Internal Report August 2014

Universiteit Leiden

Opleiding Informatica

Simulation and Mathematical Structures

for

Asynchronous Stream Circuits

Leroy van Delft

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Leiden University

Master's Thesis

Simulation and Mathematical Structures for Asynchronous Stream Circuits

Author:
Leroy van Delft BSc

Supervisors:
Dr. M.M. Bonsangue
Dr. H.J. Hoogeboom

August 29, 2014

Contents

1 Introduction **2**

2 Synchronous Stream Circuits **3**
2.1 Nodes in a Synchronous Stream Circuit . . . 3
2.2 Behaviour of a Complete Circuit . . . 5
2.3 Additional Nodes . . . 6

3 Stream Calculus **9**
3.1 Basic Stream Calculus . . . 9
3.2 Stream Differential Equations . . . 11

4 Stream Circuits & Stream Calculus **12**
4.1 Nodes and Operators . . . 12
4.2 Polynomial Functions . . . 12
4.3 Rational Circuits . . . 14
4.4 Calculating Rational Streams Directly . . . 15

5 Asynchronous Stream Circuits **17**
5.1 Asynchronous Nodes . . . 17

6 Extended Stream Calculus **19**
6.1 Periodic stream samplers . . . 19
6.2 Splitting and Merging . . . 20

7 Implementation **21**
7.1 Synchronous Stream Circuits and their Simulation . . . 21
7.2 Conversion to and from Stream Polynomials . . . 23
7.3 Asynchronous Stream Circuits and their Simulation . . . 24
7.4 Conversion from Asynchronous Circuits . . . 25

8 Conclusion and Discussion **27**
8.1 Future Work . . . 27

A How to use the Software **28**

B Examples **31**
B.1 Example Streams . . . 31
B.2 Moessner's theorem . . . 32

C Code **33**
C.1 Animation . . . 33
C.2 Stream Calculus . . . 37
C.3 Stream to polynomial conversion . . . 38

1 | Introduction

The goal of this master's thesis is to investigate stream circuits and while doing that design and produce an application which can manipulate these circuits. The reason for doing this is that there currently exists no application which satisfy our needs. By designing this application we want to make it easier to understand such circuits which is known to be difficult. We extend our application by asynchronous stream circuits and by doing this we present the first application capable of producing and simulating asynchronous stream circuits.

A stream is an infinite sequence of data from a given set, for example the rational numbers. A stream circuit is a directed graph which models function from streams to streams, i.e., an inputted stream will be converted into another. We define two variants of stream circuits, the synchronous and the asynchronous stream circuits. The mayor difference between them is that the connecting arrows in the asynchronous circuits can contain data while in the synchronous circuits they will not.

Stream circuits are an interesting area of research. They can be used to express mathematical formulas denoting streams, which is what we are interested in. A subset of these circuits called signal flow graphs are used to represent relations of cause and effect and are commonly used in electrical engineering. There are even connections to music.

This thesis is based on a previously done research project [2]. In this project we worked on a tool, which we called PolySFG, aiming to draw and animate signal flow graphs as well as be able to convert fractions of polynomial streams to them. In this thesis we continue this work and add the conversion the other way around; from polynomial streams to circuits. Furthermore we extend signal flow graphs to full stream circuits and also consider asynchronous nodes. The resulting tool is called TASCI (Tool for simulating Asynchronous Stream CIrcuits).

In Chapter 2 we introduce the synchronous stream circuits and explain their behaviour in detail. Chapter 3 is a natural continuation of the previous as we introduce the mathematics behind stream circuits: the stream calculus. The relation between the two is explained in detail in Chapter 4. In Chapter 5 we proceed by adding asynchronous nodes to circuits and explain the challenges they present. The calculus of these new structures is explained in Chapter 6. A detailed explanation of TASCI is given in Chapter 7 in which we discuss decisions made as well as explain the back-end of the tool. Finally in Chapter 8 we conclude this thesis. The appendices contain an explanation of the tool, more focused on its use. It also contain examples and some parts of the actual source code.

2 | Synchronous Stream Circuits

In this chapter we will give a definition of all the nodes in a synchronous stream circuit [10, 12]. Also we show how a full circuit behaves. A *stream circuit* is a data flow network represented by a directed graph. A subset of these networks are synchronous stream circuits and are also known *signal flow graphs* [9]. The synchronous version discussed in this chapter are limited as they only do scalar multiplication and addition. The main reason why these circuits are considered synchronous is that the connecting arrows do not have any memory, i.e, there will never be any data on the arrows. In this chapter we also present extended nodes to the synchronous streams which are not part of the SFGs, but will keep the circuit synchronous.

2.1 Nodes in a Synchronous Stream Circuit

A stream circuit is a data flow network represented by a directed graph. In this thesis we will limit the data in these circuits to the rational numbers in $\mathbb{Q}$. The nodes in the synchronous version of the circuit come in four types which we describe below. The syntax used to represent stream circuits varies among the literature. Throughout this thesis we will use the same syntax as we have used in our implementation.

The first node in our circuit is the a-multiplier, which we will refer to simply as *multiplier*, and is also known as an *amplifier*. A multiplier has exactly one input end and one output end. When a value $r \in \mathbb{Q}$ is presented at its input end the multiplier will produce a new value $r \cdot a$, where $a \in \mathbb{Q}$ is a constant value from multiplier and r is the input. The result of the multiplication is passed to its output end. The graphical representation is shown in Figure 2.1

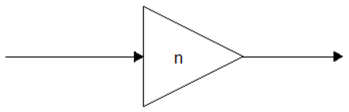


Figure 2.1: Our representation for a multiplier

The second node present in stream circuits is the *adder*, with its graphical representation given in Figure 2.2. Unlike the multiplier the adder has two input ends but like the multiplier it has exactly one output end. The behaviour of the node is as expected, it takes the value on both of its input ends r and s and passes the sum $(r + s)$ to its output. The basic adder can easily be extended to an adder with three inputs. The way to do this is to use the output of a basic adder and provide it as input of a second adder, see Figure 2.3. This will result in a circuit which takes three inputs and produces the sum of them. This process can repeated an arbitrary number of times in order to produce an adder with any (but ≥ 2) amount of inputs. For this reason we will always use the general adder in this thesis.

Thirdly we have a *copier*. The copier has exactly one input and two output ends. Like the adder its behaviour is as expected. The value presented at its input will be copied (without any modifications) to both of its outputs ends. The graphical representation of the copier can be found in Figure 2.4. Similar to the adder the copier can be extended to a more general variant by combining multiple basic copiers. If we want a copier with three outputs we take a basic copier and provide one of its output (it does not matter which one as they are the same) to the input of a second copier. We now have a circuit with three

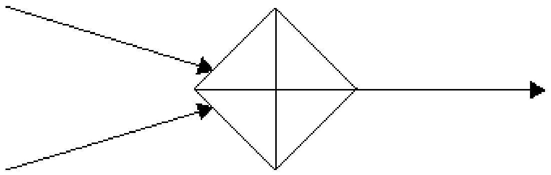


Figure 2.2: Our representation for an adder.

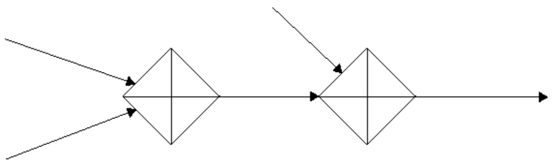


Figure 2.3: Combining two adders to make an adder with three input ends.

output ends which all contain the value of the single input as can be seen in Figure 2.5. Of course this process can be repeated to make a general copier with $n \geq 2$ output ends. From now on we will always use a general copier in this thesis.

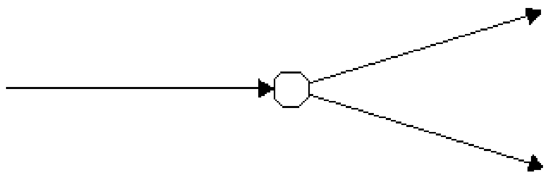


Figure 2.4: Our representation for an copier.

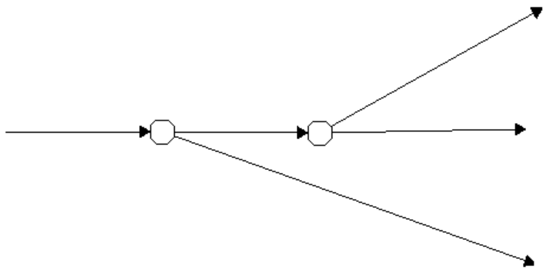


Figure 2.5: Combining two copiers to make a copier with three output ends.

