# Fault-Tolerant Architecture Design for Flow-Based Biochips

Morten Chabert Eskesen


Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

# Summary (English)

Microfluidic biochips revolutionise biology by placing laboratory functionality on a very small chip. In this thesis, the focus is on flow-based biochips. Flow-based microfluidic biochips are used for the manipulation of continuous fluid through fabricated microchannels, using external pressure sources or integrated mechanical micro-pumps. In these biochips, the basic building block is a microvalve, which can be fabricated at very high densities, e.g., 1 million valves per $cm^2$ [8]. By combining these valves, more complex units such as mixers, switches and multiplexers can be built. Flow-based biochips are manufactured using multilayer soft lithography.

A potential roadblock in the deployment of microfluidic biochips is the lack of test techniques to screen defective devices before they are used for biochemical analysis. Defective chips lead to repetition of experiments. This is undesirable due to high reagent cost and limited availability of samples. Flow-based biochips are also affected by faults, and the defects can escape the after-fabrication inspection and can thereby affect the operation. Recent work has addressed fault-modeling and the automated testing of flow-based biochips.

Based on these fault models and testing techniques, the objective of this thesis is to propose approaches for the fault-tolerant design of flow-based biochips, such that the biochips can tolerate several permanent faults, given a cost budget and a biochip area. During the physical design of the biochip layout, redundancy can be introduced for on-chip components such as valves, channels and microfluidic units in order to improve fault-tolerance, thereby increasing the yield.

The thesis proposes a fault model as part of the biochip architecture model and

introduces a component library with fault-tolerant components. Using these models, two algorithmic approaches to solving the problem of fault-tolerant architecture synthesis are proposed. The fault-tolerant architecture synthesis considers the application model, fault-tolerant routing and the physical constraints of the biochip such that the minimum amount of redundancy is added to achieve fault-tolerance. The proposed approaches have been evaluated using real-life case studies and synthetic benchmarks.

# Summary (Danish)

Mikrofluidiske biochips revolutionerer biologi ved at lægge laboratorie funktionalitet på en meget lille chip. Denne afhandling fokuserer på flow-baserede biochips. Flow-baserede mikrofluidiske biochips er baseret på behandling af kontinuerlig væske gennem fabrikeret mikrokanaler ved at bruge eksterne trykkilder eller integreret mikro-pumper. I disse biochips er byggestenen en mikrovalve som kan blive fabrikeret ved høj tæthed, fx. 1 million valves per $cm^2$. Ved at kombinere disse valves kan mere komplekse enheder fremstilles - herunder mixere, switches, multipleksere. Flow-baserede biochips er fremstillet ved brug af multilags blød litografi.

En potentiel forhindring i implementeringen af mikrofluidiske biochips er manglen på test-teknikker til at screene defekte enheder før de bruges til biokemisk analyse. Defekte chips fører til gentagelse af eksperimenter. Dette er uhensigtsmæssigt på grund af høje reagens omkostninger og begrænset tilgængelighed af prøver. Flow-baserede biochips er også påvirket af fejl, og disse defekter kan undslippe efter-fabrikation inspektion og dermed påvirke operationen. Fejlmodellering og automatiske test af flow-baserede biochips er blevet adresseret fornyligt i en teknisk rapport.

Baseret på disse fejlmodeller og fejlfindingsteknikker er målet for denne afhandling at foreslå tilgange til det fejltolerante design af flow-baserede biochips således at biochippen kan tolerere flere permanente fejl givet et budget og et biochip område. Under det fysiske design af biochippens layout kan redundans introduceres for komponenterne på chippen såsom valves, kanaler og microfluidiske enheder, og derved øge udbyttet af biochips.

Denne afhandling foreslår en fejlmodel som en del af biochip arkitekturmodellen og introducerer et komponentbibliotek med fejltolerante komponenter. Ved brug af disse modeller er to algoritmiske tilgange foreslået til at løse problemet med fremstilling af fejltolerante arkitekturer. Fremstillingen af fejltolerante arkitektur tager applikationsmodellen og fejltolerant rutebestemmelse på biochippen med i overvejelserne samt de fysiske begrænsinger således at den minimale mængde af redundans er tilføjet for at opnå fejltolerance. De foreslåede tilgange er blevet evalueret ved brug af real-life case studies og syntetiske benchmarks.

# Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with synthesis of fault-tolerant architectures for flow-based microfluidic Very Large Scale Integration (mVLSI) biochips. The work has been supervised by Associate Professor Paul Pop.

Lyngby, 26-June-2015

Morten Chabert Eskesen

# Acknowledgements

Firstly, I would like to thank my supervisor Paul Pop for his close involvement and excellent guidance throughout my work which have been extremely helpful. His capability to always be available for questions and discussion contributed a great deal in the completion of this thesis.

Secondly, I want to thank my colleague and close friend Andreas Hallberg Kjeldsen, who has been an invaluable partner throughout my studies for five years and contributed with small algorithmic ideas for this thesis.

Lastly, I want to extend my gratitude to my family who have supported and encouraged me. I dedicate this thesis to my brother, Julian, who has been and will always be my inspiration in life.

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Microfluidics is the science of handling and manipulating very small volumes of fluids. It is a multidisciplinary field that involves engineering, physics, chemistry, biochemistry, nanotechnology, and biotechnology. Microfluidic biochips combine different biochemical analysis functionalities, e.g. mixers, filter, detectors, on-chip. It miniaturises the macroscopic chemical and biological processes to a sub-millimetre scale [12].
There are several types of microfluidic biochip platforms. Based on the fluid manipulation on the chip, biochips can broadly be divided into two categories [8].

- Droplet-based biochips
- Flow-based biochips

In this thesis the focus is on flow-based biochips. The following sections will explain the flow-based technology and its application areas.

## 1.1 Flow-based Biochips

Flow-based biochips are fabricated using multilayer soft lithography. Figure 1.1 shows a flow-based biochip. Polydimethylsiloxane, *PDMS*, is used as the fabrication substrate [7]. PDMS is used as it is a cheap and rubber-like elastomer with good biocompatibility and optical transparency.

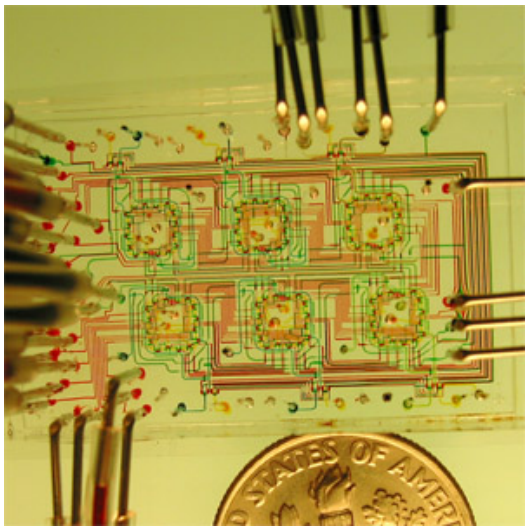


**Figure 1.1:** Flow-based biochip [1].

Flow-based biochips can have multiple physical layers, but the layers are logically divided into two types: *flow layer* and *control layer*. The flow layer is depicted in blue and the control layer in red as shown in Figure 1.2a. The liquid is in the flow layer and it is manipulated using the control layer [7].

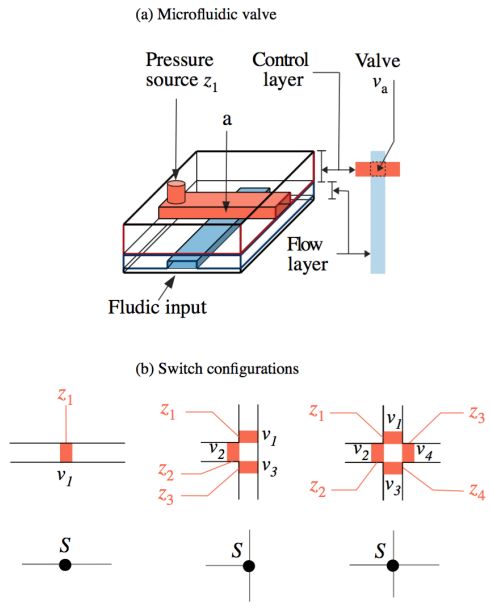



**Figure 1.2:** Flow-based valve and switch [8].

A valve (shown in Figure 1.2a) is the basic building block of such a biochip. The control layer is connected to an external air pressure source through the punch hole $z_1$. This is referred to as a control pin. The flow layer is connected to a fluid reservoir through a pump which generates the fluid flow. When the external air pressure in the control layer is not active, the fluid is permitted to flow freely, i.e. the valve is open. When the pressure source is activated, the high pressure causes the elastic control layer to pinch the underlying flow layer (point $a$ in Figure 1.2a) blocking the fluid flow, i.e. the valve is closed. These valves are used to manipulate fluids in the flow layer as the valves either restrict or permit the fluid flow. More complex units such as switches, mixer, micropumps, etc., can be formed by combining these valves. An example of valves combining to form a component is that of a switch. Three different switch configurations are depicted in Figure 1.2b. As shown here a switch can consist of one or more valves. Multiple valve switches are present at channel junctions and are used to control the path of the fluids entering the switch from different sides. The fluid flow can be generated by either using off-chip or on-chip pumps. The control layer is not necessarily above the flow layer as shown in Figure 1.2a. It can also be below the flow layer by creating a "push-up" valve, and having the control layer above the flow layer is done by creating a "push-down" valve. The connections to external ports (fluidic ports for the flow layer and pressure

sources for the control layer) are made by punching holes in the chip and placing external tubings into the punch holes [7]. All input ports are connected to the off-chip pumps.

### 1.1.1 Application Areas

Since the introduction of this technology, several biochips have been designed which target a variety of biochemical applications [3]. A few are listed below:

- *Microreaction Technology*: These chips allow the production of fine chemicals. The superior mixing and reaction control properties of microfluidic systems are used to perform chemical reaction or syntheses at much better yields and better selectivity than in conventional systems. Chemical reactions can take place much faster by reducing the diffusion length [3].

- *Cell Biology*: As the typical dimensions of cells are 5-20 $\mu m$, it is an ideal size for the size range of typical microfluidic structures. The applications within cell biology range from the observation of the physical and biological behaviour of single cells in different culturing media, chemotaxis experiments to observations of growth patterns, the guidance of growth. This can potentially be of great importance in drug research [3].

- *Diagnosis Testing*: Certain chips allow the diagnosis of diseases. Known examples are chips testing for Human Immunodeficiency Virus ($HIV$) and syphilis [8]. The chip designed for this purpose is cheap, easy to use, requires only micro-litres of the blood sample and it simultaneously tests for HIV and syphilis producing the result within 20 minutes [8].

One of the biggest beneficiaries of microfluidic devices and systems is the diagnostic market, especially molecular diagnostics. Microfluidics and miniaturisation technologies have a crucial enabling role for new product development in this field due to the required integration density, portability and speed for such applications can only be realised in miniaturised solutions. Additionally, many of the diagnostic procedures require the integration of methods of molecular biology like DeoxyriboNucleic Acid (DNA) extraction or Polymerase Chain Reaction (PCR) which can only be performed in their microfluidics-based protocols outside a specialised laboratory [3].

## 1.2 Motivation

A potential roadblock in the deployment of microfluidic biochips is the lack of test techniques to screen defective devices before they are used for biochemical analysis. Defective chips lead to repetition of experiments. This is undesirable due to high reagent cost and limited availability of samples. Recent work has addressed the fault-modeling and the automated testing of flow-based biochips [4].

Based on these fault models and testing techniques, the objective of this thesis is to propose approaches for the fault-tolerant design of flow-based biochips, such that the biochips can tolerate several permanent faults, given a cost budget and a biochip area. During the physical design of the biochip layout, redundancy can be introduced for on-chip components such as valves, channels and microfluidic units to improve fault-tolerance, thereby increasing the yield.

This thesis explores techniques and algorithms for designing fault-tolerant flow-based biochips. The purpose is to design and implement a tool to assist the designer in designing a fault-tolerant biochip. Implementing such a tool requires a biochip architecture model, biochemical application model, heuristics for introducing fault-tolerance and a way to evaluate the fault-tolerance of the biochip.

### 1.2.1 Related Work

In [8] there are proposed system models for flow-based biochips which are used in this thesis. The proposed models used in this thesis are the biochip architecture model and the biochemical application model. Additionally, [8] also contributes towards application mapping, architecture synthesis and control synthesis for flow-based biochips.

The physical placement of components is done by the designer of the flow-based biochip, which is a time-consuming and error-prone phase. In [10], an automated tool for the physical placement of components is proposed which assists the designer in choosing the best placement.

Fault-tolerant design of microfluidic biochips has been done in [2]. However, the thesis in [2] introduces fault-tolerant design for droplet-based biochips. The thesis proposes algorithms to generate application-specific biochip architectures that are able to tolerate a certain number of permanent faults. The aim of the thesis was to increase the yield of fabricated biochips.

## 1.3 Thesis Overview

This thesis is organised in eight chapters. A brief summary of the chapters are provided here.

**Chapter 2** presents the different types of faults in flow-based biochips and an automated testing strategy to determine the faults and locations thereof in a flow-based biochip.

**Chapter 3** describes the system models used in this thesis and proposes a fault-model for flow-based biochips. Furthermore it outlines the problem of application mapping.

**Chapter 4** focuses on the fault-tolerant architecture synthesis problem for flow-based biochips. It proposes two algorithmic solutions to obtain a fault-tolerant architecture.

**Chapter 5** proposes an evaluation method for architectures in order to determine if a given architecture is fault-tolerant.

**Chapter 6** describes the implementation of the tool developed in this thesis.

**Chapter 7** experimentally evaluates the proposed algorithmic approaches on a number of benchmarks. The evaluation is done in terms of solution quality and performance of the algorithms.

**Chapter 8** presents the conclusions of this thesis and the options for future work.

Chapter 2

# Faults in Flow-Based Biochips

This chapter will outline the possible faults and their causes in flow-based biochips. Furthermore, it will outline the fault-modeling of the defects and describe the recently developed automated testing technique for flow-based biochips.

## 2.1 Possible Faults and Causes

Defects in a flow-based microfluidic biochip can be attributed to fabrication steps and environmental reasons such as imperfections in molds, pollutants, bubbles in PDMS gel, and failure in hard baking. Furthermore, as feature sizes are scaled down, the sizes of and distances between microchannels are reduced in order to achieve higher degrees of microfluidic integration. This increasing density raises the likelihood of defects [4]. Some typical defects are listed below.

- *Block:* Channels may be disconnected, blocked or even missing. Figure 2.1(a)-(c) show some examples of the block defect in fabricated microfluidic biochips. The possible causes for block defects are environmental particles or imperfection in silicon wafer mold.

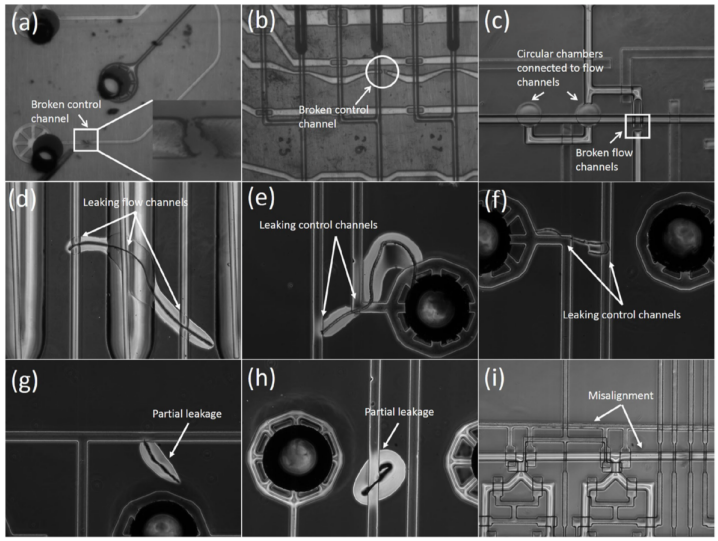



**Figure 2.1:** Possible faults in flow-based biochips [4].

- *Leak:* It is possible that some defective spots on the wall can connect independent channels. Thereby the flows in either of the channels infiltrate the other channel, which will result in a cross-contamination that can be catastrophic. The probability of a leaked channel pair increases as the length of the channels increases. Additionally the probability becomes greater if the distance between parallel channels decreases, and similarly the probability is smaller for channels that do not run in parallel. Figure 2.1(d)-(f) shows some examples of leak defects that are caused by fiber pollutant in fabricated microfluidic biochips. Some partial leak defects are shown in Figure 2.1(g) and Figure 2.1(h). These defective spots may become full leakage when high pressure is injected into the channels.

- *Misalignment:* The control layer and the flow layer of the biochip are misaligned. Figure 2.1(i) shows the defect. The result of this is that membrane valves either cannot be closed or are not even formed.

- *Faulty Pumps:* Pumps with defects fail to generate pressure when actuated. The transmission of pressure is interrupted.

- *Degradation of Valves:* The membranes of valves may lose their flexibilities or they might even be perforated after a large number of operations. The result is that the valves cannot seal flow channels.

- *Dimensional Errors:* The actual fabricated microfluidic biochip might be

too narrow compared to the designed dimensions. The mismatch of height-to-width ratio may lead to a valve that cannot be closed.

## 2.2 Defects and Fault Modeling

Despite the complexity of flow-based biochips the consequences of the defects can be described as either a block or a leak [4]. These two generic faults (block and leak) can be observed in both the control layer and the flow layer, however their respective faulty behaviours are different. Their faulty behaviours are described in Table 2.2. Therefore we can describe the possible defects in a flow-based biochip and their respective fault model, which is given in Table 2.1.

**Table 2.1:** Defects and fault modeling

| **Defect** | **Fault model** |
|---|---|
| Block | Block in control/flow channel |
| Leak / partial leak | Leak in control/flow channel |
| Misalignment between flow and control layer | Block in control channel |
| Faulty pumps | Block in control/flow channel |
| Degradation of valves | Block in control channel |
| Dimensional errors | Block in control channel |

Misalignment between flow and control layer can be modeled as the faulty behaviour of block in the control channels. This is possible as the result of the defect is that membrane valves either cannot be closed or are not even formed [4]. The behaviour of faulty pumps is similar to the faulty behaviour of a block as the transmission of pressure is interrupted [4]. Degradation of valves can be modeled as a block in a control channel as the valves cannot seal flow channels [4]. Dimensional errors have similar faulty behaviour as that of a block in a control channel as a valve cannot be closed, i.e. the flow cannot be stopped in flow channels underneath the valve [4].

