A Software Toolchain for Physical System Description and Synthesis, and Applications to Microfluidic Design Automation

by

Murphy Berzish

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Microfluidic circuits are currently designed by hand, using a combination of the designer's domain knowledge and educated intuition to determine unknown design parameters. As no microfluidic circuit design software exists to assist designers, circuits are typically tested by physically constructing them *in silico* and performing another design iteration should the prototype fail to operate correctly. Similar to how electronic design automation tools revolutionized the digital circuit design process, so too do microfluidic design packages have the potential to increase productivity for microfluidic circuit designers and allow more complex devices to be designed.

Two of the primary software engineering problems to be solved in this space relate to design entry and design synthesis. First, the circuit designer requires a programming language to describe the behaviour and properties of the device they wish to build, and a compiler toolchain to convert this description into a model that can then be processed by other software tools. Second, once such a model is constructed, the remaining portions of the design toolchain must be constructed. It is necessary to implement software that can find unknown design parameters automatically to relieve the designer of much of the complexity that goes into creating such a circuit. Furthermore, automated testing and verification tools must be used to simulate the device and check for correctness and safety requirements before the engineer can have confidence in their design.

In this thesis I outline work that has been done towards both of these goals. First, I describe a new programming language that has been developed for the purpose of describing and modelling physical systems, including but not limited to microfluidic circuits. This programming language, called "Manifold", has been implemented following principles and features of modern functional programming languages, as well as drawing inspiration from VHDL and Verilog, the two industry-standard programming languages for EDA. The Manifold high-level language compiler carries out the process of translating a system description into a domain-agnostic intermediate representation. This representation is then passed to a domain-specific backend compiler which can perform further operations on the design, such as creating simulations, performing verification, and generating appropriate output products.

Second, I perform a case study with respect to the creation of such a domain-specific backend for the domain of multi-phase microfluidic circuits. The process involved in taking a circuit description from design entry to device specification has a number of significant steps. I discuss in detail these steps with respect to the design of a multi-way droplet generator circuit. Such a circuit is difficult to design because of the behaviour of the key

design parameter, the volume of generated droplets. The design goal is for each droplet generator on the device to produce droplets of a certain specified volume. However, the equation relating the properties of a droplet generator to the predicted droplet volume is complex and contains several nonlinearities, making it very difficult to solve by traditional methods. Recent advances in constraint solvers which can reason about nonlinear equations over real-valued terms make it possible to solve this equation efficiently for a given set of design constraints and goals, and produce many feasible specifications for droplet generators that meet the requirements.

Another difficulty in designing these circuits is due to interactions between droplet generators. As the produced droplets have a significant hydrodynamic resistance, they affect the behaviour of the circuit by causing perturbations in the flow rates into the droplet generators. This has the potential to alter the volume of droplets that are being produced. Therefore, a means of regulating or controlling the flow rates must be found. I describe a potential solution in the form of a passive element analogous to a capacitor in an electrical circuit. Once an appropriate value for the capacitor is chosen, it remains to verify that it operates correctly under manufacturing variances in fabrication of the device. To perform this verification, a bounded model checker for real-valued differential equations is employed to demonstrate correctness or discover robustness issues. Furthermore, a simulation file for the MapleSim numerical simulation engine is generated in order to perform whole-design tests for further validation.

The sequence in which these steps are performed closely follows the concept of "abstraction refinement" in formal methods, in which successively more detailed models are checked and a failure in one step can invoke a previous step with new information, allowing errors to be caught early and introducing the ability to iterate on the design. I describe such a refinement loop in place in the microfluidics backend that integrates these three steps in a coherent design flow, able to synthesize and verify many specifications for a microfluidic circuit, thereby automating a significant portion of the design process.

The combination of the Manifold high-level language and microfluidics backend introduces a new design automation toolchain that demonstrates the effectiveness of constraint solvers in the tasks of design synthesis and verification. Further enhancements to the performance and capabilities of these solvers, as well as to the high-level language and backend, will in the future produce a general-purpose design package for microfluidic circuits that will allow for new, complex designs to be created and checked with confidence.

Acknowledgements

I am grateful for the support and encouragement of my supervisor, Dr. Derek Rayside, and my mentor, Dr. Vijay Ganesh, without whom this thesis would not have been possible.

I would like to acknowledge the support of Cody Chen, whose expertise and insight in the field of multi-phase microfluidic circuits was invaluable to the creation of these design tools.

I would like to acknowledge the support and hard work of Lucas Wojciechowski, Max Chen, and Tyson Andre, who collaborated with me on development of the Manifold front-tend compiler and intermediate representation.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter, I give a brief overview of the software engineering problems which I am attempting to solve, and summarize the major contributions detailed in the rest of the thesis.

1.1 Motivation

Much of the modern success of digital circuit design can be traced back to 1980 and the publication of the textbook "Introduction to VLSI Systems" by Carver Mead and Lynn Conway. The idea introduced in this text was that digital devices could be designed and synthesized using programming languages that described hardware instead of software. Since that time, the industry for electronic design automation has exploded and driven several decades of advancement in terms of toolchain power, design expressibility, and device complexity. Today it is possible for relatively inexperienced engineers to create a digital circuit design from scratch, bring it to completion, and physically fabricate it with the assistance of programming languages such as VHDL and Verilog and powerful EDA toolchains. These software tools perform a large number of intermediate steps for the designer, including the determination of many unknown parameters that the designer may not have specified, such as the exact location of every gate on the entire device and the length and shape of every wire between each set of connected pins. To require the designer to enter this information goes against the principle of design automation, and such an approach would not scale to the massive and intricate processor designs commonly seen today.

A second advantage of using a software representation of a physical design is the ability to use software verification tools to reason about the design. While an engineer may be able to construct a test suite that exercises the behaviour of a device on some representative set of inputs, formal verification tools are capable of checking the correctness of a design on an entire class of inputs. The ability to use these tools as part of a comprehensive validation and verification chain is enjoyed by companies such as Intel and AMD, who use extensive verification to find defects in processor designs that would not have likely been found by hand-generated tests.

It follows that a combination of these principles – specifically, design entry using a programming language, design automation using a software toolchain, and design verification using computer-aided reasoning tools – can be applied to other domains besides digital hardware as well. One domain of interest is multi-phase microfluidic circuits, in which micro- or nano-scale chemical reactions are performed using tiny droplets of reagents. These droplets are created by "droplet generators", in which two fluids meet at a junction and form a suspension of one fluid inside the other, and are subsequently moved around on a "chip" through systems of channels.

Unlike digital circuits, no such design automation revolution has yet occurred for microfluidics. In developing a design automation toolchain for microfluidic circuits, a number of critical design and engineering problems must be solved. The first of these issues is the choice of a high-level language for design entry. The two most popular languages for digital circuit design, VHDL and Verilog, were specifically tailored to this purpose, and as such may not be suitable for description of other classes of devices such as microfluidic circuits. Furthermore, these programming languages do not offer some of the capabilities and conveniences of modern programming languages. This motivates the development of a new high-level programming language, which I introduce as **Manifold**, for physical systems description. The ability to take a fresh start allows us to leverage over 30 years of programming language research and development and draw inspiration from widely-used programming languages such as ML and Haskell. Furthermore, this also allows us to design a language that is suitable for describing *general* physical systems, rather than being restricted to a specific domain such as digital circuits or microfluidic devices.

The second issue is the process of microfluidic device synthesis itself. In designing a microfluidic device, the engineer must take into account a large number of physical concerns, such as constraints on the geometry of the channels, non-linear relationships between the shape of droplet generation devices and the desired droplet volume, and interference between droplet generators due to changes in the resistance of the circuit because of droplet formation. The major software engineering problem is how to automate the design process over as many of these details as possible. By leveraging recent advances in constraint

solvers, I believe it is possible to automate a large portion of microfluidic design synthesis and verification, thereby reducing both the overhead in design entry of specifying a large number of unknown parameters and the amount of trial and error involved in creating and testing such a device. Although development of a general-purpose microfluidics design automation tool is beyond the scope of any one thesis, I present some preliminary work done in this area and demonstrate the effectiveness of these engineering principles on a case study involving the design of a specific class of microfluidic circuit. This tool is incorporated as a backend to the Manifold toolchain, allowing use of the high-level language for design entry and synthesis.

1.2 Summary of Contributions

In this thesis, I present the following major contributions:

1. I present the Manifold programming language, a concurrent functional language suitable for description and synthesis of physical systems. I also present a toolchain for the Manifold language, including a compiler for the high-level language, and a format for the intermediate representation.

2. I present a backend compiler for the Manifold toolchain that is intended to be used as a design automation tool for microfluidic circuits. Specifically, I discuss the process involved in translating descriptions of a specific class of microfluidic devices into specifications. I describe the synthesis and verification steps executed by this compiler, and discuss an abstraction-refinement strategy that allows the compiler to perform iterations on the design process, improving its performance and effectiveness.

The remainder of this thesis is structured as follows. In Chapter 2 I describe the design and implementation of the Manifold systems description language, which is intended to be useful in describing and modelling physical systems with software tools. In Chapter 3 I outline a case study in which the Manifold design language is used to create a design automation and verification toolchain for microfluidic circuits. Finally, in Chapter 4, I discuss the results and potential future work in this area, and conclude.