The last node present in a synchronous stream circuit is the *register*, this node is also known as a *buffer* or *unit delay.* The register is a memory cell capable of storing a single value, and nothing more. The register is also never empty and generally has an initial value of 0, however this is not necessarily the case. The register has a single input end and a single output end. When a new value arrives on its input end it will placed in the register while the value that was residing there is passed on to the output end. Figure 2.6 contains the graphical representation for the register.

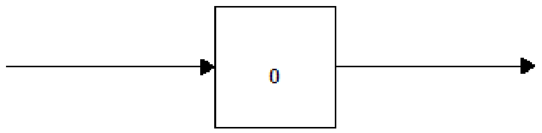


Figure 2.6: Our representation for a register

Besides these four main components there are two other “semi components” we need which are the input and the output node. The input of a stream circuit does not have to be present. There are circuits which do not have an input but still have behaviour which is interesting. The input is a stream, a formal definition of a stream will be given in Chapter 3, for now it will suffice to think of a stream as an infinite sequence of data. As stated previously our data is limited to the rational numbers $\mathbb{Q}$. Unlike the input, the output must always be present in a stream circuit. The output node will of course show the result of the circuit's computation. The graphical representation of the input node is found in Figure 2.7 and for the output node in Figure 2.8. We name the input and output in this thesis, usually something simple as `O1` for the output or `I1` for the input.

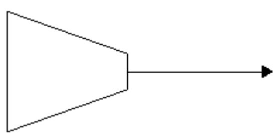


Figure 2.7: Our representation for the input.

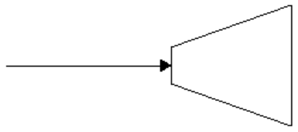


Figure 2.8: Our representation for the output.

2.2 Behaviour of a Complete Circuit

Having defined all the basic building blocks for making a complete circuit we give some examples and will show some restrictions that apply when making stream circuits. Before we go into the examples it is crucial to know the behaviour of a complete circuit. The behaviour of a stream circuit is *clock-based.* This means that there exists an imaginary clock and on every tick of the clock values are consumed by nodes moving through arrows (or wires). This movement is done synchronously, so all components produce output and consume input at the same time.

Consider the example in Figure 2.9. This is a very basic circuit with simple behaviour. Given an input it is copied to both of the multipliers. They multiply the incoming input by their internal values, 5 and -2 respectively. The results are passed to the adder who adds them to produces a new value, which is the output of our circuit. If the input I is given, then the circuit produces $(I \cdot 5) + (I \cdot -2)$, or simply $I \cdot 3$.

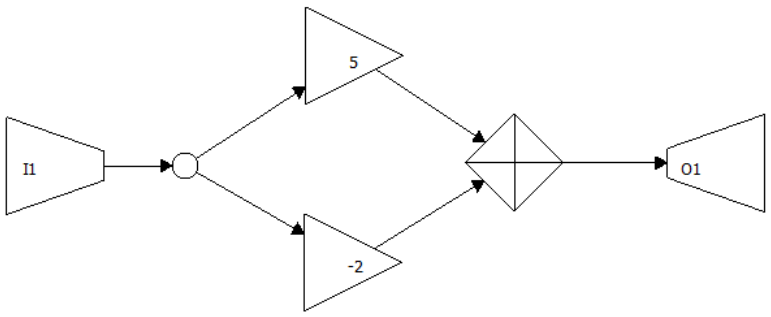


Figure 2.9: A simple stream circuit.

As another example, consider the simple circuit shown in Figure 2.10. This circuit has a major difference when compared to the previous one, in that in contains a *feedback loop.* At a glance this does not seem to matter, however if we look closely at this circuit we see there is a problem. When input is provided we have half of the inputs required for the adder but to get the second input we need to get the output of the adder first. This is a problem as the input of the adder is waiting for the output of the adder and vice versa. So we now have a *causality problem.* The solution for this problem is that we need to make sure that the the loop can always provide a value. There is only one node which does this and that is the register.

The improved version of the simple circuit is found in Figure 2.11. The behaviour of the circuit with the register is as follows: given an input we require the second input for the adder. At the first clock cycle this is 0, the initial value of the register. That result is passed to the copier, which copies it to the output and the multiplier. The result of the multiplier is than passed to the register again. Note that we said the circuit behaves synchronously, which it does, however when we reason about it we do it atomically.

The final example is more extensive and is a circuit whose behaviour is not clear at the first glance of the circuit. The initial circuit is found in Figure 2.13(a). For readability we removed the output symbol from Figures 2.13(b)–2.13(f) and placed the output there instead. This stream circuit does not have an input so our starting point is the rightmost register (initially 0). The value of that register is copied to the input of the adder, as well as to the output, hence the first output value is 0. As the adder needs a second value we look at the second register and as a second input and get 1. Meanwhile the value of the second register has also moved to the first register where it will remain for this clock-cycle. The result of

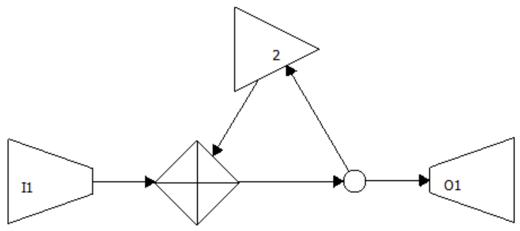


Figure 2.10: A simple circuit with a feedback loop.

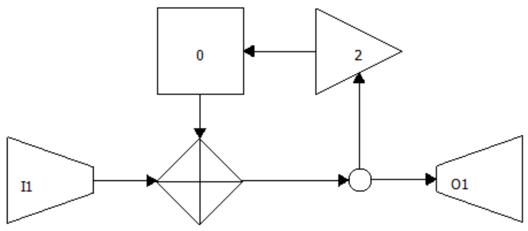


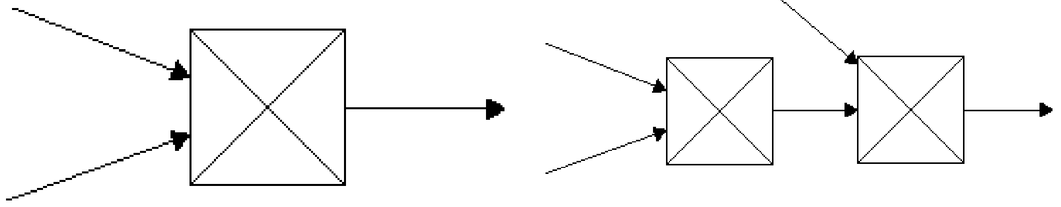
Figure 2.11: A simple circuit with a feedback loop containing a register.

the adder (1) is passed to the multiplier which multiplies it by 1 and puts it into the leftmost register. This is all that happens in the first clock-cycle and the result is shown in Figure 2.13(b). The remaining clock cycles go according to a similar pattern and all the results are shown in the remaining figures. Also note that in this example the multiplier is not necessary as it multiplies by one. In general multipliers with one as a value may be removed. The final result of this circuit are the Fibonacci numbers.

In the implementation we have solved the problem of doing all this by hand. One has probably noticed that in the last example we need to keep track of the registers constantly and if we make an error there we get the wrong result. Also doing all of this has shown that a simple looking circuit (the last example had only six nodes) can be difficult to understand until we have at least done several clock cycles. Our implementation allows us to easily simulate these circuits and because of that we can understand circuits faster.