**Table 2.2:** Faulty behaviour of block and leak [4]

| | Flow Layer | Control Layer |
|---|---|---|
| Block | Fluid flow cannot go through the obstacle inside the channel so transport is blocked. | Pressure cannot reach the flexible membrane, which prevents the corresponding valve from closing. |
| Leak | Fluid flow permeates the adjacent microchannels. | Control channels of two independent valves are unintentionally connected. Pressure on either valve activates both. |

## 2.3 Fault Model

The errors due to defects can be modeled in terms of faulty behaviours of valves [4]. For example a block in a flow channel can be modeled as a valve that cannot be opened, i.e. the valve cannot be deactivated. While a block in a control channel can be modeled as a valve that cannot be closed, i.e. the valve cannot be activated. Similar behaviour models can be defined for leaks.

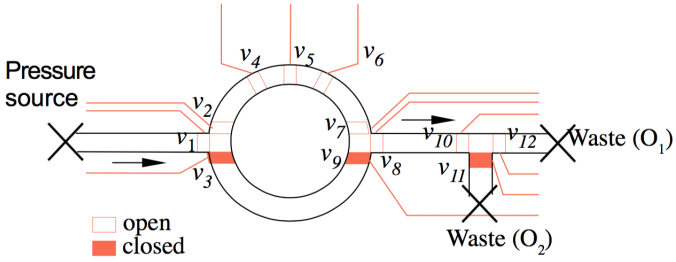



**Figure 2.2:** Simple microfluidic biochip

Figure 2.2 shows a layout of a simple microfluidic biochip with a mixer. The rectangles indicate positions of valves and the black lines indicate flow channels. Consider Figure 2.2 and consider the following defects.

- *Block in flow channel*: A block defect in the flow channel between valves $v_3$ and $v_9$ leads to the behaviour that valve $v_3$ cannot be opened. A valve and the channels connected to it are considered to be a single entity.

- *Leak in flow channel*: If a leak occurs between the flow channels $v_3$-$v_9$ and $v_2$-$v_4$, the liquid in channel $v_3$-$v_9$ infiltrates channel $v_2$-$v_4$
- *Block in control channel*: If a block occurs in a control channel then pressurised air cannot reach the flexible membrane to seal the flow channel. In this case valve $v_9$ cannot be closed.
- *Leak in control channel*: If a leak occurs between the control channels of $v_4$ and $v_9$, the two shorted valves effectively form one valve. The result is that whenever either valve is activated then both valves are activated, i.e. whenever $v_4$ is closed then $v_9$ is also closed.

## 2.4 Fault-Specific Testing Strategy

We use the testing strategy for flow-based biochips as proposed in [4]. The used test set up has feedback generated when pressure sensors are connected to the outlets and pumps are connected to the inlets. If a path exists between the pump sources (inlets) and pressure sensors (outlets), pressure sensors at the outlets will detect a high pressure generated by the pumps. The measured high pressure is defined as output "1". If all paths between inlets and outlets are blocked, pressure sensors cannot sense the high pressure injected by the pumps. This is defined as output "0". All ports in flow-based biochips are identical, regardless of functional classification. When testing, only one of the ports in the flow layer is connected to a pressure source and the rest to pressure sensors. Consequently the definitions for valve conditions can be formulated. Table 2.3 connects the logic representation of valve states to the corresponding pressure response.

**Table 2.3:** Logic representation of valve states and pressure response [4]

| Logic | Valve state | Valve condition | Pressure response |
|---|---|---|---|
| 1 | open | deactivated | high |
| 0 | closed | activated | low |

A binary pattern (test vector) is applied to all valves to set their/open close states. The actual responses of pressure sensors are compared to the expected responses. If these two sets match then the biochip is considered good.

Table 2.4 shows the testing strategy to target the faults in Table 2.2. The test effectiveness is dependent on the quality of the test patterns. The more

complicated the biochip structure is, the harder it is to determine a test pattern set that covers every fault type for each valve and channel.

**Table 2.4:** Testing strategy for different kinds of faults [4]

| | **Flow Layer** | **Control Layer** |
|---|---|---|
| Block | Position: $v_3$-$v_9$. Both valves $v_3$ and $v_9$ are deactivated to form a route *inlet*-$v_1$-$v_3$-$v_9$-$v_8$-$v_{10}$-$v_{11}$-$O_2$. If the output at $O_2$ is "0", the defect is detected. | Position: valve $v_9$. The block in the control layer prevents valve from closing. Deactivate valve $v_1$, $v_3$, $v_8$, $v_{10}$, $v_{11}$ and $O_2$ but activate the rest including $v_9$. If $O_2$ is "1" the defect is detected. |
| Leak | Position between $v_3$-$v_9$ & $v_2$-$v_4$. Deactivate valve $v_1$, $v_2$, $v_9$, $v_8$, $v_{10}$ and $v_{11}$. If high pressure is sensed at $O_2$, the leaking defect is detected. | Position: $v_7$ & $v_9$. Turn on valve $v_1$, $v_3$, $v_9$, $v_8$, $v_{10}$ and $v_{11}$ but activate $v_7$. If there is a leakage, high pressure in control channel $v_7$ will activate $v_9$ and block the route. |

Due to the difficulty of determining a test pattern set the more complicated a biochip structure is, there is a need to further abstract faults and microfluidic structures to facilitate automatic test-vector generation.

## 2.5 General Testing Strategy

Table 2.5 defines the behavioural-level fault models for a flow-based biochip. This definition is possible since the defects can be modeled as faulty behaviour of a valve and a binary logic framework can be defined where an activated valve can be defined as "0" and a deactivated valve as "1".

**Table 2.5:** Behavioural-level fault model for flow-based biochips [4]

| | **Flow Layer** | **Control Layer** |
|---|---|---|
| Block | stuck-at-0 | stuck-at-1 |
| Leak | OR-bridge (1-dominant) | AND bridge (0-dominant) |

All types of defects in control and flow channels can be mapped to specific behaviour-level fault at a valve. By virtue of this classification the testing prob-

lem is simplified from a 3-D structure to that of a 2-D design. Biochips with complicated networks of channels and valves have their test generation simplified because of this.

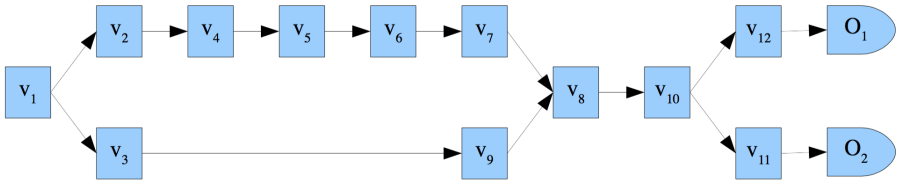



**Figure 2.3:** Schematic of a valve network corresponding to Figure 2.2. Adapted from [4]

In order to easily describe and analyse biochip channel networks, a discretised schematic is developed in [4] in place of a continuos fluid-flow topology. Figure 2.3 illustrates the design by using the biochip design in Figure 2.2. This schematic infers logic relationships that define flow-based biochips. Figure 2.3 infers that valve $v_2$ is serially connected to valve $v_4$, $v_5$, $v_6$, and $v_7$. Hence either valve can potentially block the route, i.e. there is an "AND" logic relationship among the valves. Contrary routes $v_2$-$v_7$ and $v_3$-$v_9$ are in parallel. Therefore the activation of either of the two routes can lead to output "1". There is an "OR" logic relationship between them.

The flow-based biochips can be further abstracted from the schematic representative of valve networks to valve-based logic gate circuit diagrams [4]. This is shown in Figure 2.4 using Figure 2.2 as an example.

The logic expression of Figure 2.4 is $\{O_1, O_2\} = \{v_{11}, v_{12}\} \cdot v_1 \cdot v_8 \cdot v_{10} \cdot (v_2 \cdot v_4 \cdot v_5 \cdot v_6 \cdot v_7 + v_3 \cdot v_9)$. The primary inputs are nodes in the schematic of Figure 2.3. The following is the important attributes of the logic circuit model.

- Only primary inputs (valves) and outputs (pressure sensors) have physical meaning. All other circuit connections are used to represent logical relationship. By virtue of this, it is only necessary to target faults at the primary inputs of this circuit.
- A series connection of valves in a flow route is mapped to an AND gate. Contrary a parallel connection of valves is mapped to an OR gate.

A physical defect in a flow-based biochip can be mapped to a fault at a primary

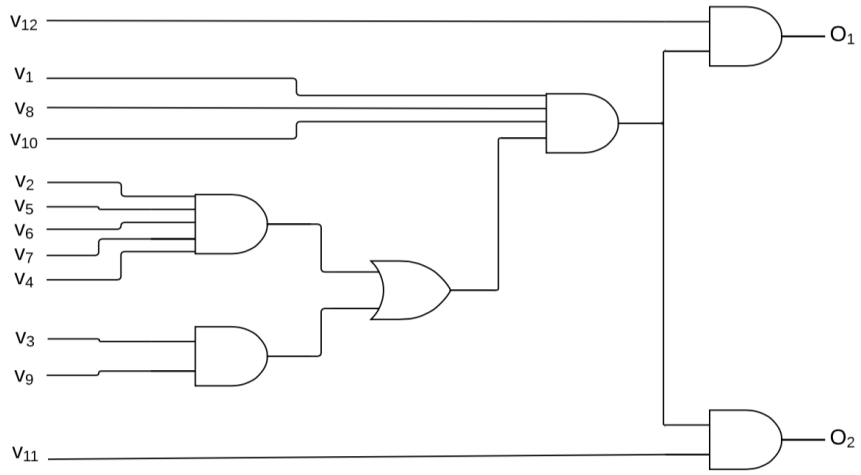



**Figure 2.4:** Logic circuit model of the biochip in Figure 2.2. Adapted from [4]

input of a logic circuit. This is based on Figure 2.4 and Table 2.5. Targeting the block defect in the flow channel $v_3$-$v_9$ is done by mapping the defect to a stuck-at-0 according to Table 2.5. Afterwards, the fault is associated with the primary input $v_3$ in the logic circuit model in Figure 2.4. A leak defect between valve $v_7$ and $v_9$ can be represented by an AND bridge fault between primary inputs $v_7$ and $v_9$ of Figure 2.4. As a result of the logic circuit model it is possible to readily determine the actual (with faults) and expected (fault-free) responses of pressure sensors and hence accelerate the search for test stimuli. If the actual output differ from the expected output, it is possible to not only conclude that the biochip is faulty, but also infer the positions and types of defects.

## 2.6 Summary

In this chapter the possible faults and causes in a flow-based biochip are outlined. The faults and causes can be attributed to fabrication steps and environmental reasons. The consequences of the faults can be modeled as either a block or a leak in flow or control channels. The errors due to defects can be modeled in terms of faulty behaviours of valves. This fact is used to devise an automated testing strategy that uses a logic circuit model to determine if a given biochip is faulty and the types and positions of the defects.

Chapter 3

# System Models

This chapter will outline the system models of flow-based biochips. We use the biochip architecture model and the biochemical application model proposed in [8]. An extension to this biochip architecture model is proposed by the addition of fault-tolerant components and a fault model. Lastly the problem of application mapping will be described.

## 3.1 Biochip Architecture Model

The biochip architecture model is composed of three models: component model, architecture model and fault model. The following subsections will explain these models.

### 3.1.1 Component Model

In [8], a dual-layer component modeling framework is proposed consisting of a *flow layer model* and a *control layer model*. The flow layer model $(\mathcal{P}, C, H)$ of each component $\mathcal{M}$, is characterised by a set of operational phases $\mathcal{P}$, the

execution time C and the geometrical dimensions $H$. The control layer model captures the valve actuation details required for the on-chip execution of all operational phases of a component.

The component library $\mathcal{L} = \mathcal{M}(\mathcal{P}, C, H)$ is shown in Table 3.1. The component library describes the set of components that a biochip architecture can have. The geometrical dimensions $H$ are given as length×width and are scaled with a unit length being equal to $150\mu m$. Therefore a length of 10 in Table 3.1 corresponds to $1500\mu m$. FT is an abbreviation for Fault-Tolerant. Considering Table 3.1 the only change from a component to its fault-tolerant version is in width, however this is not the case. The introduction of fault-tolerant components is the difference between Table 3.1 and the component library given in [8]. The components and their fault-tolerant counterpart will be described in this section.

**Table 3.1:** Component Library ($\mathcal{L}$): Flow Layer Model

| **Component** | **Phases($\mathcal{P}$)** | $C$ | $H$ |
|---|---|---|---|
| Mixer | Ip1 / Ip2 / **Mix** / Op1 / Op2 | 0.5 s | 30 × 30 |
| FT-Mixer | Ip1 / Ip2 / **Mix** / Op1 / Op2 | 0.5 s | 30 × 30 |
| Filter | Ip / **Filter** / Op1 / Op2 | 20 s | 120 × 30 |
| FT-Filter | Ip / **Filter** / Op1 / Op2 | 20 s | 120 × 60 |
| Detector | Ip / **Detect** / Op | 5 s | 20 × 20 |
| FT-Detector | Ip / **Detect** / Op | 5 s | 20 × 40 |
| Separator | Ip1 / Ip2 / **Separate** / Op1 / Op2 | 140 s | 70 × 20 |
| FT-Separator | Ip1 / Ip2 / **Separate** / Op1 / Op2 | 140 s | 70 × 40 |
| Heater | Ip / **Heat** / Op | 20° C/s | 40 × 15 |
| FT-Heater | Ip / **Heat** / Op | 20° C/s | 40 × 30 |
| Storage | Ip or Op | - | 90 × 30 |
| FT-Storage | Ip or Op | - | 90 × 40 |
| Metering | Ip / **Met** / Op1 / Op2 | - | 30 × 15 |
| Multiplexer | Ip or Op | - | 30 × 10 |

#### 3.1.1.1 Fault-tolerant Switch

Figure 3.1a shows switches formed by combining valves. A switch can consist of one valve that restricts or allows flow in a channel or a switch can consist of more than one valve [8]. Multiple valve switches are present at channel junctions and are used to control the path of the fluids entering the switch from different sides. The conceptual view of a switch is shown below the actual component design. A switch can fail in different ways. Each valve in the switch can be stuck open or stuck closed. For example, if valve $v_3$ in the third switch with four valves is

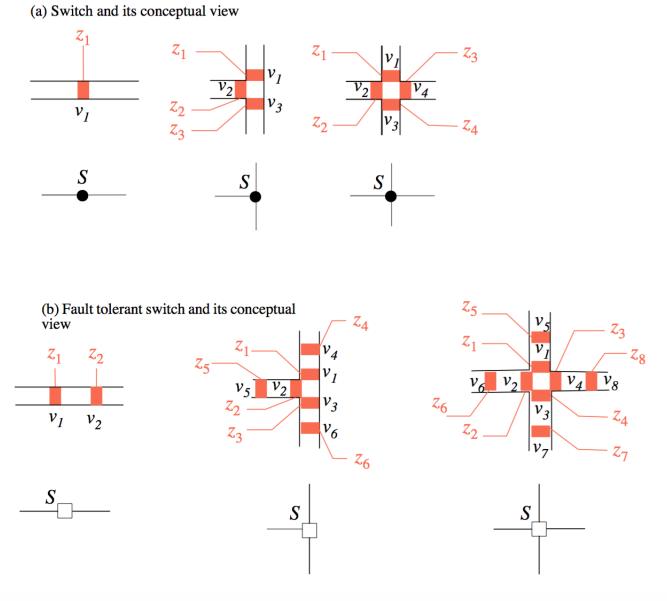



**Figure 3.1:** Switch and fault-tolerant switch

stuck closed the effect is that fluid is restricted from entering the switch from the channel controlled by $v_3$ and the fluid is restricted from leaving the switch going to the channel controlled by $v_3$. However if the valve $v_3$ is stuck open, the fluid is allowed to enter the switch from the channel controlled by $v_3$ and leave the switch going to any other channel. But if the fluid is entering the switch from any other valve ($v_1$, $v_2$ or $v_4$) than $v_3$, the fluid can only leave the switch going to the channel controlled by $v_3$.

Figure 3.1b shows the fault tolerant version of each switch in Figure 3.1a. These are called fault-tolerant switches or *ft-switches* for short. The ft-switch is formed by combining valves as regular switches and function like regular switches. The conceptual view of a ft-switch is shown below the actual component design. A ft-switch has an extra added valve for each valve in a regular switch. The ft-switch tolerates faults that causes the valves to be stuck open as the extra added valve can compensate for the failing valve. However, if a valve is stuck closed, the switch is still faulty and the fluid is still restricted from leaving the switch to the channel the valve controls, and from entering the switch from that channel. A switch cannot consist of more than 4 valves, i.e. it cannot restrict / allow flow in more than 4 channels. This also applies for the ft-switch however the maximum number of valves is 8, but it cannot restrict / allow flow in more than 4 channels. This is due to cleaning difficulties and for the reason that fluid

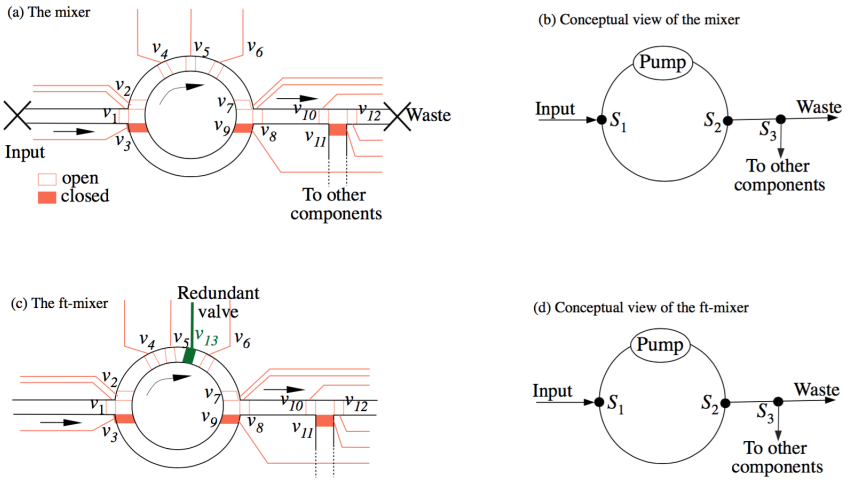



**Figure 3.2:** Mixer and fault-tolerant mixer

can be trapped inside the switch if it has more valves (i.e. channels).

### 3.1.1.2 Fault-tolerant Mixer

Figure 3.2a shows a pneumatic mixer, which is implemented by nine microfluidic valves, $v_1$ to $v_9$. Figure 3.2b shows the conceptual view of the same mixer. The valve set $\{v_4, v_5, v_6\}$ works as an on-chip pump which is used to perform the mixing. The valve set $\{v_1, v_2, v_3\}$ is termed as switch $S_1$ and facilitates the inputs. The valve set $\{v_7, v_8, v_9\}$ is termed as switch $S_2$ and facilitates the outputs. The mixer has five operational phases. The first two phases represent the input of two fluid samples to be mixed, which is followed by the mixing phase. The mixed sample is then transported out of the mixer in the last two phases. The mixer can fail in various ways. Each valve in the mixer can be stuck closed or stuck open. The two channels inside the mixer can also fail as both channels can suffer from a block defect or a leakage. For example, any valve in the valve-set $\{v_4, v_5, v_6\}$, that acts as the pump can suffer from being stuck open or closed and the mixer will therefore not be able to perform its mixing operation.