Chapter 2

A Programming Language for Physical Systems Design

This chapter introduces the Manifold programming language, a high-level language for describing physical systems.

2.1 Background

Manifold is a high-level language and intermediate representation for the description and verification of physical systems. It has characteristics in common with hardware description languages such as VHDL and Verilog, but it was designed to be a domain-agnostic language and therefore can describe many kinds of devices, including digital circuits and microfluidic devices.

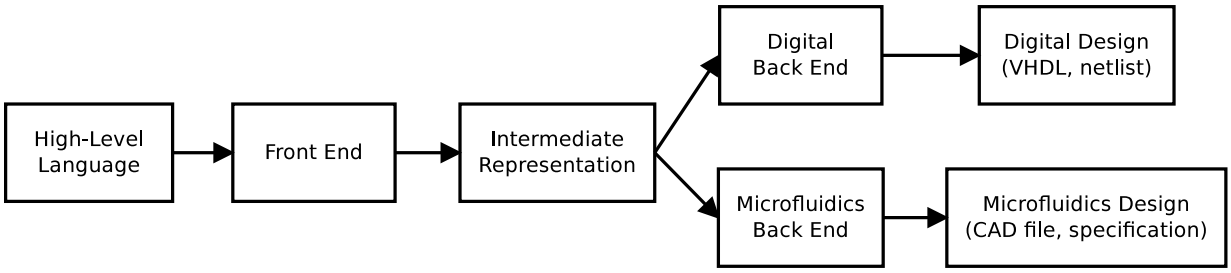

Figure 2.1: The Manifold toolchain architecture.

```
a = in();
out(a);
```

(a) Assigning `a` before it is used.

```
out(a);
a = in();
```

(b) Using `a` before it is assigned.

Figure 2.2: An illustration of concurrent statements in the Manifold language. The two listings are both correct and produce identical output.

The Manifold toolchain provides a *frontend compiler* for the high-level Manifold language that translates a high-level system description into a low-level intermediate representation. The domain-specific reasoning for microfluidic devices takes place in a custom *backend compiler* that operates on the intermediate representation and eventually outputs a specification for the microfluidic device. Figure 2.1 shows the toolchain architecture and the path from the input language to the final output product.

2.2 The Manifold Systems Description Language

The Manifold high-level language is a concurrent functional programming language that is inspired by modern functional languages such as Haskell and ML, as well as the hardware description languages VHDL and Verilog. The grammar of the language is given in Listing 2.1. The language is concurrent in the sense that all expressions are considered to be evaluated simultaneously, much as is the case with non-blocking assignments to signals in VHDL. Thus the listings shown in Figure 2.2 are both correct and both result in identical outputs, despite the apparent use of `a` before its definition in the second example.

The paradigm of a concurrent functional language is a sensible one for the description of physical systems because we do not expect a blueprint for the construction of a system to be stateful, nor do we expect it to be time-varying.

Listing 2.1: Grammar for the front-end language.

```
1  INTEGER_VALUE: [0−9]+;
2  BOOLEAN_VALUE: 'false' | 'true';
3
4  tupleTypeValueEntry: (IDENTIFIER ':')? typevalue (':' expression)?;
5  tupleTypeValue: '(' tupleTypeValueEntry (',' tupleTypeValueEntry)* ')';
6
7  tupleValueEntry: (IDENTIFIER ':')? expression;
```

```
8  tupleValue:
9  '(' tupleValueEntry (',' tupleValueEntry)* ')' |
10 '(' ')';
11
12 functionTypeValue: tupleTypeValue '->' tupleTypeValue;
13 functionValue: functionTypeValue '{' (expression EXPRESSION_TERMINATOR)* '}';
14
15 IDENTIFIER: [a-zA-Z] [0-9a-zA-Z_]*;
16 namespacedIdentifier: (IDENTIFIER '::')* IDENTIFIER;
17
18 typevalue:
19 namespacedIdentifier # Typename
20 | tupleTypeValue # TupleType
21 | functionTypeValue # FunctionType
22 ;
23
24 expression:
25 BOOLEAN_VALUE # Boolean
26 | INTEGER_VALUE # Integer
27 | tupleValue # Tuple
28 | functionValue # Function
29 | expression expression # FunctionInvocationExpression
30 | expression '.' (IDENTIFIER | INTEGER_VALUE) # StaticAttributeAccessExpression
31 | namespacedIdentifier # VariableReferenceExpression
32 | expression '=' expression # AssignmentExpression
33 | 'primitive' 'port' typevalue (':' tupleTypeValue)? # PrimitivePortDefinitionExpression
34 | 'primitive' 'node' functionTypeValue # PrimitiveNodeDefinitionExpression
35 ;
36
37 schematic: (expression ';')*;
```

Designs in the Manifold language are expressed in terms of primitive functions that correspond to atomic objects that are represented in the intermediate language (see Section 2.3). The front-end compiler has no knowledge of the semantics of primitive functions beyond what kind of object they represent in the IR. As a result, evaluation of a primitive function is equivalent to creation of a particular kind of object in the output.

Other functions in the language may then be defined in terms of primitive functions. When the compiler attempts to evaluate these functions, it performs an elaboration step

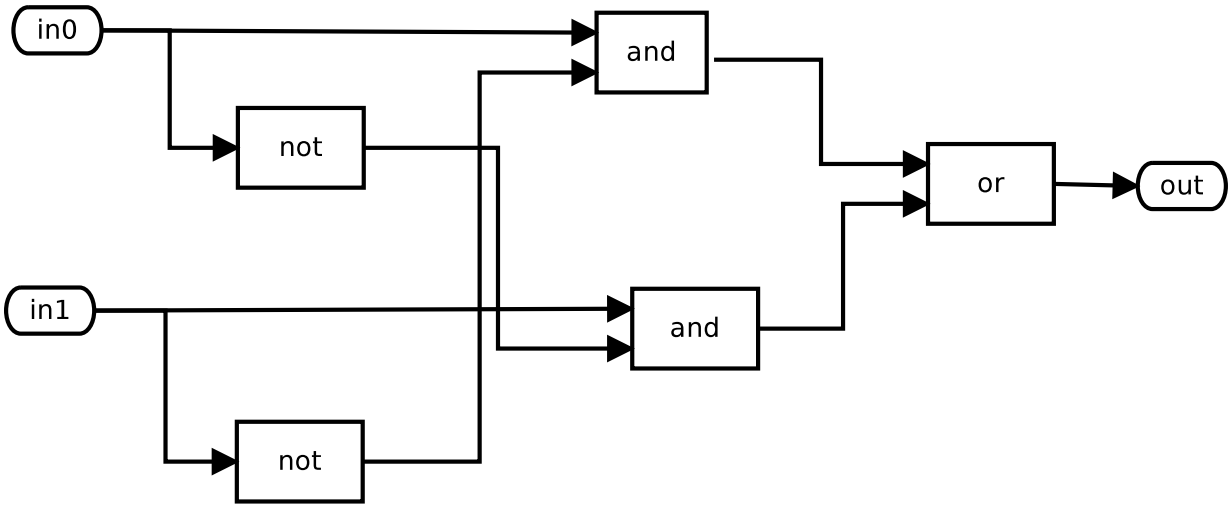

Figure 2.3: Schematic produced by compiling the function described in Listing 2.2.

in which a function invocation is replaced with the body of the function being invoked. This is similar in spirit to elaboration of components in VHDL, where circuitry from other modules can be instantiated multiple times, with the end result being a single "flat" set of gates.

As an illustrative example, consider the description of a two-input exclusive-OR gate in terms of AND, OR, and NOT gates (Listing 2.2). The functions `and`, `or`, and `not` are provided by the digital circuits library as primitive functions that correspond to nodes representing logic gates. The input and output ports of the nodes are expressed implicitly in terms of the arguments and return values from the primitive functions. Connections are also formed implicitly by performing a dataflow analysis between ports on nodes. For example, by assigning the return value of the `and` function in line 2 to `e1` and then using `e1` as an argument to the `or` function in line 4, a connection is created between the output port of one AND gate in the circuit and one of the input ports of the OR gate. The resulting nodes and connections are shown in Figure 2.3.

Listing 2.2: An XOR gate expressed in terms of simpler gates.

```
1  xor = (in0: Bool, in1: Bool) → (out: Bool) {
2    e1 = and(in0, not(in1));
3    e2 = and(in1, not(in0));
4    out = or(e1, e2);
5  }
```

When attempting to compile this function, the compiler first substitutes the function call with the body of the defined function. It then creates connections between values passed as arguments to the function and uses of those arguments, and between values returned as outputs from the function and uses of those returned values. Because elaboration of the function created more function calls, the compiler adds those call sites to a worklist and will evaluate them at a later time.

When primitive functions such as `and` are elaborated, the compiler obviously cannot perform substitution of function bodies here, since primitive functions have no body. Instead it creates a primitive node in the intermediate representation and forms connections to adjacent primitive nodes through the inputs and outputs of the function.

The compiler performs elaboration of function calls until no function calls on non-primitive functions exist and all primitive functions have been translated to the intermediate representation. Just as a software compiler for a high-level programming language translates high-level semantics into simple assembly language instructions, the Manifold high-level compiler translates complex assemblies of components into a graph of simple objects. Domain-specific backend compilers are not burdened with the responsibility of understanding and compiling the high-level Manifold language.

