Synthesis of Flow-Based Biochip Architectures from High-Level Protocol Languages

Mathias Kaas-Olsen

Kongens Lyngby 2015

Abstract

This thesis presents solutions to two problems of biochip synthesis; namely the application model synthesis problem and the architectural synthesis problem.

The application model synthesis derives an application model from an assay written in Aqua. We use ANTLR4 to generate a parser and then traverse the parse tree to derive the application graph. During the synthesis we also show how to achieve a mixture of fluids using only one-to-one mixer components, as these common.

The architectural synthesis focuses on deriving a netlist for a biochip architecture to efficiently execute a specific assay. The synthesis uses a resource-constrained list-base scheduling algorithm to determine an allocation of components from a component library, as well a preliminary binding and scheduling of the application. The binding and application model is then used to derive interconnections of components in the allocation and this yiels the netlist.

Contents

Abstract **i**

1 Introduction **1**
- 1.1 Flow-Based Microfluidic Biochips 2
- 1.2 Biochip Design Methodology 3
- 1.3 Thesis Objective 4

2 System Model **7**
- 2.1 High-Level Protocol Language 8
  - 2.1.1 Declarations 9
  - 2.1.2 Statements 10
- 2.2 Biochemical Application Model 12
- 2.3 Biochip Architecture Model 14

3 Problem Formulation **17**

4 Application Model Synthesis **19**
- 4.1 High-Level Language Grammar 21
  - 4.1.1 Notation 21
  - 4.1.2 Lexer Rules 22
  - 4.1.3 Parser rules 23
- 4.2 Generating the Application Graph 25
  - 4.2.1 Declarations 26
  - 4.2.2 Expressions and Identifiers 27
  - 4.2.3 Assignments 29
  - 4.2.4 Control Flow Statements 29
  - 4.2.5 Operation Statements 30
- 4.3 Solving the Mixing Problem 32

5 Architectural Synthesis **37**
5.1 Allocation 38
5.2 Schematic Design 41

6 Implementation and Results **43**
6.1 Implementation 43
6.2 Results 45

7 Conclusion and Future Work **47**
7.1 Conclusion 47
7.2 Future Work 48

A Example Parse Tree **51**

B Aqua Assays **53**
B.1 Simple Mix 53
B.2 Complex Mix 53
B.3 Example 54
B.4 Enzyme Test 54
B.5 Glucose Test 56

C Application Graphs **57**
C.1 Simple Mix 57
C.2 Complex Mix 58
C.3 Example 58
C.4 Enzyme Test 59
C.5 Glucose Test 59

D Architecture Netlists **61**
D.1 Simple Mix 61
D.2 Complex Mix 62
D.3 Example 63
D.4 Enzyme Test 64
D.5 Glucose Test 65

Bibliography **67**

Chapter 1

Introduction

Miniaturized Total Chemical Analysis Systems (μTAS) were introduced in 1990 by Manz et al. [MGW90]. These systems carry out chemical analyses automatically, handling fluid sample transportation and preparation, and performing laboratory functions such as mixing, heating, measurement and detection for the analyses. The systems are called miniaturized, or micro, when they handle the fluid samples very close to the place of detection, for example on a small chip. Miniaturized total analysis systems have several advantages compared conventional analysers, due to their small size. Some advantages are requiring smaller sample and reagent volumes, having faster biochemical reaction times, and giving more accurate detection.

Microfluidic biochips, also known as Lab-on-a-Chip, are examples of μTAS. Biochips are made of a biocompatible material using microfabrication techniques, and are typically classified as either droplet-based biochips or flow-based biochips depending on how fluids are manipulated on the chip. Biochips are used to automate performance of many biochemical protocols, also called assays, and have multiple application areas, such as cell analysis, DNA analysis, drug discovery, forensics, and environmental analysis [CMSJ$^+$14].

In this thesis we will focus on flow-based microfluidic biochips and the first steps of how to automate their architectural design.

1.1 Flow-Based Microfluidic Biochips

Flow-based micriofluidic biochips manipulate continuous flow of discrete volumes of fluids. The biochips consist of a substrate in which there are integrated microfluidic channels, micromechanical valves and biochemical components such as heaters, filters and sensors. The valves are the basic building blocks on the chips, and can be used to implement pumps, mixers, multiplexers, and switches to control the flow of fluids in the channels. The valves can be manufactured at very high densities on the chip, approaching 1 million valves per cm^2, giving rise to the technology name *microfluidic very large scale integration* (mVLSI) analogous to the electronic counterpart VLSI [AQ12]. The chips can be manufactured with soft lithography using PDMS as substrate, making them cheap and fast to produce.

We consider biochips as logically having two layers; a flow layer and a control layer. The flow layer is the layer where fluid is routed through channels to perform the assays. Fluids are typically emulsified in oil or another emulsifier to avoid evaporation and contamination from fluid remnants on the channel surfaces. Discrete volumes of fluids are achieved by knowing the metrics of the flow channels and metering out a known length of fluid to achieve a multiple of a predetermined volume, referred to as a unit volume. The flow in the flow layer is controlled by the valves, which are activated in the control layer. The control layer's channels are filled with air and are applied pressure to open and close the valves in the flow layer. When no pressure is applied in the control layer of a valve, the valve is open and fluid can flow through the underlying flow layer channel. When pressure is applied to the valve, a thin membrane is pressed into the flow layer, completely cutting off the flow in the underlying channel. This kind of valve is called a normally open valve. A conceptual illustration of a valve with flow- and control layer (blue and red respectively) is shown in Figure 1.1. The pressure for the control layer is typically supplied and controlled by an off-chip pump and runtime system.

The current practise for designing biochips takes a full-custom, bottum-up approach where an application-specific chip is designed manually using CAD programs to lay out the channels, valves and component in the two layers and to schedule the control signals for the valves and components. This approach is very labour intensive and error-prone, and is not very cost-effective if the implemented assay later needs to be modified or integrated into another biochip. Every change to the assay requires a revisit to the manual design phases of the biochip which is time-consuming, thus reducing the usefulness of biochips.

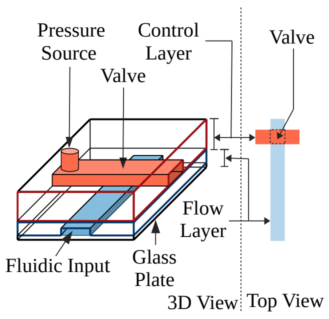


Figure 1.1: Conceptual illustration of a microvalve [Min12].

1.2 Biochip Design Methodology

A top-down design methodology is proposed by Minhass in [Min12] as an alternative to the full-custom, bottom-up approach. Models for the biochemical components, biochip architecture and biochemical application are defined and the design process is subdivided into phases. We will present the models in more detail in chapter 2. An overview of the design methodology is shown in Figure 1.2. The three major design phases are architectural synthesis, application mapping and control synthesis.

The architectural synthesis derives a biochip architecture given a biochemical application and a component library. The component library is a listing of all the available types of biochemical components. The synthesis is divided into two steps:

1. Allocation and Schematic Design: Components are allocated from the component library and the schematic design, also called the netlist, of the biochip is extracted from the application.
2. Physical Synthesis: The physical placement of components and routing of channels for the flow- and control layer is decided.

The application mapping takes a biochemical application and a biochip architecture and determines a binding and scheduling of the operations, which attempts to minimize the application completion time while taking fluid routing and channel contention into account.

The control synthesis takes a biochip architecture and a correspondingly mapped biochemical application and generates the control logic, i.e. timing of

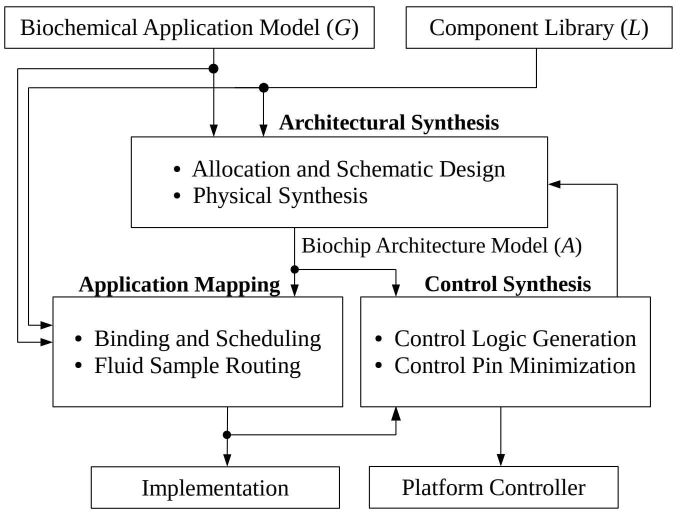


Figure 1.2: Overview of biochip design methodology [Min12].

valve and component activations, required to execute the application on the biochip.

1.3 Thesis Objective

In this thesis we will focus on the representation of biochemical applications in a high-level protocol language, and implement the first step of the architectural synthesis, namely allocation and schematic design. The objective of the thesis is to provide a tool which automatically derives an application-specific schematic design of a flow-based biochip architecture given a biochemical assay written in the high-level protocol language. The workflow of the tool is illustrated in Figure 1.3.

First, we will specify an appropriate high-level protocol language to describe biochemical applications. Next, we will build a parser which extracts a biochemical application model from an assay written in the high-level language. The application model will be represented as a sequencing graph, which is a directed, acyclic, and polar graph. Finally, from the application model we will generate the schematic design for a biochip architecture suitable to perform the application.

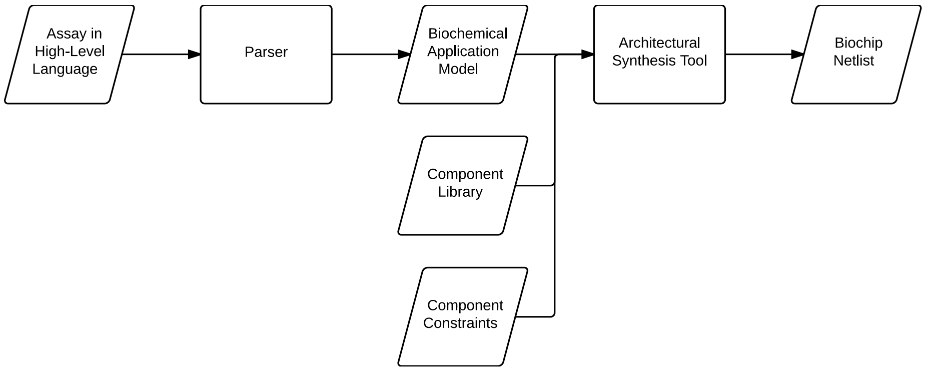

Assay in High-Level Language
Parser
Biochemical Application Model
Architectural Synthesis Tool
Biochip Netlist
Component Library
Component Constraints

Figure 1.3: Overview of the tool workflow.

We will apply our proposed methods and algorithms on several biochemical protocols and show the resulting application models and generated schematics for biochip architectures.

Chapter 2

System Model

In this chapter we will give an informal presentation of the syntax and semantics of our high-level protocol language and define the biochemical application model and biochip architecture model used in the thesis. The application and architecture models are adopted from [Min12].

The purpose of the high-level protocol language is to describe biochemical assays in a precise and unambiguous way to allow automatic extraction of the biochemical application.

The biochemical application model represents the microfluidic operations of an assay and their interdependencies in terms of input and output relations.

The biochip architecture model represents the schematic and physical design of the biochip. The schematic design specifies the components and their interconnections, and the physical design specifies the actual placement and routing of the components and channels.

2.1 High-Level Protocol Language

For a high-level protocol language we will use the Aqua language [Ami15]. We use Aqua since its syntax and semantics are simple and intuitive with the purpose of making it accessible to users who are not necessarily programmers (e.g. biologists or chemists). However, the Aqua language is designed to program assays for multi-purpose biochip architectures developed by Microfluidic Innovations, whereas our goal is to derive the architecture for an application-specific biochip. Thus, some features of the Aqua language are tied to knowing the biochip architecture when writing the assay and are not suited for architectural synthesis. Therefore we will use a slightly modified version of the language where we focus on the parts relevant for architectural synthesis. In section 4.1 we will give a formal definition of the language, including the features that we do not use otherwise. In this section we will give an informal presentation of the modified language, excluding unused features.

We list the language with keywords highlighted in bold, optional parts enclosed in curly brackets, and parts which should be replaced in italic:

```
KEYWORD
{optional}
replace
```

The language is case-sensitive and ignores excess whitespace.

The basic structure of an assay described in the language is:

```
ASSAY name START
  declarations
  statements
END
```

The name of the assay should be indicative of the purpose of the assay and will be used as the name of the extracted biochip architecture. The name must be a valid identifier, but does not have to be distinct from other identifiers.

A valid identifier consist of an alphabetic or underscore character followed by zero or more alphanumeric or underscore characters. Keywords in the language are not valid identifiers.