Figure 3.2c shows a fault-tolerant version of the pneumatic mixer called fault-tolerant mixer or *ft-mixer* for short. Figure 3.2d shows the conceptual view of the same ft-mixer. The ft-mixer has the same operational phases as the regular mixer and performs them in the same way. The difference is the added valve $v_{13}$.

The purpose of this valve is to tolerate the fault of any valve in the pump being stuck open. In case, any of the valves in the pump are stuck open, the pump will still be functional by virtue of $v_{13}$ and the mixing can still be performed. However, in case, any of the other possible faults discussed previously happen, the ft-mixer will not be able to perform the mixing operation. It is possible to have a pump consisting of four valves as long as the amount of space between the valves is kept small, i.e. the valves should be close together to perform the pumping action. The result is that the pump can still function as intended and thereby perform the mixing.

It is possible to route through the mixers as valves are controlling its mixing operation. The valves are opened and closed by the designer. Therefore, the fluid can be routed through the mixer without being unintentionally affected by a mixing operation. Additionally, it is possible to route through the mixers even with faults affecting it. For example, if the mixer suffers from a blocked channel ($v_3$-$v_9$), the mixing operation cannot be performed, but it can use the other non-faulty channel ($v_2$ to $v_7$) to route through the mixer. Furthermore, it is possible for the mixer to receive input from both sides and similarly send output to both sides.

#### 3.1.1.3 Fault-tolerant Heater, Filter, Detector and Separator

This section outlines the components heater, filter, detector, separator and their fault-tolerant counterparts. These components are similar in the sense that they consist only of a channel where the channel has a specific operational functionality. The functionality is the only thing that differs between the components. Therefore Figure 3.3 will be explained first with the possible faults and the introduced fault-tolerant component. Afterwards the details specific to each component will be described.

Figure 3.3a shows a component which consists of two valves $v_1$ and $v_2$. Figure 3.3b shows the conceptual view of the same component. The valve $v_1$ is termed as switch $S_1$ and facilitates the input. The valve $v_2$ is termed as switch $S_2$ and facilitates the output. The two valves also trap the fluid in the channel such that the component-specific operation can take place. The component can fail in various ways. The valves $v_1$ and $v_2$ can either be stuck open or stuck closed. If either valve is stuck open the component cannot trap the fluid as needed by using its valves. If $v_2$ is stuck closed the fluid cannot leave the channel. Additionally if the channel of the component is blocked, the channel is not usable and similarly if $v_1$ is stuck closed the channel is not reachable. Furthermore the channel can also suffer from a leakage.

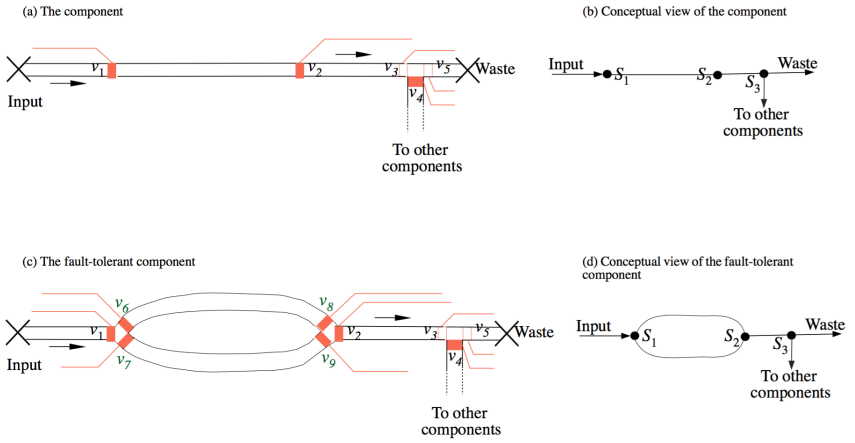



**Figure 3.3:** Heater, Filter, Detector and Separator and their fault-tolerant counterpart

Figure 3.3c shows a fault-tolerant version of the component which consists of six valves $v_1$, $v_2$, $v_6$, $v_7$, $v_8$ and $v_9$. Figure 3.3d shows the conceptual view of the same fault-tolerant component. The valve set $\{v_1, v_6, v_7\}$ is termed as switch $S_1$ and facilitates the input. The valve set $\{v_2, v_8, v_9\}$ is termed as switch $S_2$ and facilitates the output. The component has two channels, where the fluid can be trapped while the component performs its component-specific operational phase(s). The valve set $\{v_6, v_8\}$ can trap the fluid between them and similarly $\{v_7, v_9\}$. The fault-tolerant component can tolerate one channel suffering from a block defect as it is possible to use the other non-faulty channel. It can also tolerate $v_1$ and $v_2$ being stuck open as the valves are no longer essential for trapping the fluid inside the channels. However, if either $v_1$ or $v_2$ are stuck closed then it is not possible to enter or leave the component, respectively. Therefore, the component would still be considered faulty in the event of these two faults occuring.

**Detector** The channel in a detector acts as a detection channel which identifies and quantifies analyte by various methods. Fluorescence is one among the many techniques to perform detection [6]. The fault-tolerant version is called *ft-detector*. The detector and ft-detector have 3 operational phases - Input / Detect / Output.

**Heater** The fluid trapped in the heating channel is heated by a metal plate under the flow layer, which is activated when needed. The fault-tolerant version is called *ft-heater*. The heater and ft-heater have 3 operational

phases - Input / Heat / Output.

**Filter** The channel acts as a filtration channel, which treats and isolates the analyte of interest from crude biological samples. Current methods of filtration include the use of membranes, gels, electrophoresis and dielectrophoresis [6]. The fault-tolerant version is called *ft-filter*. The filter and ft-filter have 4 operational phases - Input / Filter / Output1 and Output2.

**Separator** The channel acts as a separation channel which isolates samples after other processes. Capillary electrophoresis is a used technique to perform the separation. In this technique, the ionic particles are isolated by their charge and frictional forces [6]. The fault-tolerant version is called *ft-separator*. The separator and ft-separator have 5 operational phases - Input1 / Input2 / Separate / Output1 / Output2.

#### 3.1.1.4 Fault-tolerant Storage

Figure 3.4 shows a storage component with eight reservoirs $Res_1$-$Res_8$ and 28 valves total. The storage component is used to store fluid output from chemical or biological reactions on the biochip. The fluid is stored in the channels between the valve sets $\{v_5, v_5\}$ and $\{v_6, v_6\}$. The storage component only receives input or outputs fluid. The storage component can fail in many ways as it has a lot of valves and channels. Each valve in the component can be stuck open or stuck closed or any channel in the component could be suffering from a block defect or leakage. However since the storage component has so many channels and valves it has some fault-tolerance in itself depending how much storage is needed in a given application at the same time. If it is necessary to store four fluids at the same time and four of the storage reservoirs' channel are blocked, then the application can still run.

Additionally, it can also receive input and output from/to both sides and thereby the two channels controlling the input can also be blocked and the storage component is still usable. Furthermore, similar to the mixer, it is possible to route through the storage component as the operations of the storage are performed using only valves and thereby the fluid cannot be unintentionally affected in any way.

Figure 3.5 shows a fault-tolerant version of the storage component with nine reservoirs $Res_1$-$Res_9$ and 34 valves. It is called fault-tolerant storage or *ft-storage* for short. The ft-storage has one more channel to store fluid than the regular storage component, thereby adding additional fault-tolerance to the storage component if there is a need to store eight fluids at the same time and one of the reservoir's channel is blocked.

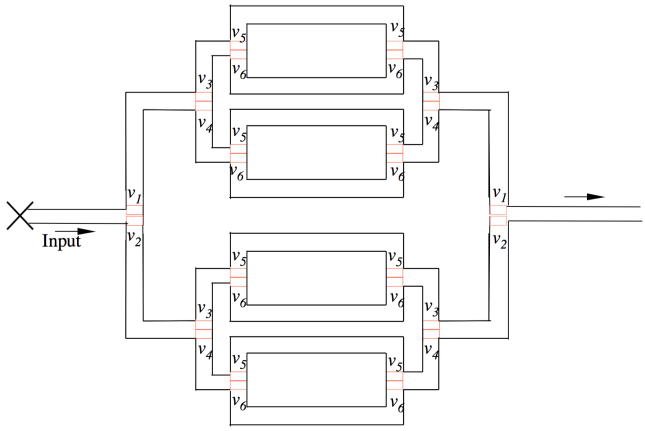



**Figure 3.4:** Storage component

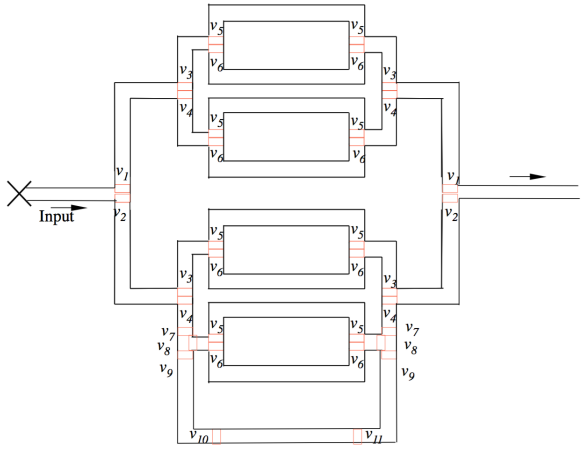



**Figure 3.5:** Fault-tolerant storage component

### 3.1.2 Architecture Model

The architecture model outlined is proposed in [8]. The biochip architecture is modeled by a topology graph $\mathcal{A} = (\mathcal{N}, \mathcal{S}, \mathcal{D}, \mathcal{F}, \mathcal{K}, c)$, where $N$ is a finite set of vertices, $\mathcal{S}$ is a subset of $\mathcal{N}$, $\mathcal{S} \subseteq \mathcal{N}$, $\mathcal{D}$ is a finite set of directed edges, $\mathcal{F}$ is a finite set of flow paths and $\mathcal{K}$ is a finite set of routing constraints. A vertex $N \in \mathcal{N}$ has two distinguished types, a vertex $S \in \mathcal{S}$ represents a switch, whereas a vertex $\mathcal{M} \in \mathcal{N}, \notin \mathcal{S}$ represents a component or input/output node. A directed edge $D_{i,j}$ denotes a direct connection from vertex $N_i$ to $N_j$ where $N_i, N_j \in \mathcal{N}$. A flow path, $F_i \in \mathcal{F}$, is a subset of two or more directed edges of D, $F_i \subseteq \mathcal{D}$, $|F_i| > 1$, denoting a direct communication link between two vertices $\in N$ using a chain of directed edges $\in \mathcal{D}$. A routing constraint $K_i \in \mathcal{K}$ is a set of flow paths that are mutually exclusive with the flow path $F_i \in \mathcal{F}$. These flow paths can not be activated in parallel. The function $c(y)$ where $y$ is either a directed edge $D_{i,j} \in \mathcal{D}$ or a flow path $F_i \in \mathcal{F}$ denotes the routing latency, i.e. the time required by a fluid sample to traverse $y$.

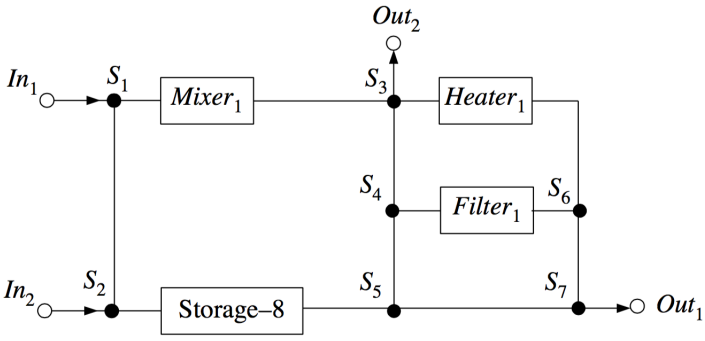



**Figure 3.6:** Biochip architecture

Figure 3.6 shows a biochip architecture equipped with two inputs, two outputs, one mixer, one heater, one filter and eight storage reservoirs $Res_1$-$Res_8$ which are contained in the component 'Storage-8'.

#### 3.1.2.1 Netlist

The netlist defines which components are to be placed on the biochip and their interconnections. Therefore, it models the functionality of the chip but not the

physical layout. The netlist is modeled as a directed graph where components are nodes and the connections between components are the edges, i.e. the nodes are $\mathcal{N}$ from the architecture model and the edges are $\mathcal{D}$.

### 3.1.3 Fault Model

The following is a proposed fault model for flow-based biochip architectures. The faults are modeled as a set of faults $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, v, c)$, where $\mathcal{VF}$ is a finite set of valve faults and $\mathcal{CF}$ is a finite set of channel faults. A valve fault $VF(N, w, t) \in \mathcal{VF}$ can happen to a vertex $N \in \mathcal{N}$ in the architecture. The component model, which specifies the valves, is part of the architecture. The valve affected by the fault is denoted by $w$. The type of fault is denoted by $t$. The type of fault is either stuck open or stuck closed. The $v$ in the fault model specifies the maximum number of valve faults happening at the same time in the biochip architecture. Therefore more than $v$ valve faults, $VF \in \mathcal{VF}$, can be specified in the fault model, but no more than $v$ valve faults are happening at the same time.
A channel fault can happen to a component $M \in \mathcal{N}, \notin \mathcal{S}$ in the architecture. A channel fault can also happen to a connection $D_{i,j} \in \mathcal{D}$. The type of fault is denoted by $t$, where the type of fault can either be a block defect or leak. A channel fault on a component is denoted by $CF(M, t) \in \mathcal{CF}$. A channel fault on a connection is denoted by $CF(D_{i,j}, t) \in \mathcal{CF}$. The $c$ in the fault model specifies the maximum number of channel faults happening in the biochip architecture at the same time. Hence more than $c$ channel faults, $CF \in \mathcal{CF}$, can be specified in the fault model, but no more than $c$ are happening at the same time.

This means that the set of faults in the fault model are divided into two categories - valve faults and channel faults where valves can either be stuck open or stuck closed and channels can either suffer from a block defect or leakage. The set of faults is specific to a biochip architecture. The faults are specific, i.e. when the faults are specified, they denote a specific channel or valve suspected to be faulty. The faults in the fault model can be faults, known by the designer, to be common faults in the architecture.

Considering the architecture in Figure 3.6 a possible fault model could be: $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, 2, 2)$.

**Table 3.2:** Example set of valve faults $\mathcal{VF}$

| **Name** | **Vertex** ($N \in \mathcal{N}$) | **Valve affected** ($w$) | **Type** ($t$) |
|---|---|---|---|
| $VF_1$ | $Mixer_1$ | $v_5$ | Open |
| $VF_2$ | $S_6$ | $v_3$ | Open |
| $VF_3$ | $S_5$ | $v_2$ | Open |
| $VF_4$ | $S_3$ | $v_3$ | Open |

**Table 3.3:** Example set of channel faults $\mathcal{CF}$

| **Name** | **Component** ($M \in \mathcal{N}, \notin \mathcal{S}$) / **Connection** $D_{i,j} \in \mathcal{D}$ | **Type** ($t$) |
|---|---|---|
| $CF_1$ | $Heater_1$ | Block |
| $CF_2$ | $Filter_1$ | Block |
| $CF_3$ | $S_2 \rightarrow$ Storage-8 | Block |
| $CF_4$ | $S_1 \rightarrow Mixer_1$ | Block |

A possible fault scenario using this fault model example could be the combination $\{VF_1, VF_3, CF_1, CF_3\}$. In this scenario these four faults are considered to be affecting the architecture. This fault scenario has a lot of effects on the architecture. $CF_3$ specifies that the connection from $S_2 \rightarrow$ Storage-8 is blocked and therefore reaching the Storage-8 component from $In_2$ a longer route is necessary going through $S_1$, $Mixer_1$, $S_3$, $S_4$, $S_5$ and then reaching Storage-8. $VF_1$ effectively means that $Mixer_1$ is faulty and cannot perform its mixing operation as valve $v_5$ (in the pump) is stuck open, however it is still possible to route through $Mixer_1$. $CF_1$ means that the $Heater_1$ is suffering from a block defect and therefore the channel is not usable which renders $Heater_1$ faulty. $VF_3$ specifies that $v_2$ of $S_5$ is stuck open and therefore when reaching $S_5$ in the architecture the routing possibilities depend on which channel is used to enter the switch. Valve $v_2$ restricts / allows the flow to channel between $S_5$ and Storage-8. Therefore, if $S_5$ is entered from Storage-8 the fluid can leave $S_5$, using any other channel. However, if $S_5$ is entered from any other channel it can only use the channel going to Storage-8.

A fault scenario is a combination of faults, where any combination of the faults specified in the fault model is possible to create a fault scenario. The defects in the biochip architecture are tested for and known after fabrication. Effectively, this eliminates the tolerance of faults presented at run-time. A valve stuck closed and a blocked channel fault lead to the same result as the channel cannot be used due to the valve controlling the input and/or output of the channel.

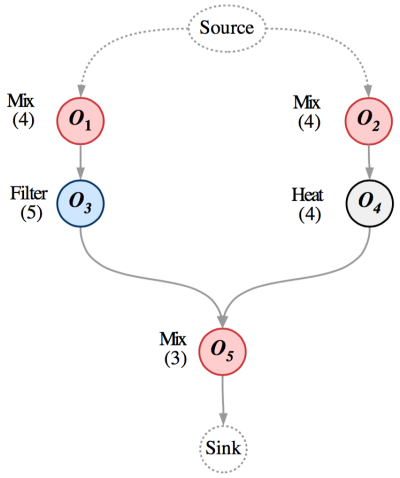



**Figure 3.7:** Application model

## 3.2 Biochemical Application Model

The biochemical application model outlined is proposed in [8]. The biochemical application is modeled by a sequencing graph. The graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ is directed, acyclic and polar. Therefore, the graph will always have a source vertex with no predecessors and a sink vertex with no successors. Each vertex $O_i \in \mathcal{O}$ represents an operation in the biochemical application. The dependency constraints in the assay are modeled by the edge set $\mathcal{E}$. An edge $e_{i,j}$ from $O_i$ to $O_j$ describes that the output of $O_i$ is the input of $O_j$. All the inputs need to arrive before an operation can be activated. The operations in the application can be bound to a component by the binding function: $\mathcal{B} : \mathcal{O} \rightarrow \mathcal{M}$. Each vertex has an associated weight $C_i^{M_j}$. This denotes the execution time required for operation $O_i$ to be completed on component $M_j$. The application is assumed to have a deadline which it should complete within. This deadline is denoted as $d_{\mathcal{G}}$. Furthermore it is also assumed that the biochemical application has been correctly designed, i.e. all the operations will have the correct volume of liquid available for their execution.