Similarly, the frontend compiler does not need to be burdened with domain-specific logic when dealing with particular primitives. The high-level compiler does not know anything about what an AND gate is; it only knows how to create an `and` node when it sees a particular function being used. This makes it very simple to extend the Manifold language to work with different problem domains; the primitives of that domain can be defined in the high-level language without requiring the front-end compiler to be modified, and the back-end compiler for that domain can concern itself with the domain-specific logic and reap the benefits of a simplified representation rather than having to re-implement a compiler for the entire high-level language.

2.3 Intermediate Representation

The Manifold intermediate representation (IR) is intended to be a low-level structure that describes the connections between primitive objects in a given design. An instance of an intermediate representation file for Manifold is called a *schematic*. The intermediate representation deals with four types of primitive object (further elaborated in Listing 2.3), corresponding to the primitive functions defined in the high-level language:

1. *Nodes*, which are elementary devices such as logic gates or fluid input channels;

2. *Connections*, which connect two nodes together;

3. *Ports*, which describe places on nodes where connections can be made; and

4. *Constraints*, which describe design rules or goals that are too complex to be described in terms of the other three primitives.

Each of the four primitive objects in the IR may also be annotated with attributes, which may be interpreted as constant design parameters. For example, a fluid input node may have an attribute that describes a property of the fluid that will enter the circuit, such as its viscosity. Attributes may be given as concrete values, or may be denoted as "inferred" quantities. Specifying that an attribute is to be inferred means that the backend compiler which receives such a schematic may choose any value it wishes for the value of that attribute, provided that other design goals and concrete attributes are respected. It also allows the backend to reject such a schematic if it expects a concrete attribute or is not able to choose a suitable value. It is considered an error if the type definition of a primitive object requires an attribute to be present, but neither a concrete nor inferred value is specified.

Listing 2.3: Grammar for the intermediate representation.

```
1  Schematic ::= {
2    Name ,
3    UserDefinedTypes ,
4    PortTypes ,
5    NodeTypes ,
6    ConstraintTypes ,
7    Nodes ,
8    Connections ,
9    Constraints
10 }
11 Name ::= "name" : String
12 UserDefinedTypes ::= "userDefinedTypes" : { UDT* }
13 PortTypes ::= "portTypes" : { PortType* }
14 NodeTypes ::= "nodeTypes" : { NodeType* }
15 ConstraintTypes ::= "constraintTypes" : { ConstraintType* }
16 Nodes ::= "nodes" : { Node* }
17 Connections ::= "connections" : { Connection* }
18 Constraints ::= "constraints" : { Constraint* }
```

```
19 UDT ::= String : String
20 PortType ::= String : { ‘‘signalType’’ : String , ‘‘attributes’’ : { AttributeType∗ } }
21 AttributeType ::= String : String
22 NodeType ::= String : { ‘‘attributes’’ : { AttributeType∗ } , ‘‘ports’’ : { (String : String)∗
       } }
23 ConstraintTypes ::= String : { ‘‘attributes’’ : { AttributeType∗ } }
24 Node ::= String : { ‘‘type’’ : String , ‘‘attributes’’ : { Attribute∗ } , ‘‘portAttrs’’ : {
       PortAttributes∗ } }
25 Attribute ::= String : Value
26 PortAttributes ::= String : { Attribute∗ }
27 Connection ::= String : { ‘‘attributes’’ : { Attribute∗ } , ‘‘from’’ : PortSpecifier , ‘‘to’’ :
       PortSpecifier }
28 PortSpecifier ::= ‘‘ character+ : character+ ‘‘
29 Constraint ::= String : { ‘‘type’’ : String , ‘‘attributes’’ : { Attribute∗ } }
30 Value ::= String
31 Value ::= Integer
32 Value ::= Real
33 Value ::= Array
34 String ::= ‘‘ character∗ ‘‘
35 Integer ::= 0
36 Integer ::= −? [1−9] [0−9]∗
37 Real ::= Integer . Integer
38 Real ::= Integer (. Integer)? [e|e+|e−|E|E+|E−] Integer
39 Array ::= [ Value∗ ]
```

The advantage of working with an intermediate representation is that it forms a layer of abstraction between the high-level language and the backend compiler. Instead of having to implement the entire Manifold high-level language in every backend, it is only necessary to understand how to work with the much simpler low-level schematics provided by the IR. In addition, this allows schematics to be generated by other programs, such as CAD tools, and used by existing backend compilers with no modifications. The introduction of an intermediate representation is a common strategy used to modularize compilers and separate language-specific processing from target-specific code generation. This paradigm is also common for digital hardware design; many VHDL and Verilog toolchains perform synthesis to the EDIF netlist format and then pass a collection of netlists to the backend device-specific code generator. Similarly, the Manifold toolchain performs synthesis to a common intermediate representation able to describe *all* designs, and then pass the schematic to a domain-specific backend.

2.4 Domain-Specific (Back-End) Compilers

The output from the front-end compiler is a flat list of nodes, connections, and constraints, along with the associated attributes for each object. Recall that the front-end compiler has no knowledge of any domain-specific devices; its only job is to perform design elaboration, take care of high-level language features, and output the schematic of the entire design. The task of the back-end compilers is to provide this domain knowledge and perform further translation to domain-specific output products. For instance, compiling a design that instantiates digital logic nodes and passing it to the digital circuits back-end could produce an output product in the form of a VHDL or Verilog description of the circuit. It is also the job of the backend to perform domain-specific design rule checks and interpret the meaning of each constraint passed to it from the intermediate representation.

The decision to perform these checks in the backend is motivated by the goal to keep as much domain knowledge out of the frontend as possible, in order to make it domain-agnostic. This makes the front-end compiler much simpler as it only concerns itself with the structure of the design. This also has the consequence that many design rules cannot be checked until relatively late in the toolchain; the front-end is limited to basic domain-agnostic static analysis such as type safety. I do not believe that this is a flaw of the design; the trade-off is that the front-end remains simple, and as such can be expected to perform code generation relatively quickly compared to the time expected to be taken by the backend compiler.

I have implemented two backend compilers, in order to demonstrate the applicability of the Manifold toolchain to different domains. One backend deals with microfluidic devices, and is the subject of a case study in Chapter 3. This backend uses a variety of constraint solvers and simulators to determine the unknown parameters in a microfluidic device specification and produce a feasible design. The second backend targets digital circuits, and can translate a circuit description to either VHDL or Verilog, both of which can be used by other tools such as synthesis tools for FPGAs or circuit simulators. The digital circuits backend will not be discussed further in this thesis; while it certainly demonstrates the versatility of the Manifold toolchain and the ease with which the language can be extended to new domains, the software engineering principles at work in the microfluidics backend form the basis of a much more interesting case study.

Chapter 3

Design Automation and Verification for Microfluidic Devices

In this chapter, I discuss the microfluidics backend of the Manifold toolchain in a detailed case study, describing the motivation for designing microfluidics circuits in this way and explaining the techniques and challenges encountered in designing and implementing this software.

3.1 Background

Currently, microfluidic circuit designers work by hand to produce and test specifications for devices, using domain knowledge and educated intuition to find unknown design parameters. They rely upon simulation, using tools such as MATLAB and Comsol, to check whether the values for the design parameters satisfy the design rules and constraints. In some cases, microfluidic engineers proceed directly to physical fabrication, as producing one device is relatively inexpensive, and perform tests *in silico*. If the simulation or test reports that a design rule is violated, the designer must try again, attempting to determine which parameter or parameters are causing the problem and selecting different values for the next iteration.

Araci and Brisk [7] have identified a need to automate this process, similar to how electronic design automation revolutionized the field of digital systems design. Two primary difficulties can be identified: first, representing a microfluidic circuit in a way that a designer can describe and that a software tool can understand; second, constructing such

a tool that can perform the process of discovering design parameters in an efficient way, and simulating and verifying the design to determine whether the requirements are met.

The behaviour of microfluidic devices can be described by systems of equations. Solving these equations for unknown parameters, in the presence of additional constraints that set requirements (such as fabrication process limitations or physical impossibilities), allows one to find a specification for a particular device and characterize how it will behave. In other words, the design problem can be viewed as a constraint satisfaction problem. Finding a satisfying assignment to all the variables such that all the constraints are satisfied is analogous to finding a specification for the design that meets all the requirements and design rules. Recent advances in solvers for the Satisfiability Modulo Theories problem (SMT solvers), in particular new classes of solvers that can effectively reason about systems of nonlinear equations over real-valued variables, make them particularly suitable to this task.

The creation of design automation tools for microfluidic circuits would allow microfluidic engineers to spend more time designing and improving circuits, and less time (and money) in physical fabrication and iteration of design parameters. This, in turn, means increased productivity, allowing larger and more complex circuits to be constructed and built. This is similar to the consequences of electronic design automation tools for digital circuits. With more power tools, designs can be completed and sent to production more quickly, and designers can be confident in the correctness of more complex devices that would be infeasible to characterize or verify using the "guess-and-check" approach.