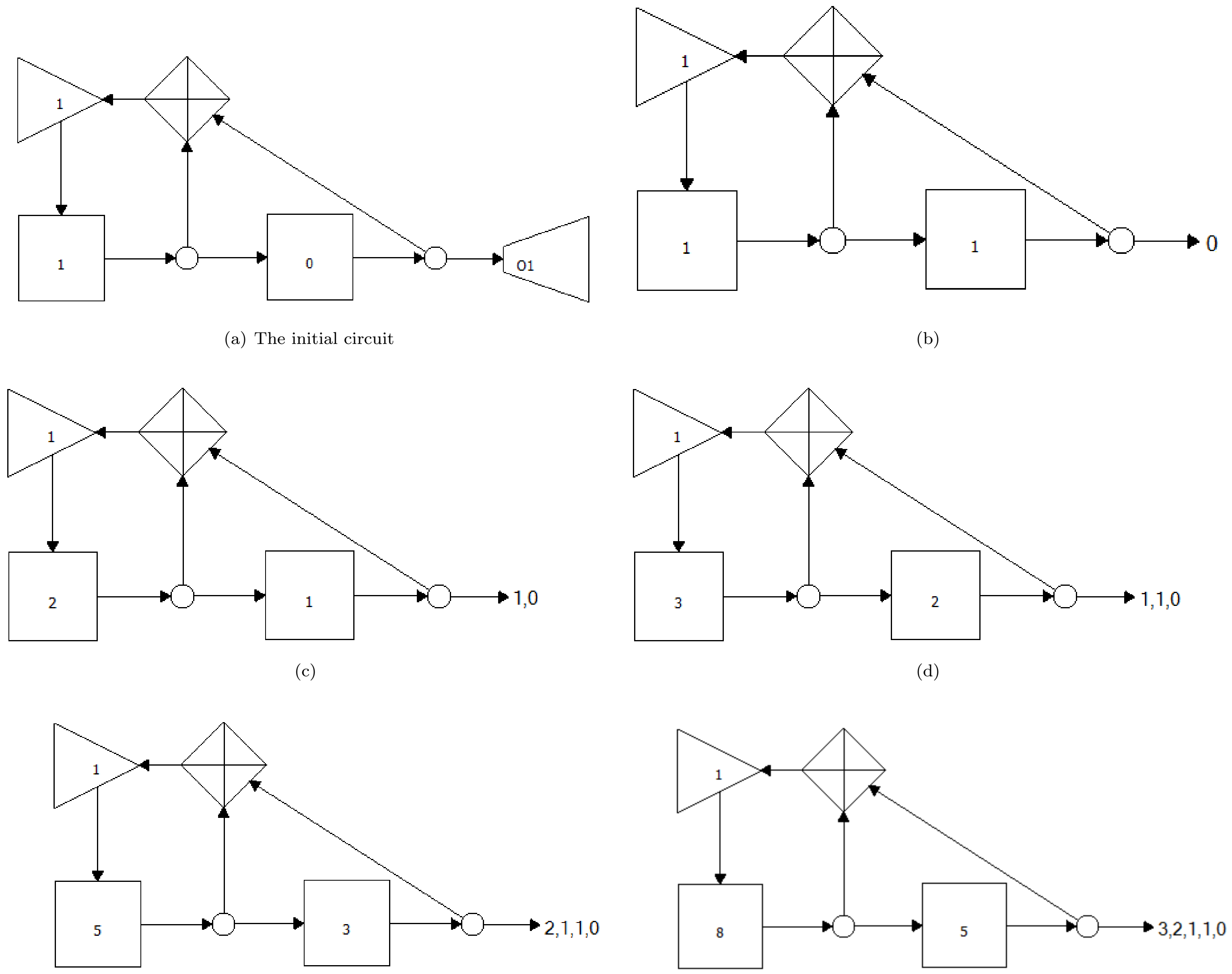
2.3 Additional Nodes

So far we have discussed the nodes for the synchronous circuits, however we can add more nodes to get more complicated behaviour while still keeping the circuit's behaviour synchronous. The first of the two nodes that are going to discuss is the *pointwise multiplier*, whose graphical representation in found in Figure 2.12(a). This node is more similar to the adder than to the multiplier as it also has exactly two inputs and one output. This node will calculate the Hadamard product, which means that the node will take both of its inputs and multiply their values, the result of the calculation is passed to the output. Like both the copier and the adder we can combine multiple point wise multipliers to make a more general variant. How to do this is shown in Figure 2.12(b). This general point wise multiplier will multiply the values on all of its inputs and multiply them. The result of the calculation is passed as the output.



(a) Our representation for a point wise multiplier. (b) Combining two point wise multipliers to make a point wise multiplier with three inputs.

Figure 2.12: Convolution product and their composition.



(a) The initial circuit

(b)

(c)

(d)

(e)

(f)

Figure 2.13: The behaviour of a stream circuit producing the Fibonacci numbers.

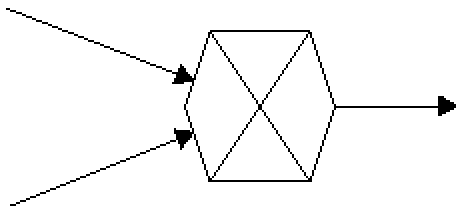


Figure 2.14: Our representation for a convolution product.

The second node we add is the "brother" of the point wise multiplier which is the convolution product. The graphical representation of the convolution product can be found in Figure 2.14. The (original) multiplier calculates a scalar product because it only takes a single input and multiplies that by a rational number. The convolution product node allows us to input two streams and calculate the convolution product of these input streams. The formal definition of the convolution product is given in the next chapter. However for now it will suffice to know that it is a method for multiplying two streams. The method also involves addition and requires all previous elements of both streams in order to calculate the next value. Because of this the node needs internal memory, moreover it requires two unbounded lists, one for each input.

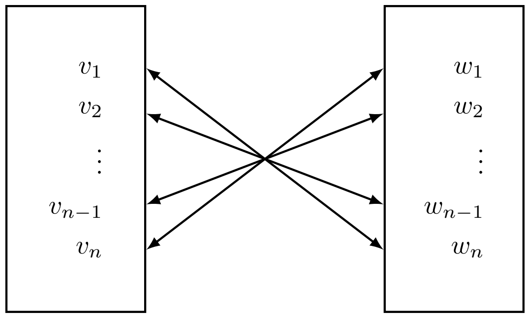


Figure 2.15: An indication on how the internal lists of the convolution product node work.

In Figure 2.15 we show how the internal memory works and how we calculate the next value. Given that we have gotten the input for the n-th value we can calculate the output as follows. First we multiply v_1 with w_n, next we multiply v_2 with w_{n-1} and we continue this process for all our values. The results of the multiplications are added together and passed as the output of the node. We cannot make a general version of this node as the convolution product is only defined for two streams.

3 | Stream Calculus

Stream calculus is the mathematics behind the stream circuits explained in the previous chapter. In this chapter we will explain the calculus as well as what a stream is formally. We will furthermore show the derivatives of streams and how stream calculus is similar to "ordinary" calculus.

3.1 Basic Stream Calculus

A *stream* [1, 5, 13] is an infinite amount of data. More formally, a stream is a mapping from the natural numbers to a set A. The set of streams over a set A is defined by $A^\omega = \{\sigma \mid \sigma : \mathbb{N} \to A\}$. In this thesis we consider only the streams over $\mathbb{Q}$, thus we have $\mathbb{Q}^\omega = \{\sigma \mid \sigma : \mathbb{N} \to \mathbb{Q}\}$. Streams are usually denoted using the Greek letters $\sigma, \tau, \rho, \ldots$. To denote the n-th element of a stream we use $\sigma(n)$. We can therefore denote a stream as $\sigma = (\sigma(0), \sigma(1), \sigma(2), \ldots)$.

We can do manipulation on streams via stream calculus. The main operations are sum, convolution and Hadamard product of two streams, as well as the inverse of a stream.

The *sum* of two streams $\sigma, \tau \in \mathbb{Q}^\omega$ is defined in Equation 3.1. As can be seen this operation is fairly trivial as we just need to sum all individual elements. Note the double use of the symbol $+$, it is used both for the sum of two elements, as well as two streams. The context will make sure which variant we are using.

$$(\sigma + \tau)(n) = \sigma(n) + \tau(n), \forall n \geq 0 \tag{3.1}$$

For two streams $\sigma, \tau \in \mathbb{Q}^\omega$ the *convolution product* is defined in Equation 3.2. The convolution product is more complex as it requires to sum over products. Unlike the sum we do have two different symbols for the product of two streams($\times$) and the product of two natural numbers ($\cdot$). The *Hadamard product* (sometimes also called point-wise multiplication) is a different method of multiplying streams and is defined in Equation 3.3. The Hadamard product is easier than the convolution product as we multiply individual elements only and do not need to record previous elements.

$$(\sigma \times \tau)(n) = \sum_{k=0}^{n} \sigma(k) \cdot \tau(n-k), \forall n \geq 0 \tag{3.2}$$

$$(\sigma \otimes \tau)(n) = \sigma(n) \cdot \tau(n), \forall n \geq 0 \tag{3.3}$$

Lastly we define the inverse of a stream $\sigma \in \mathbb{Q}^\omega$ in Equation 3.4. Two different notations can be used for the inverse of a stream either σ^{-1} or $\frac{1}{\sigma}$, we use the latter one. Note that we have two different equations for the inverse. The first is for the the first element which is trivial. To calculate all other elements we use the following recurrence relation.

$$\begin{aligned} \frac{1}{\sigma}(0) &= \frac{1}{\sigma(0)}, \text{ with } \sigma(0) \neq 0 \\ \frac{1}{\sigma}(n) &= -\frac{1}{\sigma(0)} \cdot \sum_{k=0}^{n-1} \sigma(n-k) \cdot \frac{1}{\sigma}(k), \ \forall n \geq 1 \end{aligned} \tag{3.4}$$