Figure 3.7 shows a simple application model that has 3 mixing operations ($O_1$, $O_2$ and $O_5$), 1 filtration operation ($O_3$) and 1 heating operation ($O_4$). The execution times for each operation are given in the parameter below the operation type. The execution times provided in Table 3.1, are of the actual functional phase, which is given in bold in the table. The execution times from this table

are the typical execution times for the component types, however a biochemical application description may specify a longer time, if required for a particular operation.

## 3.3 Application Mapping

Application mapping is the problem of mapping a biochemical application onto a given biochip architecture. Mapping the application onto the architecture involves *binding* onto the allocated components, *scheduling* the operations and performing the required fluidic *routing*. The architecture is modeled as described in subsection 3.1.2 and the application is modeled as outlined in section 3.2. The allocated components are therefore captured by the vertex set $\mathcal{M}$, $\mathcal{M} \in \mathcal{N}$, in the architecture model $\mathcal{A}$. The component placement and interconnections are also captured in the topology graph $\mathcal{A}$ modeling the architecture.
During the binding step, each vertex $O_i$, $O_i \in \mathcal{O}$, representing a biochemical operation in the application model is bound to an available component $M_j$, i.e. $\mathcal{B}(O_i) = M_j$. Since the fluid transport latencies in microfluidic chips are comparable to the operation execution times, fluid routing also needs to be considered. Therefore, the binding function must also capture the binding of the edge set $\mathcal{E} \in \mathcal{G}$ to an available route. The route can be a flow path, $F \in \mathcal{F}$, or a collection of flow paths called a *composite route*. A composite route is used when the source and destination components are such that no direct flow path exists between them.
A scheduling strategy is necessary to efficiently execute the biochemical operations on the biochip components, while considering the dependency captured by the application model and the resource constraints captured by the biochip architecture. Alongside, the set of operations $\mathcal{O} \in \mathcal{G}$ given in the application model the edge set $\mathcal{E} \in \mathcal{G}$ also needs to be scheduled on the biochip while considering the routing constraints. Before scheduling a specific edge, the implementation needs to evaluate if a flow path $F \in \mathcal{F}$ is sufficient to bind the edge or if a composite route is necessary.
Throughout the scheduling phase, the storage requirement analysis needs to be performed. Consequently, after completion of an operation, a decision on whether the output fluid should be moved to a storage reservoir or not, needs to be made.

## 3.4 Summary

In this chapter, the used biochip architecture model has been outlined. The biochip architecture model has been proposed before. However an addition is proposed to the component model in form of an extended component library with fault-tolerant components in the flow layer. Furthermore, a fault model for describing faults in the biochip architecture has been proposed with the distinction between valve faults and channel faults. Lastly, the application model used in the thesis has been defined and the problem of mapping a biochemical application onto a given biochip architecture has been outlined.

Chapter 4

# Architecture Synthesis

This chapter focuses on the architecture synthesis, i.e. synthesising a fault-tolerant architecture. It takes the initial architecture (as a netlist) to be made fault-tolerant, a component library, an application model and fault model as an input and synthesises a fault-tolerant architecture. The chapter will state the formal problem formulation. Furthermore it will present the solution strategy.

## 4.1 Problem Formulation

The problem can be formulated as follows: Given a netlist (components and their interconnections), a component library $\mathcal{L}$, an application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ with a deadline $d_{\mathcal{G}}$ and a fault model $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, v, c)$, determine a fault tolerant netlist such that the biochip is fault tolerant and it minimises the cost of the architecture.

The architecture provided by the designer is assumed to be an architecture that the designer knows well. By virtue of this the designer can specify exact faults that are typical faults or critical faults in the architecture. The inputs are modeled as described in chapter 3.

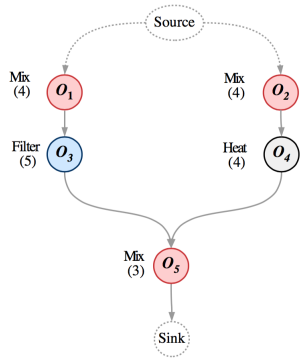



**Figure 4.1:** Example application graph $\mathcal{G}$ for architecture synthesis

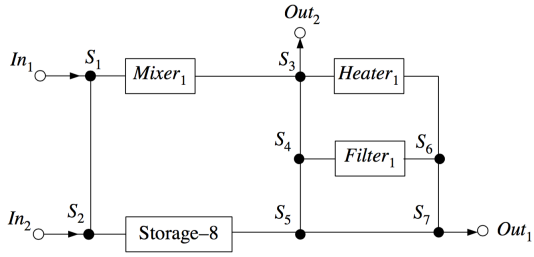



**Figure 4.2:** Example architecture for architecture synthesis

**Table 4.1:** Example set of valve faults $\mathcal{VF}$ for architecture synthesis

| **Name** | **Vertex** $(N \in \mathcal{N})$ | **Valve affected** $(w)$ | **Type** $(t)$ |
|---|---|---|---|
| $VF_1$ | $Mixer_1$ | $v_5$ | Open |
| $VF_2$ | $S_6$ | $v_3$ | Open |
| $VF_3$ | $S_5$ | $v_2$ | Open |
| $VF_4$ | $S_3$ | $v_3$ | Open |

**Table 4.2:** Example set of channel faults $\mathcal{CF}$ for architecture synthesis

| **Name** | **Component** ($M \in \mathcal{N}, \notin \mathcal{S}$) / **Connection** $D_{i,j} \in \mathcal{D}$ | **Type** ($t$) |
|---|---|---|
| $CF_1$ | $Heater_1$ | Block |
| $CF_2$ | $Filter_1$ | Block |
| $CF_3$ | $S_2 \rightarrow$ Storage-8 | Block |
| $CF_4$ | $S_1 \rightarrow Mixer_1$ | Block |

Considering the application graph $\mathcal{G}$ in Figure 4.1 and the architecture $\mathcal{A}$ in Figure 4.2 we are interested in synthesising an architecture such that is tolerant to the faults given in Table 4.1 and Table 4.2 and that the cost of the architecture is minimised.
Figure 4.3a shows a straightforward solution to this problem, whereas Figure 4.3b shows an optimised solution to this problem. In the straightforward solution, which we assume that the designer could create without the help of our tool, a redundant component is added for each component that would be unable to perform its operation considering the fault model which thereby implies that the application graph in Figure 4.1 cannot complete. Fault-tolerant switches, $S_3, S_5$ and $S_6$, have been added to compensate for the failing valves. Similarly a redundant channel has been added to compensate for the blocked channel $S_2 \rightarrow$ Storage-8. No channel has been added to make the $S_1 \rightarrow Mixer_1$ channel redundant as it is not needed by virtue of $Mixer_2$. The cost of the architecture in Figure 4.3a is calculated to be 129. The calculation is the sum of the total number of valves and the total number of channels.

In the optimised solution the fault tolerant variants of filter, heater and mixer have replaced the original components. Consequently the components tolerate the faults in the fault model and are therefore able to perform their operations. Two redundant channels have been added to compensate for the two blocked channels. Routing is still possible even though there are valves failing in some switches. The cost of the architecture in Figure 4.3b is calculated to be 96. The optimised solution is therefore less costly than the straightforward solution, while at the same time tolerating all the fault scenarios possible under the given fault model. Considering the application $\mathcal{G}$ in Figure 4.1 there are four routes that will be affected by the valves failing in the switches. When the filtration and heating operation finish the resulting fluids from both operations will need to route back to the $FT\text{-}Mixer_1$. Consequently they need to route through $S_6$ to $S_7$ however as the valve $v_3$ is the failing valve it is still possible use the channel $S_6 \rightarrow S_7$. Recall that when the valve stuck open in the switch restricts / allows flow to the channel, we want to route to, then the fault is inconsequential. However when arriving to $S_5$ the failing valve $v_2$ in $S_5$ forces the fluid to go through Storage-8 and therefore the route is a bit longer than the

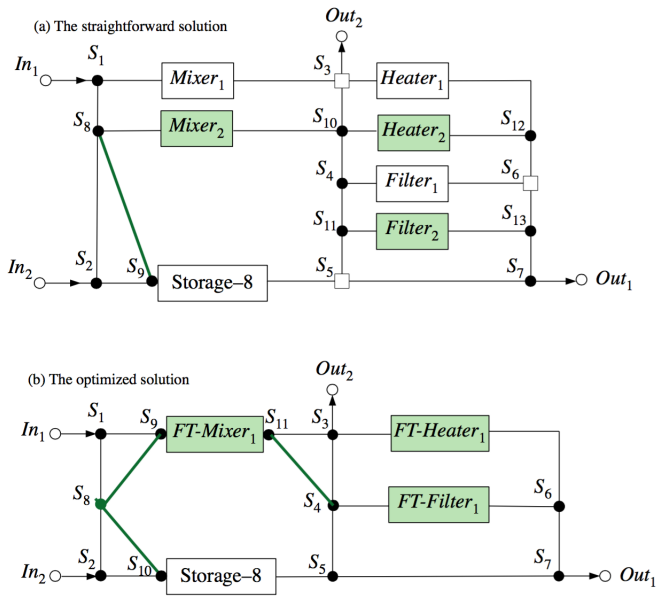



**Figure 4.3:** Possible solutions for fault-tolerant architecture synthesis of the example

optimal route. The other two routes affected are $FT\text{-}Mixer_1 \rightarrow FT\text{-}Heater_1$ and $FT\text{-}Mixer_1 \rightarrow Out_2$. The optimal routes would be to use the channel $FT\text{-}Mixer_1 \rightarrow S_3$ however as the valve $v_3$ of $S_3$ is failing it is not possible to use this channel and the desired optimal route. However the routes can use the redundant channel $FT\text{-}Mixer_1 \rightarrow S_4$. This allows the usage of the channel $S_4 \rightarrow S_3$ and thereby route to $FT\text{-}Heater_1$ and $Out_2$. It is possible now to route to any channel we want to from $S_3$ as we are routing to $S_3$ from the channel that the stuck open valve restricts / allows flow from.

The assumption is that the fault-tolerant architecture synthesis is part of a methodology. The steps in the methodology are as follow.

1. *Architecture design.* An architecture $\mathcal{A}$ is designed by the designer. This architecture can be generated as an application-specific architecture, which is possible using a tool developed in [8].

2. *Fault-tolerant architecture synthesis.* A fault-tolerant netlist is synthesised for the architecture $\mathcal{A}$ designed in the previous step while considering an application graph $\mathcal{G}$ with a deadline $d_{\mathcal{G}}$ and a fault model $\mathcal{Z}$.

3. *Physical architecture synthesis.* A physical architecture is synthesised (generating the placements of the component) using the fault-tolerant netlist generated in step 2. The physical architecture synthesis is possible using the tool developed in [10].

4. *Fabrication.* The biochips are fabricated using the architecture $\mathcal{A}$ obtained in the previous step.

5. *Testing.* All the biochips are tested to determine if they have permanent faults using techniques such as the one proposed in [4], which is outlined in section 2.5. The exact locations of faults are determined during this step. If there are any permanent faults on the biochip not present in the fault model $\mathcal{Z}$, the biochip is discarded.

The focus of this thesis is on the second step of the methodology: the fault-tolerant architecture synthesis problem. The thesis proposes two solutions based on metaheuristics, Simulated Annealing and Greedily Randomized Adaptive Search Procedure. The flow of the metaheuristic implementations is outlined in Figure 4.4. A metaheuristic explores the solution space using design transformations called moves. These are applied to the current solution in order to obtain neighbouring architecture alternatives.

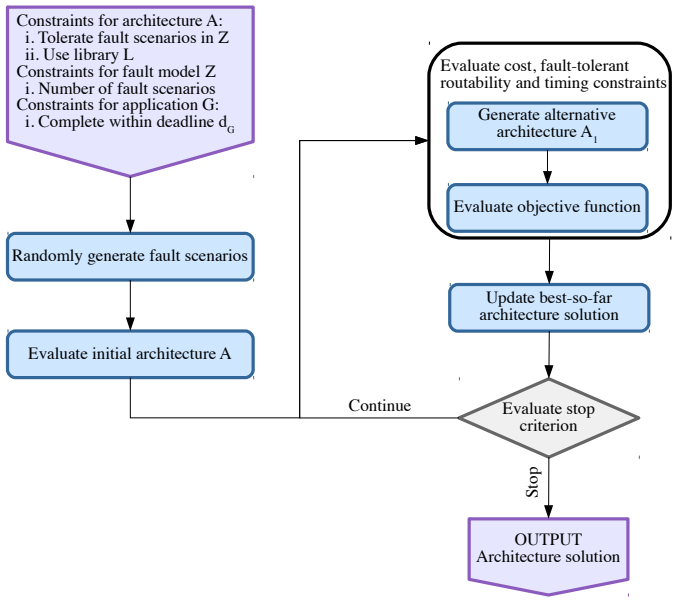



**Figure 4.4:** Architecture synthesis

One of the architecture alternatives is then selected to be the current solution. Therefore each architecture alternative is evaluated using an objective function which is described in chapter 5. When a new architecture solution improves the objective function it is accepted and chosen as the current solution. However in some cases the Simulated Annealing metaheuristic accepts worse solution in order to escape the local minima. The metaheuristic continues to apply the moves on the determined current solution and uses the objective function to evaluate the obtained neighbouring architectures. The search terminates when a stop criterion is satisfied.

## 4.2 Design Transformations

In order to generate an alternative architecture a way to obtain neighbouring solutions has to be defined. This is also called *moves* as mentioned. It is possible to specify multiple moves. The possible moves implemented in this thesis are as follows.

1. *Add redundant component*: In order to make a component redundant another component of that type is added to the architecture.

2. *Convert regular component to the fault-tolerant version*: By virtue of the component library $\mathcal{L}$ given in Table 3.1 with fault-tolerant components a component can be changed to its fault-tolerant version provided it is a component with a fault-tolerant version.

3. *Add connection between two components*: A redundant connection is added between the two components.

4. *Remove redundant component*: A redundant component is removed from the architecture. Note that it is only a redundant component that can be removed, i.e. none of the original components in the architecture are removed.

5. *Convert fault-tolerant component to the regular version*: This move is the reverse of move 2 as no component is fault-tolerant in the initial architecture.

6. *Remove connection*: A redundant connection is removed from the architecture. Note that it is only a redundant connection that can be removed. None of the original connections in the architecture are removed.

Let us consider the architecture from Figure 4.5a as the current solution $\mathcal{A}^{current}$. By applying the following moves: add redundant connection between $S_4$ and $Heater_1$, convert $Mixer_1$ to its fault-tolerant version and add redundant component $Filter_2$, we obtain the neighbouring architectures from Figure 4.5b, c and d respectively.

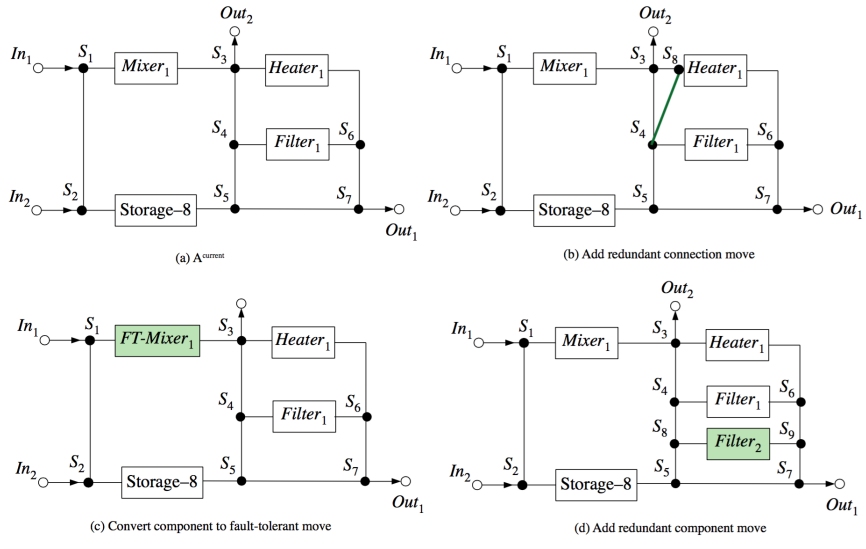



**Figure 4.5:** Example of neighbouring solutions

## 4.3 Simulated Annealing Architecture Synthesis

As a solution to the fault-tolerant architecture synthesis problem we propose a Simulated Annealing approach. Simulated Annealing, henceforth *SA*, is a metaheuristic whose inspiration originates from a technique in metallurgy. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state [11]. In the following sections the concept of SA will be explained and how it is implemented for the fault-tolerant architecture synthesis problem.

### 4.3.1 Concept

The idea of SA is to randomly create an initial solution, $s_0$, and choose an initial temperature, $t_0$. A neighbouring solution of $s_0$ is explored to see if that solution is an improvement. The quality of the solutions is determined by a *cost function*. If the neighbouring solution improves the current solution, that solution is chosen as the current one and the temperature is reduced according to a *cooling schedule*. However if the neighbouring solution is worse it is accepted with a certain probability. If the algorithm only accepted a solution determined to be an improvement it would converge towards the local optimum and not the global optimum.

The idea is that when the temperature is high a large part of the solution space is explored. Accordingly at high temperatures almost any solution is accepted regardless of the quality. As the temperature decreases the simulation gradually becomes more and more reluctant to accept inferior solutions. When the temperature becomes low enough SA will only accept solutions that improves the current solution. This causes it to become stable at a local optimum. If the cooling is slow then the local optimum is a good approximation on the global optimum. The probability, $p(s)$, that an inferior solution, $s$, will be chosen is given by the exponential expression.

$$p(s) = e^{-\delta/t}$$

where $\delta$ is the cost increase from the previous solution, and $t$ is the temperature. The probability is inspired by the laws of thermodynamics.

### 4.3.2 Implementation

SA takes as input the initial architecture $\mathcal{A}$, the component library $\mathcal{L}$, the application graph $\mathcal{G}$, and a fault model $\mathcal{Z}$ and produces a fault-tolerant netlist of minimum cost. For the evaluation of each architecture alternative the SA-based architecture synthesis uses the cost function described in chapter 5. By the fact that SA receives the initial architecture $\mathcal{A}$ as an input there is no need to randomly generate an initial solution. The algorithm is shown in Figure 4.6.

```
1: function SimulatedAnnealing(A, L, G, Z)
2:     t ← t_0                                  ▷ Temperature is initialized, t_0 > 0
3:     while t ≤ t_termination do
4:         for i ← 0 to iterations do
5:             A' ← RandomNeighbor(A)
6:             if Cost(A') < Cost(A) ∨ Random(0, 1) < e^{−δ/t} then
7:                 A ← A'
8:             end if
9:         end for
10:        t ← ReduceTemperature(t)             ▷ Temperature is reduced
11:    end while
12:    return A
13: end function
```

**Figure 4.6:** Implementation of Simulated Annealing

Deciding whether or not to accept the architecture $\mathcal{A}'$ or keep the current one $\mathcal{A}$ is done in line 6. A superior architecture is always accepted ($\delta < 0$) but an inferior solution is accepted with the probability $p(s) = e^{-\delta/t}$. The criterion is met by generating a number uniformly at random between 0 and 1, which must be less than the acceptance probability, $p(s)$.