3.1.1 Multi-Phase Microfluidic Devices

In multi-phase microfluidic devices, each channel may contain two or more different immiscible fluids. In a typical configuration, small droplets of one fluid (the substance being analyzed or reacted), known as the "continuous phase", are suspended in a flow of a second fluid (such as water or oil), known as the "dispersed phase". In such systems, there are several design problems to solve. The pressures at the inputs and outputs of the system must be chosen to achieve the desired flow rate, which in turn affects the rate of reaction or rate of analysis. In addition, the dimensions of the channels in the system as well as the flow rates affect the size of droplets that are formed.

The primary object of interest in a multi-phase microfluidic circuit is the droplet. Each droplet consists of a quantity of dispersed-phase fluid suspended in the continuous-phase fluid. There are several operations that can be performed on droplets; they can be created by mixing two fluids together, sorted, separated, combined, mixed, or trapped. In

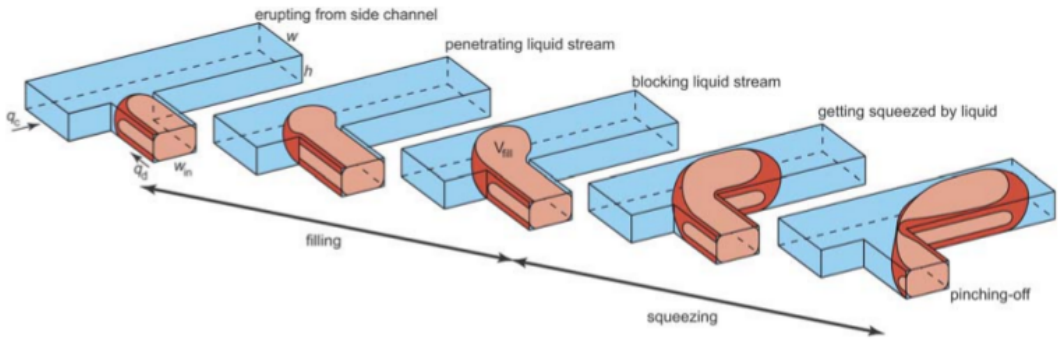

Figure 3.1: Formation of a droplet at a T-junction droplet generator. The design parameters are h, the height of the junction; w and w_{in}, the width of the continuous and dispersed channels; and q_c and q_d, the continuous and dispersed flow rates. (This figure has been reproduced from [26] for illustrative purposes.)

microfluidics, each droplet "functional unit" has clearly-defined inputs and outputs, and behaves statically (that is, there are no moving parts and no or little state). This makes the Manifold language suitable for describing the behaviour of these circuits.

In this thesis, I primarily consider the action of droplet formation. This can occur when a stream of dispersed-phase fluid meets a stream of continuous-phase fluid. The simplest droplet generator is the T-junction. In this droplet generator, the continuous and dispersed fluids meet at a channel intersection that forms a 'T' shape. If the two fluids do not mix together, the dispersed fluid will begin to form a bubble inside the continuous fluid. At the same time, the flow of continuous fluid will push the bubble down the channel. This action eventually causes the bubble to be "pinched off" and disconnected from the stream of dispersed fluid, forming a droplet ([17], [26]). An illustration of droplet formation is shown in Figure 3.1.

3.1.2 Problem Description

The specific design I will consider as a case study in this thesis is a multi-way droplet generator. The purpose of the device is to achieve a high sustained rate of droplet generation by using multiple droplet generators in parallel. The continuous-phase inputs of all droplet generators are connected together and are supplied from a single input channel, as are the dispersed-phase inputs. A schematic of this design is shown in Figure 3.2.

In the ideal case, the flow rate into each droplet generator is the same across the entire

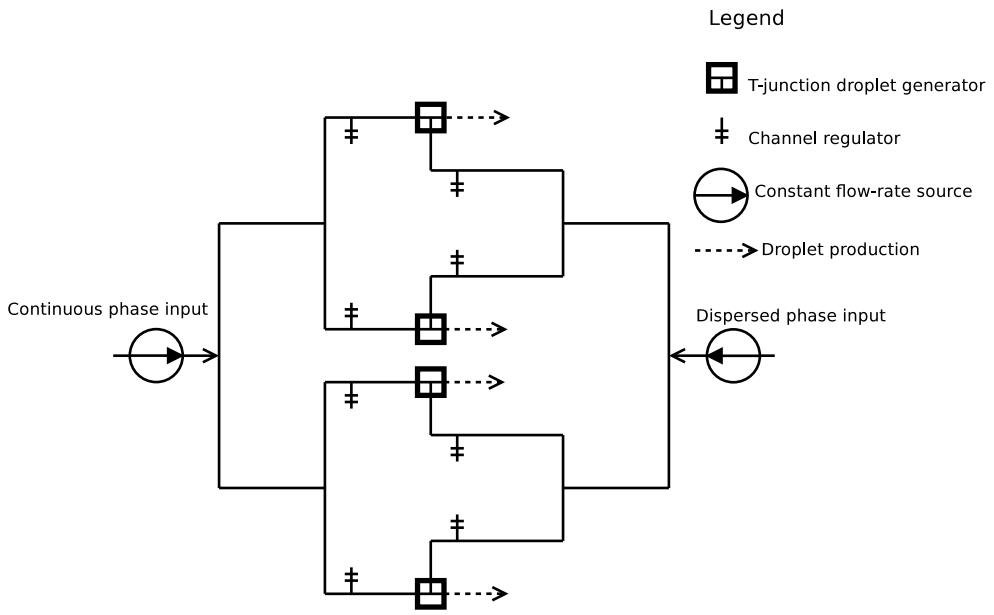

Figure 3.2: Schematic of a 4-junction multi-way droplet generator circuit.

chip, and with uniform droplet generator geometry, this implies that uniform droplets are produced at each droplet generator continuously. However, variance in the manufacturing process means that two channels that are specified to be of equal dimensions will not necessarily have identical dimensions on the physical device. The difference in resistance between channels that are specified to be identical will in turn result in different flow rates into different droplet generators. The discrepancy in flow rates has two consequences. First, if the flow rates differ from the specified values, droplets will be produced with volumes that are different from what is desired. This in turn means that other devices downstream from this one which use these droplets will receive the wrong quantities of reagents. The second consequence relates to the timing of droplet formation. As the rate of flow into a droplet generator changes the rate at which droplets are filled up, and hence the rate at which they are produced, two droplet generators with identical geometry but different flow rates will produce droplets at different times. Additionally, droplets in channels contribute to the overall resistance of the channel. This means that if a droplet generator on one part of the chip forms a droplet before any others do, the change in resistance at the output channel may affect the flow rates for other droplet generators elsewhere on the chip ([18], [25]).

One potential solution is a closed-loop feedback system that measures the difference

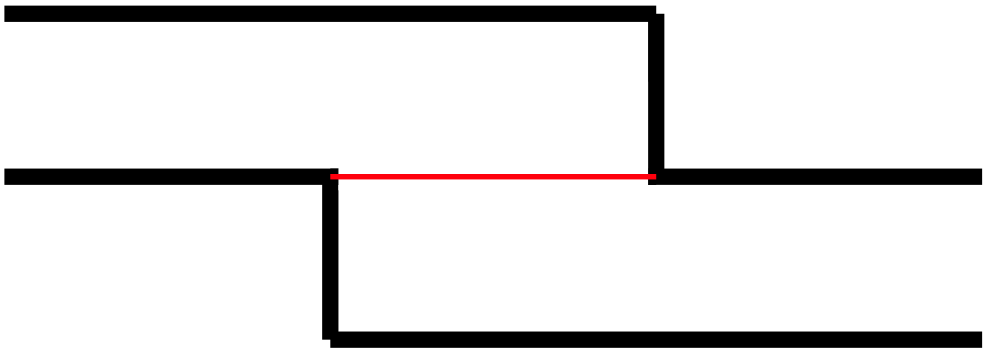

Figure 3.3: Schematic of a microfluidic regulator. The thick lines are the walls of two adjacent channels; the thin line intersecting both channels is a flexible membrane separating the fluids.

between the expected and actual flow rates and changes the flow into the system in order to attain the desired droplet volume. This solution is not feasible, for several reasons. First, adding flow sensors to each channel is prohibitively expensive and requires a large number of components. Second, using an active control loop means that additional equipment is necessary to perform the control action, which may not be available to all labs that wish to design and use such a device. Finally, adding extra control points to adjust flow rates defeats the purpose of the multi-way droplet generator design, which is to supply fluid to many droplet generators with the minimum number of external inputs (one for the continuous fluid and one for the dispersed fluid).

The solution explored here is to introduce devices that are the analogue of capacitors in an electrical circuit. These devices are flexible inter-channel membranes that behave analogously to capacitors, where the capacitance is determined by the thickness of the membrane. A schematic of this device is shown in Figure 3.3. The purpose of these membranes is to act as regulators in the circuit. Just as capacitors can regulate fluctuations in voltage and current in an electronic circuit, so too can membranes behave as regulators of pressure and flow rate in a microfluidic circuit.

The design problem, from a microfluidics point of view, is to minimize the effect of manufacturing variance on the performance of the multi-way droplet generator. In particular, the following design goals and constraints must be observed in order to produce a usable specification:

- Each produced droplet must be within 2% of the specified droplet volume. It is acceptable for an initial "startup period" in which droplets have incorrect volumes

while the device reaches steady-state conditions, but if this is done, the startup period must be shorter than 5 seconds.

