*, 48(6):579–590. IEEE.

Raedtke, S., Bargfrede, J., and Anheier, W. (1995). Distributed Automatic Test Pattem Generation with a Parallel FAN Algorithm. In *International Conference on Computer Design: VLSI in Computers and Processors - ICCD 1995*, pages 698–702. IEEE.

Reed, I. S. (1973). Boolean Difference Calculus and Fault Finding. *Journal on Applied Mathematics*, 24(1):124–143. Society for Industrial and Applied Mathematics - SIAM.

Reorda, M. S., Sterpone, L., and Violante, M. (2005a). Efficient Estimation of SEU effects in SRAM-based FPGAs. In *11th International On-Line Testing Symposium - IOLTS 2005*, pages 54–59. IEEE.

Reorda, M. S., Sterpone, L., and Violante, M. (2005b). Multiple errors produced by single upsets in FPGA configuration memory - a possible solution. In *European Test Symposium*, pages 136–141.

Rudnick, E. M. and Patel, J. H. (1999). Efficient Techniques for Dynamic Test Sequence Compaction. *Transactions on Computers*, 48(3):323–330. IEEE.

Schulz, M. H., Trischler, E., and Sarfert, T. M. (1988). SOCRATES: A Highly Afficient Automatic Test Pattern Generation System. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1):126–137. IEEE.

Sharma, R. and Saluja, K. K. (1988). An Implementation and Analysis of a Concurrent Built-In Self-Test Technique. In *18th International Symposium on Fault-Tolerant Computing - FTCS 18*, pages 164–169. IEEE.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Pérez-Uribe, A., and Stauffer, A. (1997). A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *Transactions on Evolutionary Computation*, 1(1):83–97. IEEE.

Sreeramareddy, A., Josiah, J. G., Akoglu, A., and Stoica, A. (2008). SCARS: Scalable Self-Configurable Architecture for Reusable Space Systems. In *NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2008*, pages 204–210.

Sreeramareddy, A., Kallam, R., Dasu, A. R., and Akoglu, A. (2010). Self-configurable architecture for reusable systems with Accelerated Relocation Circuit (SCARS-ARC). In *International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum - IPDPSW 2010*, pages 1–4. IEEE.

Sterpone, L. and Violante, M. (2005). A Design Flow for Protecting FPGA-Based Systems Against Single Event Upsets. In *20th International Symposium on Defect and Fault Tolerance in VLSI Systems - DFT 2005*, pages 436–444. IEEE.

Sterpone, L. and Violante, M. (2007). A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs. *Transactions on Nuclear Science*, 54(4):965–970. IEEE.

Sudarsanam, A., Kallam, R., and Dasu, A. (2009). PRR-PRR Dynamic Relocation. *Computer Architecture Letters*, 8(2):44–47. IEEE.

Sutton, A. (2012). No Room for Error: Creating Highly Reliable, High-Availability FPGA Designs. Technical report, Synopsis.

Tarakanov, A., Goncharova, L., and Tarakanov, O. (2005). A Cytokine Formal Immune Network. In *8th European Conference on Advances in Artificial Life - ECAL 2005*, volume 3630 of *Lecture Notes in Computer Science*, pages 510–519. Springer.

Tarakanov, A. O. (2008a). Formal Immune Networks: Self-Organization and Real-World Applications. In Prokopenko, M., editor, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 271–290. Springer.

Tarakanov, A. O. (2008b). Formal Immune Networks: Self-Organization and Real-World Applications. In Wu, X. and Prokopenko, M., editors, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 271–290. Springer.

Tarakanov, A. O., Skormin, V. A., and Sokolova, S. P. (2003a). *Immunocomputing: Principles and Applications*. Springer.

Tarakanov, A. O., Skormin, V. A., and Sokolova, S. P. (2003b). *Immunocomputing, Principles and Applications*. Springer.

Tarlinton, D. (1998). Germinal centers: form and function. *Current Opinion in Immunology*, 10(3):245–251. Elsevier.

Tempesti, G., Mange, D., Mudry, P.-A., Rossier, J., and Stauffer, A. (2007). Self-Replicating Hardware for Reliability: The Embryonics Project. *Journal on Emerging Technologies in Computing Systems - JETC*, 3(2):Article No. 9. ACM.

Tempesti, G., Mange, D., and Stauffer, A. (1997). A Robust Multiplexer-Based FPGA Inspired By Biological Systems. *Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems*, 43(10):719–733. Elsevier.

Theodoridis, S. (2009). *Introduction to Pattern Recognition: A MATLAB Approach*. Academic Press.

Theodoridis, S. and Koutroumbas, K. (2008). *Pattern Recognition*. Academic Press, 4 edition.

Timmis, J. and Neal, M. (2001). A resource limited artificial immune system for data analysis. *Knowledge Based Systems*, 14(3-4):121–130. Elsevier.

Timmis, J., Neal, M., and Hunt, J. (2000). An artificial imune system for data analysis. *BioSystems*, 55(1-3):143–150. Elsevier.

Torresen, J. (2004). An Evolvable Hardware Tutorial. In *14th International Conference on Field Programmable Logic and Applications - FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 821–830. Springer.

Touba, N. A. and McCluskey, E. J. (1997). Logic Synthesis of Multilevel Circuits with Concurrent Error Detection. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(7):783–789. IEEE.

Traut, A. (2010). Fehler-Injektor fur digitale Schaltungen. Master's thesis, University of Paderborn.

Tyrell, A. M. and Barker, W. (2006). The Poetic Hardware Device: Assistance for Evolution, Development and Learning. In Higuchi, T., Liu, Y., and Yao, X., editors, *Evolvable Hardware*, Genetic and Evolutionary Computation, pages 99–119. Springer.