The SA-based synthesis generates new architecture alternatives by performing moves on the current solution which is line 5 of the algorithm. The moves are described in detail in section 4.2. The moves are chosen uniformly at random. Each move is randomised in itself. Adding a connection picks two components uniformly at random and adds a connection between them. Similarly converting a component to its fault-tolerant version move choses the component uniformly at random, provided it has a fault-tolerant version. This randomness applies to each move done by SA to obtain a neighbouring architecture.

The temperate cooling schedule consists of four parameters: initial value, reduction function, termination value and number of iterations at each temperature. The initial value should be high such that initially the acceptance rate is close to 100%. This ensures that the entire solution space is explored. The reduction function determines the speed of the temperature cooling. The implementation of SA uses this reduction function.

$$\text{ReduceTemperature}(t) = k \cdot t$$

where $k < 1$. In order to ensure the cooling down is slow a value of 0.999 has been used for $k$.
Performing many iterations at each temperature ensures a more thorough exploration of the solution space. Considering that superior solutions are accepted

unconditionally spending more time at each temperature has a tendency to lead to better final solutions. The number of iterations at each temperature, *iterations*, can be used to scale the running time of the algorithm to ensure more exploring of the solution space. A value of 1 has been used to obtain results. Experimenting more with the number of iterations is a possible optimisation. The cooling should continue until the temperature is zero. A value of 0.1 for the termination temperature has provided good results in this thesis. However often the heuristic settles for a non-improving stable solution long before the temperature reaches zero. There could also be experimented more with the termination temperature.

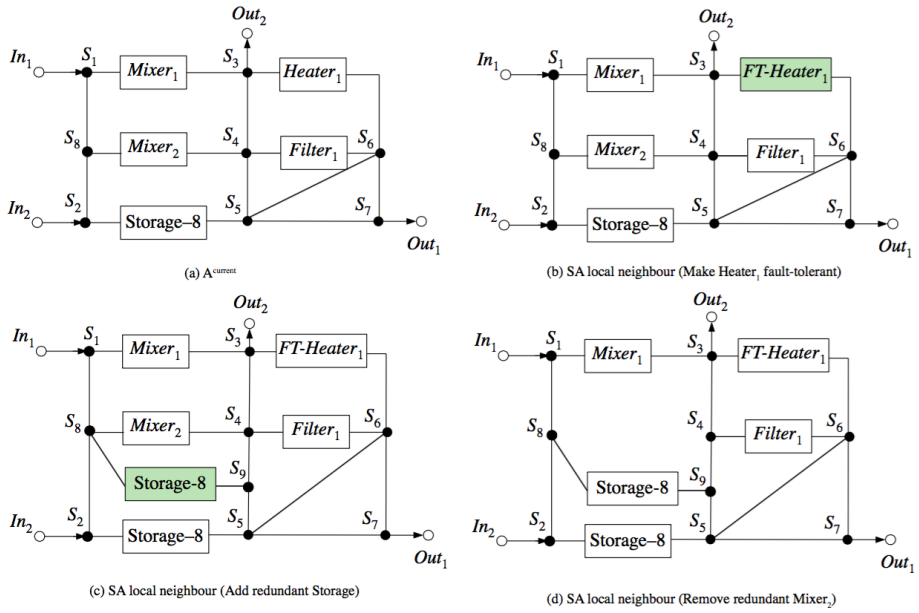



**Figure 4.7:** Simulated Annealing example

Let us consider the example in section 4.1 with the initial architecture given in Figure 4.2, the application in Figure 4.1 and the faults in Table 4.1 and Table 4.2 to illustrate how SA works. Figure 4.7a is the current solution, $\mathcal{A}^{current}$, where a number of moves have been applied to the initial architecture in order to obtain $\mathcal{A}^{current}$. From here we can apply most of the moves described in section 4.2. Uniformly at random we pick a move, which gives us the architecture alternative Figure 4.7b. This move is accepted since it now tolerates the channel fault of $Heater_1$, which means it improves the current solution and is thereby chosen as the current solution, $\mathcal{A}^{current}$. From this solution we apply a random move

again to obtain the architecture in Figure 4.7c, which is adding a redundant storage component. This move is accepted and chosen as $\mathcal{A}^{current}$ although it does not improve the current solution. It only adds more valves and connections without contributing to the completion of the application. The move is accepted due to the randomness of SA when deciding whether to accept inferior solutions. As in the previous steps we pick a random move to see how it affects the current solution. This move chosen is removing the redundant component, $Mixer_2$. However this move is not accepted by SA as $Mixer_1$ is affected by a fault such that is unable to perform mixing, and by virtue of removing $Mixer_2$ the architecture cannot perform mixing. Therefore this move is not accepted even though SA randomly accepts inferior solutions. This continues until SA reaches its termination temperature.

## 4.4 GRASP Architecture Synthesis

Greedily Randomized Adaptive Search Procedure, henceforth *GRASP*, is a metaheuristic for combinatorial optimisation [9]. Combinatorial optimisation is a problem that consists of finding an optimal object from a finite set of objects.

### 4.4.1 Concept

GRASP is an iterative metaheuristic. Each iteration consists of two phases: a construction phase and a local search phase. In the construction phase the GRASP algorithm builds a solution based on a list of candidate design transformations. The local search phase explores the neighborhood of the built solution to find the local optimum. GRASP runs for a number of iterations and returns the best solution found, i.e. the found solution that minimises the cost function the most.
In a greedy algorithm a solution is constructed one element at a time. At each step in the construction a set $C$ of candidate elements that can be added to the solutions at that step is constructed. A greedy function is applied to each element in $C$ and the elements are ranked according to their greedy function values. A best ranked element is added to the solution. By virtue of this the set $C$ is updated and the greedy function changes. The process continues until $C = \varnothing$. By randomly generating initial solutions a multi-start procedure will eventually find the global optimum. Using greedy solutions as starting points for local search however in a multi-start method will generally lead to good although often suboptimal solutions due to the small amount of variability in

greedy solutions and the small likeliness that a greedy starting solution is in the vicinity of the global optimum [9]. A *semi-greedy* heuristic adds variability to the greedy algorithm. The candidate elements are ranked according to their greedy function values. Afterwards the well ranked candidates are placed in a *restrictive candidate list* (RCL) and an element from RCL is selected at random and added to the solution. There are two schemes to build an RCL [9]:

1. In the *cardinality* based scheme an integer $k$ is fixed and the $k$ top ranked elements are placed in the RCL
2. In the *value* based scheme all candidate elements with greedy function values within $a\%$ of greedy value are placed in the RCL, where $a \in [0, 100]$

The GRASP algorithm therefore builds initial solutions by applying design transformations on elements from the RCL.

### 4.4.2 Implementation

The implementation of GRASP in this thesis is inspired by the implementation given in [5] and adapted to the fault-tolerant architecture synthesis problem. The implementation is shown in Figure 4.8. GRASP takes as input the initial architecture $\mathcal{A}$, the component library $\mathcal{L}$ and a fault model $\mathcal{Z}$. Additional parameters to the algorithm are the number of iterations, $NoI$, for GRASP to run, the size of the candidate list $cl$, the number of unsuccessful iterations, $ui$, after which the algorithm increases the size of $cl$ with $cli$. Similarly there is a parameter specifying the number of iterations, $NbI$, for the local search.

The algorithm initializes the cost of the best known solution to the cost of the initial architecture. Furthermore the number of unsuccessful iterations $\#ui$ is initialized as 0 where an unsuccessful iteration is defined as not improving the best known solution. GRASP performs two phases: solution construction and local search. During the construction phase GRASP creates the candidate list $CL$ of $cl$ potential candidate components that will be transformed. The selection of potential candidates is done by ranking each component $M \in N$ of $\mathcal{A}$ by the number of fault scenarios affecting the component. A fault scenario is affecting a component if it contains a valve fault or channel fault affecting the component or if it contains a channel fault that blocks an input or output channel of the component. Only $cl$ components with the highest ranking are selected by GRASP. However the number $cl$ of potential candidate components does not remain constant throughout the search. If $ui$ number of iterations has passed since improving the best known solution $\mathcal{A}^{best}$ the size $cl$ of the potential

```
1: function GRASP(A, L, G, Z, NoI, cl, ui, cli, NbI)
2:     A^best ← A
3:     Cost^best ← Cost(A)
4:     #ui ← 0
5:     while i < NoI do
6:         CL ← CreateCandidates(A, cl)                    ▷ Phase 1
7:         RCL ← ChooseCandidates(CL)
8:         A' ← ApplyDesignTransformations(A, RCL)
9:         A' ← HillClimbing(A', NbI)                      ▷ Phase 2
10:        if Cost(A') < Cost^best then
11:            A^best ← A'
12:            Cost^best ← Cost(A^best)
13:            #ui ← 0
14:        else
15:            #ui ← #ui + 1
16:        end if
17:        if #ui ≥ ui then
18:            cl ← cl × cli
19:            #ui ← 0
20:        end if
21:        i ← i + 1
22:    end while
23:    return A^best
24: end function
25: function HillClimbing(A, iterations)
26:    j ← 0
27:    while j < iterations do
28:        A' = RandomNeighbor(A)
29:        if Cost(A') < Cost(A) then
30:            A ← A'
31:        end if
32:        j ← j + 1
33:    end while
34:    return A
35: end function
```

**Figure 4.8:** Implementation of GRASP

candidate list is increased by $cli$ (lines 17-20).
From the candidates list $CL$ GRASP selects a random subset $RCL$ of candidates (line 7). The size of the subset is chosen uniformly at random ranging from one candidate to the full set of potential candidates. Each component from the restricted candidate list RCL is then randomly applied to one of five design transformations (line 8 in the algorithm):

1. Add a redundant component to the architecture to compensate for the component
2. Convert the component to its fault-tolerant version
3. Convert the component to its fault-tolerant version and add a redundant incoming connection to the component
4. Convert the component to its fault-tolerant version and add a redundant outgoing connection from the component
5. Convert the component to its fault-tolerant version and add redundant connections for the component both incoming and outgoing

In the second phase GRASP performs local search (line 9). In this GRASP implementation a simple Hill Climbing, henceforth *HC*, algorithm is used. The HC algorithm generates new architecture alternatives by performing moves on the current solution (line 28). The possible moves are described in section 4.2. The moves are chosen uniformly at random. Similarly to SA the moves are also randomized in itself, i.e. it will pick two random components to connect when adding a connection. It then evaluates the architecture alternative and if it is deemed less costly it is chosen as the new solution in HC (line 29-31). When the local search finishes it will return the best solution found. If the solution found by HC is better than the best known solution in GRASP the new solution is chosen as the best solution and $\#ui$ is reset to 0 as it has been a successful iteration. If it is deemed more costly than the best known solution then it is an unsuccessful iteration and $\#ui$ is incremented by 1 (line 10-16). The GRASP algorithm returns the best found solution $\mathcal{A}^{best}$ in terms of the objective function defined in chapter 5.

The GRASP algorithm consists of many defined parameters. The number of iterations of the algorithm, size of the candidate list, number of unsuccessful iterations after which the algorithm increases the size of the candidate list with a certain amount. Lastly the number of iterations in the local search also has to be defined. In this thesis the number of iterations of GRASP has been 50 which has provided good results. The initial size of the candidate list has been 2 where it has been increased by a factor of 2 each time the maximum number of

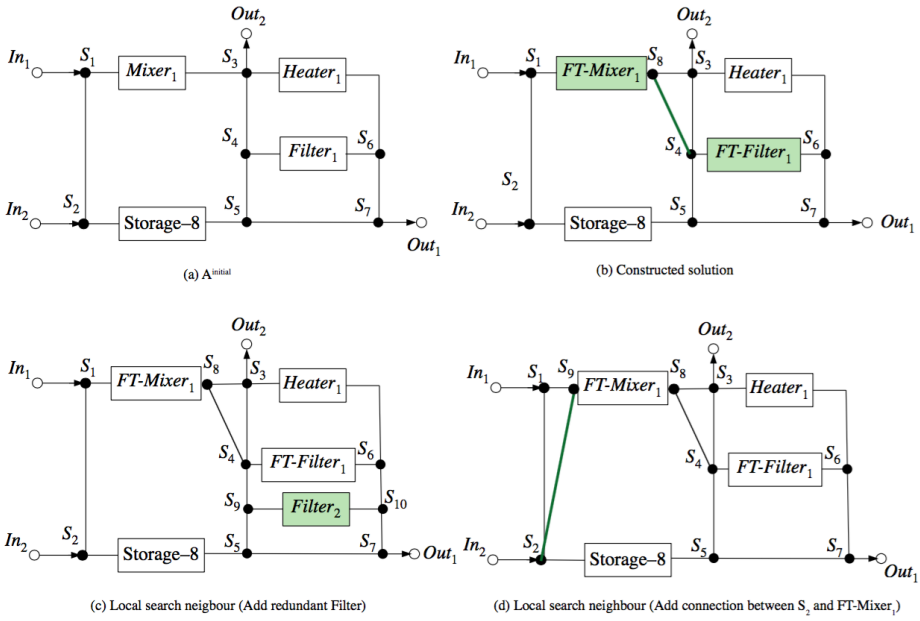



**Figure 4.9:** Greedily Randomized Adaptive Search Procedure example

unsuccessful iterations has been reached. The number of unsuccessful iterations has been ranging from 5 to 10 where both have provided good results. The number of iterations in the local search phase has ranged between 50 and 100 where good results have been produced within this range.

Let us consider the example in section 4.1 with the initial architecture given in Figure 4.2, the application in Figure 4.1 and the faults in Table 4.1 and Table 4.2 to illustrate how GRASP works. Each iteration of GRASP starts with the initial architecture, $\mathcal{A}^{initial}$, shown in Figure 4.9a. The candidate list has well ranked components, where the ranking is decided by the amount of fault scenarios affecting the component. Therefore $Mixer_1$, $Heater_1$ and $Filter_1$ are in this candidate list. Randomly a subset of this candidate list is chosen and each component in the subset is randomly applied to one of the five moves described earlier. $Mixer_1$ is converted to its fault-tolerant version and an outgoing connection is added, $FT\text{-}Mixer_1 \rightarrow S_4$, and $Filter_1$ is converted to its fault-tolerant version. This gives us the constructed solution shown in Figure 4.9b, which is the first phase of GRASP. Then the local search starts from Figure 4.9b to obtain the local optimum, i.e. it only accepts improving solutions. At random the local search picks a move from the possible moves described in section 4.2. It adds a redundant $Filter_2$ to the architecture, which provides the architecture in Figure 4.9c. This architecture is however not accepted as it

adds more valves and connections without contributing to the completion of the application. $FT\text{-}Filter_1$ can perform the filtering and therefore the redundant $Filter_2$ is not needed. Therefore Figure 4.9b is still the current solution and it applies another a random move, which is adding a connection between $S_2$ and $FT\text{-}Mixer_1$. This move contributes to the completion of the application as it was not possible to route from $In_1$ or $In_2$ to $FT\text{-}Mixer_1$ due to blocked channels. The local search continues until the maximum number of iterations in the local search has been reached. When the local search finishes GRASP will update the best known solution if, and only if, the solution found by local search is deemed better than the best known. Afterwards GRASP restarts from the initial architecture, $\mathcal{A}^{initial}$.

## 4.5 Summary

This chapter specifies that the problem addressed in this thesis is obtaining a fault-tolerant netlist given an initial architecture, a component library, an application graph, and a fault model. The general flow of the architecture synthesis is described which consists of generating an architecture alternative by performing a design transformation where six possible design transformations have been defined. The chapter proposes two solutions to the fault-tolerant architecture synthesis problem, Simulated Annealing and Greedily Randomized Adaptive Search Procedure. These two algorithms are metaheuristics that have been adapted to this problem. Both proposed solutions rely on randomisation to obtain architecture alternatives.

Chapter 5

# Architecture Evaluation

This chapter focuses on architecture evaluation. The chapter will define how an alternative architecture generated by SA and GRASP is evaluated. The objective function consists of three parts: graph connectivity, scheduling of the application onto the architecture and the cost of the architecture. These three parts will be explained in detail in this chapter.

## 5.1 Objective Function

Before evaluating an architecture the set of fault scenarios, $\mathcal{FS}$, must be randomly generated from the fault model $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, v, c)$. The number of fault scenarios is given by the designer. The generated fault scenarios are iterated and each iteration applies a fault scenario to the architecture, i.e. the faults in the fault scenario are injected into the architecture (see Figure 5.1). In each iteration the connectivity of the architecture, $ft$, and the finish time, $\delta$, of the application on the architecture are determined. The architecture evaluation is then the sum of three variables.

$$Objective(\mathcal{A}) = \left( \sum_{f \in \mathcal{FS}}^{\mathcal{FS}} \neg ft \right) \times W_{ft} + \left( \sum_{f \in \mathcal{FS}}^{\mathcal{FS}} max(0, \delta - d_{\mathcal{G}}) \right) \times W_s + Cost_{\mathcal{A}}$$

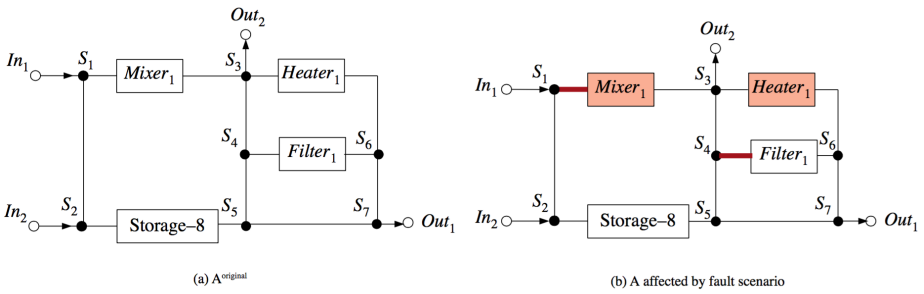



**Figure 5.1:** Architecture affected by fault scenario

$\left(\sum_{f \in \mathcal{FS}} \neg ft\right)$ is the number of fault scenarios that causes the architecture to not be connected. Connectivity is denoted by $ft$ where $ft$ will be 1 if the architecture is connected and 0 if it is not connected. The number is then multiplied with $W_{ft}$ which denotes a penalty value for the connectivity. Consequently this variable will become smaller the more fault scenarios that pass the connectivity test. The penalty value $W_{ft}$ is specified as 10000.