- Due to fabrication process requirements, all channels have a minimum and maximum width and height. Furthermore, all channels must have the same height across the entire chip.
- The input flow rates of the continuous and dispersed fluids are constrained to a range defined by the syringe pump technology being used to inject fluids into the device.
- The maximum dimensions of the device are constrained, and the entire circuit must fit on a 2-dimensional plane with fixed length and width.

In terms of design automation, the problem is to design a Manifold backend that performs the following three steps. First, given a specified droplet volume, design a droplet generator that produces droplets of this volume. Second, given a target rate of droplet production, design a multi-way droplet generator, incorporating the single droplet generator component from the previous step, that achieves the desired output. Finally, given information about potential manufacturing variances in the design from the previous step, introduce regulators to minimize the effects of such variances on the performance of the device.

3.2 Design Synthesis

The designer of a microfluidic device in the Manifold language need not specify concrete values for all parameters in the design. To require this would be a waste of effort and would provide little benefit over the guess-and-check by-hand approach that is currently used. Instead, the designer is free to specify concrete values only for the design parameters they are interested in controlling or identifying up front. The backend compiler also reads a file separate from the main schematic that describes the parameters of the fabrication process that will be used to manufacture the device. This includes constraints such as the minimum and maximum size of the chip or substrate upon which the device is built and the minimum and maximum dimensions of channel that can be fabricated. These parameters are kept separate so that they can be reused across different designs, or alternatively to allow a single design file to be shared among different groups with different fabrication processes without requiring recompilation in the front-end. The back-end compiler generates clauses for a constraint solver using this specification that constrain the values that can be chosen

by the solver for relevant aspects of the design. This saves the designer from having to specify, by hand and for each component in the design, all the relevant design rules related to fabrication (which would be tedious, error-prone, and non-portable).

The constraint solver is used to find values for the other design parameters that satisfy the design rules and constraints with respect to the designer's concrete choices. To accomplish this, the backend generates equations and constraints for various aspects of the design as abstract syntax trees for the SMTLIB2 low-level language [10]. These ASTs are then passed to a solver such as dReal [16] that is able to deal with nonlinear inequalities over real-valued terms.

The motivation for using a constraint solver is that packages such as MATLAB and Comsol, which are typically used by microfluidic device designers to simulate systems, are not able to conduct design parameter searches automatically. These packages are primarily intended as simulation engines, not as equation solvers. Moreover, MATLAB in particular does not have native support for solving systems of nonlinear inequalities over the reals.

Note that this is not a typical use case for a constraint solver, and certainly not the intended use case for dReal, which was originally designed to check safety properties of systems. In that use case, an answer of SAT is considered negative, as it indicates a counterexample to the safety property that is being checked. In this use case, dReal is used to perform a synthesis task; an answer of SAT is considered positive, as it indicates that a feasible design exists – or in this case, *might* exist. The major limitation of dReal is that the solution it returns is not expressed in terms of an assignment of single real numbers to variables. Rather, for each variable, dReal returns an interval in which a solution *may* lie, to within a perturbation δ from the exact solution. This property is known as δ-completeness. [1] When this occurs and the span of a given range is larger than the appropriate design tolerance, we must choose a candidate value from that range. By doing so, it is possible that the value or values that we choose do not result in a feasible specification. This introduces two complications into our design flow.

The first issue is that, while dReal returns an interval of values for each variable, producing an actual specification requires us to choose a single value from this interval. I discuss a means of approaching this problem in Section 3.2.1. A second issue is that, empirically, dReal is not able to check safety properties efficiently (i.e. in a reasonable period of time) on the full-scale model, including multiple droplet generators and regulators. Furthermore, the constraints that dReal is given are approximations to the true physical laws, and omit details of the physics that are assumed to be insignificant. The strategy I choose

[1] Note that dReal's solution procedure is sound; if it returns UNSAT, no solution exists, and this is an exact answer.

to deal with this is an iterative approach, referred to in formal methods as "abstraction refinement". I discuss this further in Section 3.2.2.

A simple application of dReal as a synthesis tool in this backend is to find combinations of channel length and width that specify a droplet generator that produces droplets of a given volume. van Steijn et al. present a predictive droplet generator model that characterizes the droplet volume based on the input flow rates and channel dimensions of a given droplet generator [26]. The expression that is used is fairly complex and contains many nonlinearities, including trigonometric functions. To a human designer, this model would not be useful in designing a droplet generator directly, except by trial and error. However, the model defines a constraint set that dReal is capable of solving. In order to design a droplet generator, therefore, we create an instance for dReal to solve consisting of the droplet generator model, a constraint on the droplet volume, and constraints on the channel dimensions and flow rates from the designer's requirements. If dReal decides that this instance is satisfiable, the model it returns can then be used to find a specification for the droplet generator, which is then used in later design phases.

3.2.1 Design-Space Search

Recall that the satisfying assignment returned by dReal is not a mapping from variables to single real numbers, but a mapping from variables to real intervals. From this assignment, we wish to extract a single, concrete real value for each variable that can form our specification and be used for further steps of the design process. Let δ be the maximum allowed perturbation in a solution. Then if each interval returned by dReal is of size δ or less, we can choose the center value from each interval as the concrete assignment to the corresponding variable. More concretely, if variable x is mapped to the interval $[l, h]$, and $h - l < \delta$, choose $x = l + \frac{h-l}{2}$. Then our specification is derived directly by computing the representative value for each variable in this way. The choice of concrete value is arbitrary; any point in the interval assignment that is returned, in fact, is a witness to the δ-satisfiability of the input formula, provided the intervals are smaller than δ [16]. The more general case, which we must deal with now, is that dReal returns one or more intervals that are larger than δ. I make the assumption that all such intervals are finite, or that the designer can provide physically realistic upper and lower bounds to make them finite. We wish to check all possible designs within this space in an efficient manner and leveraging the power of the constraint solver. A brute-force search through each possible subinterval of size δ is possible, but an alternative approach is to search larger subintervals first in the hope of eliminating larger portions of the search space (thereby allowing us to add constraints to future searches to guide the solver). To search the design space given a

system of constraints and a top-level mapping of variables to intervals, some of which may be larger than δ, the program performs the following search procedure:

1. Let C be the original constraint set. Construct the constraint set C' by adding a constraint of the form $v \in I_v$ for each mapping $v : I_v$ of a variable to an interval.

2. If all intervals are δ or smaller, emit a specification composed of a concrete value chosen from each interval.

3. Otherwise, choose the largest interval $I = [I_l, I_h]$ and subdivide it into two smaller intervals, $I_1 = [I_l, I_l + \frac{I_l + I_h}{2}]$ and $I_2 = [I_l + \frac{I_l + I_h}{2}, I_h]$. Let x be the variable corresponding to the interval I.

4. Check whether the constraint set C', augmented with the constraint $x \in I_1$, has any solutions. If it does not, skip the remainder of this step; otherwise, recursively search the space of intervals in this manner except using the mapping $x : I_1$ instead of $x : I$.

5. Check whether C' augmented with $x \in I_2$ has any solutions. If it does not, skip the remainder of this step; otherwise, recursively search again using $x : I_2$ instead of $x : I$.

The algorithm given here is inspired by the well-known DPLL algorithm for SAT solvers, except instead of branching on variables, we branch on discrete partitions of the solution space. This search procedure is terminating, as at each recursive call, the size of some interval is reduced until all intervals are smaller than δ (given the assumption that each original interval is finite). It also finds exactly the same set of solutions that would be found by searching each subinterval of size δ one-by-one, as in the worst case, where the constraint solver always returns SAT, this procedure reduces to brute-force search over all subintervals. The choice of which subinterval to branch on is influenced by the idea that, in the absence of a better heuristic, we would find it most useful to eliminate the largest solution space first; therefore, the procedure chooses the largest remaining interval in the hopes of eliminating the largest possible space in as few calls to the solver as possible.

3.2.2 Abstraction Refinement

As discussed in Section 3.2, using dReal as a means of performing synthesis requires the additional step of verifying that the chosen parameters do correspond to a feasible design. Recall that the first step of design is to synthesize a specification for a droplet generator

that produces droplets of a given size. However, this step does not take into account the presence of a regulator component or interactions with multiple droplet generators in the same device. To accomplish this, I introduce a technique known as "abstraction refinement", which is adapted from the formal verification community (as described, for instance, in [15] as an application to symbolic model checking).

In an abstraction-refinement loop, analysis of a model is initially performed with an approximate description of the system that is simpler to reason about. If we fail to find a model here, since our description is taken to be an overapproximation of the actual system, we know that there are also no solutions with respect to the more detailed system. A model found in this step, however, need not be an actual solution to the more detailed set of constraints; in this case, a second analysis step is performed. Should the model fail the second step, we can learn from this counterexample and block it from being generated again in the approximation by adding additional constraints to our model. The goal of this process is twofold; we can catch errors more quickly by performing a faster analysis step over an approximate model, and we can learn things from failed designs that will allow us to improve the designs we consider in the future.

The microfluidics backend leverages an abstraction-refinement loop by splitting the design and synthesis task across several steps, each time refining the approximation with respect to either a larger model or to more accurate descriptions of the design. Furthermore, a detected error in the design at any step (except the first) will return the process to the previous step, introducing the infeasible design as a counterexample to guide the search for future designs.