An assay then consists of one or more declarations followed by one or more statements. We will distinguish between two types of statements: control-flow statements and microfluidic operation statements. We will often refer to the latter type as just an operation.

2.1.1 Declarations

The language has two data types: *integers* and *fluids*.

Integers use the regular decimal system and support the standard arithmetic operators +, -, * and / for addition, subtraction, multiplication and integer division respectively. Arithmetic expressions may be enclosed in parenthesis to specify operator precedence. The result of an integer division will discard the fractional part if there is any, rounding the result towards zero. For example the expression

```
5 / 2
```

will evaluate to the value 2. It is worth noting that due to integer division rounding, the order of multiplication and division operands is significant. For instance, the two expressions

```
5 / 2 * 2
5 * 2 / 2
```

will not evaluate to the same values. The first expression will evaluate to 4 and the second will evaluate to 5.

Fluids represent a specific type of fluid rather than a single sample or a specific volume. The fluid can either be an input which is supplied to the biochip by the user, or the result of an on-chip operation such as a mixing or heating operation. A fluid may be used arbitrarily many times without regard to consumption of the actual sample.

A declaration has one of the following forms:

```
FLUID name {dimensions};
VAR name {dimensions};
INPUT name {volume};
```

The **FLUID** keyword declares a fluid variable. The *name* must be a valid and unique identifier. Fluid variables may be assigned to results of operations as shown later.

The **VAR** keyword declares an integer variable which may later be assigned to literal values or results of arithmetic expressions and may be used in other statements and expressions. Assignment to integer variables is done with the equals operator:

```
a = 2;
```

```
b = 3 * a * a;
```

Both types of variables may optionally be declared with a series of one or more dimensions to declare the variable as an array. Each dimension is specified as the dimension size enclosed in square brackets, e.g.

```
FLUID f[5];
VAR i[2][7];
```

Variable arrays are 1-indexed and values are accessed using square bracket indices, e.g. i[1][1] for the first integer in the i array.

The **INPUT** declaration designates a previously declared fluid variable to be an input. The optional volume part is an integer specifying the amount of available fluid in nL. Input variables cannot be assigned to the results of operations.

A special fluid variable called **it** is automatically declared and will always reference the result of the last executed operation. This variable cannot be assigned manually.

2.1.2 Statements

Our language has two control-flow statements and three microfluidic operation statements.

The two control-flow statements are:

```
REPEAT num_times START statements ENDREPEAT
FOR var FROM var_start TO var_end START statements ENDFOR
```

The language does not support conditional statements and as such is deterministic at compile-time allowing every reachable program state to be computed.

For the **REPEAT** statement, *num_times* must be an expression and *statements* must be one or more statements. The given statements are then executed the specified number of times.

The **FOR** statement sets the value of the integer variable *var* to each value in the range from *var_start* to *var_end*, both inclusive, and executes *statements* for each value in increasing order. The *var_start* and *var_end* parameters must both be expressions.

The syntax for each of the three microfluidic operations is:

```
MIX f1 AND f2 {AND f3...} {IN RATIOS r1 : r2 {: r3...}} FOR duration;
INCUBATE fluid AT temperature FOR duration;
SENSE sense_type fluid INTO var;
```

The **MIX** operation takes two or more fluids and mixes them in the ratios specified for the given duration. If the **IN RATIOS** part is omitted then all fluids are assumed to be in ratios of 1. The number of specified fluids and ratios must always match if the ratios are not omitted entirely.

The **INCUBATE** operation takes a single fluid and heats it up to and maintains the given temperature for the given duration.

The **SENSE** operation moves the given fluid into a sensor and stores the result in the given variable. The *sense_ type* argument specifies the type of sensor to use; it can be either **OPTICAL** or **FLUORESCENCE**. It is unclear from the documentation on Aqua how this is actually achieved. In particular the interface between the sensor and the runtime system is not specified, as well as what the format of the stored value will be, and if and how it may be used in other statements. We will assume that the variable that stores the result will not be used in other statements as the outcome is undefined.

For the **MIX** and **INCUBATE** operations, the resulting fluid type may be stored in a fluid variable with the equals operator:

```
f3 = MIX f1 AND f2 IN RATIOS 1 : 2 FOR 15;
f4 = INCUBATE f3 AT 60 FOR 30;
```

An example assay described in the high-level language is shown in Listing 2.1.

```
ASSAY example START
    FLUID f1;
    FLUID f2;
    FLUID f3;
    FLUID f4;

    VAR v1;
    VAR v2;
    VAR v3[4];

    INPUT f1;
    INPUT f2;
    INPUT f3;

    MIX f1 AND f2 AND f3 IN RATIOS 1 : 3 : 7 FOR 60;
    INCUBATE it AT 75 FOR 15;
    SENSE FLUORESCENCE it INTO v1;

    f4 = MIX f1 AND f2 FOR 30;
    FOR v2 FROM 1 TO 4 START
        INCUBATE f4 AT 15*v2 FOR 30;
        SENSE OPTICAL it INTO v3[v2];
    ENDFOR
END
```

Listing 2.1: A small example of an assay specified in the high-level language.

2.2 Biochemical Application Model

A biochemical application consists of a set of microfluidic operations and a set of dependencies between the operations. An operation O_i depends on another operation O_j if the input to O_i is the output of O_j.

We model a biochemical application as a sequencing graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$. The model captures the operations of an assay as the vertices $\mathcal{O}$ and the dependencies between operations as the edges $\mathcal{E}$. The graph is directed, acyclic, and polar, having a *source* vertex with no ingoing edges and a *sink* vertex with no outgoing edges. If v and u are two vertices in the graph, representing the operations O_i and O_j respectively, then there exists a directed edge from u to v if and only if O_i depends on O_j. Each operation has a specified execution time, or duration, which is modelled as the weight of the corresponding vertex.

The application model assumes that all operations have the correct volume of fluids available from their dependencies and that any excess fluid of an operation is discarded. The volumes for the operations are not specified in the application model, as the actual volumes of operations will depend on the components on which they are executed. For example, one mixer component may take two unit volumes of fluids and mix them, resulting in two units of mixed fluid, whereas another mixer might take four units of fluid and mix them, resulting in four units of mixed fluid.

For the purpose of this thesis we will let the source and sink vertices be implicit in the application graphs and assume that every operation without an ingoing edge is connected to the source vertex and every operation without an outgoing edge is connected to the sink.

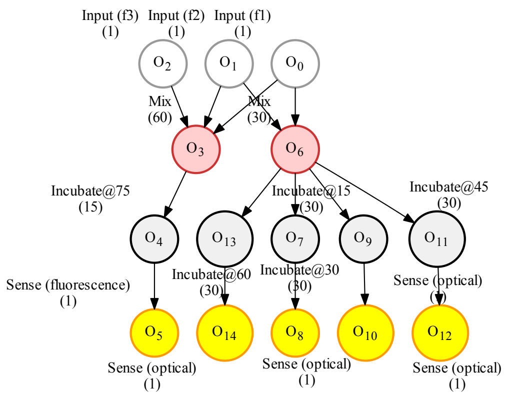


Figure 2.1: Application graph for the example assay in Listing 2.1.

Each operation has a specified type, which defines what components it can be executed on. The types used are

- *input*;
- *mix*;
- *heat*; and
- *detect*.

The last three correspond to the operations **MIX**, **INCUBATE** and **SENSE** available in the high-level language, respectively. The *input* operation type indicates a pseudo-operation which is only used to reflect which fluids are used in the following operations. In practise, an input operation simply corresponds to moving a fluid sample from any of its input ports into the component where it is needed. Collectively, all the input operations in an application graph can be thought of as the source vertex.

An example application graph is shown in Figure 2.1, corresponding to the example assay in Listing 2.1. The source and sink vertices are not shown, as they are implicit. If they were shown, the source vertex would have outgoing edges to O_0, O_1 and O_2 as they're the operations without any ingoing edges. Likewise, the sink vertex would have incoming edges from $O_8, O_{11}, O_{13}, O_{15}$ and O_{17}. The operation execution times are shown as the number in parenthesis under the operation names.

2.3 Biochip Architecture Model

The biochip architecture model captures the schematic and physical design of biochip components as well as the biochip itself.

The components are defined in a *component library* $\mathcal{L}(\mathcal{M},\mathcal{U})$. Each component is characterized by its name $\mathcal{M}$, dimensions, functions $\mathcal{U}$, valves, and input and output ports. The component's name $\mathcal{M}$ identifies each specific type of component in the library and is unique within the library. A component's functions $\mathcal{U}$ are the operation types that the component can execute, giving a mapping of components to and from operation types. Basic components normally only perform a single type of operation, but more complex components might implement more functions. The dimensions specify the size of the component as the width and height of a bounding rectangle. The valves specify how many valves are used to implement the component. Input and Output ports specify the coordinates of the entry and exit points for the component on the bounding rectangle. Neither dimensions, valves, nor input or output ports are relevant to automating the schematic design of a biochip, but are relevant to the physical design of the flow- and control layers.

The component library used in the thesis is shown in Table 2.1. Two of the components have functions that our high-level language does not support operations for; an extension to the language could implement these. Also, some components do not have a function specified because they are used to control the flow on the biochip rather than performing a microfluidic operation. We will assume that the Mixer component takes two units of fluids as input, thus mixing in one-to-one ratios. For all other components we assume that they take a single unit of fluid as input.

The schematic design of a biochip defines which components are on the biochip and how they are connected. This is represented in a netlist, which we model as a directed graph $\mathcal{N}(\mathcal{C},\mathcal{P})$ where the vertices $\mathcal{C}$ represent components and the directed edges $\mathcal{P}$ represent the connections between the components. A directed edge exists between components if fluids can flow from one component to the other. This implies that there must be a path of flow channels in the physical design that has a pressure source at the beginning and a waste outlet at the end and goes through the two designated components in the correct order. Components are allowed to be connected to themselves, so the netlist graph may have loops.

An example netlist is shown is Figure 2.2. The netlist is tailored to execute the example assay presented earlier in Figure 2.1. The netlist graph does not in any way indicate what the physical layout of the biochip should be.

Type Name	Function	Size	Valves
Input	-	(5, 5)	1
Output	-	(5, 5)	1
Filter	filter	(120, 30)	2
Heater	heat	(40, 15)	2
Mixer	mix	(30, 30)	9
Detector	detect	(20, 20)	2
Separator	separate	(70, 20)	2
Metering	-	(30, 15)	6
Multiplexer	-	(30, 10)	2
Storage	-	(90, 30)	28
Switch	-	(1, 1)	3
SwitchI	-	(1, 1)	2
SwitchT	-	(1, 1)	3
SwicthX	-	(1, 1)	4

Table 2.1: The Component Library $\mathcal{L}$ used for this thesis.

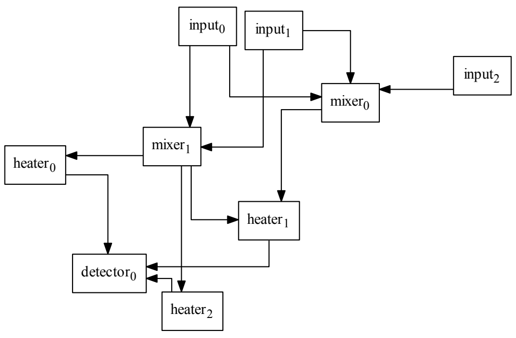


Figure 2.2: An example netlist graph.

Name	**Limit**
Filter	∞
Heater	3
Mixer	3
Detector	1
Separator	∞

Table 2.2: Example component constraints.

During the architectural synthesis, we will decide how many components should be on the biochip as well as how they're connected. To that end, we take a set of resource constraints $\mathcal{R}$ which specify an upper limit on the number of components of each type that should be on the chip. The resource constraints define a mapping from component types to a limit of that component, $\mathcal{R} : \mathcal{M} \to \mathbb{N}$. An example of resource constraints is shown in Table 2.2, which are also the constraints used to generate the example netlist.

CHAPTER 3

Problem Formulation

In this thesis we will present methods to solve the problem of generating a flow-based biochip architecture to run a biochemical application specified in a high-level language. We will divide this problem into two subproblems:

1. *The application model synthesis problem*: Generate the application graph for a biochemical assay written in the high-level protocol language.
2. *The architectural synthesis problem*: Generate the schematic design for a biochip architecture to run the biochemical application.

The application model synthesis problem takes a biochemical assay written in the high-level protocol language and derives an application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ as per the biochemical application model. For example, we derive the application graph shown in Figure 2.1 from the assay written in Listing 2.1. Our solution to this subproblem is presented in chapter 4.