$\left(\sum_{f \in \mathcal{FS}} max(0, \delta - d_{\mathcal{G}})\right)$ denotes the scheduling of the application on the architecture affected by the different fault scenarios. In each fault scenario the maximum of either 0 or the application finish time minus the application deadline is added to the sum. Thereby the sum grows larger as the application is not schedulable on the architecture or if it completes after the deadline. The sum is then multiplied with $W_s$ which denotes a penalty value for the schedule. Therefore the variable will grow larger as the application is not able to complete within its deadline or no schedule has been found. The penalty value $W_s$ is defined as 5000.

$Cost_{\mathcal{A}}$ denotes the physical constraints. This variable is the sum of the total number of valves and the total number of channels in the architecture.

Let us consider Figure 5.1a as the architecture to be evaluated, $\mathcal{A}^{original}$. Consider the fault scenario, where the channel $S_1 \rightarrow Mixer_1$ is blocked and the channel of $Heater_1$ suffers from a block defect. Additionally valves $v_6$ (a valve in the pump) of $Mixer_1$, $v_1$ (channel towards $S_3$) and $v_3$ (channel towards $S_5$) of $S_4$ are stuck open. The effects of this fault scenario is shown in Figure 5.1b, where channels marked with a thick red line represent unusable channels and components marked with red are unable to perform their operation. Therefore

we are unable to perform mixing, heating and route to $Filter_1$. This injection of the faults in a fault scenario is done for all generated fault scenarios, and the connectivity and scheduling are determined for each fault scenario.

The following sections will describe how the fault scenarios are generated, how the connectivity of the architecture is determined and how scheduling the application onto the architecture is done.

## 5.2 Generation of Fault Scenarios

The fault scenarios, $\mathcal{FS}$, are randomly generated from the fault model $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, v, c)$. A fault scenario $f \in \mathcal{FS}$ is a set of faults containing a set of valve faults from the set of valve faults, $\mathcal{VF}$, in the fault model and a set of channel faults from the set of channel faults, $\mathcal{CF}$, in the fault model. The fault scenarios are generated such that they are unique, i.e. $f \in \mathcal{FS}$ will occur once and only once.

The random generation of fault scenarios is divided into two phases.

**First phase** The first phase consists of generating all the possible combinations of subsets of both the set of valve faults, $\mathcal{VF}$, and the set of channel faults, $\mathcal{CF}$. Recall that $v$ is the maximum number of valve faults happening in the architecture at any point and likewise $c$ is the maximum number of channel faults. Therefore all permutations of the set of valve faults needs to be generated where cardinality of the subsets are $0 \leq k \leq v$. Likewise for the set of channel faults where the cardinality of the subsets are $0 \leq j \leq c$.

**Second phase** The second phase picks a random subset from all the possible combinations of subsets of valve faults. Similarly it picks a random subset from the subsets of channel faults. Combining these two subsets constitutes a fault scenario, $f \in \mathcal{FS}$. This continues until the specified number of fault scenarios to generate are generated.

Normally we would have to use all the fault scenarios when doing the tests for connectivity and scheduling. However, as these are too numerous we have decided to use a subset. By virtue of this we are not guaranteed to create a fault-tolerant architecture which is fault-tolerant to all scenarios. However, our argument is that by using a subset, we can synthesise an architecture that can tolerate most of the fault scenarios. This argument has been investigated in

chapter 7. However, it is not a problem if an architecture is not fault-tolerant. As mentioned in section 4.1 this tool is part of a methodology. If after testing we determine that a fault, which is not tolerated, is present, we will discard the chip.

The implementation of the generation of fault scenarios is shown in Figure 5.2. $NoFS$ is the number of fault scenarios to be generated.

```
1:  function GenerateFaultScenarios(𝒵, NoFS)
2:      vsubsets ← PossibleCombinations(𝒱ℱ, v)                ▷ Phase 1
3:      csubsets ← PossibleCombinations(𝒞ℱ, c)
4:      i ← 0
5:      ℱ𝒮 ← ∅
6:      while i < NoFS do                                     ▷ Phase 2
7:          v ← RandomSet(vsubsets)
8:          c ← RandomSet(csubsets)
9:          f ← v ∪ c
10:         if f ∉ ℱ𝒮 then   ▷ If the fault scenario does not exist add it to ℱ𝒮
11:             ℱ𝒮 ← ℱ𝒮 ∪ f
12:             i ← i + 1
13:         end if
14:     end while
15:     return ℱ𝒮
16: end function
17: function PossibleCombinations(set, k)
18:     j ← 1
19:     subsets ← ∅
20:     while j ≤ k do
21:         r ← permutations(set, j))
22:         subsets ← subsets ∪ r
23:     end while
24:     subsets ← subsets ∪ ∅          ▷ The empty set (no faults) is possible
25:     return subsets
26: end function
```

**Figure 5.2:** Implementation of random generation of fault scenarios

## 5.3 Connectivity

Due to permanent faults, an architecture can become disconnected and thereby routing of fluid to the desired destination is no longer possible. Therefore it is

```
1: function IsConnected(𝒜)
2:     start ← ChooseRandomInput(𝒜)
3:     visited ← ∅                          ▷ Keep track of the visited components
4:     Q.enqueue(start)                                  ▷ Q is a FIFO queue
5:     visited ← visited ∪ start
6:     while Q is not empty do
7:         v ← Q.dequeue()
8:         for all connection from v to w in 𝒜.OutConnections(v) do
9:             if connection not faulty and w ∉ visited then
10:                visited ← visited ∪ w
11:                Q.enqueue(w)
12:            end if
13:        end for
14:    end while
15:    if (𝒩 \ 𝒜.inputs) ∈ visited then      ▷ If all vertices N ∈ 𝒩 excluding
   inputs are in the visited set. The architecture is connected
16:        return true
17:    else
18:        return false
19:    end if
20: end function
```

**Figure 5.3:** Implementation of graph connectivity algorithm

important that an architecture is connected. The initial architecture given by the designer is assumed to be connected. The connectivity of the architecture is determined by using the algorithm Breadth First Search or *BFS* for short. BFS is an algorithm for traversing a graph data structure. The BFS algorithm starts at one of the input nodes and traverses the architecture. If BFS visits all the components, excluding the input nodes, in the architecture then the architecture is connected. The input nodes are excluded due to being a source of input and they never receive any fluid from other components on the chip. Contrary if some components are not visited by the BFS then the architecture is not connected. The implementation considers connections (channels) that are blocked and switches that are affected by a valve fault and therefore have their routing affected. The implementation of the BFS algorithm is shown in Figure 5.3. If in a given fault scenario a switch is suffering from more than one valve fault that causes it to disallow fluid going to a certain component the affected connection is deemed faulty and removed from the architecture while being affected by that fault scenario. It is added again to the architecture when the architecture is restored from being affected by the fault scenario. Similarly blocked channels are considered. Therefore the implementation shown in Figure 5.3 is possible. The connectivity test is done for each fault scenario in the set of generated fault scenarios.

## 5.4 List Scheduling

An architecture is only fault-tolerant if it can run the application within its deadline. To determine the finishing time of the application, on a given architecture affected by a fault scenario, we use the List Scheduling Algorithm, henceforth *LS*. Mapping the application onto the architecture involves binding of operations onto the allocated components, scheduling the operations and performing the required fluidic routing as outlined in section 3.3. LS is a heuristic approach to solve the problem of application mapping in a computationally efficient manner. Together with the operation binding and scheduling, the heuristic approach also considers the fluidic routing and channel contention. The input to this tool is a netlist, which is not yet routed, i.e. the physical placement of components is not known yet. We only know the exact routing latencies if we know the routes. However, the routing latencies should not be completely ignored when determining a schedule using LS. Therefore we assume that the designer gives an average latency, which we use when determining a schedule. This may mean that the application is actually not schedulable as we could be using routing latencies, which are smaller than those resulted after the physical synthesis. However it is still a reasonable estimation of the application completion time. In case the application is actually not schedulable, this will be known

after the testing, when we have the physical architecture. If the application turns out not to be schedulable, the chip is discarded.

LS is a while loop that runs until all operations ($\mathcal{O}$) and edges ($\mathcal{E}$) from the application ($\mathcal{G}$) are scheduled. The operations are topologically sorted based on the dependency constraints. At each step a ready operation and its edges are bound and scheduled to a component. The algorithm tries all possible bindings and chooses a binding that produces the shortest completion time for that operation. The list of ready operations are prioritised using the urgency criteria. The urgency of an operation is specified as the length of the longest path from the operation to the sink, i.e. summing the execution weights of the vertices. When an operation has been scheduled its ready successor operations are added to the list of ready operations. If no component is found during the binding then there is no schedule for the application on the architecture. If the number of ready operations exceeds the number of available resources the most urgent operations are scheduled and the remaining ones are deferred. Ready operations are defined as operations whose predecessors have completed. Determining routes between components is done by using BFS where the BFS considers channels affected by faults and switches affected by valve faults as when determining the connectivity.

The implementation of LS in this thesis is given in Figure 5.4. The application is preprocessed such that two operations are added with execution times of 0. Recall the application $\mathcal{G}$ has a source and a sink as shown in Figure 3.7. These two operations are added where the source is added such that all the operations which originally had no dependencies (input operations) depend on the source operation. Contrary the sink is added such that all operations who originally no other operation depended on (output operations), the sink operation depends on the completion of them. Thereby the completion time of the application graph $\mathcal{G}$ onto the architecture $\mathcal{A}$ is the finish time of the sink operation. Furthermore the source operation provides the LS algorithm with a starting point.

In the $BindAndSchedule()$ function the algorithm tries all the possible bindings for an operation and chooses the one that produces the shortest completion time for the operation. It also considers the routing time and routing constraints. If the component is occupied by another operation it will consider the time it takes for the operation occupying the component to route to the storage. Therefore a storage reservoir is only used if the component to which the previous operation was bound is needed for performing another operation. The function $ScheduleOperation()$ then schedules the operation on the component specified. It also binds and schedules the edges such that the channel(s) used by the edges cannot be used by other edges during that time. The operation also schedules the flows to storage if needed for the previous operation occupying the component. If no route can be found to a component needed by the operation or if no

```
1: function ListScheduling(A, G)
2:     PQ.put(G.source)                                   ▷ PQ is a priority queue
3:     while PQ is not empty do
4:         o ← PQ.ExtractMax()
5:         success ← BindAndSchedule(o)
6:         if success is true then
7:             for all op ∈ ReadySuccessors(o) do
8:                 PQ.put(op)
9:             end for
10:        else
11:            return d_G × 2          ▷ No schedule is found and a high value is
   returned to make the cost of the architecture higher
12:        end if
13:    end while
14:    return G.sink.finishtime
15: end function
16: function BindAndSchedule(operation)
17:    time ← ∞
18:    best ← null
19:    components ← A.GetComponentsForOperation(operation) ▷ The
   architecture keeps track of faulty components
20:    for all component ∈ components do
21:        t = ReadyTime(operation, component)
22:        if t < time then
23:            time ← t
24:            best ← component
25:        end if
26:        if best ≠ null then
27:            ScheduleOperation(operation, best) ▷ This also schedules the
   edges and renders the channels unusable while the operation is using them
28:            return true
29:        else
30:            return false          ▷ The operation is not possible to schedule
31:        end if
32:    end for
33: end function
```

**Figure 5.4:** Implementation of the List Scheduling algorithm

component is able to perform that operation then no schedule exists and the LS algorithm returns the a high value to make the cost of the architecture higher, i.e. $d_{\mathcal{G}} \times 2$.

## 5.5 Summary

This chapter defines how an architecture is evaluated. The architecture is evaluated by determining if the architecture is connected and if the application is able to finish within its deadline when affected by a fault scenario in the set of generated fault scenarios. Connectivity of the architecture is determined using the algorithm Breadth First Search. The schedule for the application graph on the architecture is determined using the List Scheduling algorithm. The fault scenarios are generated at random from the fault model where the number of generated fault scenarios is determined by the designer. Additionally the objective function also considers the physical constraints of the architecture, which is the number of valves and channels (connections) in the architecture.

Chapter 6

# Analysis, Design and Test

This chapter describes how the tool has been implemented using Python (version 3.4) as the programming language. The input to the tool are given as JavaScript Object Notation (*JSON*) files and the tool also outputs a JSON file.

## 6.1 Design

The implementation has been done by dividing responsibilities into separate modules and classes. In total, the tool consists of 8 modules and 27 classes. To explain how the tool is implemented Figure 6.1 shows a high level class diagram. Figure 6.1 does not show all the classes to simplify the class diagram. Furthermore not all methods and attributes in the classes are shown.

In the following sections each module will be described and each class within the module will be clarified.

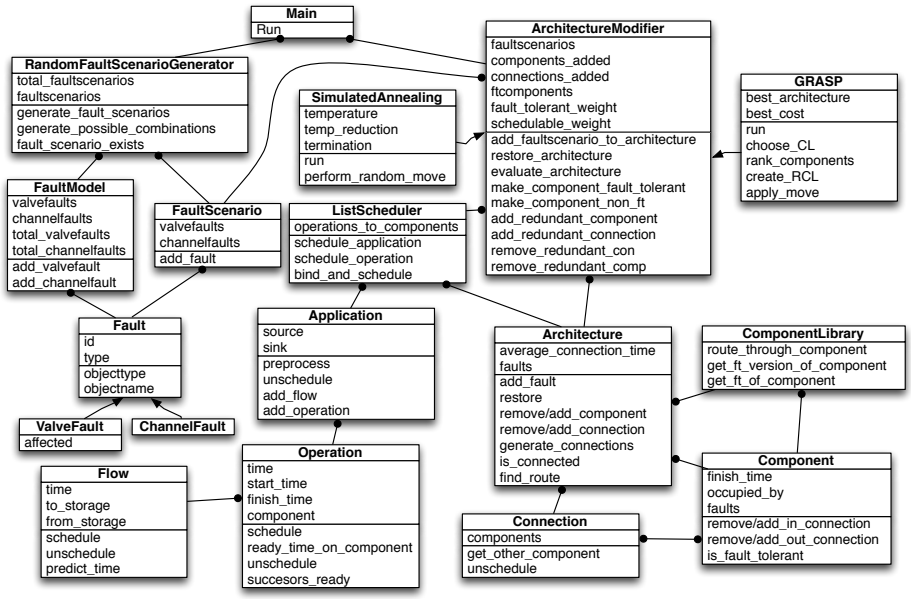



**Figure 6.1:** Class diagram of the implementation

### 6.1.1 Architecture

The architecture module contains the classes for the architecture model, components, connections, component library and routes. The *Architecture* class implements the architecture model and it has the responsibility of adding and removal of components and connections, finding routes for the fluid and generating the effects of faults. When finding routes between components, the architecture considers the effects of valve faults on switches, i.e. it might only be able to use certain connections if the valve is faulty. Similarly, the route search also considers blocked channels. Furthermore, it contains methods to add faults and restore from faults. The architecture uses the *Component*, *Connection* and *ComponentLibrary* classes. The component class specify the type of the component, the in and out connections it has and furthermore it has scheduling related specifications. The connection class specify a simple graph edge and knows the two component it connects and it has details about scheduling. The reasoning behind components and connections having details about the scheduling is that it simplifies the scheduler, and unscheduling is also simple as it is just the resetting of used attributes. The component library has all the possible components and it keeps track of which components have and do not have fault-tolerance, the fault-tolerance of the components and which

components they are the fault-tolerant version of.

### 6.1.2 Fault Model

The fault model module contains classes for the fault model, the faults, fault scenarios and the random generation of fault scenarios. The *FaultModel* class is simple as it contains two sets which distinguish channel and valve faults, the maximum number of valve faults and the maximum number of channel faults. *Fault* is a superclass which has two subclasses *ValveFault* and *ChannelFault*. The fault class specifies the object type the fault affects, either component or connection, and the name of component or connection, where the names of components and connections are specified in the JSON file for the architecture. The *FaultScenario* class contains a set of valve faults and a set of channel faults. The *RandomFaultScenarioGenerator* class implements the fault scenario generation algorithm described in detail in section 5.2.

### 6.1.3 Application

The application module implements the application graph, the operations and the flows (edges) in the application graph. The *Application* class contains a preprocess method for preprocessing the application graph and functionality to unschedule the application such it can be rescheduled and thereby a new schedule in the objective function can be found. The preprocessing consists of adding a source and a sink as operations to the graph. The source is added such that all the input operations depend on the source and the sink is added such that the sink depend on all the output operations. The *Operation* class represent operations in the application graph. The class implements methods to find the ready time on specific components, which considers the routing time and constraints. The *Flow* class model the dependencies (edges) in the application graph. The class implements methods to schedule the flows to storage and from storage if needed when scheduling, and they can easily be unscheduled.

### 6.1.4 Parsing

The parsing module contains all the parsers for the JSON files. The module contains a superclass named *Parser*. In total, the tool receives 5 files and there is a class to parse each type of file, where all are subclasses of Parser. The *ArchitectureParser* is responsible for parsing the netlist from the JSON file and

create an object of the architecture class. Similarly, *ApplicationParser*, *ComponentLibraryParser*, *FaultModelParser*, and *ConfigParser* create the application graph, component library, fault model and the config data, respectively. The config data specifies the average routing latency, application deadline, number of fault scenarios, which algorithm to use and the algorithm specific details.

### 6.1.5 Scheduling

The scheduling module implements a superclass *Scheduler* such that it is easily extendable to implement more schedulers if more schedulers are needed. The *ListScheduler* class is a subclass of Scheduler. The ListScheduler class implements the List Scheduling algorithm which is described in detail in section 5.4.

### 6.1.6 Architecture Modifier

The architecture modifier module implements the design transformations, architecture evaluation, SA and GRASP. It has three classes *ArchitectureModifier*, *SimulatedAnnealing* and *GRASP*. The ArchitectureModifier is a superclass which implements the design transformations, evaluates the architecture using the fault scenarios and keeps track of the added components, connections and fault-tolerant components converted. The moves are described in detail in section 4.2. Converting a component to fault-tolerant is simple as the type of the component just needs to be changed to the fault-tolerant type. Similar is the case, when converting it back to its regular version. When adding a connection, we just need to consider that the components do not get more incoming or outgoing channels than they are able to handle. Recall that a switch can only consist of four valves (channels) which also means that, e.g. a mixer cannot have more than two incoming and outgoing channels. Removing a connection is also simple as we keep track of the connections, we add to architecture and only redundant connections can be removed. Adding a redundant component however adds more complexity. The move is implemented such that the incoming and outgoing connections are always from / to switches. Consider the situation where all switches already have the maximum allowed channels, then two connections between two switches has to be modified to add a switch between them such that we can add the new component. Removing a redundant component therefore has to take this into consideration. It is implemented such that if a component has added switches to the architecture and we want to remove it, then it also removes the switches if, and only if, the switches do not have a connection to other components than the one, we wish to remove, and the switches from the connection, we modified. If the switches have connections to other

components, they are kept in the architecture and added to the list of added components such that it is possible to remove. If it is removed later on, it will add the connection that we modified to the architecture.