Now we have all operations on streams defined we want a method for using rational numbers as streams. This will allow us to write 4σ to represent $\sigma + \sigma + \sigma + \sigma$. Given a number $r \in \mathbb{Q}$, the stream $[r] \in \mathbb{Q}^\omega$ is defined as

$$\begin{aligned} [r](0) &= r, \\ [r](n) &= 0, \forall n \geq 1 \end{aligned}$$

or simply $[r] = (r, 0, 0, 0, \ldots)$. We often omit the brackets from $[r]$, the context will make clear whether we mean the stream $[r]$ or the rational number r. This definition allows us to get multiples of streams using the convolution product, this results in the equation below. Even though we are using the convolution product in this multiplication, it is often called the *scalar product* when we multiply a stream and a rational number.

$$\begin{aligned} (r \times \sigma)(0) &= r(0) \cdot \sigma(0) \\ (r \times \sigma)(n) &= \sum_{k=0}^{n} r(n) \cdot \sigma(n-k) \\ &= r(0) \cdot \sigma(n) + 0 = r(0) \cdot \sigma(n) \end{aligned}$$

A special stream we need to define is the stream X, where

$$\begin{aligned} X(0) &= 0 \\ X(1) &= 1 \\ X(n) &= 0, \forall n \geq 2. \end{aligned}$$

This stream has some special interactions with other streams defined below.

$$\begin{aligned} r \times X &= (0, r, 0, 0, 0, \ldots) \\ \sigma \times X &= (0, \sigma(0), \sigma(1), \sigma(2), \sigma(3), \ldots) \\ X^n &= \underbrace{X \times X \times \ldots \times X}_{n \text{ times}} = (\underbrace{0, \ldots, 0}_{n \text{ times}}, 1, 0, 0, 0, \ldots) \end{aligned}$$

These properties allow us to define *polynomial streams.* A general polynomial stream is defined as:

$$r_0 + r_1 X + r_2 X^2 + \cdots + r_n X^n = (r_0, r_1, r_2, \ldots, r_n, 0, 0, 0, \ldots)$$

For instance, $2 + 3X - 8X^3 = (2, 3, 0, -8, 0, 0, 0, \ldots)$. Be careful that although a polynomial stream looks like a (polynomial) *function* $f(x) = 2 + 3x - 8x^3$, for which x is a variable, it is not. This is in fact a stream build up of the constant streams $2, 3, -8$ and X.

A final stream we need to define is the *rational stream.* As defined in [12, Definition 2] this is the product of a polynomial stream and the inverse of a polynomial stream. Equivalently, a stream σ is rational if there exist $n, m \geq 0$ and coefficients $r_0, \ldots, r_n, s_0, \ldots, s_m$ with $s_0 \neq 0$, such that

$$\sigma = \frac{r_0 + r_1 X + r_2 X^2 + \cdots + r_n X^n}{s_0 + s_1 X + s_2 X^2 + \cdots + s_m X^m}$$

The restriction for $s_0 \neq 0$ is not arbitrary. The denominator defines the inverse of a polynomial stream τ, i.e., $\frac{1}{\tau}$. Looking at Equation 3.4 we see that for the first element we need calculate $\frac{1}{\tau(0)}$. And as defined above for a polynomial stream this will become $\frac{1}{s_0}$, which is not defined if $s_0 = 0$.

Finally we define some basic properties of our operators in Proposition 3.1.1. These properties are very useful as they show that some properties of "basic" arithmetic carry over to stream calculus. Properties such as, commutativity, associativity and distributivity allow us to reason more easily about streams.

Proposition 3.1.1 *For all $r, s \in \mathbb{Q}$ and $\sigma, \tau, \rho \in \mathbb{Q}^\omega$,*

$$\begin{aligned}
-\sigma &\equiv [-1] \times \sigma \\
\sigma - \sigma &= 0 \\
[r] + [s] &= [r + s] \\
\sigma + 0 &= \sigma \\
\sigma + \tau &= \tau + \sigma \\
\sigma + (\tau + \rho) &= (\sigma + \tau) + \rho \\
[r] \times [s] &= [r \cdot s] \\
[0] \times \sigma &= [0] \\
[1] \times \sigma &= \sigma \\
\sigma \times \tau &= \tau \times \sigma \\
\sigma \times (\tau + \rho) &= (\sigma \times \tau) + (\sigma \times \rho) \\
\sigma \times (\tau \times \rho) &= (\sigma \times \tau) \times \rho \\
\sigma \times \frac{1}{\sigma} &= [1]
\end{aligned}$$

$$\begin{aligned}
\frac{1}{1-X}(n) &= 1, \forall n \geq 0 \\
[r] \otimes [s] &= [r \cdot s] \\
[0] \otimes \sigma &= [0] \\
[1] \otimes \sigma &= [\sigma(0)] \\
\frac{1}{1-X} \otimes \sigma &= \sigma \\
\sigma \otimes [r] &= [[r](0) \cdot \sigma(0)] \\
\sigma \otimes \tau &= \tau \otimes \sigma \\
\sigma \otimes ([r] \times \tau) &= [r] \times (\sigma \otimes \tau) \\
\sigma \otimes (\tau + \rho) &= \sigma \otimes \tau + \sigma \otimes \rho \\
\sigma \otimes (\tau \otimes \rho) &= (\sigma \otimes \tau) \otimes \rho
\end{aligned}$$

3.2 Stream Differential Equations

We currently have shown operators on streams. Something that also needs to be discussed are the *stream differential equations* [10]. Taking the derivative of a stream is a systematic method of finding any element in such a stream. We can define a stream in terms of derivatives and initial values. The initial value or *head* of a stream σ is $\sigma(0)$. The derivative or *tail* of a stream $\sigma = \sigma' = (\sigma(1), \sigma(2), \sigma(3), \ldots)$. In Proposition 3.2.1 we give an overview of the stream operators, their initial values and derivatives.

Proposition 3.2.1 *For all $r \in \mathbb{Q}$ and $\sigma, \tau \in \mathbb{Q}^\omega$,*

Initial value	*Derivative*
$[r](0) = r,$	$[r]' = [0]$
$X(0) = 0,$	$X' = [1]$
$(\sigma + \tau)(0) = \sigma(0) + \tau(0),$	$(\sigma + \tau)' = \sigma' + \tau'$
$(\sigma \times \tau)(0) = \sigma(0) \cdot \tau(0),$	$(\sigma \times \tau)' = \sigma' \times \tau + \sigma(0) \times \tau'$
$(\sigma \otimes \tau)(0) = \sigma(0) \cdot \tau(0),$	$(\sigma \otimes \tau)' = \sigma' \otimes \tau'$
$\left(\frac{1}{\sigma}\right)(0) = \frac{1}{\sigma(0)},$ *where* $\sigma(0) \neq 0,$	$\left(\frac{1}{\sigma}\right)' = -\left(\frac{1}{\sigma}\right)(0) \times \sigma' \times \frac{1}{\sigma}$

4 | Stream Circuits & Stream Calculus

With both the synchronous stream circuits and stream calculus explained in this chapter we show how the two are related. The most important relation we show is that between the polynomial streams and synchronous stream circuits.

4.1 Nodes and Operators

In the last 2 chapters we defined both operators on streams, as well as all the nodes that make up the synchronous stream circuits. Now we will investigate how these are related. We start of by going through the four components defined in Chapter 2 and explain how they relate to stream calculus.

The a-multiplier its input by a. If we look at the stream calculus behind this node it is calculating $a \times \sigma$, where σ is the input for this node. The adder follows naturally from the analysis above. Given input streams σ and τ it calculates $\sigma + \tau$. The copier does not do any stream calculus and we therefore do not need to consider it in this section.

Finally we have the register still remaining which is the most interesting node. Assuming that the value in the register is 0, then the calculation performed is $X \times \sigma$, where σ is the input and X is the constant X defined in Chapter 3. We already know this results is $(0, \sigma(0), \sigma(1), \sigma(2), \ldots)$. Of course the initial register does not need to have a value of 0 in it. If we have an arbitrary value n as out initial value the calculation done is $(X \times \sigma) + [n]$. It is easy to see this is correct as we can take our previous result and just add $[n]$, resulting in $(n, \sigma(0), \sigma(1), \sigma(2), \ldots)$.

4.2 Polynomial Functions

Using the properties of the nodes above in relation to stream calculus one can recognize that we can convert a stream circuit into a *stream function* (something we already hinted at in Section 2.2). A stream function is a function on a stream σ and calculates $f(\sigma)$, where $f : \mathbb{Q}^\omega \to \mathbb{Q}^\omega$ i.e., a function of streams to streams. It turns out that every synchronous circuit computes such a stream function of the form: $f(\sigma) = \rho \times \sigma$, for all σ and some fixed rational stream ρ, where ρ is the circuit. For the proof of this claim we refer the reader to [11, Theorem 4.25].