In the first step, the software designs a single droplet generator using dReal to synthesize a channel geometry. If dReal determines that this cannot be done, we can immediately declare failure; there is no reason to continue designing regulators and performing more detailed simulations if the desired droplet generator does not exist. In the second step, the software performs safety property verification on this droplet generator design, with a more detailed set of constraints. This constraint set includes the presence of a regulator, and the allowance for the channel geometry and flow rates to vary within a specified manufacturing tolerance. The backend verifies whether there exists a realization of the droplet generator, within manufacturing variance, such that the actual droplet volume lies outside of a 2% interval of the specified target volume. This step incorporates an extension of dReal known as dReach [20], which can reason about ordinary differential equations over hybrid (discrete-continuous) state-space systems. If dReach finds that this is unsatisfiable, then the safety property holds, and we can proceed to the next step. However, should dReach find a δ-satisfying model, this indicates a robustness problem with the design; the regulator is not able to compensate for variations in channel flow rates under these

conditions. In this case, we must synthesize another droplet generator. The solver from the first step is called again, with an added constraint that the previously-found specification (which has just been found to be infeasible) is disallowed. This forces dReal to synthesize a different droplet generator specification, which is then moved to the next step. Each time the regulator verification fails in the second step, an additional constraint is collected, and this repeats until no more droplet generator designs can be found.

We do not use dReal here to perform synthesis of the regulators themselves. Instead, it is assumed that all regulators across the device are uniform, and that the "capacitance" of the regulator is chosen from a pre-defined set of discrete values. The reason for this is that, pending further characterization of microfluidic membrane regulators, we do not have a good model of how the geometry of a membrane influences the regulator's capacitance. Therefore is not possible at this time to use a synthesis-driven approach as we do for the droplet generator, for which we have a good predictive model of its behaviour.

The third step is to perform a full-scale simulation of the design which includes all droplet generators, connected together from two inputs, and all regulators across the entire chip. Unlike in the previous step, we do not use dReal or dReach to perform this task as, empirically, these tools have not performed efficiently on large designs such as a 32-way droplet generator with all regulators. However, this simulation is still performed at a "behavioural" level. It does not incorporate all the physical laws that would be present under real operating conditions; instead, the goal of this step is to validate that the regulators continue to control the droplet volumes when the droplet generators are allowed to "interfere" with each other. This simulation is currently done in the MapleSim simulation engine by leveraging an interpretation of a microfluidic circuit as an analog electrical circuit (as described, for example, in [23]). A Modelica description of the design is created and simulated at several operating points, each time perturbing the values of one or more channel dimensions across the circuit. We then simulate the model and verify that the flow rates are indeed within the required bounds; deviation from this results in the design returning to the previous step, where a different regulator capacitance is chosen. If no more choices for regulators are possible, the design process returns to the step before that one, causing the process to be repeated with a totally new droplet generator specification.

The final step is to perform a detailed physical simulation of the entire design. This step takes the longest to perform, and introduces the possibility that a design may be rejected at this stage because of physical laws or interactions that were not considered in previous steps. A multiphysics simulation package such as Comsol is ideal for performing such a simulation as it contains built-in support for simulating multi-phase microfluidics circuits, including droplet generators, and can be considered in some respect to be "trusted code" that has been written and checked by domain experts. Should a design perform well

in simulations performed at this level of detail, we can have a high degree of confidence that the specification is correct.

Chapter 4

Observations

4.1 Results and Limitations

To motivate the proposed solution of regulators for the multi-way droplet generator design, I constructed by hand a MapleSim model in the electrical domain that simulates the effect of introducing regulators to a design with time-varying resistance. A schematic of this model is shown in Figure 4.1. The model is driven by two constant current sources simulating the flow of the continuous and dispersed fluids. There are two droplet generators in this model, passively simulated by the trapezoidal signal generators which control the value of a variable resistor in the output channel from each droplet generator. This simulates the effect of droplet formation on the total resistance of that channel. The currents and resistances were chosen to be on the same scale as the flow rates and channel resistances of a real microfluidic circuit. I performed two simulations with this model; one without any passive control, and one with added channel regulators, here modelled as capacitors to ground on each of the continuous and dispersed input channels to the droplet generators. The capacitor values were chosen by considering the resistor-capacitor combination in each channel as an RC filter circuit, and solving for the capacitance C in the equation $\tau = RC$, where τ is the period of droplet formation and R is the channel resistance. This gives a capacitance on the order of 10^{-6} farads. Figure 4.2 shows the resultant flow rate in the dispersed-phase channel for each of these configurations. The flow rate fluctuates considerably without regulators, but once regulators are introduced the flow rate becomes stabilized.

Note that although the model is constructed as an analogue of a microfluidic circuit, the ideas being explored here currently exist in the electrical domain only. The results

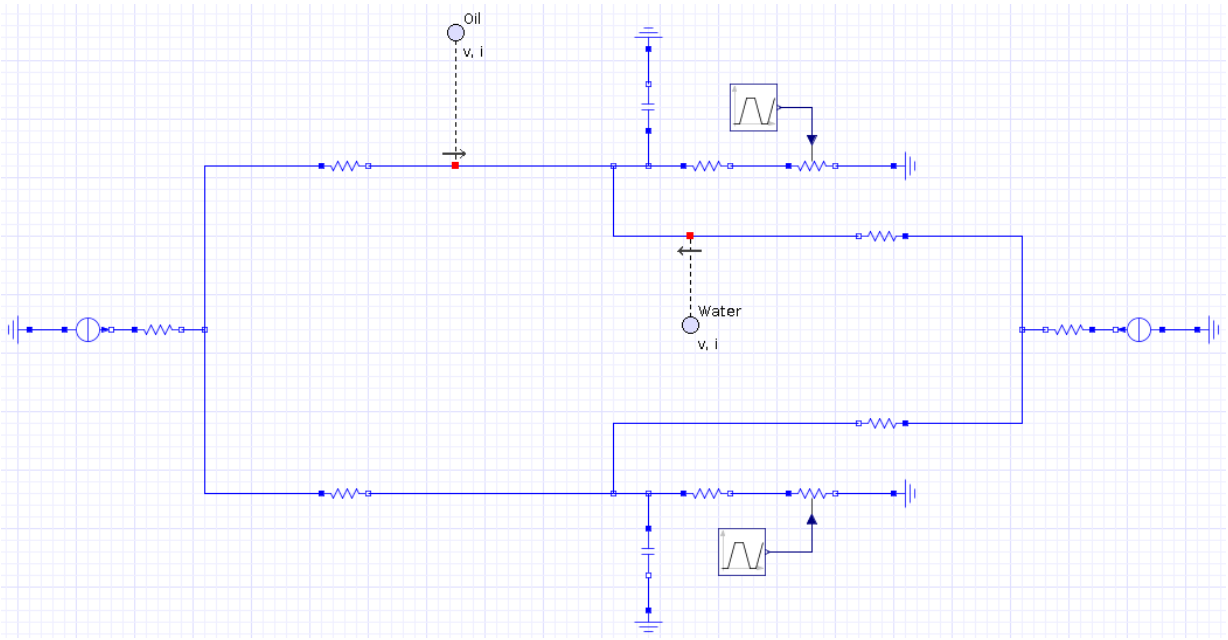

Figure 4.1: MapleSim schematic of the electrical-domain droplet generator model.

presented here would need to be improved by repeating the simulation with a more accurate droplet generator model and a better-characterized regulator component, as currently I do not know whether the regulators we plan to design will behave identically to capacitors.

As a test of the effectiveness of using dReal as a synthesis tool to design droplet generators, I altered the refinement loop to reject all designs immediately after droplet generator synthesis, forcing the first step of the design flow to repeat until all possible droplet generator designs within the given constraints were exhausted. I measured the performance of dReal by allowing it to run for 60 minutes, producing as many satisfying solutions as possible. The analysis was conducted with the full predictive model found in [26], providing bounded real interval constraints for all seven of the design parameters (channel height, continuous and dispersed channel width, junction "roundedness", continuous and dispersed flow rate, and droplet volume). Over a variety of design constraints and droplet volume targets, in the cases where dReal did not immediately return UNSAT (signifying that no droplet generator designs existed at all), after 60 minutes it had found an average of 700 droplet generator designs, and in all cases had not yet exhausted the solution space (i.e. had not yet terminated with UNSAT). This demonstrates that dReal is extremely effective when used as a synthesis tool in this manner.