Simulated Annealing and GRASP are subclasses of the ArchitectureModifier class as they use the moves and the architecture evaluation of their superclass. SA and GRASP are explained in detail in section 4.3 and section 4.4, respectively. The ArchitectureModifier superclass makes it easy to extend the tool with more algorithms to produce a fault-tolerant netlist that use the same moves and evaluation method.

### 6.1.7 Serializing

The serializing module contains two classes, a superclass *Serializer* and a subclass *NetlistSerializer*. This is to output the JSON file specifying the fault-tolerant netlist produced from this tool. The superclass is created such that it is easy to extend with more serializers, e.g. a serializer for applications.

### 6.1.8 Run

The run module is the main file for running the program. It takes the command line input which is the JSON files specifying the initial netlist, application, fault model and config file.

## 6.2 Testing

The tool has been tested in an incremental manner as the different aspects was implemented. The scheduling was one of the first functionalities that was implemented. It was tested on different architectures with the associated application models using the architectures and applications provided by the work of [8] and [10]. The example in section 4.1 has been used as the key example for all steps in the implementation process. When the logic to generate faults was implemented, this example was used to test that the architecture responded in the correct way when finding routes and determining the connectivity of the architecture. Furthermore, we also tested how the faults affected the scheduling by affecting it both with faults that would make it impossible to schedule the application and contrary with faults where scheduling would still be possible.

When it was determined that the effects of the faults were implemented correct, the moves were implemented. The moves were tested in different stages. First they were tested to see if when applied to an architecture the outcome was appropriate. When that test succeeded, it was then tested by affecting the architecture with faults that would affect connectivity and scheduling. Redundancy was then added to compensate for the faults and the outcome should be that the connectivity test and the scheduling would pass. After successfully implementing the moves, SA was implemented to use these moves and it was tested many times using the key example from section 4.1 to make sure that the moves and their counter-moves were acting correctly when applied many times. GRASP was then implemented and tested similarly to SA.

## 6.3 Summary

This chapter outlines how the tool was implemented using a class diagram and explaining the different modules and classes. The implementation has been done such that it is easy to extend with new algorithms that will use the same objective function and moves as SA and GRASP and implement new scheduling algorithms. Similarly, every aspect of the tool is implemented. The testing was done incrementally where examples have been used to test every aspect of the tool.

Chapter 7

# Experimental Evaluation

The algorithms proposed in chapter 4 are experimentally evaluated by applying them to a number of benchmarks. The evaluation is done in terms of solution quality and performance.

The algorithms are implemented in Python3.4. All the experiments were run on DTU High Performance Clusters.

## 7.1 Benchmarks

In order to evaluate the proposed design methodology we use both synthetic and real-life benchmarks. The following sections will describe the benchmarks used.

### 7.1.1 S-1 Benchmark

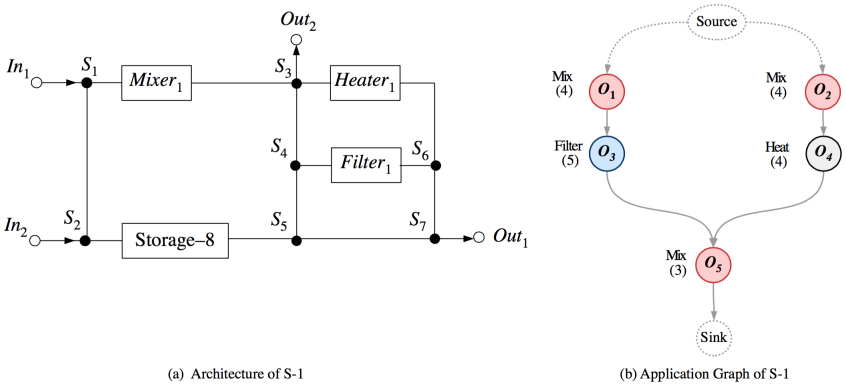



**Figure 7.1:** Architecture and application graph of S-1

S-1 is a synthetic benchmark. The architecture for S-1 is shown in Figure 7.1a and application graph for S-1 is shown in Figure 7.1b (5 operations).
The fault model for S-1 is: $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, 2, 2)$ where $\mathcal{VF}$ and $\mathcal{CF}$ are shown in Table 7.1 and Table 7.2, respectively.

**Table 7.1:** The set of valve faults $\mathcal{VF}$ for S-1

| **Name** | **Vertex** ($N \in \mathcal{N}$) | **Valve affected** ($w$) | **Type** ($t$) |
|---|---|---|---|
| $VF_1$ | $Mixer_1$ | $v_5$ | Open |
| $VF_2$ | $S_6$ | $v_3$ | Open |
| $VF_3$ | $S_5$ | $v_2$ | Open |
| $VF_4$ | $S_3$ | $v_3$ | Open |

**Table 7.2:** The set of channel faults $\mathcal{CF}$ for S-1

| **Name** | **Component** ($M \in \mathcal{N}, \notin \mathcal{S}$) / **Connection** $D_{i,j} \in \mathcal{D}$ | **Type** ($t$) |
|---|---|---|
| $CF_1$ | $Heater_1$ | Block |
| $CF_2$ | $Filter_1$ | Block |
| $CF_3$ | $S_2 \rightarrow$ Storage-8 | Block |
| $CF_4$ | $S_1 \rightarrow Mixer_1$ | Block |

### 7.1.2 PCR Benchmark

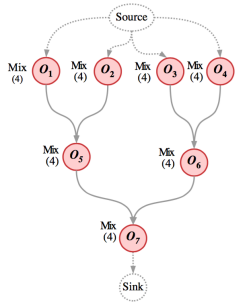



**Figure 7.2:** Application graph of PCR

Polymerase Chain Reaction, henceforth *PCR*, is a real-life benchmark. The application graph for PCR is shown in Figure 7.2 (7 operations). The PCR architecture consists of 2 input nodes, 2 output nodes, 2 mixers, 1 storage component and 7 switches.
The fault model for PCR is: $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, 2, 2)$ where $\mathcal{VF}$ and $\mathcal{CF}$ are shown in Table 7.3 and Table 7.4, respectively.

**Table 7.3:** The set of valve faults $\mathcal{VF}$ for PCR

| **Name** | **Vertex** ($N \in \mathcal{N}$) | **Valve affected** ($w$) | **Type** ($t$) |
|---|---|---|---|
| $VF_1$ | $Mixer_1$ | $v_5$ (pump) | Open |
| $VF_2$ | $Mixer_2$ | $v_6$ (pump) | Open |
| $VF_3$ | $S_1$ | $v_2$ (towards $S_2$) | Open |
| $VF_4$ | $S_3$ | $v_3$ (towards Storage1) | Open |

**Table 7.4:** The set of channel faults $\mathcal{CF}$ for PCR

| **Name** | **Component** ($M \in \mathcal{N}, \notin \mathcal{S}$) / **Connection** $D_{i,j} \in \mathcal{D}$ | **Type** ($t$) |
|---|---|---|
| $CF_1$ | $S_4 \rightarrow S_7$ | Block |
| $CF_2$ | $Mixer_1 \rightarrow S_3$ | Block |
| $CF_3$ | $S_5 \rightarrow Mixer_1$ | Block |

### 7.1.3 IVD Benchmark

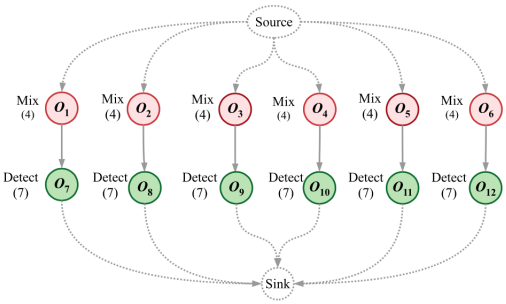



**Figure 7.3:** Application graph of IVD [8]

In-vitro Diagnostics, henceforth *IVD*, is a real-life benchmark. The application graph for IVD is shown in Figure 7.3 (12 operations). The IVD architecture consists of 2 input nodes, 2 output nodes, 2 mixers, 2 detectors, 1 storage component and 43 switches.
The fault model for PCR is: $\mathcal{Z} = (\mathcal{VF}, \mathcal{CF}, 2, 2)$ where $\mathcal{VF}$ and $\mathcal{CF}$ are shown in Table 7.5 and Table 7.6, respectively.

**Table 7.5:** The set of valve faults $\mathcal{VF}$ for IVD

| **Name** | **Vertex** $(N \in \mathcal{N})$ | **Valve affected** $(w)$ | **Type** $(t)$ |
|---|---|---|---|
| $VF_1$ | $Mixer_2$ | $v_5$ (pump) | Open |
| $VF_2$ | $S_{23}$ | $v_3$ (towards $S_{14}$) | Open |
| $VF_3$ | $S_{42}$ | $v_2$ (towards $Detector_2$) | Open |
| $VF_4$ | $S_2$ | $v_3$ (towards Storage1) | Open |
| $VF_5$ | $S_5$ | $v_3$ (towards $S_{15}$) | Open |
| $VF_6$ | $S_{10}$ | $v_2$ (towards $S_3$) | Open |
| $VF_7$ | $S_3$ | $v_1$ (towards $S_{21}$) | Open |

**Table 7.6:** The set of channel faults $\mathcal{CF}$ for IVD

| **Name** | **Component** ($M \in \mathcal{N}, \notin \mathcal{S}$) / **Connection** $D_{i,j} \in \mathcal{D}$ | **Type** ($t$) |
|---|---|---|
| $CF_1$ | $Detector_1$ | Block |
| $CF_2$ | $Detector_2$ | Block |
| $CF_3$ | $S_9 \rightarrow Detector_2$ | Block |
| $CF_4$ | $Mixer_1$ | Block |
| $CF_5$ | $S_{21} \rightarrow S_{28}$ | Block |
| $CF_6$ | $S_{16} \rightarrow S_5$ | Block |
| $CF_7$ | Storage1 | Block |

## 7.2 Solution Quality

The parameters of which to evaluate fault-tolerant architectures depend on many factors. Two of the important factors are that we have to be able to reach every component on the chip (connectivity) and that the application has to be able to complete on the chip, even in the presence of any fault scenario possible under the given fault model. Meanwhile the physical constraints of the biochip also have to be considered as the physical size of the biochip should be kept as small as possible. Otherwise it would be easy to introduce fault-tolerance as we could just introduce many redundant components. Therefore the fault-tolerant architecture produced should be evaluated according to these three parameters, and the parameters are also considered by the objective function outlined in chapter 5. A more practical and equally important feature considering solution quality is the yield of the biochips, i.e. the number of fabricated biochips that, after testing, are considered fault-tolerant hence they can be used by the end-users. Note that in principle, when introducing fault-tolerance to a given fault model, all the manufactured biochips should be fault-tolerant, so the yield should be 100%. However, in practice, it may happen that after testing we detect a fault which is not captured by the fault model. Also, since our solution is based on a metaheuristic that uses a subset of the possible fault scenarios to decide on introducing redundancy, it may happen that the fault-tolerant biochip design is actually not tolerant to all the possible fault scenarios from the fault model.

The experiments evaluating solution quality are divided into two sets of experiments. The first set of experiments have the purpose to evaluate the quality of the obtained solutions and compare SA and GRASP. The second set of experiments will evaluate the yield of the obtained solutions.

### 7.2.1 Objective Function Evaluation

The initial features of the benchmarks are presented in Table 7.7. In Table 7.7 $|\mathcal{N}|$ is the number of components, $|\mathcal{D}|$ is the number of connections, the cost is the number of valves and connections, the objective function calculated cost, $\text{Obj}_{cost}$ of the initial architecture $\mathcal{A}_{initial}$, the number of possible fault scenarios, the number of fault scenarios, $|\mathcal{FS}|$, generated for the specific benchmark, and $d_{\mathcal{G}}$ denotes the deadline for the application. Note that the $\text{Obj}_{cost}$ of the initial architecture is quite high due to the penalty values of the objective function.

**Table 7.7:** The benchmarks and their initial features

| Name | Type | $\mathcal{A}_{initial}$ | | | | Possible $\lvert\mathcal{FS}\rvert$ | $\lvert\mathcal{FS}\rvert$ | $d_{\mathcal{G}}$ |
|---|---|---|---|---|---|---|---|---|
| | | $\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | Cost | $\text{Obj}_{cost}$ | | | |
| S-1 | Synthetic | 15 | 17 | 84 | 22350084 | 121 | 100 | 50 |
| PCR | Real-life | 14 | 16 | 88 | 1695088 | 77 | 50 | 65 |
| IVD | Real-life | 52 | 78 | 274 | 1800274 | 841 | 100 | 90 |

In Table 7.8 the resulting fault-tolerant netlists are presented. The specifications are the same as in Table 7.7 with the addition that FT-$|\mathcal{N}|$ is the number of fault-tolerant components. The netlist obtained by Simulated Annealing is denoted as $\mathcal{A}_{SA}$ and the netlist obtained by GRASP is denoted as $\mathcal{A}_{GRASP}$. The initial netlists and the resulting netlists are shown in Appendix A where they are drawn using GraphViz.

**Table 7.8:** The resulting fault-tolerant netlist of the benchmarks

| Name | $\mathcal{A}_{SA}$ | | | | $\mathcal{A}_{GRASP}$ | | | |
|---|---|---|---|---|---|---|---|---|
| | $\lvert\mathcal{N}\rvert$ | FT-$\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | Cost | $\lvert\mathcal{N}\rvert$ | FT-$\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | Cost |
| S-1 | 24 | 1 | 35 | 185 | 15 | 3 | 20 | 102 |
| PCR | 19 | 1 | 27 | 124 | 14 | 1 | 17 | 92 |
| IVD | 53 | 2 | 80 | 285 | 52 | 2 | 78 | 279 |

These experiments show that both algorithms have been able to find a solution deemed fault-tolerant by using the same cost function presented in chapter 5. This is the case for all benchmarks used. Thereby a fault-tolerant architecture has been synthesised for each benchmark such that the architecture is connected and the application can complete within its deadline considering their respective fault models.

In terms of obtaining a less costly fault-tolerant netlist GRASP performs bet-

ter than SA in all benchmarks. However considering the IVD benchmark the algorithms perform almost equally. GRASP uses less components and connections to obtain a fault-tolerant solution and thereby the cost of the solutions are smaller for GRASP. On average the cost of the solutions found by GRASP are 20% less costly than the solutions found by SA. A possible reason that SA does worse than GRASP is that SA performs a number of bad moves in order to converge towards the global optimum, where the moves are hard to recover from. However, this requires further investigation in order to determine the exact reasons.

### 7.2.2 Yield Evaluation

In this set of experiments only one benchmark is used, which is S-1. Furthermore it will use only one algorithm to evaluate the yield. The algorithm used is GRASP considering it performed better than SA in the first set of experiments. The purpose of these experiments is evaluating the yield and the influence of the number of randomly generated fault scenarios on the obtained solution. The result of the experiments are given in Table 7.9. In Table 7.9 $|\mathcal{N}|$ is the number of components, $|\mathcal{D}|$ is the number of connections, the objective function calculated cost is denoted by $\text{Obj}_{cost}$, the cost is the number of valves and connections, the initial architecture is denoted $\mathcal{A}_{initial}$, $\mathcal{A}_{result}$ is the resulting architecture, $|\mathcal{FS}|$ is the number of fault scenarios generated for the specific benchmark, $|FT|$ denotes the number of fault scenarios, $f \in \mathcal{FS}$, the resulting architecture tolerates, $|\neg FT|$ denotes the number of fault scenarios not tolerated and $FT\%$ is the percentage of fault scenarios tolerated.

**Table 7.9:** The benchmark and the yield evaluation thereof

| $\mathcal{A}_{initial}$ | | | $\lvert\mathcal{FS}\rvert$ | $\mathcal{A}_{result}$ | | | | $\lvert FT\rvert$ | $\lvert\neg FT\rvert$ | $FT\%$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | $\text{Obj}_{cost}$ | | $\lvert\mathcal{N}\rvert$ | FT-$\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | Cost | | | |
| 15 | 17 | 5760084 | 25 | 16 | 2 | 20 | 98 | 105 | 16 | 86.78 |
| 15 | 17 | 10540084 | 50 | 15 | 3 | 19 | 99 | 117 | 4 | 96.69 |
| 15 | 17 | 18580084 | 85 | 16 | 2 | 21 | 101 | 121 | 0 | 100 |
| 15 | 17 | 27610084 | 121 | 15 | 3 | 19 | 99 | 121 | 0 | 100 |

Table 7.9 shows that the number of fault scenarios generated has some influence on the obtained solutions, although it is not a critical factor. When generating only 25 fault scenarios we obtain a solution that tolerates 86.78% of the possible fault scenarios. This is because if a fault-tolerant solution in terms of the objective function, i.e. the application can be scheduled and the architecture is connected in all the generated fault scenarios, is found then it tolerates each

individual fault in the fault scenarios. However the combination of certain faults will make the biochip unusable. This experiment shows that even generating a fraction of the possible fault scenarios can lead to good fault tolerance to all possible fault scenarios. In this case 21% of the possible fault scenarios were generated and the percentage of fault scenarios tolerated is 86.78%. By virtue of this it is possible to get good fault-tolerance results on large architectures where generating a large portion of the fault scenarios to use in the cost function would be computationally infeasible. Furthermore the experiment also shows that it is able to obtain a fault-tolerance of 100% to the possible fault scenarios when generating 70% percent of the fault scenarios. Thus the yield can be increased considerably by using the tool in developed in this thesis.

## 7.3 Performance

In Table 7.10 the running time of the experiments presented earlier are outlined. The number of vertices in initial architecture $\mathcal{A}_{initial}$ are denoted as $|\mathcal{N}|$ and the number of connections is $|\mathcal{D}|$. The number of operations in the application graph $\mathcal{G}$ are denoted as $|\mathcal{O}|$ and the number of generated fault scenarios for the benchmarks are denoted as $|\mathcal{FS}|$. $\mathcal{A}_{SA}$ describes the running time of Simulated Annealing where the time is presented as hh:mm:ss. $\mathcal{A}_{GRASP}$ presents the running time of GRASP with the same time formatting as $\mathcal{A}_{SA}$. Note that test 2, 3, 4 and 5 in the table has no running time for SA as these benchmarks were only evaluated with GRASP in subsection 7.2.2.