To show how it all works we give an example and we use the circuit in Figure 4.1, which we already analyzed informally. Our input (`I1`) is a stream σ, the output (`O1`) is a stream function $f(\sigma)$. Going over all the marked arrows in the figure we can easily find a formula for both $P1$ and $P2$.

$$P1 = \sigma, \qquad P2 = \sigma.$$

Continuing naturally through the circuit we next find formulas for $P3$ and $P4$:

$$P3 = [5] \times P1 = 5 \times \sigma, \qquad P4 = [-3] \times P2 = -3 \times \sigma$$

Finally we have the output of the adder which is also our final formula.

$$f(\sigma) = P3 + P4 = (5 \times \sigma) + (-3 \times \sigma) = 2 \times \sigma$$

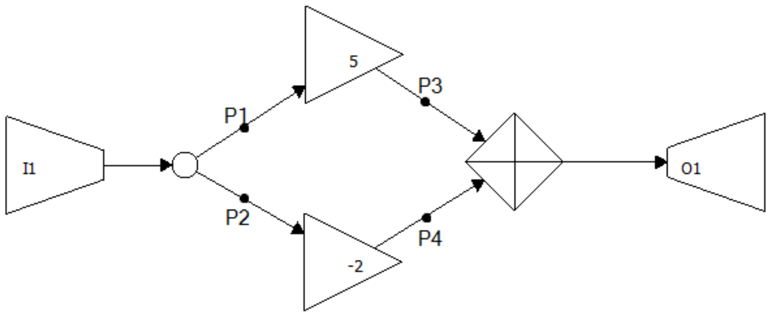


Figure 4.1: A simple stream circuit (as seen in Figure 2.9).

This formula is of the required form $f(\sigma) = \rho \times \sigma$. In our example the constant stream ρ is [2], which is a rational stream.

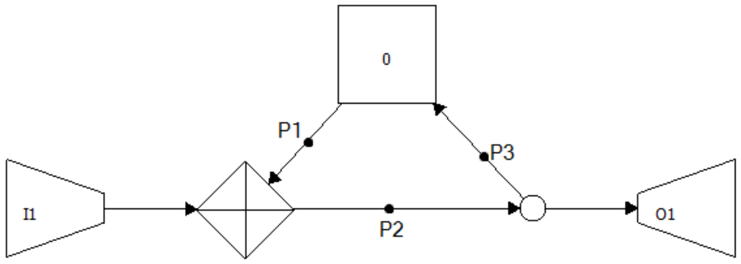


Figure 4.2: A simple circuit with a feedback loop.

The previous example is quite trivial as we do not have a feedback loop. Things get a little more complicated when we have a circuit with a feedback loop. We take an example from [12] which is shown in Figure 4.2. We have already shown how to get the formulas for each of the marked arrows, so we just give the result here:

$$\begin{aligned} P1 &= X \times P3 \\ P2 &= \sigma + P1 \\ P3 &= P2 \\ f(\sigma) &= P2 \end{aligned}$$

Solving this set of equations is possible by using Gaussian elimination. Using this we obtain:

$$\begin{aligned} P1 &= X \times P3 \\ &= X \times P2 \\ &= X \times (\sigma + P1) \\ &= (X \times \sigma) + (X \times P1) \end{aligned}$$

This implies (using the stream calculus):

$$\begin{aligned} P1 - (X \times P1) &= X \times \sigma \\ P1 &= \frac{X}{1-X} \times \sigma \end{aligned}$$

Now we have a solution for $P1$ we can start replacing from $f(\sigma)$ onwards: $f(\sigma) = P2 = \sigma + P1 = \sigma + \left(\frac{X}{1-X} \times \sigma\right) = \frac{1}{1-X} \times \sigma$.

For smaller circuits such as the one in this example the conversion process is manageable. However the complexity increases with the number of nodes, creating a larger set of equations. Therefore in TASCI we can do this conversion automatically. This allows the user to easily convert most[1] circuits into an equivalent mathematical form.

We added the Hadamard and convolution product to our set of nodes. The calculus behind these is already defined in Chapter 3 in both Equations 3.2 and 3.3. When converting to a stream we often define them as $convolution(\sigma, \tau)$ for the convolution product and $Hadamard(\sigma_0, \ldots, \sigma_{k-1})$ for the Hadamard product. We do this because we have already used the symbols we have available in the implementation.

4.3 Rational Circuits

We have shown a method for transforming a stream circuit into a stream function. Now we look at the other way around. The streams we are going to translate are the rational streams, as described by Rutten in [12]. The conversion process is easiest explained with an example. Assume we have the polynomial stream:

$$\sigma = 3 + 5X - 2X^2$$

Then the equivalent stream circuit would be the one defined in Figure 4.3. What has happened is that the coefficients have moved to the multipliers in the circuit. We can easily find them in order on the multipliers. Note that the number of registers is dependent on the degree of the polynomial stream, in this example the degree is 2. The result from the stream circuit is of course dependent on the input stream. We can choose any stream we like, but if we want expected output, our input needs to be the stream $[1] = (1, 0, 0, 0, \ldots)$. The result of this circuit is simply $(3, 5, -2, 0, 0, 0 \ldots)$, given input [1].

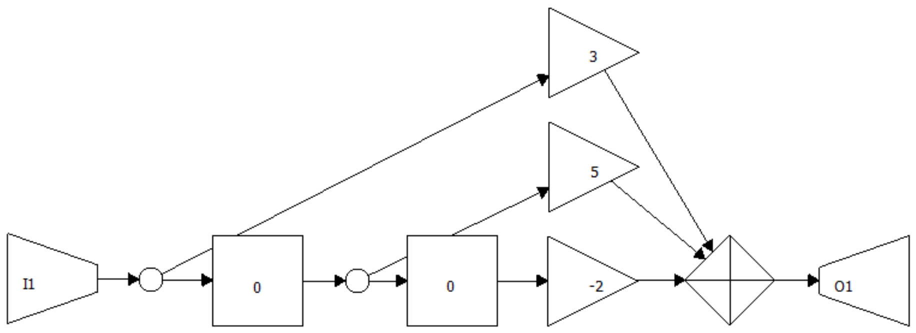


Figure 4.3: The stream circuit matching the polynomial stream above.

Next consider the example in Figure 4.4 which is the circuit corresponding to the ration stream below:

$$\sigma = \frac{2 + 4X + 8X^2}{1 + 6X + 4X^2}$$

Both figures are similar in the fact that we connect a multitude of registers with multipliers which contain the coefficients. The coefficients from numerator are on the multipliers connected via the arrows going left to right, as in our previous example. Similarly the coefficients from the denominator have moved to the multipliers with arrows going in the opposite direction. Notice that in the process the coefficients have been negated and that the first coefficient of the denominator has been placed before the first adder. The reason for this is the formula for the inverse of a stream, found in Equation 3.4. There is both a different formula for the first coefficient as well a negation sign in front of the recurrence relation. This is the cause for these two facts. With an input of [1], the first values of the output are: $(2, -8, 48, -256, 1344, \ldots)$.

[1]We will later show an example of a circuit which cannot be converted automatically.

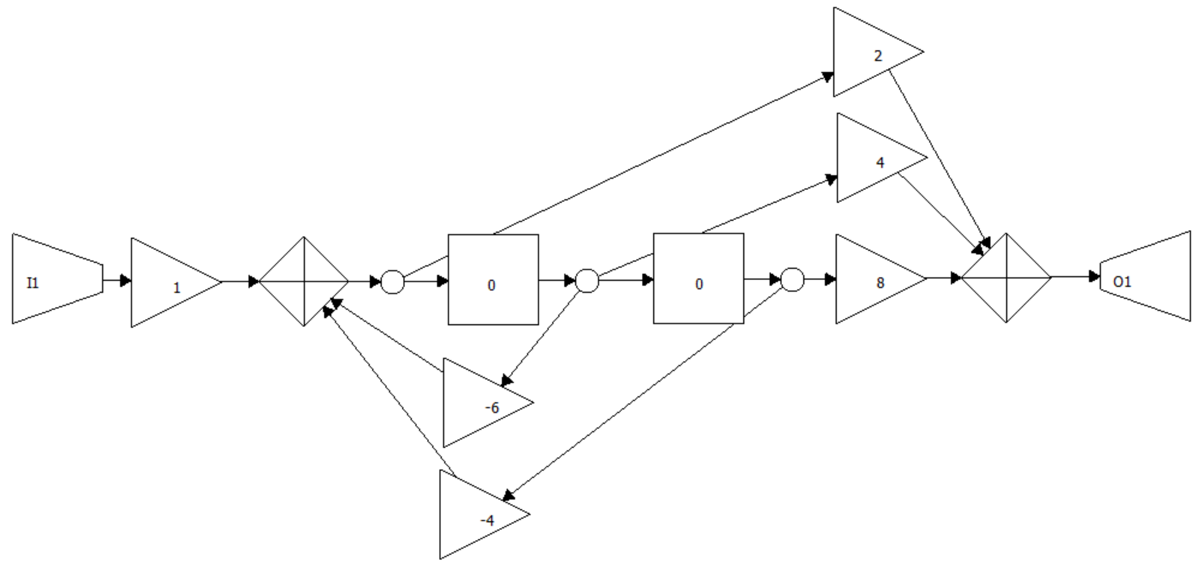


Figure 4.4: The stream circuit matching the rational stream above.