Tyrell, A. M., Sanchez, E., Floreano, D., Tempesti, G., mange, D., Moreno, J.-M., Rosenberg, J., and Villa, A. E. (2003). POEtic Tissue: An Integrated Architecture for Bio-inspired Hardware. In *Evolvable Systems: From Biology to Hardware*, volume 2606 of *Lecture Notes in Computer Science*, pages 129–140. Springer.

Venishetti, S. K., Akoglu, A., and Kalra, R. (2007). Hierarchical Built-in Self-testing and FPGA Based Healing Methodology for System-on-a-Chip. In *2nd NASA/ESA Conference on Adaptive Hardware and Systems - AHS 2007*, pages 717–724.

Voyiatzis, I., Gizopoulos, D., and Paschalis, A. (2009). An Input Vector Monitoring Concurrent BIST Scheme Exploiting X Values. In *15th On-Line Testing Symposium - IOLTS 2009*, pages 206–207. IEEE.

Wang, L.-T., Cheng, K.-T., and Chang, Y.-W., editors (2009). *Electronic Design Automation: Synthesis, Verification, and Test.* Systems on Silicon. Morgan Kaufmann.

Wang, L.-T., Wu, C.-W., and Wen, X. (2006). *VLSI Test Principles and Architectures: Design for Testability.* Systems on Silicon. Morgan Kaufman.

Wikipedia (2010a). Searched words: artificial intelligence, disease, cell, software agent, intelligent agent, apoptosis, necrosis, pathogen, virus, bacterion, fungus, lymphatic system, lymph, thymus, spleen, tonsils, leukocyte, protein, enzyme, allosteric regulation, cell signaling, communication, intracrine, autocrine signalling, juxtacrine signalling, paracrine signalling, idiotope, interferon, histamine, tumor necrosis factor, toxin, growth factor, hormone, cytokine, neurotransmitter, DAMPs, PAMPs, major histocompatibility complex, pattern recognition receptor, toll like receptor, signal transduction, gene, genetic, genetic code, dendritic cell, T-cell, B-cell, T helper 17 cell, lymph node, clonal selection, affinity maturation.

Wikipedia (2010b). Searched words: dimension reduction, singular value decomposition, Karhunen-Loève theorem, Hamming distance, feature selection k-nearest neighbor algorithm, Mahalanobis distance, pattern recognition, pattern matching, machine learning, molecular recognition, bilinear form, Lagrange multiplier.

Wikipedia (2011). Searched words: standard score, standard deviation, normalization, norm, unit vector, variance, Mahalanobis distance, design for test, scan chain, NAND logic.

Wikipedia (2012a). Searched words: fault-tolerant system, fault-tolerant design, single point of failure, integrated circuit, chip, partial reconfiguration, system on chip, network on chip, hardware register, CMOS, fault model, SRAM, flash memory.

Wikipedia (2012b). Searched words: register-transfer level, clock signal, reset, high impedance, arithmetic logic unit, associative array, content-addressable memory, hash function, hash table.

Williams, M. J. Y. and Angell, J. B. (1973). Enhancing Testability of Large-Scale Integrated Circuits via Test Points and additional Logic. *Transactions on Computers*, 22(1):46–60. IEEE Computer Society.

Wong, J. S. J., Sedcole, P., and Cheung, P. Y. K. (2007). Self-characterization of Combinatorial Circuit Delays in FPGAs. In *International Conference on Field-Programmable Technology - ICFPT 2007*, pages 17–23.

Xie, T., Mueller, W., and Letombe, F. (2011). HDL-Mutation Based Simulation Data Generation by Propagation Guided Search. In *14th Euromicro Conference on Digital System Design*, pages 608–615.

Xilinx (2000). *Using Block RAM for High Performance Read/Write CAMs*. XAPP204 (v1.2).

Xilinx (2004). *ML401/2/3 Block Diagram*.

Xilinx (2005). *Development System Reference Guide*.

Xilinx (2006a). *Command Line PR Implementation*.

Xilinx (2006b). *Early Access Partial Reconfiguration User Guide. For ISE 8.1.01i*. UG208 (v1.1).

Xilinx (2006c). *ML401/ML402/ML403 Evaluation Platform User Guide*. UG080 (v2.5).

Xilinx (2007). *Partial reconfiguration Design with PlanAhead*.

Xilinx (2008). *Early Access Partial Reconfiguration User Guide. For ISE 9.2.04i*. UG208 (v1.2).

Xilinx (2009a). *Command Line Tools User Guide*. UG628 (v11.4).

Xilinx (2009b). *Constraints Guide*. UG625 (v11.4).

Xilinx (2009c). *Embedded System Tools Reference Manual. EDK 11.3.1*. UG111.

Xilinx (2009d). *System ACE CompactFlash Solution*. DS080 (v1.4).

Xilinx (2009e). *Virtex-4 FPGA Configuration User Guide*. UG071 (v1.11).

Xilinx (2009f). *XST User Guide*. UG627 (v11.3).

Xilinx (2010). *Partial Reconfiguration User Guide*. UG702 (v12.3).

Yu, S.-Y. (2001). *Fault Tolerance in Adaptive Real-Time Computing Sytems*. PhD thesis, Stanford University.

Zeng, C., Saxena, N., and McCluskey, E. J. (1999). Finite State Machine Synthesis with Concurrent Error Detection. In *International Test Conference*, pages 672–678.

Zeppenfeld, J., Bouajila, A., Herkersdorf, A., and Stechele, W. (2010). Towards Scalability and Reliability of Autonomic Systems on Chip. In *3rd International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing - ISORC 2010 - Workshop on Self-Organizing Real-Time Systems*, pages 73–80. IEEE.

Zwolinski, M. (2003). *Digital System Design with VHDL*. Prentice Hall, second edition.