The application model synthesis gives rise to another subproblem which we will also address. It is common in modern biochips that the mixer components take exactly two units of fluids and mix them in one-to-one ratios, and we assume the mixers in our component library to do the same. Since our high-level language supports statements for mixing several fluids in arbitrary ratios, we will address the problem of how to achieve mixing of fluids in arbitrary ratios using only one-to-one mixer components. Under the assumption that all mixing operations are

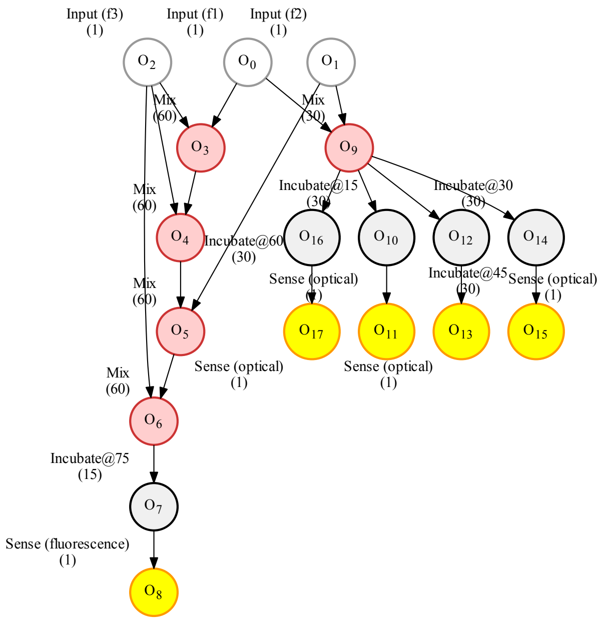


Figure 3.1: An application graph with mixing operations in one-to-one ratios.

in one-to-one ratios, an amended application graph for the assay in Listing 2.1 is shown in Figure 3.1. We will describe the approach to solve the mixing problem in section 4.3.

The architectural synthesis problem takes an application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$, a component library $\mathcal{L}(\mathcal{M}, \mathcal{U})$ and a set of resource constraints $\mathcal{R}$, and derives the schematic design, or netlist $\mathcal{N}(\mathcal{C}, \mathcal{P})$, of a biochip architecture which is suitable to execute the biochemical application. The netlist shown in Figure 2.2 is generated from the application graph shown in Figure 2.1, the component library shown in Table 2.1 and the resource constraints shown in Table 2.2. We will present our solution to the architectural synthesis in chapter 5.

The biochemical assay, component library and resource constraints are the inputs to the system, and the application graph and architecture netlist are the outputs.

Chapter 4

Application Model Synthesis

In this chapter we will describe our solution to the application model synthesis problem, i.e. how we parse an assay described in the high-level language to derive an application graph as per the biochemical application model.

The strategy is to build a parser which will generate a parse tree from an assay written in the high-level language. We will then utilize the deterministic property of the language to statically analyse the parse tree and derive all the microfluidic operations and their dependencies. During the static analysis we will apply a transformation to solve the mixing problem.

Parsers are generally divided into two major categories, depending on how they parse the input. The two types are LL parsers, short for **L**eft-to-right **L**eftmost derivation, and LR parsers, for **L**eft-to-right **R**ightmost derivation. Both types of parsers are further divided into more specific categories and able to parse different subsets of context-free languages. General parsing algorithms also exist which can parse any context-free language. Whereas it is certainly possible to build parsers from scratch, they are usually constructed automatically by a parser generator. Parser generators, also known as compiler-compilers, are tools which build parsers from a formal specification of the language they must parse, usually described as a context-free grammar.

For this project we considered a few different parser generators to build our parser:

- ACCENT [ACC15] generates parsers implementing the Earley parsing algorithm [Ear70] in C. This is a general parsing algorithm and thus ACCENT can generate parsers for any context-free language. The Earley algorithm in general takes $O(n^3)$ time and $O(n^2)$ space where n is the size of the parser's input. However, these are upper-bounds and the algorithm performs significantly better for many grammars.
- PLY (Python Lex-Yacc) [Lex15] is an implementation of the canonical Lex and Yacc tools in Python, generating LALR(1) parsers (1-token Look-Ahead LR). LALR(1) parsers are memory and time efficient, and parse a fairly large subset of context-free languages. The parsing algorithm runs in $O(n)$ time and linear space in the size of the grammar. The parser offers extensive error handling.
- ANTLR 4.5 [ANT15, Par13] generates parsers which use the ALL(*) parsing algorithm [PHF14]. The ALL(*) parsing algorithm can handle any context-free grammar which does not have left-recursion, and ANTLR automatically rewrites direct left-recursive rules. As such, ANTLR can generate parsers for any context-free language which is not inherently left-recursive. The ALL(*) algorithm runs in $O(n^4)$ worst-case time, but performs significantly better on many parser inputs.

To build the parser we decided on using ANTLR due to the fairly general parsing algorithm, prior experience with LL(*) grammars which are very intuitive, and the fact that ANTLR generates parsers in our desired programming language. ANTLR takes a language specification as a context-free grammar in Extended Backus-Naur Form and generates the source code for a lexer and parser for one of a number of supported programming languages, in our case Python. The lexer reads input and splits it up into tokens which it feeds the parser. The parser generates a parse tree from the tokens if the input is valid syntax according to the grammar. It also implements tree-walking functions to traverse the parse tree to implement semantics.

An overview of our solution is shown in Figure 4.1. First we will give the formal definition of the high-level language grammar and have ANTLR build a parser. We then use the generated parser to build a parse tree for a high-level language input and describe how we walk the parse tree to perform the static analysis and extract the application graph. We will finally present the method to generate the application graph such that it can be executed on architectures which use one-to-one mixer components.

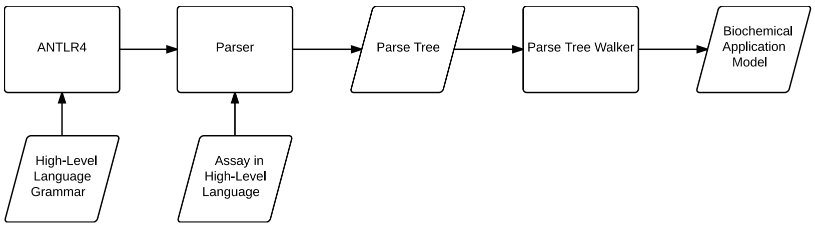

ANTLR4
Parser
Parse Tree
Parse Tree Walker
Biochemical Application Model
High-Level Language Grammar
Assay in High-Level Language

Figure 4.1: Overview of solution to the application model synthesis problem.

4.1 High-Level Language Grammar

In section 2.1 we gave an informal description of our high-level language syntax and semantics. In this section we will give the formal definition of the language syntax expressed as a context-free grammar in Extended Backus-Naur Form, as is the required input format for ANTLR.

A context-free grammar defines a context-free language by a set of terminal symbols Σ, non-terminal symbols V, a starting symbol S, and a set of production rules R. The production rules define a mapping from non-terminals into strings of terminals and non-terminals: $R : V \rightarrow (\Sigma \cup V)^*$, where $\cup$ is the closure of the symbol sets and * is the Kleene star. We write a production rule as a left-hand side which is the non-terminal, and a right-hand side which is the string of terminals and non-terminals. A production rule specifies that its left-hand non-terminal may be replaced by its right-hand string. The language defined by a grammar consists of the set of all strings which can be constructed by recursively applying production rules from the starting symbol until a string of only terminals is reached.

Our high-level language is not a context-free language, since it includes context-sensitive semantics such as the use of variables. However, we can use a context-free grammar to initially verify the syntax of the language, and then later apply logic to verify the semantics such as variable declaration before use and type-checking.

4.1.1 Notation

When listing the grammar, we separate the left- and right-hand side of a production rule with the ::= symbol and enclose every non-terminal in angled brackets and each terminal in single quotes. Each production rule has one or more productions, each alternative separated by a | operator. The | operator is the logical "or" operator and binds as loosely as possible, but may be bound tighter by grouping string parts in parenthesis. When a production string is made from several terminals and non-terminals, we separate each symbol with whitespace. An example of a production rule with two alternatives, of which the second is a string of two symbols, is:

⟨*production-rule*⟩ ::= 'terminal'
  | ⟨*non-terminal*⟩ 'terminal'

The extension to Backus-Naur Form includes additional constructs which are familiar to anyone who uses regular expressions; namely the "one-or-more" operator +, the ("zero-or-more") Kleene star operator *, and the "one-or-zero" operator ?. The operators specify that the string which they operate on may be repeated the specified number of times. These operators bind as tightly as possible, but parenthesis can be used to group the parts of the production string which is the subject of the operators. The operators + and $*$ are greedy, meaning they will consume as much input as can match the subject substring while still allowing the entire input to match the grammar (if a matching exists).

4.1.2 Lexer Rules

The lexer has three explicit token rules along with an implicit rule for each of the literal strings used in the grammar. Each lexer rule is specified as the token name followed by a regular expression which defines the valid content of the token type:

⟨*IDENTIFIER*⟩ ::= [a-zA-Z_][a-zA-Z0-9_]*

⟨*INTEGER*⟩ ::= '-'?[0-9]+

⟨*WS*⟩ ::= [\t\r\n]+

The *IDENTIFIER* rule specifies the valid identifier names and the *INTEGER* rule specifies valid integer literals.

The *WS* rule uses an ANTLR feature to suppress redundant whitespace character by sending them to the parser on a channel which indicates that they're not used. The first character in the WS character class is a literal space. This effectively causes the language to ignore excess whitespace as desired.

4.1.3 Parser rules

The starting symbol of the grammar is the *assay* non-terminal symbol. The *EOF* symbol is a special ANTLR terminal which is the end of the input file.

⟨*assay*⟩ ::= 'ASSAY' ⟨*IDENTIFIER*⟩ 'START' ⟨*decls*⟩ ⟨*stmts*⟩ 'END' ⟨*EOF*⟩

The syntax for declarations is as follows. Coupled with fluid and integer declarations is a syntax definition for conflicts, which is an Aqua specific feature that we do not implement. Information on the feature can be found in the Aqua manual [Ami15] and papers.

⟨*decls*⟩ ::= (⟨*decl*⟩ ';')+

⟨*decl*⟩ ::= ⟨*fluid*⟩
  | ⟨*input*⟩
  | ⟨*var*⟩
  | ⟨*conflict*⟩

⟨*fluid*⟩ ::= 'FLUID' ⟨*IDENTIFIER*⟩ ⟨*dimension*⟩* ('WASH' ⟨*IDENTIFIER*⟩)? ('PORT' ⟨*INTEGER*⟩)?

⟨*input*⟩ ::= 'INPUT' ⟨*IDENTIFIER*⟩ ⟨*INTEGER*⟩?

⟨*var*⟩ ::= 'VAR' ⟨*IDENTIFIER*⟩ ⟨*dimension*⟩*

⟨*dimension*⟩ ::= '[' ⟨*INTEGER*⟩ ']'

⟨*conflict*⟩ ::= 'CONFLICT' ⟨*IDENTIFIER*⟩ ('FOLLOWS' ⟨*IDENTIFIER*⟩ | ',' ⟨*IDENTIFIER*⟩) ('WASH' ⟨*IDENTIFIER*⟩)?

The syntax for the statements is as follows. The statements are again logically divided into control-flow statements (the *control_stmt* rule) and operation statements (the *stmt* rule). A statement is allowed to produce the empty string, which means that the empty statements is also a valid statement; this in turn means that excess semicolons can be ignored as they represent a no-operation. A consequence of this is that a valid assay can contain declarations and only the empty statement one or more times, which makes little sense since it's not really an assay at all, but it would run on any architecture making it a trivial problem.

⟨*stmts*⟩ ::= (⟨*stmt*⟩ ';' | ⟨*control_ stmt*⟩)+

⟨*control_ stmt*⟩ ::= ⟨*repeat*⟩
  | ⟨*for_ loop*⟩

⟨*repeat*⟩ ::= 'REPEAT' ⟨*expr*⟩ 'START' ⟨*stmts*⟩ 'ENDREPEAT'

⟨*for_ loop*⟩ ::= 'FOR' ⟨*IDENTIFIER*⟩ 'FROM' ⟨*expr*⟩ 'TO' ⟨*expr*⟩ 'START' ⟨*stmts*⟩ 'ENDFOR'

⟨*stmt*⟩ ::= ⟨*assign*⟩
  | ⟨*mix*⟩
  | ⟨*incubate*⟩
  | ⟨*sense*⟩
  | /* empty statement */

The assign statement handles assignment to both variable types, each is their own alternative production of the rule.