Note that in the previous example $s_0 = 1$, this is not a random choice, if $s_0 \neq 1$ we need to divide all coefficients by s_0:

$$\frac{4 + 3X^2 - 5X^3}{3 - 8X + 9X^2 + 7X^3} = \frac{\frac{4}{3} + X^2 - \frac{5}{3}X^3}{1 - \frac{8}{3}X + 3X^2 + \frac{7}{3}X^3}$$

A consequence of this is that s_0 must not be equal 0, which we already discovered in Section 3.1. For more example streams we refer the reader to Appendix B.

In our implementation we have created this method to allow the user to present any rational stream and TASCI will then convert the inputted rational stream into an equivalent stream circuit. While this conversion method is not necessarily too difficult it can be quite much work. The best thing about automating this system is that the we can easily find the output corresponding to a given rational circuit.

4.4 Calculating Rational Streams Directly

Making a stream circuit when we want to calculate the output of a rational stream is not always desired and we would like a method to do the calculation directly. However it will turn out that this will not make our life any easier. We need to solve $Input \times \frac{numerator}{denominator} = I \times \frac{n}{d} = I \times n \times \frac{1}{d}$. Because we have the inverse in the calculation we need to have two formulas. One for the first element and one for all other elements. Applying the stream calculus operators for the first element yields:

$$I \times \left(n \times \frac{1}{d}\right)(0) = I(0) \cdot n(0) \cdot \frac{1}{d(0)} \tag{4.1}$$

This is still manageable, however for all other elements $i \geq 1$ applying the stream calculus operators yields:

$$\begin{aligned} I \times \left(n \times \frac{1}{d}\right)(i) = &= \sum_{k=0}^{i} I(k) \times \left(n \times \frac{1}{d}\right)(i-k) \\ &= \sum_{k=0}^{i} I(k) \cdot \sum_{l=0}^{i-k} n(l) \times \frac{1}{d}(i-k-l) \\ &= \sum_{k=0}^{i} I(k) \cdot \sum_{l=0}^{i-k} n(l) \cdot \left(-\frac{1}{d(0)} \cdot \sum_{h=0}^{i-k-l-1} d(i-k-l-h) \cdot \frac{1}{d}(h)\right) \end{aligned} \tag{4.2}$$

Because of the triple sum and the fact that we have a recurrence relation the formula in Equation 4.2 this is difficult to calculate by hand. We can however make it slightly more manageable when our input stream $I = (1, 0, 0, 0, \ldots)$. This means that we only care for the case $k = 0$, because for all other values of k we multiply by 0. Therefore we can simplify the formula to the formula in Equation 4.3

$$I \times \left(n \times \frac{1}{d}\right)(i) = \sum_{l=0}^{i} n(l) \cdot \left(-\frac{1}{d(0)} \cdot \sum_{h=0}^{i-l-1} d(i-l-h) \cdot \frac{1}{d}(h)\right). \tag{4.3}$$

5 | Asynchronous Stream Circuits

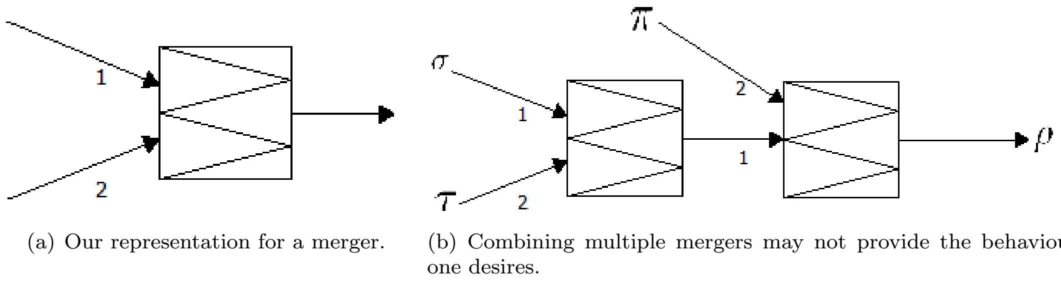
So far we presented the rational streams and synchronous circuits. In this chapter we go beyond that and introduce additional nodes for our circuits as well as explore asynchronous nodes. These new circuits are called asynchronous or extended stream circuits.

5.1 Asynchronous Nodes

Before we define the nodes that are asynchronous we need to alter our stream circuit slightly. Because these nodes are asynchronous they will not always consume all their input, or they will not always produce output. This means we need to have a look at the arrows that connect the nodes.

Assumption 5.1.1 *The connecting arrows in asynchronous circuits behave like unbounded FIFO queues.*

The first new node we define is the *merger* also called the *zip*. This node has two input ends and one output end. Its behaviour is defined as follows: it takes the input from one of its input ends and it will pass that value to its output end. The other input end will be ignored and the value present at that arrow will stay in the queue. On the next clock cycle the merger will take the input from the other input end and ignore the first end. It will continue alternating like this indefinitely. The graphical representation for the merger is found in Figure 5.1(a). The numbers on the incoming arrows indicate the order in which the arrows are handled.



(a) Our representation for a merger.

(b) Combining multiple mergers may not provide the behaviour one desires.

Figure 5.1: The merger and its composition.

Because the merger will only take input from one of its input ends in an alternating method the queues will slowly fill up. We call this fact *overproduction*, the merger does not take all of its input ends in consideration and will therefore provide too much output in comparison to its input. Another point to make is that one might want a merger with three inputs and then alternate between three inputs. The method we used for many previous nodes is shown in Figure 5.1(b). What we would want is $\rho = (\sigma(0), \tau(0), \pi(0), \sigma(1), \tau(1), \pi(1), \sigma(2), \tau(2), \pi(2), \ldots)$, however the circuit in Figure 5.1(b) will not give this result, the actual result is $\rho = (\sigma(0), \pi(0), \tau(0), \pi(1), \sigma(1), \pi(2), \tau(1), \pi(3), \sigma(2), \ldots)$. This means we cannot combine multiple mergers to create a general merger. However to allow this behaviour we will define a merger with n input ends to be alternating between all n input ends.

The counterpart to the merger is the *splitter* or *unzip*, Figure 5.2 contains its graphical representation. The splitter has only a single input end and two output ends. The splitter will take the value at its input end and pass it to one of its output ends. On the next clock cycle it will pass the input to the other output end and it will continue alternating. The order is defined by the number on the arrows.

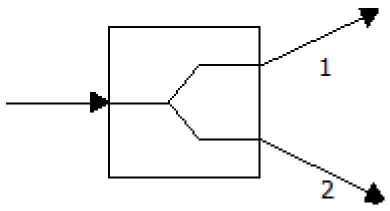


Figure 5.2: Our representation for a splitter.

Where the merger overproduces the splitter *overconsumes*. The splitter will leave one of its output ends without a value meaning that if there is an output node at the splitter's output end it will not always have an output. For all the nodes getting nothing as input means that they can output nothing. However we need to handle the special case of the register. As explained previously we are sometimes forced to "kickstart" a register. In the synchronous circuits there is no issue as a value will always be provided at the input, if the circuit is connected properly of course. However if we have splitters it may happen that we sometimes get no input. This means that we can do two things either we can accept that a register is sometimes empty or we can not accept "nothing" as an input and keep the original value. Because of the definition for the register the latter is incorrect and we therefore stick to the first one. Still there are cases where a stream circuit has no flow. The circuit in Figure 5.3 will be used to explain the issue. For easier reading we will define no value to be ε.

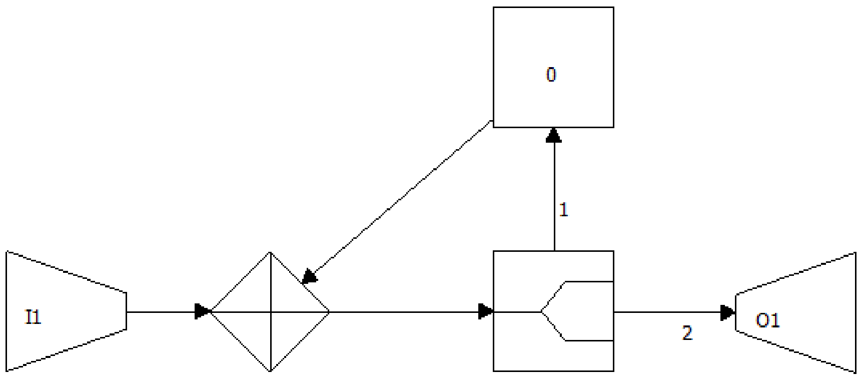


Figure 5.3: A circuit indicating the behaviour between splitters and registers.