**Table 7.10:** The benchmarks and their performance

| # | Name | Type | $\mathcal{A}_{initial}$ $\lvert\mathcal{N}\rvert$ | $\lvert\mathcal{D}\rvert$ | $\lvert\mathcal{O}\rvert$ | $\lvert\mathcal{FS}\rvert$ | $\mathcal{A}_{SA}$ | $\mathcal{A}_{GRASP}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | S-1 | Synthetic | 15 | 17 | 5 | 100 | 06:23:54 | 01:22:36 |
| 2 | S-1 | Synthetic | 15 | 17 | 5 | 25 | - | 00:23:23 |
| 3 | S-1 | Synthetic | 15 | 17 | 5 | 50 | - | 00:45:23 |
| 4 | S-1 | Synthetic | 15 | 17 | 5 | 85 | - | 01:15:45 |
| 5 | S-1 | Synthetic | 15 | 17 | 5 | 121 | - | 01:43:26 |
| 6 | PCR | Real-life | 14 | 16 | 7 | 50 | 16:05:57 | 02:23:52 |
| 7 | IVD | Real-life | 52 | 78 | 12 | 100 | 184:50:28 | 27:40:37 |

Table 7.10 shows that the running time of GRASP is much faster than SA in all benchmarks. Part of the reason is that SA uses the objective function at each iteration whereas GRASP uses it only in the local search and furthermore SA with a slow temperature cooling schedule will have more iterations than the

main loop of GRASP. The objective function is the computationally slow part of the algorithms. However GRASP still does considerably better in terms of running time. Considering the S-1 benchmark GRASP is approximately 4 times faster than SA. GRASP finds a solution to the PCR benchmark approximately 6-7 times faster than SA. Similarly with the IVD benchmark where GRASP finds a solution 7-8 times faster than SA. Therefore it can be concluded that GRASP does substantially better performance-wise in the benchmarks. However there could be implemented optimisations of the objective function that would benefit the running time of both algorithms.

## 7.4 Summary

In this chapter the algorithms proposed for the fault-tolerant architecture synthesis problem have been evaluated by applying them to three different benchmarks. The evaluation was done in terms of solution quality and performance, where GRASP did considerably better than SA in both categories. Furthermore in an effort to increase the yield even generating a fraction of the possible fault scenarios provides good fault-tolerance on all possible fault scenarios.

Chapter 8

# Conclusions and Future Work

This chapter presents the conclusions and possible extensions of the work presented in this thesis.

## 8.1 Conclusions

This thesis describes the problems involved in synthesising a fault-tolerant architecture. We use the biochip architecture model and application model proposed before in [8] and propose extensions to the biochip architecture model for achieving fault-tolerance. We propose a fault model, which consists of a set of valve faults and a set of channel faults where a maximum number of valve faults and a maximum number of channel faults can happen at any given time. Furthermore, the component library for the flow layer model has been extended with fault tolerant components which are the fault-tolerant switch, mixer, heater, filter, separator, detector and storage.

In order to achieve a fault-tolerant architecture, metaheuristics have been implemented. Two metaheuristics have been proposed as solutions to the problem. Simulated Annealing (SA) has been implemented. SA has the important prop-

erty that the found solution converges towards the global optimum. The second metaheuristic is Greedily Randomized Adaptive Search Procedure (GRASP) which constructs randomised semi-greedy solutions as starting points for local search. It then uses local search to find the local optimum.

An important factor is to decide and define what a good solution is. In this thesis, an architecture is considered fault-tolerant if, despite faults, the application that has to run on the architecture can complete within its deadline and that the architecture is connected such that all components on the chip are reachable. The fault scenarios used in the evaluation are randomly generated from the fault model considering the maximum number of valve faults and channel faults. Furthermore, the physical constraints have to be considered as the size of the architecture should be kept small. The physical constraints have been defined as the total number of valves and the total number of channels in the architecture. The total number of valves and channels should therefore be kept as small as possible while still achieving a fault-tolerant architecture. The two metaheuristics must optimise the architecture with respect to these features.

These two algorithmic approaches are evaluated on benchmarks, both synthetic and real-life benchmarks. The evaluations were done in terms of solution quality and performance. The quality of a solution is defined by two things: minimising the cost of the architecture and increasing the yield of biochips. The metaheuristic GRASP produced the best results in terms of solution quality with an average of 20% less costly solutions compared to SA. Both of the algorithms produced fault-tolerant architectures. Furthermore, the evaluation proved that generating even a fraction of the possible fault scenarios in the fault model provided good results in fault-tolerance to all possible fault scenarios. Generating 21% percent of the possible fault scenarios provided fault tolerance to 86% of all the possible fault scenarios. Generating approximately 70% of the possible fault scenarios provided a fault-tolerance of 100%. GRASP also performed better than SA in terms of performance by finding a solution between 4-7 times faster than SA depending on the benchmark.

## 8.2 Future Work

The fault-tolerant architecture synthesis algorithms can be extended and improved in many ways. Either to improve the quality of the fault-tolerance or to improve algorithm performance. Some possible extensions are listed below.

- Considering a general fault model instead of a specific fault model. A general fault model means considering, e.g., that any $k$ channel(s) can

suffer from a fault in the architecture where $k$ is the max number of failing channels. Similarly, extended the model to valves, where $k$ valves could suffer from faults. In the extreme case, the fault model could therefore be that any valve will be stuck open and any channel will be blocked.

- The objective function could be optimised such that when generating a neighbouring solution, only the affected fault scenarios should be evaluated as it adds considerable complexity to go through each fault scenario. For larger architectures, it takes a long time to produce a fault-tolerant architecture as it has to schedule and consider connectivity for each fault scenario.

- An extension could take the same approach as [2]. In this thesis, an application-specific architecture is generated, where the architecture is fault-tolerant to a maximum of $k$ permanent faults. This was done previously for droplet-based biochips as mentioned, hence this thesis introduce fault-tolerant design for the first time for flow-based biochips.

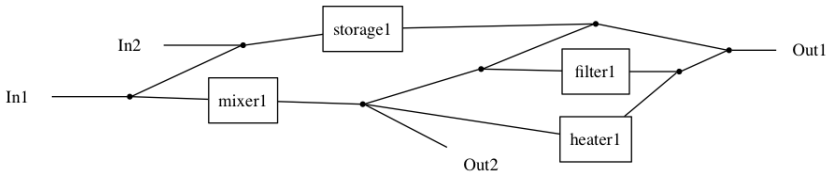
Appendix A

# Netlists

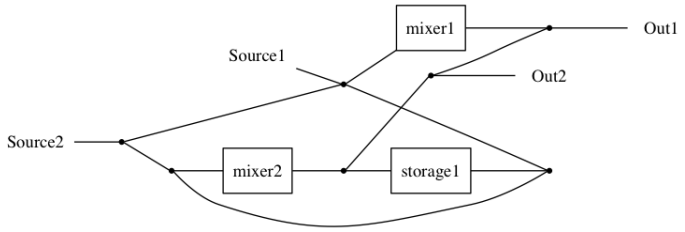
## A.1 Initial Netlists

### A.1.1 S-1

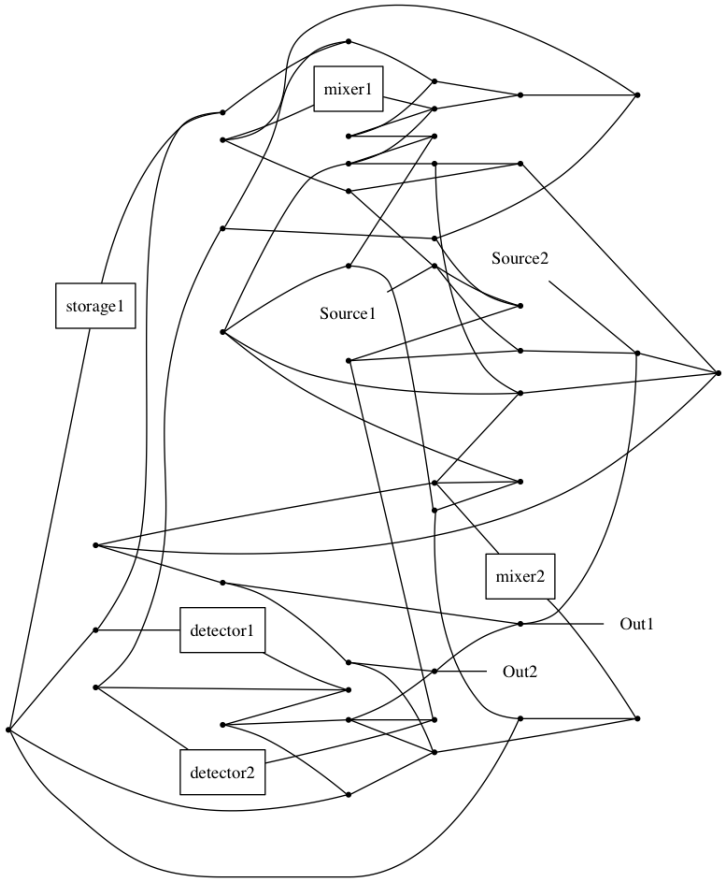
In order to better understand how the netlists look using GraphViz, the S-1 benchmark can be seen here to compare its GraphViz outcome to how it is illustrated in the thesis which is in subsection 7.1.1.



### A.1.2 PCR

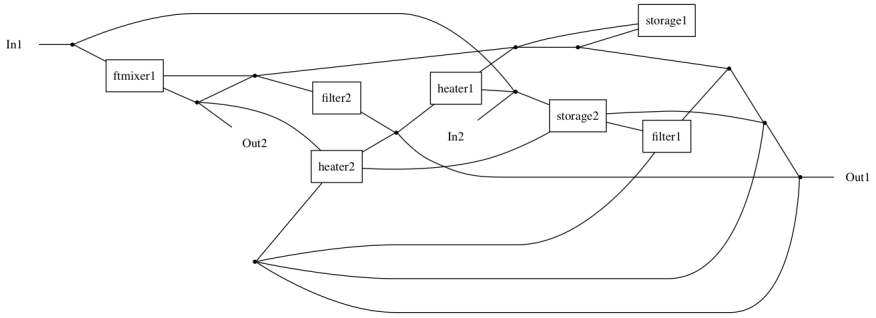



### A.1.3 IVD

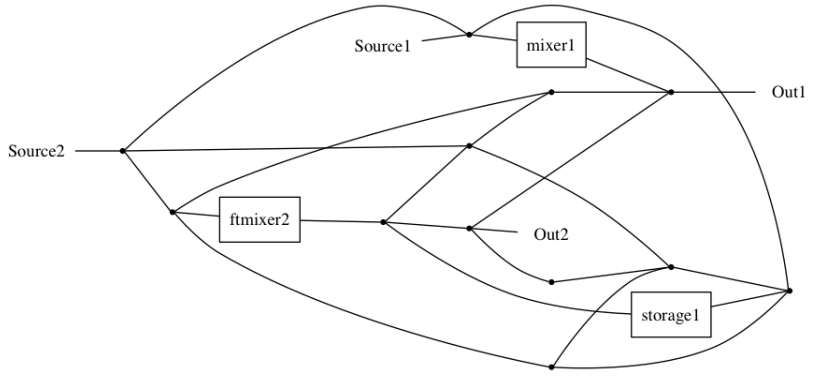


## A.2 Netlists Obtained by SA

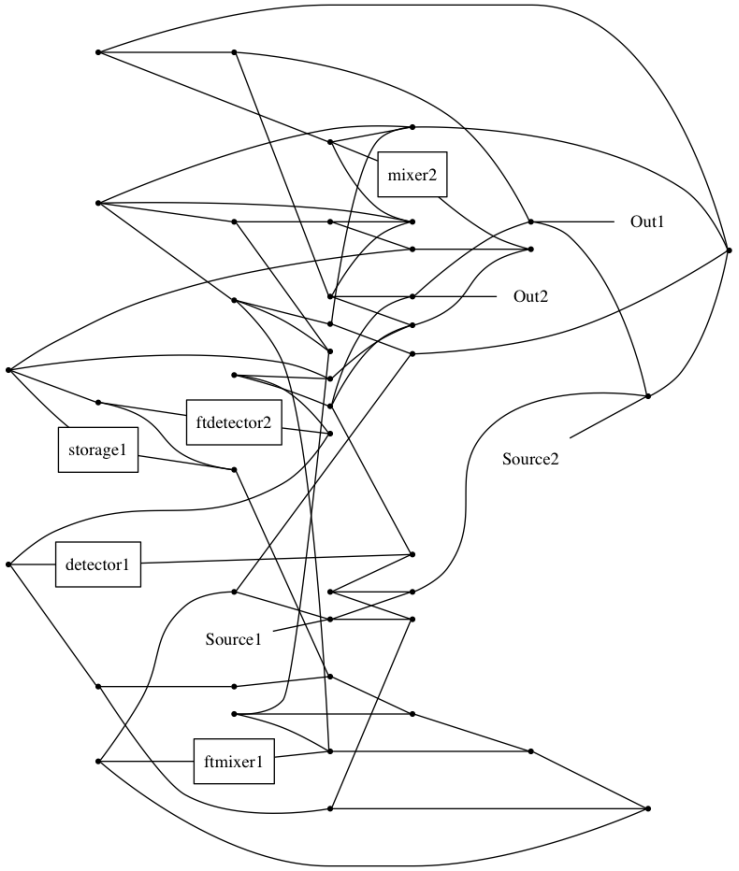
### A.2.1 S-1




### A.2.2 PCR

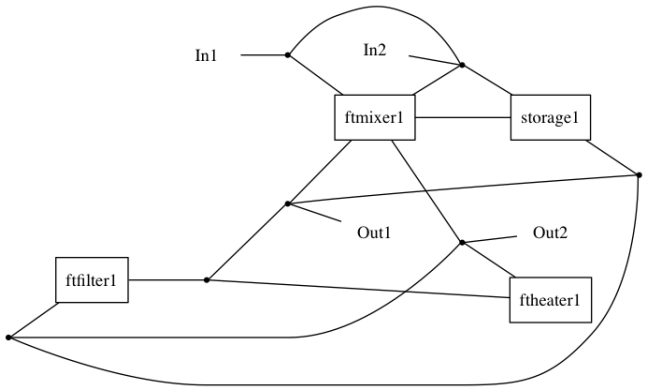


### A.2.3 IVD

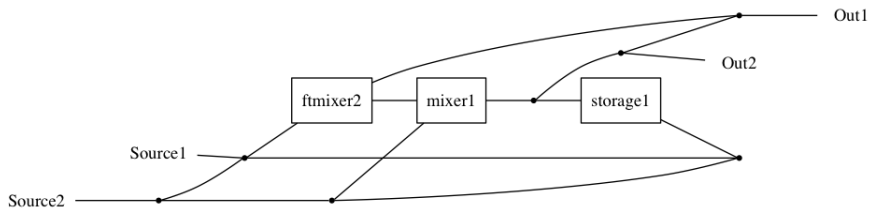

mixer2
Out1
Out2
ftdetector2
storage1
Source2
detector1
Source1
ftmixer1

## A.3 Netlists Obtained by GRASP

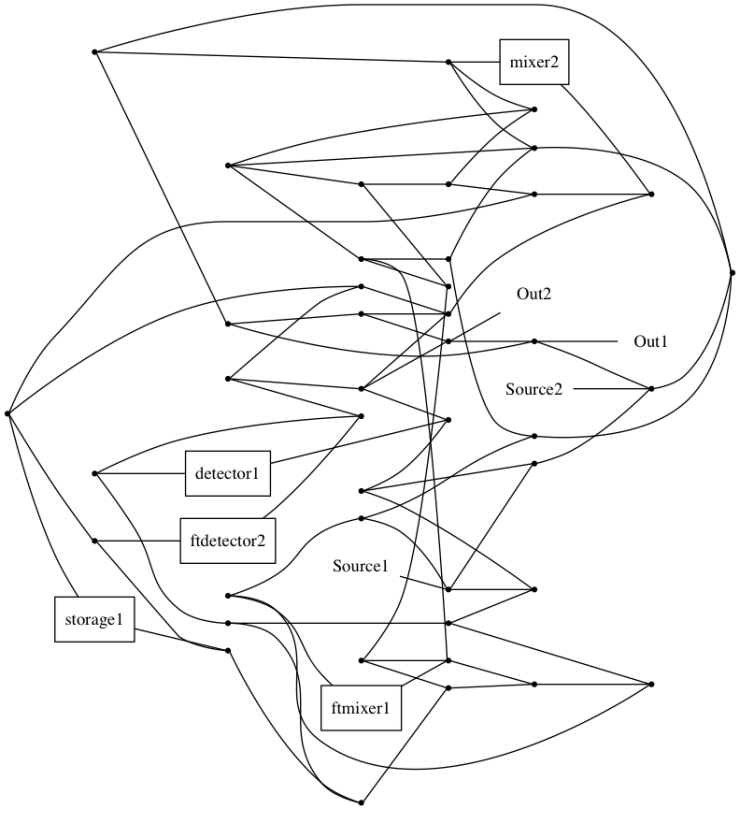
### A.3.1 S-1




### A.3.2 PCR



### A.3.3 IVD


mixer2
Out2
Out1
Source2
detector1
ftdetector2
Source1
storage1
ftmixer1

# Bibliography


[1] Stephen Quake's group at stanford university. `http://thebigone.stanford.edu/`. Accessed: 2015-06-08.

[2] Mirela Alistar. Compilation and synthesis for fault-tolerant digital microfluidic biochips. Technical report, Technical University of Denmark, DTU Compute, 2014.

[3] Holger Becker and Claudia Gartner. Microfluidics and the life sciences. *Science Progress*, 95(95):175–198, 2012.

[4] Kai Hu, Feiqiao Yu, Tsung-Yi Ho, and Krishnendu Chakrabarty. Testing of flow-based microfluidic biochips: Fault modeling, test generation, and experimental demonstration. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 33, No. 10, October 2014*, pages 1463–1475, 2014.

[5] Dobo Imre, Domitian Tamas-Selicean, and Paul Pop. Incremental scheduling of TTEthernet networks to reduce re-certification costs. Technical report, Technical University of Denmark, DTU Compute.

[6] Y.C. Lim, A.Z. Kouzani, and W. Duan. Lab-on-a-chip: a component view. *Microsyst Technol*, 16(16):1995–2015, 2010.

[7] Jessica Melin and Stephen R. Quake. Microfluidic large-scale integration: The evolution of design rules for biological automation. *Annu. Rev. Biophys. Biomol. Struct. 2007*, 36(36):213–231, 2007.

[8] Wajid Hassan Minhass. System-level modeling and synthesis techniques for flow-based microfluidic very large scale integration biochips. Technical report, Technical University of Denmark, DTU Compute, 2012.

[9] Leonidas S. Pitsoulis and Mauricio G.C. Resende. Greedy randomized adaptive search procedures. Technical report, Princeton University.

[10] Michael Raagaard. Placement algorithm for flow-based microfluidic biochips. Technical report, Technical University of Denmark, DTU Compute, 2014.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson, 2010.

[12] T. Thorsen, S. J. Maerki, S.J. Maerki, and S.R. Quake. Microfluidic large-scale integration. *Science*, (298(5593)):580–584, 2002.