⟨*assign*⟩ ::= ⟨*identifier*⟩ '=' (⟨*mix*⟩ | ⟨*incubate*⟩)
  | ⟨*identifier*⟩ '=' ⟨*expr*⟩

The syntax for each microfluidic operation statement is:

⟨*mix*⟩ ::= 'MIX' ⟨*identifier*⟩ ('AND' ⟨*identifier*⟩)+ ('IN RATIOS' ⟨*expr*⟩ (':' ⟨*expr*⟩)+)? 'FOR' ⟨*expr*⟩

⟨*incubate*⟩ ::= 'INCUBATE' ⟨*identifier*⟩ 'AT' ⟨*expr*⟩ 'FOR' ⟨*expr*⟩

⟨*sense*⟩ ::= 'SENSE' ⟨*sense_ type*⟩ ⟨*identifier*⟩ 'INTO' ⟨*identifier*⟩

⟨*sense_ type*⟩ ::= 'FLUORESCENCE'
  | 'OPTICAL'

The order in which the expression alternatives are given is intentional and cannot be changed without ruining the mathematical operator precedence. When ambiguities in parsing arise, ANTLR chooses the alternative which is listed first; this gives precedence to multiplication and division over addition and subtraction.

$\langle expr \rangle$ ::= $\langle expr \rangle$ ('*' | '/') $\langle expr \rangle$
  | $\langle expr \rangle$ ('+' | '-') $\langle expr \rangle$
  | '(' $\langle expr \rangle$ ')'
  | $\langle identifier \rangle$
  | $\langle INTEGER \rangle$

$\langle identifier \rangle$ ::= $\langle IDENTIFIER \rangle$ $\langle index \rangle$*

$\langle index \rangle$ ::= '[' $\langle expr \rangle$ ']'

4.2 Generating the Application Graph

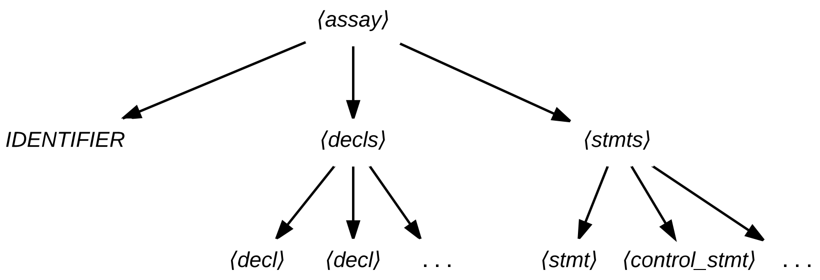
Using the above grammar, ANTLR generates a lexer and parser that can parse our high-level language. The parser in turn generates a parse tree which represents the structure of the parsed assay. The parse tree has an internal node for each non-terminal symbol derived and a leaf node for each terminal symbol. The parse tree is equivalent to the assay code; if every leaf node is printed by an in-order traversal of the parse tree, then the original assay code is printed except discarded whitespace. An example parse tree for the high-level language code example in Listing 2.1 is shown in Appendix A (due to its large size, it will not fit on A4-paper).

To generate the application model from the parse tree, we start with an empty application graph $\mathcal{G}$. We then recursively walk through the parse tree by visiting nodes, and for each visit performing appropriate actions such as modifying the assay variables, adding operations and dependencies to $\mathcal{G}$, and visiting other nodes. The actions taken are determined by the node we're visiting and the context it is in, i.e. ancestors and descendants. The first node we visit is the root node of the parse tree.

The root node of the parse tree is always an $\langle assay \rangle$ node, as this is the starting symbol of the language grammar. When we visit this node we initialize the empty application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ and set up two auxiliary functions V_i and V_f to maintain the integer and fluid variables in the assay respectively. V_i and V_f are functions which map variable names and indices into variable values;

$$V_i : var\_name \times index \rightarrow \mathbb{Z}$$
$$V_f : var\_name \times index \rightarrow operation$$

where *var_name* is a string, and *index* is an $\mathbb{N}^d$ vector of dimension $d \geq 1$. The functions are initially undefined for every variable name and index except the *it* variable with the (1) index, reflecting that no variables are explicitly declared. We then proceed to visit, from left-to-right, each descendant declaration node $\langle decl \rangle$, followed by each statement node $\langle stmt \rangle$ or $\langle control\_stmt \rangle$. The $\langle assay \rangle$ node and its immediate descendants in the parse tree are shown in the following figure. The $IDENTIFIER$ text value is stored as the assay name for outputting later.

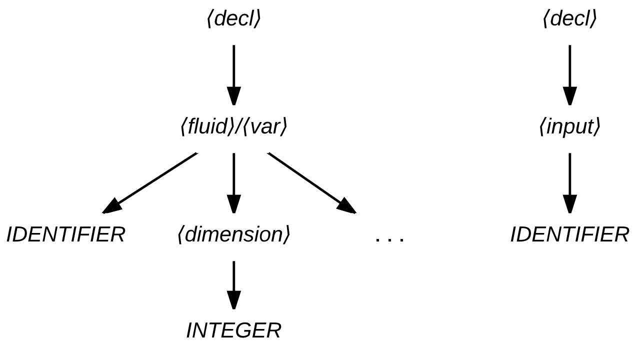


After every declaration and statement has been visited, the application graph will have been constructed and $\mathcal{G}$ contains the application model corresponding to the assay.

Since the entire parse tree of an assay can have many different structures, we will show the different possible sub-trees of the parse tree separately and explain the actions we take when we visit each sub-tree. We show internal nodes in sub-trees as the non-terminal symbol name in angled brackets, and leaf nodes as the terminal symbol name without angled brackets. We will omit nodes for any literal symbols in the production rules which do not contribute to the semantic actions taken for a particular sub-tree.

4.2.1 Declarations

During the visit of the $\langle assay \rangle$ node, all the descendant declarations are visited in order. The sub-tree structure for declarations is shown in the following figure, where there can be zero or more branches of dimensions. We will refer to the value of the $IDENTIFIER$ token as var_name as it represents the variable name to be declared.



If the visited declaration is a fluid or integer declaration, then the corresponding integer or fluid function is defined for the given variable name and dimensions. The function is initially defined to map to a special uninitialized value. Attempting to use the variable value before assignment is an error, and this special value is used to recognize uninitialized variables. The parse tree structure for both integer and fluid variable declarations is identical, and the only semantic difference is in which variable function values will be defined. If the dimensions are omitted, then the (1) vector is assumed for the dimension. Formally, if var_name is declared as a fluid with dimensions $(n_1, n_2, ..., n_d)$, then

$$V_f(var\_name, (i_1, i_2, ..., i_d)) = uninitialized \mid \forall j \in [1, d] : 1 \leq i_j \leq n_j$$

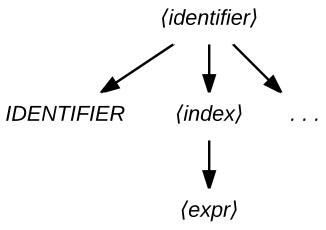
is defined, and every other point with var_name is undefined. The same applies for V_i if the declaration is for an integer variable.

If the visited declaration is an input declaration, then it is checked if the given variable name is defined in V_f with a dimension of (1). If the variable name is not defined, or if it has other dimensions, then it is illegal to declare the variable an input and the assay is invalid. If the declaration is valid, then a vertex O_i with weight 1 and type *input* is added to $\mathcal{G}$, the variable name is marked as an input, and $V_f(var\_name, (1)) = O_i$ is defined. When the variable name is marked as an input, it is no longer legal to modify the value of the variable and it will reference the input operation for the entire assay.

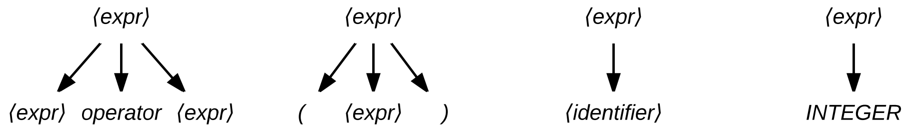
4.2.2 Expressions and Identifiers

Visiting expression and identifier nodes has no side-effects on the application graph $\mathcal{G}$, or variable functions V_i and V_f. Instead, visiting an expression node returns the integer value which is represented by the descending sub-tree, and visiting an identifier node returns the identifier name and index represented by the descending sub-tree.

The identifier tree is shown in the following figure. There can be zero or more branches of $\langle index\rangle$ nodes. If no index nodes are present, then an index of (1) is assumed. The $\langle identifer\rangle$ node returns a tuple containing the variable name and index vector. The variable name is determined by the text value of the $IDENTIFIER$ token, and the index vector is the vector of values attained by visiting all the descendant expression nodes in left-to-right order.



Expression trees take one of four different forms and may be recursively nested:



The simplest expression type is the fourth expression tree, representing a constant expression. For this sub-tree, the integer value is directly returned.

The remaining three expression trees are recursively defined, and relies on the base case of constant expressions to evaluate.

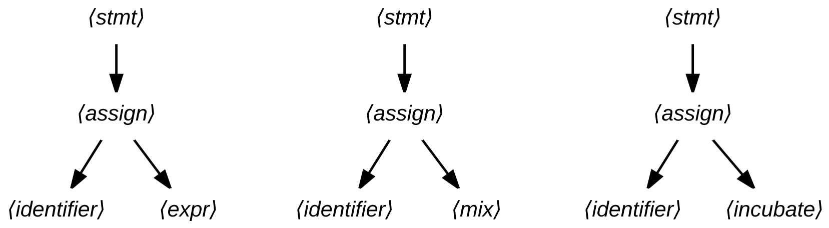
The first expression tree represents a binary arithmetic operator applied to two integers represented as expressions. The operator is either addition, subtraction, multiplication, or integer division. These trees first visit their two child expressions and then return the value of the arithmetic operator applied to the two results.

The second expression tree is used to group other expressions by forcing specific parser interpretations due to the parenthesis. This expression tree directly returns the value provided by visiting the child expression.

The third expression tree represents variable expressions. These expressions return the value of the integer variable currently defined by $V_i(var\_name, index)$ for the var_name and $index$ obtained by visiting the child identifier node. If V_i does not define a value for the variable name and index returned by the identifier node, or if the returned value is the uninitialized value, then the assay is invalid.

4.2.3 Assignments

Integer variables can be assigned to results of expressions, and fluid variables can be assigned the result of mix or incubate operations. When visiting an $\langle assign \rangle$ node, we will redefine the value of V_i or V_f for a specific variable name and index. The three sub-trees for these assignments are shown in the figure below.



The first sub-tree is for integer variable assignments. First the $\langle identifier \rangle$ node is visited to determine the variable name var_name and $index$. If $V_i(var\_name, index)$ is undefined, then the variable has not been declared and the assay is invalid. Otherwise, we visit the $\langle expr \rangle$ node and redefine $V_i(var\_name, index)$ to be the expression's returned integer value.

The two next sub-trees are for fluid variable assignments. The var_name and $index$ is determined by visiting the $\langle identifier \rangle$ node and if $V_f(var\_name, index)$ is undefined or marked as an input then the assay is again invalid. Otherwise we visit the $\langle mix \rangle$ or $\langle incubate \rangle$ node, which will return an operation O_i. We then redefine $V_f(var\_name, index) = O_i$.

4.2.4 Control Flow Statements

The control flow statements come in two variants, but they are handled in much the same way. The statements implement looping semantics for their child statements. The structure of the sub-trees are shown in the figure below.

$$\begin{tikzcd}
 & \langle\textit{control\_stmt}\rangle \arrow[d] & & \\
 & \langle\textit{repeat}\rangle \arrow[dl] \arrow[dr] & & \\
\langle\textit{expr}\rangle & & \langle\textit{stmts}\rangle \arrow[dl] \arrow[d] \arrow[dr] & \\
 & \langle\textit{stmt}\rangle & \langle\textit{control\_stmt}\rangle & \dots
\end{tikzcd}$$

$$\begin{tikzcd}
 & \langle\textit{control\_stmt}\rangle \arrow[d] & & & \\
 & \langle\textit{for\_loop}\rangle \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & & \\
\textit{IDENTIFIER} & \langle\textit{expr}\rangle & \langle\textit{expr}\rangle & \langle\textit{stmts}\rangle \arrow[dl] \arrow[d] \arrow[dr] & \\
 & & \langle\textit{stmt}\rangle & \langle\textit{control\_stmt}\rangle & \dots
\end{tikzcd}$$

Common to both is that they have a $\langle stmts\rangle$ node with a sequence of $\langle stmt\rangle$ and $\langle control\_stmt\rangle$ child nodes that they will repeatedly visit.

For the $\langle repeat\rangle$ node we will first visit the $\langle expr\rangle$ node to determine how many times the child statements must be repeated, and then we will visit all child statement the given number of times.

For the $\langle for\_loop\rangle$ node, we will first visit the two expressions to determine the $from$ and to values of the iteration variable var_name given by the $IDENTIFIER$ token value. For each integer value $x \in [from, to]$, we define $V_i(var\_name, (1)) = x$ and visit all the child statements. It is assumed that the variable name given by the $IDENTIFIER$ token has been declared with (1) as the dimension. If this is not the case the assay to be invalid.