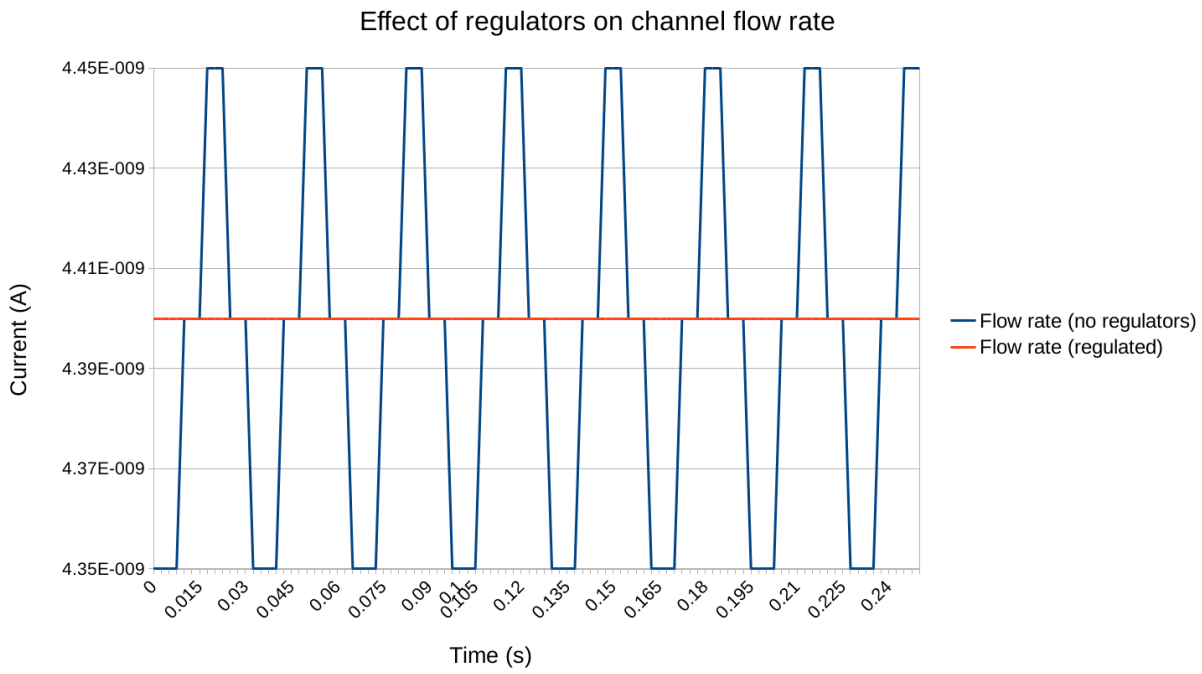

Figure 4.2: MapleSim simulation results of the model in Figure 4.1, without (blue) and with (orange) regulators. The plot is of current, i.e. flow rate, in the dispersed-phase input channel vs. time.

The results are tempered somewhat, although still indicative of success, in the subsequent step of channel regulator verification. I extracted one of the droplet generator models found by dReal from the previous experiment and started the backend at the second step, taking this design as given from the first step of synthesis. Over the search space of regulators, I observed two trends. In the case where a robustness problem existed with the design, dReach terminated almost immediately (within 30 seconds) and successfully found a counterexample. However, when the search reached a regulator design that did satisfy the safety property being checked, dReach took significantly longer (over 48 hours) to verify the absence of a counterexample and prove correctness. One the one hand, this establishes that the abstraction-refinement approach being used is successful at discovering infeasible designs early and quickly – performing such a simulation in a tool such as Comsol would have taken the same amount of time, or even longer, whether the design was correct or not. On the other hand, the performance of dReach for correct designs is much worse than expected and makes this phase of the design unacceptably time-consuming. This issue is compounded with the fact that simulation of the same design in MapleSim with the same system of differential equations can be performed over a large design space in several minutes. The immediate solution is to reverse the order of these two steps – perform the simulation first over a non-exhaustive set of conditions with each regulator, and prove correctness subsequent to this initial success. Further investigation will need to be performed to determine what, if anything, can be done to improve performance in the dReach verification step, such as simplification of the system of equations or alternate forms of expression of some of the constraints.

Empirically, a single verification task with dReach or simulation with Comsol takes significantly longer than a synthesis task with dReal. While dReal can find a specification for a droplet generator in minutes, simulating that same droplet generator in Comsol can take hours or days. This supports the idea that the backend be structured so as to use the fastest tools first and often, and perform the slow verification steps as late as possible and as few times as necessary. This strategy synergizes well with multiple abstraction-refinement loops, as discussed in Section 3.2.2. The ability to discard an infeasible design as early as possible and prevent the backend from exploring that design again is helpful in reducing the number of calls that must be made to these later, expensive steps.

4.2 Related Work

In this section I outline a number of other projects that are connected with either the Manifold toolchain itself or the general concept of microfluidic design automation.

4.2.1 System Description Languages

The most well-known system description languages are VHDL [1] and Verilog [2], both key components of modern electronic design automation (EDA) tools that allow engineers to describe the behaviour of digital logic circuits. Digital synthesis tools construct a hardware design from a VHDL or Verilog source set in two steps. First, the source-level circuit description is translated into a netlist, which describes the connections between logic gates and other components in the circuit; second, the netlist is translated into a technology-specific format, such as a firmware bitstream or a transistor-level circuit, which can then be used to fabricate a physical device. This flow has much in common with the flow of the Manifold toolchain, in terms of translation from a high-level language to an abstract intermediate form and finally to a domain-specific representation. Most notably, the concept of "elaboration" in Manifold is borrowed directly from the way VHDL and Verilog synthesis tools translate compositions of components into a flat, non-hierarchical netlist, and the design decisions made in creating the Manifold intermediate representation parallel the way netlists are represented in digital synthesis tools.

Chisel [8] is a modern hardware construction language that supports a subset of the Scala programming language, including hierarchical methods, object-oriented programming, and functional programming. It can generate code for multiple targets, including synthesizable Verilog and cycle-accurate C++ binaries for simulation. It has been used in large-scale projects, including high-performance models of processors for the RISC-V instruction set. As it is built on top of Scala, an existing and well-established programming language, it is significantly more feature-rich than Manifold, but it is not clear how difficult it would be to re-architect parts of the Chisel toolchain to work with the existing microfluidics backend.

4.2.2 Microfluidic Design Automation

Chakrabarty et al. describe the overall process of microfluidics design [13, 14]. Gleichmann et al. [19] argue that software should be used to automate this process as much as possible. The approach I describe partially realizes this proposal by integrating a number of the identified steps into one computation.

A number of IDE/drawing tools have been developed for microfluidics (e.g., [11, 24]), as well as textual languages (e.g., [21]). McDaniel et al. [21] also present a tool for simulating and debugging. That work is complementary to this one; the designer will find it useful both to draw, specify, and simulate the circuit, and to automate parts of the design and verification task.

Araci & Brisk [7] survey the current state of the art in large scale integration of microfluidic circuits. They identify the need for microfluidic design automation tools, but do not cite many examples of existing software. The only tool that they identify that uses any kind of solver is by Minhass et al. [22], which use the Gecode (gecode.org) constraint programming system to compute the routing, scheduling, and part selection for a microfluidic circuit.

Amin et al. [6] have introduced AquaCore as a general-purpose platform for microfluidics. They provide a design for a general-purpose microfluidic device that can be programmed to perform different reactions. As the problem of designing and implementing the hardware has already been solved, designers only need to specify the reaction or analysis they wish to carry out by writing a program in a domain-specific instruction set. This language allows the steps of a process to be specified in a sequential manner. The Manifold toolchain and design paradigm differs from this in an important aspect. The AquaCore platform eliminates the hardware design problem by providing a pre-existing design upon which processes can be built. In contrast, the software I describe aims to allow specification and modelling of both the chip hardware and the process it will perform. This is a more difficult problem, but allows the designer to create an application-specific device that may exhibit better performance or lower cost per unit compared to the general-purpose AquaCore platform. To draw an analogy, AquaCore is equivalent to a general-purpose CPU, able to perform many types of task with reasonable performance; I propose to create tools to design the equivalent of application-specific integrated circuits (ASICs), which are custom-built to perform one specific task with maximum performance and minimum cost per unit.

McDaniel et al [21] also introduce a domain-specific language for microfluidic device specification, called MHDL. They also propose to create design and verification tools for single-phase microfluidic devices. The MHDL language is very similar to VHDL, but differs from Manifold in that it is only able to describe the netlist of a design, whereas the Manifold language has a number of higher-level constructs. In this respect it is most similar to the intermediate representation of our toolchain, except less portable as it is specific to microfluidic devices. Furthermore, MHDL designs require the length of channels to be specified concretely, whereas our toolchain is able to solve for the channel lengths if the designer does not specify them. The verification performed is simplistic and only decides functional correctness of a process. It is not able to test whether the physical parameters of the device will result in a working design. This is because the MHDL toolchain is not powerful enough to construct the specification for a design on its own; the authors do not attempt to tackle this problem. The Manifold toolchain aims to verify both physical and functional correctness of a design.

4.3 Future Work

In this section, I describe several potential avenues for future exploration into development of the Manifold front-end language and compiler and the microfluidics backend and synthesis process.

4.3.1 Extensions to the Manifold Language and High-Level Language Compiler

Although the high-level language and compiler are currently usable to describe a variety of devices, the Manifold compiler still lacks many of the features that may be expected of a modern high-level functional language toolchain. In particular, a number of the more advanced features allowed by the language, such as higher-order functions, have not been implemented. These features are not necessary to perform the case study, as by design the backend compiler and frontend compiler never directly interact, only sharing information through the intermediate representation generated by the frontend. Therefore, a minimum viable prototype of the frontend compiler was implemented to demonstrate the feasibility of constructing such a toolchain. To make the language more widely usable, in the future I plan to extend both the frontend compiler and the language itself to support features that will be useful to developers, including loops, arrays, and higher-order functions.