We first define the input I1 $= (1, 2, 3, 4, \ldots)$. In this simple circuit the first thing that will happen is that the input will provide a value to the adder. In the first clock cycle this will be 1, also the register will be activated and provide 0 to the adder. The result of the addition will be passed to the splitter which will pass it to the register. Hence the first output will be ε. The second clock cycle is more interesting as now we get ε as input for the register. Once again the values from the input node and the register are added and passed on. Because we now get ε as an input we need to backtrack. We were not allowed to do the addition and therefore 1 must still be in the register, because there is no addition possible with only one input the input from the input node stays in the queue on the arrow. As a result our second output would now again be ε. We will also only get ε as an output. The reason is that when we get no input in a splitter (remember ε is just a substitute for "nothing") we do not switch arrows. So now we have a deadlock as we keep providing input we will not change arrows and keep going in this vicious circle.

From the previous example it is clear that asynchronous stream circuits introduce complications that can be easily missed. Because of the nature of both the splitter and the merger we need to keep track of the latest used arrows. The last circuit had only three nodes and still showed behaviour that only becomes clear after studying the circuit thoroughly. Our implementation allows for easier understanding of these circuits as it will keep track of the latest arrows used by either the splitter or merger.

6 | Extended Stream Calculus

With the extended stream circuits the extended stream calculus follows naturally. In this chapter we will explain the extended calculus and we will give two new nodes based on this calculus.

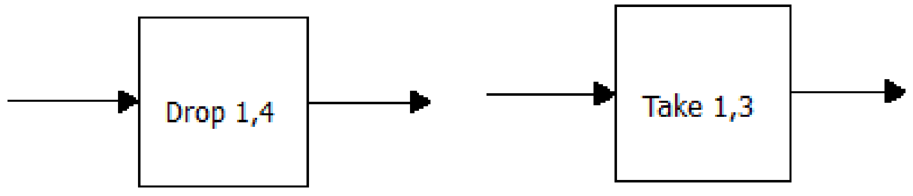
6.1 Periodic stream samplers

Up until this point we have worked with streams that are strictly monotone. Streams until now have always been a function $f : \mathbb{N} \to \mathbb{Q}^\omega$. However because of the addition of mainly the splitter the new streams will now longer always be monotone, they can now also be *periodic* streams [8]. A periodic stream produces a substream of a given input stream by repeatedly choosing elements at a fixed interval from the input stream. An example of such a function is $even : \mathbb{Q}^\omega \to \mathbb{Q}^\omega$, which produces $even(\sigma) = (\sigma(0), \sigma(2), \sigma(4), \ldots)$. The behaviour is obvious for each two incoming elements it takes the first and ignores the second.

A more general function, which behaves like *even*, is the *drop* operator. As an example consider the drop operator $D^1_4 : \mathbb{Q}^\omega \to \mathbb{Q}^\omega$, which produces $D^1_4(\sigma) = (\sigma(0), \sigma(2), \sigma(3), \sigma(4), \sigma(6), \sigma(7), \sigma(8), \sigma(10), \ldots)$. This drop operator drops every second element in a set of incoming four, hence we drop $\sigma(1), \sigma(5), \sigma(9), \ldots$ note that we start counting at zero. This drop operator has therefore a *period* of 4 and a *block size* of 2. Considering the function *even* again that function has a period of 2 and a blocksize of 1. Similarly we can define a function *odd* to be D^1_2 or $even(\sigma')$.

The twin brother of the drop operator is the *take* operator. The take operator works in the exact reverse of the drop operator. This operator also has a period and a block size like the drop operator however instead of the block size indicating the element to drop it indicates the element it needs to take. As an example consider $T^1_3(\sigma) = (\sigma(1), \sigma(4), \sigma(7), \ldots)$, this means that this take has a period of 3 and a block size of 2.

Because these operators are powerful we decided to add them as nodes to our circuits. The nodes are present in Figure 6.1. Their behaviour is as explained above, this means that like the splitter they sometimes out nothing(ε). In both the nodes in the figure below the first number is the block size whereas the second is the period.



(a) The drop operator. (b) The take operator.

Figure 6.1: the graphical representation for both the drop and rake operators.

6.2 Splitting and Merging

As we have defined both the splitter and merger nodes in the previous chapter we need the calculus for these operators as well. The easiest way to define these operators is with the stream differential equations. The *zip* operator $Z_k : (\mathbb{Q}^\omega)^k \to \mathbb{Q}^\omega$ is defined as:

$$\begin{aligned} Z_k(\sigma_0, \ldots, \sigma_{k-1})(0) &= \sigma_0(0), \\ Z_k(\sigma_0, \ldots, \sigma_{k-1})' &= Z_k(\sigma_1, \ldots, \sigma_k - 1, \sigma_0'). \end{aligned}$$

Note that σ_0, σ_{k-1} are streams and not elements of streams. An example for the zip operator would be $Z_3(\sigma, \tau, \rho) = (\sigma(0), \tau(0), \rho(0), \sigma(1), \tau(1), \rho(1), \ldots)$. In this definition we need to define a k to indicate which version of the zip we want. This is similar to how we have defined the merger node in the previous chapter, therefore the Z_2 operator defines the merger node. Of course a $Z - n$ will define a merger node with n input.

With the merger done we will switch to the splitter. This node is more complicated to define as there are multiple outgoing arrows and we therefore need multiple equations in order to define the node. Assume that we have a splitter with σ as an input and $\tau_0, \ldots, \tau_{k-1}$ as outputs. The equation for an output is: $\tau_n = T_k^n(\sigma)$, with $0 \leq n \leq k - 1$. This means that the n-th arrow will get the n-th value from the input stream σ, which is the behaviour we want.

So far we have shown how we can solve rational streams as well as give the equations for solving asynchronous streams. Unfortunately it turns out that adding asynchronous nodes can make a circuit non-rational. While we are able to simulate these streams without any problems, it is not easy to convert such a stream into a formula. We call it a formula now as because of the asynchronous nodes we no longer will always get a fraction of polynomial streams as we got previously. We refer the interested reader to [8, Section 8] which contains a detailed example of such a circuit. The conclusion of this example is that it is currently unknown how to solve these non-rational circuits automatically.

7 | Implementation

For the implementation we first had to make a choice on what programming language to use. The first option is Java, for which we considered two variants the first was “pure” Java, the second was writing a plug-in for Eclipse with the help of the the Graphical Modeling Language (GML). We discarded the latter quite early as we first of all preferred a stand alone tool instead of a plug-in and secondly we found GML poorly documented which made it difficult to understand as we never used it before. Because we have more experience in C++ we preferred this language. This eventually brought us to the Qt framework [4]. Qt is a graphical framework currently developed by the Finnish company Digia and was previously developed by Nokia.

In this chapter we will discuss to details to the implementation and explain what TASCI can do. We will explain our choices and show difficulties and how we overcome them. This chapter also contains short pieces of pseudocode to accompany our discussion.

7.1 Synchronous Stream Circuits and their Simulation

Before we can even start programming both the syntax and semantics from simulating stream circuits we first need a GUI. For the GUI we use an example from Qt itself called Diagram Scene example [3], instead of starting from scratch. This example contains functionality for both placing objects (nodes) and arrows to connect the objects, a good starting point. Note that there was no other functionality asides for the placing these objects and arrows. Like the previous chapters we use the general components in the implementation i.e., we use an adder with $n \geq 2$ inputs etc.

The first thing to do is to add all the nodes to the program. This is not very difficult as it is just a job of adding the graphical representation. We allow the user to have multiple input and output nodes in there circuits. This allows for simulation of multiple circuits at the same time as well as the ability to produce more complex circuits. More challenging is to implement the semantics. The starting point for the semantics, before even doing simulation, is to make sure we have a valid stream circuit. This includes that the nodes have the correct number of inputs and outputs. For example a register cannot have two inputs for instance. To do this we need to check if an arrow can be connected if a user draws it. While the user is still drawing his circuit we can check if an arrow can be drawn, pseudocode for doing this is shown in Listing 7.1.

```
if(itemType == buffer){
    if(newArrow == incoming && buffer.numIncoming == 1)
        return false;
    else if (newArrow == outgoing && buffer.numOutgoing == 1)
        return false;
    else
        placeArrow();
}
```

Listing 7.1: Pseudocode for checking if arrows can be placed.