4.2.5 Operation Statements

For the operation statement nodes we will be adding vertices and edges to the application graph $\mathcal{G}$. Each operation node will initially add a single vertex to the application graph, and one or more edges. To determine the dependencies of an operation, and thus which edges need to be added, we will use V_f to determine the operations that each involved fluid variable were generated by.

The structure of the sub-trees for the three operation statements is shown below.

$$\begin{tikzcd}
 & & \langle stmt \rangle \arrow[d] & & \\
 & & \langle mix \rangle \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\langle identifier \rangle & \dots & \langle expr \rangle & \dots & \langle expr \rangle
\end{tikzcd}$$

$$\begin{tikzcd}
 & \langle stmt \rangle \arrow[d] & \\
 & \langle incubate \rangle \arrow[dl] \arrow[d] \arrow[dr] & \\
\langle identifier \rangle & \langle expr \rangle & \langle expr \rangle
\end{tikzcd}$$

$$\begin{tikzcd}
 & \langle stmt \rangle \arrow[d] & \\
 & \langle sense \rangle \arrow[dl] \arrow[d] \arrow[dr] & \\
\langle sense\_type \rangle & \langle identifier \rangle & \langle identifer \rangle
\end{tikzcd}$$

The first sub-tree is for the mix operation. There are two or more $\langle identifier \rangle$ child nodes which represent the fluid variables to be mixed. Then there are an equal number of $\langle expr \rangle$ nodes which define the ratios of the mixing, and finally a last $\langle expr \rangle$ node which is the duration of the mixing operation. If there are not an equal number of $\langle identifier \rangle$ and $\langle expr \rangle$ nodes plus the final $\langle expr \rangle$ node, then ratios have not been specified for all fluids and the assay is invalid. The only exception to this is if there is exactly one $\langle expr \rangle$ node, in which case the ratios have been omitted entirely and we assume the fluids are mixed in equal ratios.

When visiting a $\langle mix \rangle$ node, we first determine the fluids that are used in the operation by visiting the $\langle identifier \rangle$ child nodes and for each var_name and $index$ returned add the $V_f(var\_name, index)$ operation to a list F. If any $V_f(var\_name, index)$ is undefined or set to the uninitialized value, then the assay is invalid. We then visit the following $\langle expr \rangle$ nodes to get the mixing ratios and add these to a list R, and visit the final $\langle expr \rangle$ node to determine the mixing duration c. We finally add a new vertex O_i to $\mathcal{G}$ with weight c and for each operation $O_j \in F$ add a directed edge (O_j, O_i) to $\mathcal{G}$. We set $V_f(it, (1)) = O_i$ to refer to the result of the last operation, which was the added mix operation.

The next sub-tree is for the incubate operation. An $\langle incubate \rangle$ node has three relevant children; first is the $\langle identifier \rangle$ node which specifies the fluid variable $(var\_name, index)$ to incubate, next are two $\langle expr \rangle$ nodes which are the temperature t to incubate at and the duration c of incubation respectively. After visiting the three child nodes, a new vertex O_i with weight c and an edge (O_j, O_i) is added to $\mathcal{G}$ where $O_j = V_f(var\_name, index)$. We finally set $V_f(it, (1)) = O_i$.

The last sub-tree is for the sense operation. First the sense type is determined by inspecting the text value of the $\langle sense\_type \rangle$ node, then the fluid variable $(var\_name, index)$ is determined by visiting the first $\langle identifier \rangle$ node. A vertex O_i for the sense operation is added to $\mathcal{G}$ and an edge (O_j, O_i) where $O_j = V_f(var\_name, index)$ is added as well. Due to the undefined semantics of the operation, we do not modify the value of the integer variable to store the result, which could also only occur at runtime.

The biochemical application model currently does not capture any details of operations other than the execution time and involved fluids. Therefore the temperature of the incubation and the type of sensor ends up being discarded, which is obviously unintended. This warrants a revision of the biochemical application model to account for different kinds of operations which require capturing different details, but we consider a revision of the system model to be out of scope of this thesis and will leave it for future work.

4.3 Solving the Mixing Problem

When we visited the $\langle mix \rangle$ node we added a single vertex to the application graph and several edges to it. We do this under the assumption that a mixer component is available which can carry out the operation directly. However, as stated in the problem formulation, it is not common to have a component capable of handling arbitrary mixing ratios of any number of fluids, and so we need another way to achieve the mixing operation. The most common mixer component is a one-to-one mixer which takes two units of fluids and mixes them in even ratios. In this section we will describe an efficient mixing algorithm which determines how to cascade one-to-one mixing operations to achieve a mixture of several fluids in specified ratios. We will use this algorithm to add vertices and edges to the application graph necessary to allow architectures to execute the application with one-to-one mixer components.

First we will note that not every set of ratios for a mixture can be achieved when using one-to-one mixers. In particular, if the constituent fluids of a mixture are in ratios $(r_1, r_2, ..., r_n) \in \mathbb{N}^n$, then the mixture can be achieved using one-to-one mixers if and only if the total sum of the ratios is a power of two, i.e. $\exists x : 2^x = \sum_{i=1..n} r_i$. If the ratios do not sum up to a power of two, then no number of one-to-one mixes can achieve a mixture with the desired ratios. If a mixture is unreachable, then the ratios can be approximated by increasing or decreasing the relative ratios of the constituent fluids until the ratios sum to a power of two. This introduces an error in the ratios of the final mixture, but the error can be made arbitrarily small at the cost of using more intermediate mixing operations.

Our approach to approximate a set of reachable ratios $(a_1, a_2, ..., a_n)$ from a set of unreachable ratios $(r_1, r_2, ..., r_n)$ is to first determine a target power of two X which we want the approximated ratios to sum up to. The larger a power of two we choose, the smaller the approximation errors will be, but the more mixes are required. If the total sum of the desired ratios is $R = \sum_{i=1..n} r_i$, then a simple target choice X is the smallest power of two greater than R, i.e. $X = 2^{\lceil \log_2 R \rceil}$. Upon choosing an X, we multiply each ratio by $\frac{X}{R}$, which will cause the ratios to no longer be integers but now sum up to X which is a power of two. To bring the ratios back into integers, we round each ratio to the nearest integer. This in turn might cause the ratios to no longer sum up to X, since each rounding introduces an error $-0.5 \leq e_i \leq 0.5$. If the rounded ratios now sum up to A, then we determine by how much we're off the target as $\delta = A - X$. If $\delta = 0$, then the rounded ratios sum to a power of two, namely X, and we use these ratios to approximate the desired mixture. If $\delta > 0$, then we choose the $|\delta|$ ratios with largest positive rounding errors e_i and decrease each of these ratios by one. Otherwise if $\delta < 0$ we choose the $|\delta|$ largest negative rounding errors e_i and increase each of those ratios by one. This brings the sum of approximated ratios back to X and now every ratio is an integer. We use these ratios to approximate the desired mixture ratios. Pseudocode for the ratio approximation algorithm is shown in Algorithm 1. The algorithm takes $O(n^2)$ time since $\delta < n$ and determining i takes $O(n)$ time with a naive implementation.

Algorithm 1 Ratio Approximation Algorithm.

```
Input: Unreachable ratios $(r_1, r_2, ..., r_n)$.
Output: Reachable approximation $(a_1, a_2, ..., a_n)$.
 1: $R := \sum r_i$
 2: $X := 2^{\lceil \log_2 R \rceil}$
 3: $(x_1, x_2, ..., x_n) := (\frac{X}{R} \cdot r_1, \frac{X}{R} \cdot r_2, ..., \frac{X}{R} \cdot r_n)$
 4: $(a_1, a_2, ..., a_n) := (\lfloor x_1 \rceil, \lfloor x_2 \rceil, ..., \lfloor x_n \rceil)$
 5: $(e_1, e_2, ..., e_n) := (a_1 - x_1, a_2 - x_2, ..., a_n - x_n)$
 6: $A := \sum a_i$
 7: $\delta := A - X$
 8: while $\delta > 0$ do
 9:     Choose $i$ such that $e_i = \max\{e_1, e_2, ..., e_n\}$.
10:     Decrement $a_i$, $e_i$ and $\delta$ by 1.
11: end while
12: while $\delta < 0$ do
13:     Choose $i$ such that $e_i = \min\{e_1, e_2, ..., e_n\}$.
14:     Increment $a_i$, $e_i$ and $\delta$ by 1.
15: end while
16: return $(a_1, a_2, ..., a_n)$
```

For example, the ratios $(2,5,6)$ are unreachable as $R = 13$. To approximate a reachable set of ratios, we can choose $X = 16$, and multiply each ratio by $\frac{X}{R} = 1.23$ giving ratios $(2.46, 6.15, 7.38)$. Rounding these to the nearest integers gives $(2,6,7)$ with rounding errors $(-0.46, -0.15, -0.38)$. Now $A = 15$, so $\delta = -1$. We then choose the (one) ratio with the largest negative rounding error, being -0.46, and increase that ratio by one, giving $(3,6,7)$ which are reachable ratios that approximate the desired ratios with a deviation of $(21.9\%, -2.5\%, -5.2\%)$ from the desired relative ratios. The deviation is calculated as the percentile deviation between the desired ratio $\frac{r_i}{R}$ compared to the approximated ratio $\frac{x_i}{X}$. If we increase our choice of X by three (binary) orders of magnitude, $X = 128$, then we get the approximated ratios $(20, 49, 59)$ deviating by $(1.6\%, -0.5\%, -0.1\%)$ from the desired ratios, which is a notably smaller deviation.

The mixing algorithm we use is the Min-Mix Algorithm presented in [TUTA06]. Pseudocode for the algorithm is shown in Algorithm 2. The algorithm takes as input the desired mixture as a set of input fluids $(f_1, f_2, ..., f_n)$ in reachable ratios $(r_1, r_2, ..., r_n)$ and outputs a mixing tree to achieve the desired mixture. For one-to-one mixing, the mixing tree is a binary tree where the leaves are input fluids and internal nodes are mixtures obtained by mixing its two child nodes. The root node is the desired final mixture. When two fluids are mixed it is assumed that half of the resulting mixture fluid is discarded, thus producing only one unit of mixed fluid. Under this assumption the algorithm is optimal with respect to the number of mixes required to reach the desired mixture.

The construction of the mixing tree relies on the observation that each time a fluid is mixed, its contribution to the final mixture is halved. This implies that an input fluid at depth d in the mixing tree constitutes 2^{-d} parts of the final mixture. The algorithm uses this to determine for each f_i at what levels in the mixing tree it must have a leaf node to contribute the ratio r_i. This is decided by inspecting the binary representation of the ratio r_i. For each ordinal position p of a 1-bit in r_i, the fluid f_i gets a leaf node at depth p in the mixing tree. After determining all the leaves of the mixing tree, the algorithm connects the leaves with internal nodes to complete the tree. If the desired ratios are not reachable, the leaves can not be connected into a binary tree.

The Min-Mix algorithm runs in $O(n \log_2 R)$ time, where n is the number of input fluids and R is the sum of the desired ratios. An upper bound on the number of nodes in the mixing tree is necessarily the same. The higher the sum R of the ratios is, the longer the algorithm takes and the larger the mixing trees can become. Our approximation algorithm improves in accuracy by exponentially increasing the sum of ratios, so each step of increase in accuracy results in a linear increase of Min-Mix runtime and mixing tree sizes due to the logarithmic scaling with R. This supports that arbitrary accuracy can be efficiently achieved in terms of execution time and application graph size.

The generated mixing tree is equivalent to a sequencing graph of mix operations that must be carried out from the inputs in order to achieve the desired mixture, so this graph can be directly inserted into $\mathcal{G}$ when visiting a $\langle mix \rangle$ node to allow the application to be executed on architectures with one-to-one mixers.

Algorithm 2 Min-Mix Algorithm.

```
Input: Fluids $(f_1, f_2, ..., f_n)$ and reachable ratios $(r_1, r_2, ..., r_n)$.
Output: A mixing tree for the desired mixture.
function MIN-MIX($(f_1, f_2, ..., f_n), (r_1, r_2, ..., r_n)$)
    $depth := \log_2(\sum r_i)$
    $bins :=$ new $stack[depth + 1]$
    for $i = 1..n$ do
        for $j = 1..depth$ do
            if the $j^{th}$ bit in $r_i$ is 1 then
                Push $f_i$ on $bins[j]$.
            end if
        end for
    end for
    return BUILD-MIXING-TREE($bins, depth$)
end function
function BUILD-MIXING-TREE($bins, depth$)
    if $bins[depth]$ is empty then
        $c_1 :=$ BUILD-MIXING-TREE($bins, depth - 1$)
        $c_2 :=$ BUILD-MIXING-TREE($bins, depth - 1$)
        Add internal node $v$ to mixing tree.
        Add edges $(c_1, v)$ and $(c_2, v)$ to mixing tree.
        return $v$
    else
        Pop $v$ from $bins[depth]$.
        Add leaf node $v$ to mixing tree.
        return $v$
    end if
end function
```

Two mixing trees are shown in Figure 4.2. Both trees are for a mixing of ratio $(2, 5, 6)$ after it has been approximated to $(3, 6, 7)$ (left) and $(20, 49, 59)$ (right). We note that even though the sums of ratios are orders of magnitude apart, the mixing trees are still comparable in size.