The toolchain is also not very user-friendly in terms of allowing the developer to identify and correct potential problems with the high-level design. Currently the intermediate representation does not carry any information about the frontend code that was involved in generating it. This can make tracing the origin of a misplaced connection or unexpected component very difficult. This disjointness between the frontend and backend motivates the potential development of debugging tools for the Manifold language. The difficulty of developing a debugger for the frontend language is that the majority of the code is not "executed", but statically describes a physical system that the backend compiler then interprets. One potential avenue of exploration for a debugger is a visualization tool that allows the designer to observe and step through the frontend compiler's elaboration process, so they can see how the intermediate representation is being constructed and changes over time. A very simplistic version of such a visualization already exists in the frontend compiler, albeit not integrated with it in a user-friendly manner (it produces diagram descriptions that must be processed with an external tool to generate graphics). This tool was used extensively in developing and testing the frontend compiler to identify faults with the elaboration process, and it seems promising that such a tool would be useful

for users of the frontend language as well to debug their own designs.

In digital hardware toolchains, the debugger or simulator usually operates on a netlist after elaboration of components, so that the designer can see how the hardware they have described will behave under some test conditions. This, however, implies that such a tool would have to be integrated with the backend rather than the frontend. Although the microfluidics backend does leverage a simulator, in the form of MapleSim, its interaction with the simulator is automated and is not exposed to the designer. Due to the flexibility of the backend compiler, it seems plausible that simulation code could be generated for a number of popular third-party tools, including Comsol, MATLAB, and MapleSim. More work will need to be done, including detailed user studies with microfluidics domain experts, to determine how these simulators will be integrated with the backend in a manner that is suitable for interaction with the user.

4.3.2 Extensions to the Microfluidics Backend

The case study presented here deals with one very particular class of microfluidic circuits. However, the backend is expected to be extended in two ways. The first extension is to add support for additional multi-phase devices, such as droplet splitters, mergers, sorters, mixers, and traps. I expect that most of the major design principles outlined in this thesis (use of the constraint solver as a synthesis tool, abstraction refinement over the search space) will continue to apply when extended to other devices. The second extension is to investigate the possibility of modelling single-phase microfluidic circuits with this tool. Single-phase devices deal only with a single fluid at a time, but have more complex stateful dynamics and necessitate the use of external controllers to change the way the fluid moves around the device. Here, the use of tools such as dReal, which can support checking of safety properties in hybrid stateful systems, will become even more crucial to producing a feasible specification.

Two aspects of the design that are currently fixed are the lengths of each channel, which are chosen as multiples of a given "standard" channel length, and the number and position of the regulators in the circuit. When fabricating a microfluidic circuit, it is desirable to keep the channel lengths as short as possible in order to minimize the total area of the device. Furthermore, introducing two regulators for each droplet generator may not be necessary. For some designs it may be possible to eliminate some of the regulators and still produce a device that behaves as specified. Implementing this functionality will make the backend more usable by microfluidics engineers, as it will allow them to optimize a design for cost and complexity. This task could be performed by a multi-objective optimization

tool that performs a search over the space of feasible channel lengths (and/or over the space of possible regulator positions) in order to find a solution that both satisfies the existing correctness requirements and minimizes the total surface area of the design.

At the time of this writing, I was not able to have the designs fabricated and tested for correct operation as a real, physical microfluidics device. This is due in part to the lack of our ability to characterize the regulators that are essential to the functionality of the design. Currently the idea of introducing these regulators is theoretical and inspired by the function of capacitors as voltage and current regulators in analog electrical circuits. Although domain knowledge suggests that these regulators will perform as specified when implemented in this way, appropriate experiments need to be run both to verify that this idea does indeed work and to produce a predictive model, similar to the one already in use for the droplet generator, that characterizes the properties of a regulator in terms of its geometry.

One design problem not discussed here is the problem of determining the physical layout of the device. This task, known as "place and route" in the electronic design automation domain, involves finding positions for each component on the surface of the chip such that components do not overlap, channels have the correct length and do not intersect or pass too close to each other, and all parts of the circuit fit in the given chip area. An extension of this problem is optimization of the device layout to minimize the total size of the chip as well as the channel lengths, in order to lower the cost of fabrication. I believe that this can be done using a similar approach to what I have outlined in this work, incorporating a multi-objective optimizer and verifier in an abstraction-refinement loop that work together to find optimized designs and verify that they still satisfy the requirements.

4.4 Conclusion

Design automation is a powerful strategy that allows engineers to create more complex devices and hold greater confidence in the correctness of their designs. Microfluidic design automation is a very broad and deep field, and poses some unique challenges that promise to make this topic an area of active research and development for many years to come.

In this thesis I have described preliminary steps down a number of avenues of exploration in microfluidic design automation. I have described the Manifold high-level language, which is suitable for design entry and systems description of not only microfluidic circuits, but many kinds of physical systems. I have implemented a compiler for this high-level language to a netlist-like intermediate representation, which allows the designs to be processed

by other tools. I have also described a second compiler that performs design synthesis for a particular class of microfluidic circuits. The implementation of this compiler uses several external tools, including a constraint solver and a numerical simulation package, to synthesize and verify design parameters. These tools are engineered to work together through the introduction of abstraction-refinement loops. Previous steps of the design flow are able to iterate on the design if a later step fails, with information about where the failure occurred. This allows the software to consider more potential designs, making it more effective at performing the synthesis task.

The preliminary work I have done in this area suggests that this combination of high-level design entry and use of constraint solvers to synthesize and verify design parameters is very powerful, and provides the microfluidic circuit designer with considerable support from software tools. Of course, the problem being tackled is very large and far from being solved by this work alone. Many steps still need to be taken to move from proof of concept to a usable, reliable toolchain. However, I envision that this work will be extended to address new microfluidic components and more complex circuit designs in order to advance the state of the art in microfluidic design automation.

References

[1] IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, 1988.

[2] IEEE standard Verilog hardware description language. *IEEE Std 1364-2001*, 2001.

[3] *Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP)*, 2012.

[4] *18th Asia and South Pacific Design Automation Conference, ASP-DAC*, Yokohama, Japan, January 2013.

[5] *Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP)*, April 2013.

[6] Ahmed M. Amin, Mithuna Thottethodi, T. N. Vijaykumar, Steven Wereley, and Stephen C. Jacobson. Aquacore: A programmable architecture for microfluidics. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 254–265. ACM, 2007.

[7] Ismail Emre Araci and Philip Brisk. Recent developments in microfluidic large scale integration. *Current Opinion in Biotechnology*, 25:60–68, 2014.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[9] Christel Baier and Cesare Tinelli, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015.

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0, 2010. www.SMT-LIB.org.

[11] Aditya Bedekar, Yi Wang, Sachin S Siddhaye, Siva Krishnamoorthy, and Stephen F. Malin. Design software for application-specific microfluidic devices. *Clinical chemistry*, 53(11):2023–6, November 2007.

[12] Maria Paola Bonacina, editor. *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*. Springer, 2013.

[13] Krishnendu Chakrabarty and Fei Su. System-level design automation tools for digital microfluidic biochips. In *CODES+ISSS'05*, 2005.

[14] Krishnendu Chakrabarty and Jun Zeng, editors. *Design Automation Methods and Tools for Microfluidics-Based Biochips*. 2006.

[15] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[16] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In Bonacina [12], pages 208–214.

[17] Piotr Garstecki, Michael J. Fuerstman, Howard A. Stone, and George M. Whitesides. Formation of droplets and bubbles in a microfluidic t-junction-scaling and mechanism of break-up. *Lab Chip*, 6:437–446, 2006.

[18] Tomasz Glawdel and Carolyn L. Ren. Global network design for robust operation of microfluidic droplet generators with pressure-driven flow. *Microfluidics and Nanofluidics*, 13(3):469–480, 2012.

[19] N. Gleichmann, P. Horbert, D. Malsch, and T. Henkel. System simulation for microfluidic design automation of lab-on-a-chip devices. In *15th International Conference on Miniaturized Systems for Chemistry and Life Sciences*, October 2011.

[20] Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dreach: δ-reachability analysis for hybrid systems. In Baier and Tinelli [9], pages 200–205.

[21] J. McDaniel, A. Baez, B. Crites, A. Tammewar, and P. Brisk. Design and verification tools for continuous fluid flow-based microfluidic devices. [4], pages 219–224.

[22] W.H. Minhass, P. Pop, and J. Madsen. Synthesis of biochemical applications on flow-based microfluidic biochips using constraint programming. [3].

[23] Kwang W. Oh, Kangsun Lee, Byungwook Ahn, and Edward P. Furlani. Design of pressure-driven microfluidic networks using electric circuit analogy. *Lab Chip*, 12:515–545, 2012.

[24] M.F. Schmidt, W.H. Minhass, P. Pop, and J. Madsen. Modeling and simulation framework for flow-based microfluidic biochips. [5].

[25] V. van Steijn, M. T. Kreutzer, and C. R. Kleijn. Velocity fluctuations of segmented flow in microchannels. *Chemical Engineering Journal*, 135(1):S159–S165, 2008.

[26] Volkert van Steijn, Chris R. Kleijn, and Michiel T. Kreutzer. Predictive model for the size of bubbles and droplets created in microfluidic T-junctions. *Lab on Chip*, 10(19):2513–8, 2010.