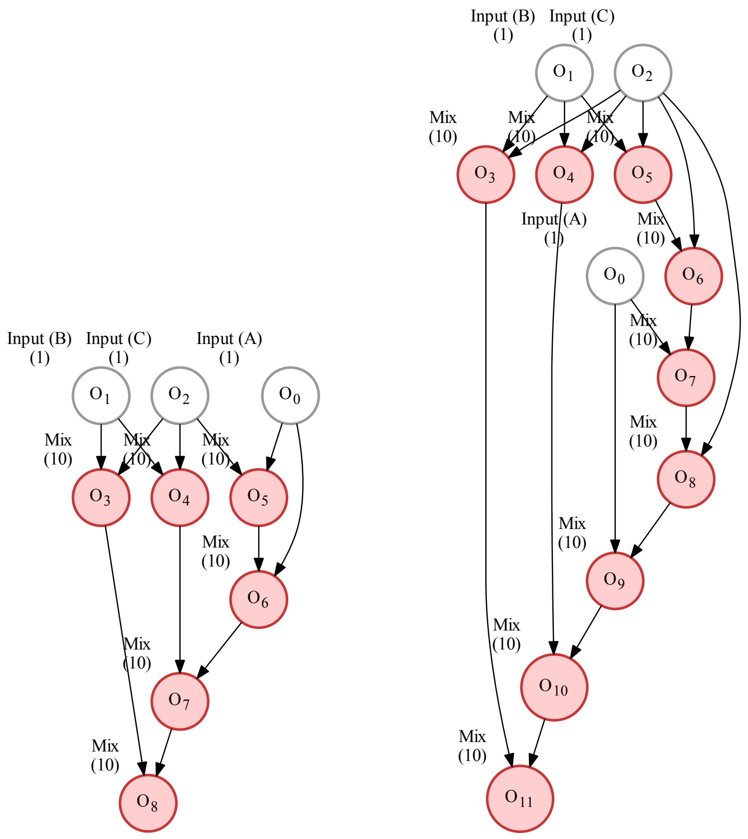


Figure 4.2: Mixing trees for approximations of $(2, 5, 6)$ ratios. The left tree is for approximation $(3, 6, 7)$ and the right tree is for approximation $(20, 49, 59)$.

Chapter 5

Architectural Synthesis

In this chapter we will describe our solution to the architectural synthesis problem. Given an application graph $\mathcal{G}(\mathcal{O},\mathcal{E})$, component library $\mathcal{L}(\mathcal{M},\mathcal{U})$ and resource constraints $\mathcal{R}$ we will derive a netlist $\mathcal{N}(\mathcal{C},\mathcal{P})$ for a biochip architecture with components from the component library while respecting the resource constraints.

Architectural synthesis is a well-known problem in high-level synthesis for electronics and several solutions to the problem exist. Strictly speaking, architectural synthesis spans a larger problem than the one we presented in chapter 3. In addition to deciding the allocation and schematic design for the biochip architecture, the physical placement and routing is also part of the architectural synthesis. Since the solutions to both subproblems are fairly involved, we will focus on a solution to the first subproblem. Thus, when we write 'architectural synthesis', we refer to the allocation and schematic design subproblem. The architectural synthesis problem is known to be NP-hard in general, and therefore no efficient and optimal algorithms are known. Instead, the problem is decomposed into simpler subproblems and heuristic algorithms are used to solve the problem.

Our solution to the problem for biochips is based on the solutions to the problem in the electronics domain provided in [Mic94]. We will focus on optimizing the latency, or execution time, of the application, which will be a trade-off of the chip size. We assume the chip size to be dominated by the allocated components on the chip, and so the resource constraints is the deciding parameter for the chip size. Since the constraints are an input, the design space can be explored by fiddling with the constraints to search for an acceptable solution in both biochip size and application execution time. Since the biochip size is decided by the input, we can narrow our focus to generating an architecture which minimizes the execution time of the application without trying to optimize the chip size. To solve the architectural synthesis problem we will first decide the *allocation*, and then decide the *schematic design*.

5.1 Allocation

The *allocation* decides how many of each component from the component library is to be placed on the biochip while ensuring that the given constraints are not exceeded. We will solve this problem by deciding a preliminary *binding and scheduling* of the application graph. The binding defines for each operation which component it must be executed on and the scheduling defines when the operation must be executed. If two operations are bound to the same component, then their schedules may not overlap. Since we are prior to the actual routing and placement phase of the architectural synthesis, we do not yet know the routing delays between components and a generated binding and schedule will not be valid on the final architecture. We use the binding and scheduling as a heuristic to estimate an efficient allocation and schematic design.

To perform the binding and scheduling while trying to minimize the application execution time, we will use a resource-constrained list-based scheduling algorithm. During the scheduling we will greedily allocate more components so long as operations are ready to be executed and the resource constraints are not exceeded for the corresponding component types.

Pseudocode for our list scheduling algorithm is shown in Algorithm 3. The algorithm takes as input the application graph $\mathcal{G}$ and the resource constraints $\mathcal{R}$, and outputs a scheduled and bound application graph as well as an allocation. First we determine an urgency criteria for each operation O_i in $\mathcal{G}$. We use the weight of the longest path from O_i to the sink node as the urgency criteria. This can easily be calculated by topologically sorting the operations in the application graph and then using a DAG longest path algorithm. Next, we start at time $t = 0$ and for this and every following time step we determine a list of ready operations and bind and schedule as many as possible. When deciding which operations to bind and schedule first, we choose the most urgent operations and check if components are available to execute them. We define a component to be available if the starting time plus duration of the last operation bound to the component is less than or equal to the current time step. If a component is available, the operation is bound to the component and scheduled to start at the current time step. If no component is available for a ready operation, then we will allocate a new component if the allocated number of components then does not exceed the given constraints. If an operation is ready, but no allocated component is available and the maximum number of this component is already allocated, then we will do nothing for the operation until a later time step when more components have become available. The next time step is determined as the next time where a scheduled operation finishes, as this is the next earliest point in time where more components become available. The algorithm is finished when every operation has been bound and scheduled. The allocated components as well the as an application graph with annotated start times and bindings is returned.

Algorithm 3 Resource-Constrained List-Scheduling.

```
Input: An application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ and constraints $\mathcal{R}$.
Output: An allocation $\mathcal{C}$ and binding and schedule of $\mathcal{G}$.
 1: Calculate urgency of each operation as longest path to sink
 2: Set time $t := 0$
 3: repeat
 4:     Determine sorted list $L$ of ready operations by descending urgency
 5:     for all $opr \in L$ do
 6:         if an allocated component $c$ of type $opr$ is available then
 7:             Bind $opr$ to $c$ with starting time $t$
 8:         else if allocated components of type $opr$ less than constraint then
 9:             Allocate a new component $c$ of type $opr$
10:             Bind $opr$ to $c$ with starting time $t$
11:         else
12:             Do nothing for operation
13:         end if
14:     end for
15:     Set $t$ to the time of the next finished operation
16: until all operations are scheduled.
```

Running the list scheduling algorithm on the application graph shown in Figure 3.1 with the constraints in Table 2.2, yields the allocation shown in Table 5.1 and the bound and scheduled application graph shown in Figure 5.1.

Component	Allocated
Input	3
Heater	3
Detector	1
Mixer	2

Table 5.1: Number of components allocated by list-based scheduling algorithm.

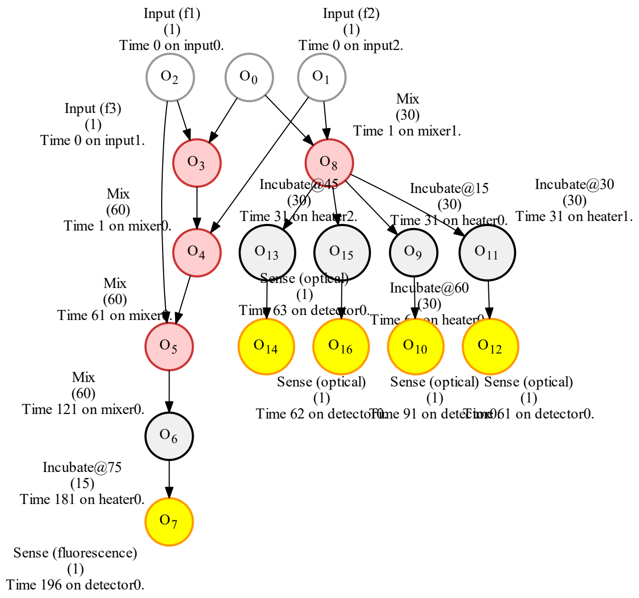

Input (f1) (1) Time 0 on input0.
Input (f2) (1) Time 0 on input2.
Input (f3) (1) Time 0 on input1.
Mix (30) Time 1 on mixer1.
Mix (60) Time 1 on mixer0.
Mix (60) Time 61 on mixer0.
Mix (60) Time 121 on mixer0.
Incubate@45 (30) Time 31 on heater2.
Incubate@15 (30) Time 31 on heater0.
Incubate@30 (30) Time 31 on heater1.
Sense (optical) (1) Time 63 on detector0.
Incubate@60 (30)
Sense (optical) (1) Time 62 on detector0.
Sense (optical) (1) Time 91 on detector0.
Sense (optical) (1) Time 61 on detector0.
Incubate@75 (15) Time 181 on heater0.
Sense (fluorescence) (1) Time 196 on detector0.
o_0
o_1
o_2
o_3
o_4
o_5
o_6
o_7
o_8
o_9
o_{10}
o_{11}
o_{12}
o_{13}
o_{14}
o_{15}
o_{16}

Figure 5.1: A bound and scheduled application graph.

5.2 Schematic Design

After deriving an allocation and preliminary binding, we can now derive the schematic design of the biochip architecture. The allocation already determines the components $\mathcal{C}$ of the netlist $\mathcal{N}(\mathcal{C}, \mathcal{P})$, so we just need to figure out the required interconnections $\mathcal{P}$ of the components. We can derive these from the bound application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$, as an interconnection of two components c_i and c_j is required exactly if two operations O_x and O_y are bound to c_i and c_j respectively and an edge (O_x, O_y) exists in $\mathcal{G}$. If we denote the binding of an operation O_x as $\beta(O_x)$, then we can define the set of component interconnections as

$$\mathcal{P} = \{(c_i, c_j) | (O_x, O_y) \in \mathcal{E} \wedge \beta(O_x) = c_i \wedge \beta(O_y) = c_j\}$$

Pseudocode for a simple algorithm to generate the set $\mathcal{P}$ is shown in Algorithm 4. The algorithm loops over every edge in the bound application graph and then adds an edge to $\mathcal{P}$ for the corresponding interconnection if it is not in $\mathcal{P}$ already.

Algorithm 4 Schematic Design.

Input: A bound application graph $\mathcal{G}(\mathcal{O}, \mathcal{E})$ and set of components $\mathcal{C}$.
Output: A set of components interconnections $\mathcal{P}$.

```
1: $P$ = empty set
2: for all $(O_x, O_y) \in \mathcal{E}$ do
3:     $p = (\beta(O_x), \beta(O_y))$
4:     if $p \notin \mathcal{P}$ then
5:         Add $p$ to $\mathcal{P}$
6:     end if
7: end for
8: return $\mathcal{P}$
```

The schematic design derived by the given algorithm for the allocation in Table 5.1 and bound application graph in Figure 5.1 is shown in Figure 5.2. The derived biochip architecture is designed to execute the assay that we first gave in Listing 2.1 using one-to-one mixer components.

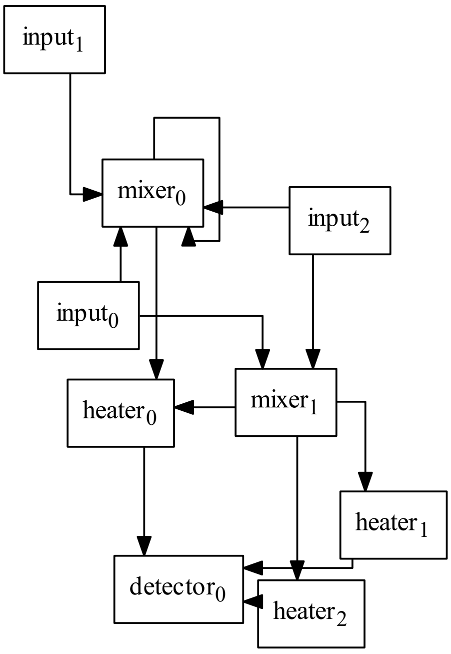


Figure 5.2: Schematic design of a biochip to run the assay in Listing 2.1.

Chapter 6

Implementation and Results

In this chapter we will describe our implementations of the proposed solutions as well as the results we get on a set of test assays.

6.1 Implementation

Our synthesis tool is implemented in Python 3.4 with ANTLR 4.5. The tool requires the ANTLR Python Runtime for Python 3 to be installed in the users python distribution. In most distributions it can be installed by running the command:

```
    pip install antlr4-python3-runtime
```

The tool consists of the following files:

```
README.txt
main.py
assays/
    complex_mix.aq
    enzyme_test.aq
    example.aq
    glucose_test.aq
    simple_mix.aq
biochip/
    Application.py
    Architecture.py
    Component.py
    ComponentLibrary.py
    Operation.py
parsing/
    Aqua.tokens
    AquaAssayVisitor.py
    AquaException.py
    AquaLexer.py
    AquaLexer.tokens
    AquaParser.py
    AquaVisitor.py
synthesis/
    Allocator.py
    Scheduler.py
```

The files in `assays/` are the Aqua assays used in testing.
The files in `biochip/` implement the system model as described in chapter 2.
The files in `parsing/` implement the Aqua parser as described in chapter 4.
The files in `synthesis/` implement the architectural synthesis described in chapter 5.

The main entry point is the `main.py` file. The instructions to run the tool are specified in `README.txt`. The tool is invoked from the command line by the command:

```
    main.py [-h] -a aqua_file -l library_file -c constraints_file
            [-x mix_accuracy]
```

If the `-h` flag is present, then the tool shows a help message displaying the usage of the tool and exits.

Otherwise, the three flags `-a aqua_file`, `-l library_file`, and `-c constraints_file` must be specified. The `aqua_file` argument must be a path to an assay in Aqua or JSON, the `library_file` argument must point to a library file in XML, and the `constraints_file` must be the path to a set of constraints in JSON. If the `-x` flag is specified with an integer argument, then the accuracy of mixing ratio approximation is increased by exponentially increasing the sum of ratios for the approximation by the argument. If -x is omitted, the approximation algorithm uses the smallest power of two larger than the sum of ratios.

The syntax for the Aqua file is specified in section 2.1. If the file is in JSON format, then it is assumed to represent the application model and the parsing step is skipped and the application model is loaded directly.

The tool generates the following files in the folder of the Aqua file:

```
/application/<aqua_file>.dot
/application/<aqua_file>.json
/architecture/<aqua_file>.dot
/architecture/<aqua_file>.json
/scheduled/<aqua_file>.dot
```

where `<aqua_file>` is the file name without extension of the Aqua file given by the `-a` flag.

The generated JSON files encode the application model and the netlist according to the format proposed in [McD]. The generated DOT files are in the Graphviz [Gra15] format and can be visualised using the Graphviz software. The files in `/application/` and `/scheduled/` are intended to be visualised with the `dot` tool, and the files in `/architecture/` with the `neato` tool:

```
dot -Tpdf -O application/<aqua_file>.dot
dot -Tpdf -O scheduled/<aqua_file>.dot
neato -Tpdf -O architecture/<aqua_file>.dot
```

6.2 Results

We have tested our solutions on five different assays. Two of the assay are from the Aqua language manual and the rest are synthetic examples. The two assay from the Aqua language manual are the enzyme test and the glucose test. The assays are listed in Appendix B, the generated application graphs in Appendix C and architecture netlists in Appendix D. The netlists have been generated using the component library shown in Table 2.1 and constraints in Table 2.2.

The assays' code line counts, total time for parsing and synthesis, and application model operation counts are shown in Table 6.1. The majority of the time

Assay	Lines	Total time (seconds)	Operations
simple_mix	10	0.10	3
complex_mix	11	0.10	9
example	24	0.17	17
enzyme_test	57	0.39	326
glucose_test	30	0.19	17

Table 6.1: Statistics of test assay synthesis.

-x	Approximation	Errors	Operations
0	[3, 6, 7]	[21.9, -2.5, -5.2]%	9
1	[5, 12, 15]	[1.6, -2.5, 1.6]%	10
2	[10, 25, 29]	[1.6, 1.6, -1.8]%	11
3	[20, 49, 59]	[1.6, -0.5, -0.1]%	12
4	[39, 99, 118]	[-1.0, 0.5, -0.1]%	15
5	[79, 197, 236]	[0.3, 0.0, -0.1]%	16

Table 6.2: Results of ratio approximation algorithm for increasing accuracies.

for the assays is spent in the ANTLR runtime to build the parse tree for input Aqua code. The enzyme test assay shows that Aqua is useful for expressing large assays with many operations compactly and that the synthesis is fast even for large application graphs.

It is difficult to determine a meaningful metric for how well the generated netlists can execute the assays. Execution time would be the most interesting metric, but to measure the execution time, the placement and routing is required and the application graph must be scheduled on the complete architecture. At that time, the performance of solutions to the two other major problems will affect the execution time, and it is hard to conclude on the performance of just the allocation and schematic design. For the same reason, fiddling with constraints at this point will not give insight into the performance of out solution.

The ratio approximation algorithm has been tested on the complex mix assay for varying values of the `-x` flag. The results are shown in Table 6.2. The parsing time was 0.02 seconds for all approximations and the target mixing ratios are $(2, 5, 6)$. We see that the errors reduce drastically at first then slowly converge towards zero while the number of operations required for the mixing scales linearly. This support the claim that mixing ratios can be approximated arbitrarily well.

Chapter 7

Conclusion and Future Work

In this chapter we will conclude on our proposed solutions and present future work in relation to the seen problems.

7.1 Conclusion

This thesis presents and solves two problems of high-level synthesis of biochips. First is the application model synthesis problem, where an application model must be derived from a representation of a biochemical application in a high-level language. Second is the architectural synthesis problem of deriving a netlist for an application-specific biochip architecture from the application model, a component library, and a set of resource constraints.

For the application model synthesis we use Aqua as the representation of biochemical applications. We provide a formal definition of the language in the form of a context-free grammar and use this grammar with ANTLR4 to generate a parser for Aqua. The parser generates a parse tree from an input assay, and we use the deterministic property of Aqua to traverse the parse tree and derive an application graph according to the application model.

During the application model synthesis we also address the mixing problem of how to achieve a mixture of any number of fluids in given ratios using only one-to-one mixer components. As not every mixture ratio is reachable using one-to-one mixers, we present an algorithm to determine reachable ratios which approximate the desired ratios. The algorithm can increase the approximation accuracy at the cost of more required mixing operations. When we have a set of reachable ratios, we use the Min-Mix algorithm to decide how to cascade one-to-one mixing operations to achieve a mixture of the target ratios.

For the architectural synthesis we propose a resource-constrained list-based scheduling algorithm to derive a preliminary binding and scheduling of an application graph as well as an allocation of biochip components. We use the allocation, binding, and application model to derive the required interconnections of the allocated components which gives the desired netlist.

An implementation of the proposed solutions is provided to solve both problems. The proposed solutions are tested on five assays written in Aqua which shows that the synthesis is feasible for large size applications. The generated application models and derived architectures are provided. The mixing ratio approximation algorithm is tested for different levels of accuracy which shows that it scales well.

7.2 Future Work

In conventional electronics, the result of an operation can be used arbitrarily often since it is stored in a register which can be read multiple times while retaining its value. In biochips however, once a unit of fluid is used in an operation, the original fluid no longer exists, so other operations that require a unit of the same fluid may no longer have access to it. This gives rise to problems when the result of an operation is required by more following operations than the actual yield of the first operations. An example of the problem is seen in Figure 3.1, where executing O_9 on a one-to-one mixer takes two units of fluids and results in two units of mixed fluid. However, the result is required by the four following operations O_{10}, O_{12}, O_{14} and O_{16}, requiring a total of four units of mixed fluid. Solving this fluid volume management problem requires knowledge of both the application and the architecture which it must run on.

A possible solution is to assume that operations use and yield predetermined volumes and then adapt the application model accordingly. Another solution is to adapt the application model when doing the final scheduling, since then both the application and architecture are known. The first solution is sub-optimal because it imposes assumptions and restrictions on the biochip architecture in the application model. The second solution is also sub-optimal, since synthesis of a biochip architecture is then reliant on a non-final application graph which can change drastically (c.f. the mixing problem).

This issue was considered during the work on the application model synthesis in this thesis, but no sound solution was obvious and the problem is left for future work.

Another problem encountered in the application model synthesis is the lack of detail in the application model. The model does not capture intrinsic details of different operations and a refinement of the application model could be future work.

Appendix A

Example Parse Tree

assay
ASSAY example START decls
stmts END <EOF>
decl ; decl ; decl ; decl ; decl ; decl ; decl ; decl ; decl ; decl;
fluid fluid fluid fluid var var var input_ input_ input_
FLUID f1 FLUID f2 FLUID f3 FLUID f4 VAR v1 VAR v2 VAR v3 dimension INPUT f1 INPUT f2 INPUT f3
[4]
stmt ; stmt ; stmt ; stmt ; control_stmt
mix incubate sense assign for_loop
MIX identifier AND identifier AND identifier IN RATIOS expr : expr : expr FOR expr
f1 f2 f3 1 3 7 60
INCUBATE identifier AT expr FOR expr
it 75 15
SENSE sense_type identifier INTO identifier
FLUORESCENCE it v1
identifier = mix
f4 MIX identifier AND identifier FOR expr
f1 f2 30
FOR v2 FROM expr TO expr START stmts ENDFOR
1 4
stmt ; stmt ;
incubate sense
INCUBATE identifier AT expr FOR expr
f4 expr * expr 30
15 identifier
v2
SENSE sense_type identifier INTO identifie
OPTICAL it
v3 index
[expr]
identif
v2

Appendix B

Aqua Assays

B.1 Simple Mix

```
ASSAY simple_mix START
    FLUID fluid1;
    FLUID fluid2;
    FLUID fluid3;

    INPUT fluid1;
    INPUT fluid2;

    fluid3 = MIX fluid1 AND fluid2 IN RATIOS 1 : 1 FOR 10;
END
```

B.2 Complex Mix

```
ASSAY complex_mix START
    FLUID a;
    FLUID b;
    FLUID c;

    INPUT a;
    INPUT b;
    INPUT c;

    MIX a AND b AND c IN RATIOS 2 : 5 : 6 FOR 10;
END
```

B.3 Example

```
ASSAY example START
    FLUID f1;
    FLUID f2;
    FLUID f3;
    FLUID f4;

    VAR v1;
    VAR v2;
    VAR v3[4];

    INPUT f1;
    INPUT f2;
    INPUT f3;

    MIX f1 AND f2 AND f3 IN RATIOS 1 : 3 : 7 FOR 60;
    INCUBATE it AT 75 FOR 15;
    SENSE FLUORESCENCE it INTO v1;

    f4 = MIX f1 AND f2 FOR 30;
    FOR v2 FROM 1 TO 4 START
        INCUBATE f4 AT 15*v2 FOR 30;
        SENSE OPTICAL it INTO v3[v2];
    ENDFOR
END
```

B.4 Enzyme Test

```
ASSAY enzyme_test START
    VAR inhibitor_diluent;
    VAR enzyme_diluent;
    VAR substrate_diluent;
```

```

FLUID Diluted_Inhibitor[4];
FLUID Diluted_Enzyme[4];
FLUID Diluted_Substrate[4];
FLUID inhibitor;
FLUID enzyme;
FLUID diluent;
FLUID substrate;
FLUID temp[4][4][4];

VAR i;
VAR j;
VAR k;
VAR RESULT[4][4][4];

INPUT inhibitor;
INPUT enzyme;
INPUT diluent;
INPUT substrate;

inhibitor_diluent = 1;
enzyme_diluent = 1;
substrate_diluent = 1;

FOR i FROM 1 TO 4 START
    Diluted_Inhibitor[i] = MIX inhibitor AND diluent IN RATIOS
        1 : inhibitor_diluent FOR 30;
    inhibitor_diluent = inhibitor_diluent * 10;
ENDFOR

FOR i FROM 1 TO 4 START
    Diluted_Enzyme[i] = MIX enzyme AND diluent IN RATIOS
        1 : enzyme_diluent FOR 30;
    enzyme_diluent = enzyme_diluent * 10;
ENDFOR

FOR i FROM 1 TO 4 START
    Diluted_Substrate[i] = MIX substrate AND diluent IN RATIOS
        1 : substrate_diluent FOR 30;
    substrate_diluent = substrate_diluent * 10;
ENDFOR

FOR i FROM 1 TO 4 START
    FOR j FROM 1 TO 4 START
```

```
            FOR k FROM 1 TO 4 START
                MIX Diluted_Inhibitor[i] AND Diluted_Enzyme[j]
                    AND Diluted_Substrate[k] FOR 60;
                INCUBATE it AT 37 FOR 300;
                SENSE OPTICAL it INTO RESULT[i][j][k];
            ENDFOR
        ENDFOR
    ENDFOR
END
```

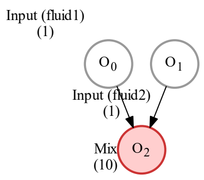
B.5 Glucose Test

```
ASSAY glucose_test START
    FLUID Glucose WASH soap;
    FLUID Reagent;
    FLUID Sample;
    FLUID a;
    FLUID b;
    FLUID c WASH h;
    FLUID d WASH w;
    FLUID e WASH x;
    VAR Result[5];

    INPUT Glucose;
    INPUT Reagent 100;
    INPUT Sample 120;

    a = MIX Glucose AND Reagent IN RATIOS 1 : 1 FOR 10;
    SENSE OPTICAL it INTO Result[1];

    b = MIX Glucose AND Reagent IN RATIOS 1 : 2 FOR 10;
    SENSE OPTICAL it INTO Result[2];

    c = MIX Glucose AND Reagent IN RATIOS 1 : 4 FOR 10;
    SENSE OPTICAL it INTO Result[3];

    d = MIX Glucose AND Reagent IN RATIOS 1 : 8 FOR 10;
    SENSE OPTICAL it INTO Result[4];

    e = MIX Sample AND Reagent IN RATIOS 1 : 1 FOR 10;
    SENSE OPTICAL it INTO Result[5];
END
```

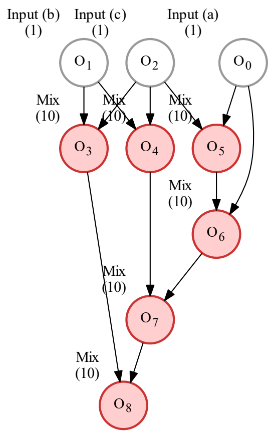
Appendix C

Application Graphs

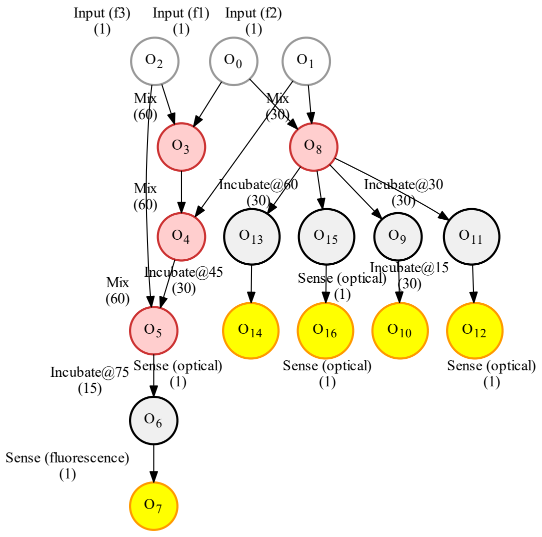
C.1 Simple Mix



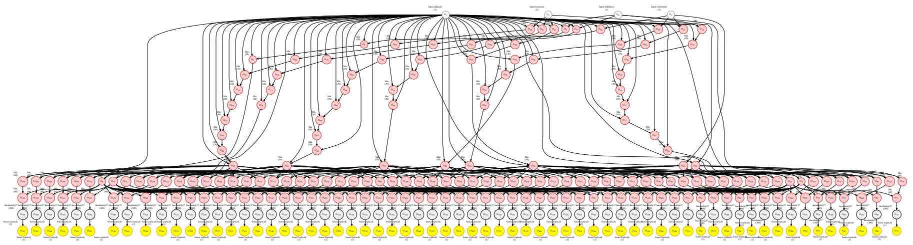
C.2 Complex Mix



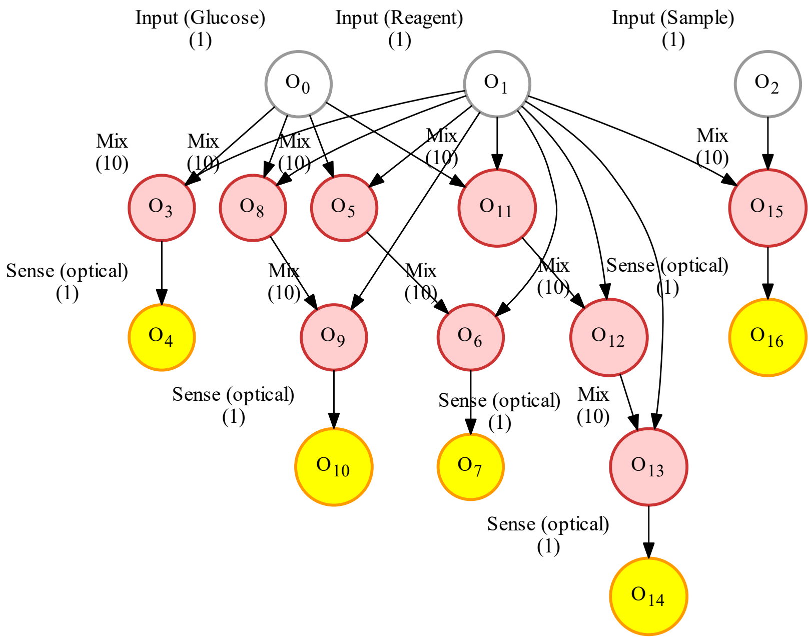
C.3 Example



C.4 Enzyme Test



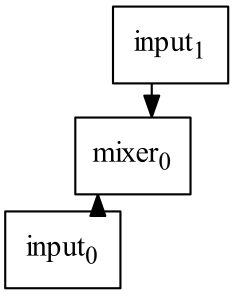
C.5 Glucose Test



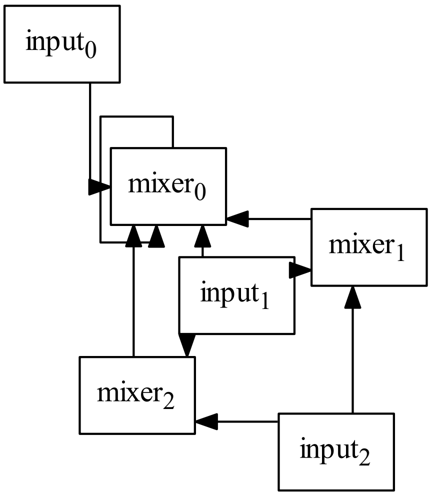
Appendix D

Architecture Netlists

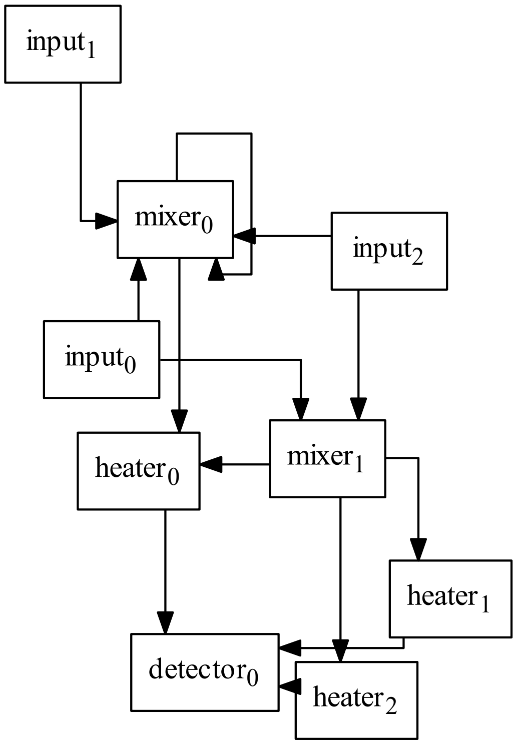
D.1 Simple Mix



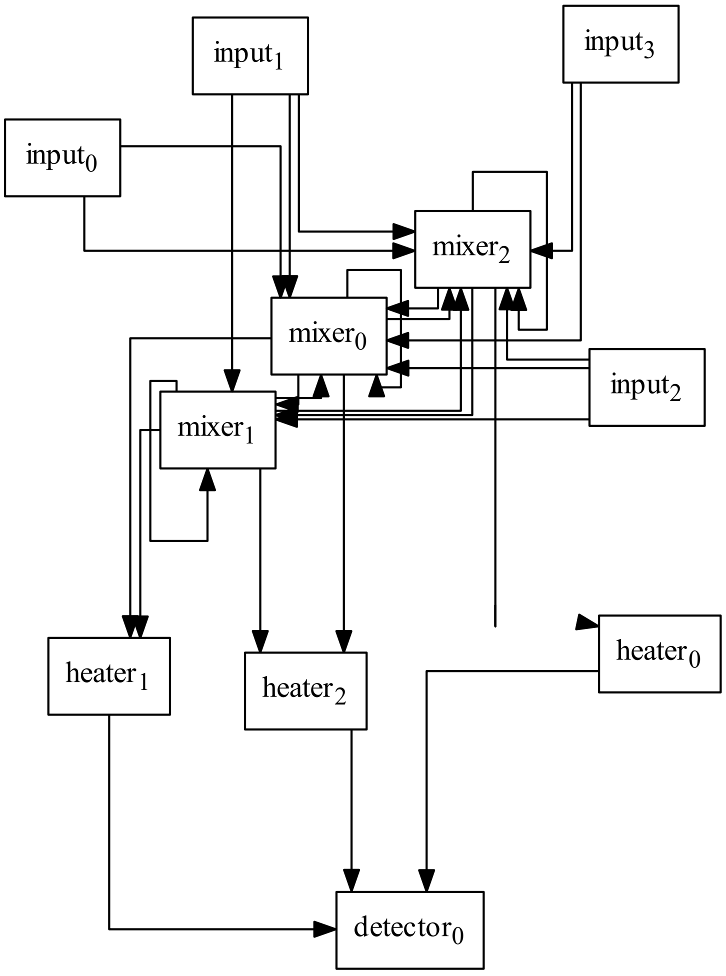
D.2 Complex Mix



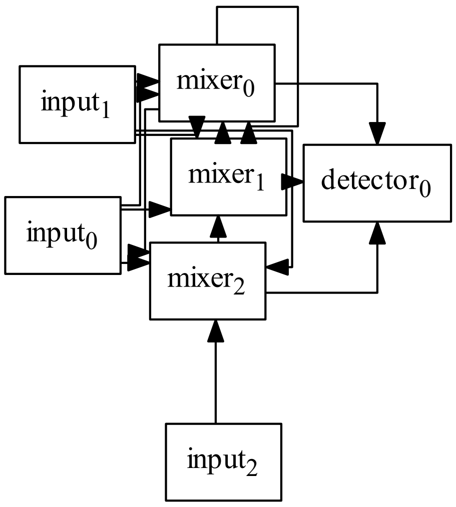
D.3 Example


input$_1$
mixer$_0$
input$_2$
input$_0$
heater$_0$
mixer$_1$
heater$_1$
detector$_0$
heater$_2$

D.4 Enzyme Test



D.5 Glucose Test



Bibliography

[ACC15] Accent website. http://accent.compilertools.net/, April 2015.

[Ami15] Ahmed M. Amin. *Aqua+ Manual.* Microfluidic Innovations, 1281 Win Hentschel Blvd., West Lafayette, IN 47906, 2015. http://microfluidicinnovations.com/.

[ANT15] Antlr website. http://www.antlr.org/, April 2015.

[AQ12] Ismail Emre Araci and Stephen R. Quake. Microfluidic very large scale integration (mVLSI) with integrated micromechanical valves. *Lab Chip*, 12:2803–2806, 2012.

[CMSJ$^+$14] Christopher T. Culbertson, Tom G. Mickleburgh, Samantha A. Stewart-James, Kathleen A. Sellens, and Melissa Pressnall. Micro total analysis systems: Fundamental advances and biological applications. *Analytical Chemistry*, 86(1):95–118, 2014. PMID: 24274655.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[Gra15] Graphviz website. http://graphviz.org/, May 2015.

[Lex15] Python lex-yacc website. http://www.dabeaz.com/ply/, April 2015.

[McD] Jeffrey McDaniel. Benchmark suite for continuous flow based microuidic biochip design automation. University of California, Riverside.

[MGW90] A. Manz, N. Graber, and H.M. Widmer. Miniaturized total chemical analysis systems: A novel concept for chemical sensing. *Sensors and Actuators B: Chemical*, 1(1–6):244 – 248, 1990.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[Min12] Wajid Hassan Minhass. *System-Level Modeling and Synthesis Techniques for Flow-Based Microfluidic Very Large Scale Integration Biochips*. PhD thesis, Technical University of Denmark, 2012.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL (*) parsing: the power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 579–598. ACM, 2014.

[TUTA06] William Thies, JohnPaul Urbanski, Todd Thorsen, and Saman Amarasinghe. Abstraction layers for scalable microfluidic biocomputers (extended version). Technical report, Massachusetts Institute of Technology, May 2006.