In the tool we do not restrict the user in drawing his circuit, however before we start the simulation we do check the diagram for correctness. The first thing we check then if the nodes have the correct number of arrows. For example a register with only an incoming arrow and no outgoing arrow does not function and needs to be fixed. The same holds for an adder with only one input. This check will look similar to the code in Listing 7.1. Moreover we need to check in case a feedback loop is present in the circuit if a register occurs in this loop. Pseudocode for finding a loop without registers is given in Listing 7.2. Note that we use `buffer` instead of register in the pseudocode as register is a keyword in C++.

What we do is we take a node in the circuit and follow the arrows. This means we either find the output and fall back in the recursion or we find ourself again. If we found ourself again we have a feedback loop and need to check if we have passed a register. We can do this easily because we kept our history in a list. If we have a register in the loop we are fine and fall back in the recursion. Otherwise we can flag this circuit as invalid via the bool `foundLoopWithoutBuffer`.

```
bool foundLoopWithoutBuffer = false;
void loopCheck{
    foreach (node in nodes){
        if(!foundLoopWithoutBuffer){
            list nodeList;
            followArrows(node, nodeList);
        }
        else
            return;
}

void followArrows(node, nodeList){
    if (nodeList.contains(node) && !nodeList.contains(buffer)){
        foundloopWithoutBuffer = true;
        return;
    }
    else if (nodeList.contains(node) && nodeList.contains(buffer))
        return;
    else{
        nodeList.append(node);
        foreach(arrow in node.arrows){
            followArrows(node, nodeList);
        }
    }
    return;
}
```

Listing 7.2: Pseudocode for checking if a feedback loop is present without a register.

After the circuit is checked we can start its simulation. Before we will explain the simulation we need to elaborate on certain points. First of all because the implementation also has to work with asynchronous circuits the arrows behave as unbounded FIFO queues and will therefore also be used in the synchronous implementation even though this is not necessary. Secondly as stated the circuit behaves synchronously, however as this is not possible to achieve in an implementation we do the simulation atomically.

The starting point for the animation usually will be the input for the circuit. However as shown in Chapter 2 not every circuit has a input and works as a closed circuit. Even if a circuit has an input we sometimes need to "kickstart" one or multiple registers. This is most often the case if that circuit contains a feedback loop. The way we have designed the algorithm is shown in Listing 7.3, which contains the main loop of this algorithm.

```
while(!allUpdated()){
    foreach(node in nodes){
        animation.handleNode(node);
    }
}
```

Listing 7.3: Pseudocode for the main loop of the simulation of a stream circuit.

The loop in the listing above works as follows. The function `allUpdated()` checks whether all nodes in the circuit are updated. Here updated means that it has received its required input and passed the output on. If they are all up-to-date we stop and are done, if not we go to loop through all nodes in the circuit. Every node will then be handled by a function `handleNode()`. The functionality of this function is fairly simple to explain. If a given node's incoming arrows all have values present we take these values, in synchronous circuits there will be at most one value, and do the corresponding operand on them. The odd one out is the register. As stated previously the register may need to be “kickstarted”. This means that even if no input is present it will still place the value that is currently in the register on its outgoing arrow.

7.2 Conversion to and from Stream Polynomials

Having a method of drawing stream circuit via a drag-and-drop system is good and useful. However, it is considerably faster and convenient to be able to input a fraction of stream polynomials and immediately get the equivalent stream circuit. Before we can begin with the actual conversion we first need to read the user’s input and convert it into something that is usable for us. The main information we need are the coefficients and the position of the coefficients. The reason we need the coefficients is that these are the values that are placed on the multipliers as shown in Section 4.3. Therefore we will convert a user’s input into a list of coefficients. For example if the user would input $4 + 3X - X^3 + 7X^5$ we convert that to a list $[4, 3, 0, -1, 0, 7]$. We require two of these list if we have a fraction of polynomial streams, one for the numerator and one for the denominator.

Given the lists we can then start building the stream circuit. The initial thing to do is to make sure that the first coefficient of the denominator is 1 and to divide all other coefficients if necessary. Of course if that value is 0 we have an error. Building the circuit itself is according to the algorithm described in Section 4.3.

The user can input multiple polynomials at the same time in order to either solve many at once, or even combine all their output together and get a new result. The polynomials come in two variants, open or closed. The advantage of a closed circuit is that it is independent of an input. In this case the “input” will always be the stream $[1] = (1, 0, 0, 0, \ldots)$. The small circuit that will produce this stream is shown in Figure 7.1. The main difference being that in our case we will not use the output, but that is the point where our “real” circuit starts. The open circuit uses an input instead which can of course be set to [1].

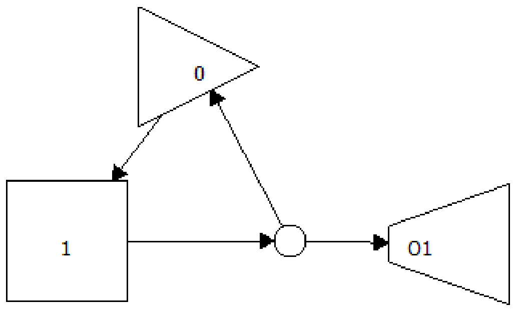
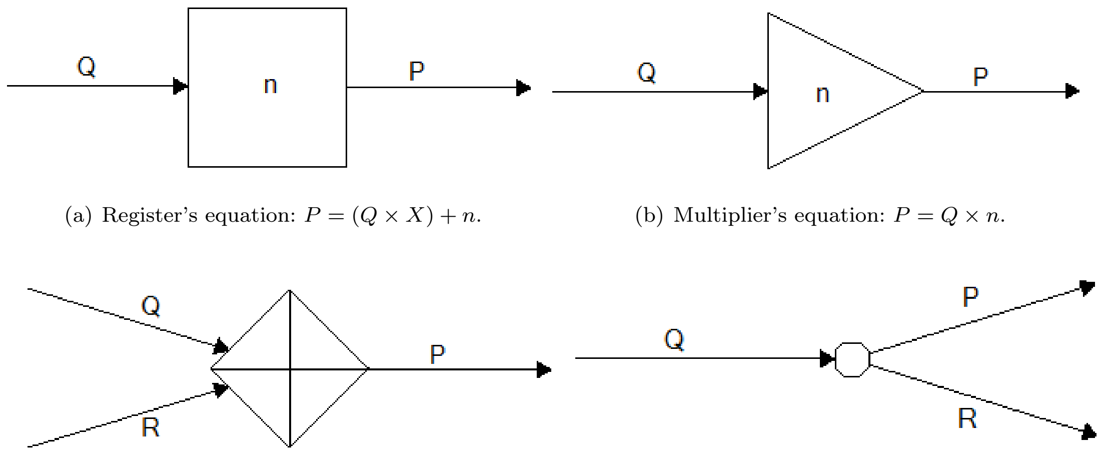


Figure 7.1: A simple loop for producing the stream $[1] = (1, 0, 0, 0, \ldots)$.

The method for doing the conversion the other way around is shown in Section 4.2. The first thing to do is to get the set of equations from the circuit which we need to solve. If we look at Figures 4.1 and 4.2 we can see that the points where we will produce the equations are the arrows in the circuit. Therefore the first thing we have to do is name each arrow which is done like the in the figures. The input and output streams are always marked as their provided name.

We will call the side of the arrow at the arrow’s head the *front* end, the opposite end we call the *back* end. To get the actual equations we need to loop over all the arrows and for every arrow we check the node that it is connected to at its back end. That node will decide what the equation will be. An overview of the synchronous nodes’ equation is found in Figure 7.2.

Given a set of equations for a certain circuit we still need to solve this for all the outputs. To solve these equations automatically we use MATLAB [6]. MATLAB has a symbolic toolbox which allows use to work with sets of equations with symbols instead of numbers. Via C++ we can call MATLAB and input our set of equations, when MATLAB is done we extract the result.



(a) Register's equation: $P = (Q \times X) + n$.

(b) Multiplier's equation: $P = Q \times n$.

(c) Adder's equation: $P = Q + R$.

(d) Copier's equations: $P = R$ and $Q = R$.

Figure 7.2: An overview for making the equations for all the synchronous nodes.

7.3 Asynchronous Stream Circuits and their Simulation

Before we explain the asynchronous nodes we first explain the convolution product node. This node has two internal lists. We need these lists in order to keep track of the values we have already received at the incoming arrows. At every clock cycles we pass the values coming on the arrows to their specific lists. With both the lists up-to-date we can then multiply and add all the values required according to Equation 3.2.

For the asynchronous components we require that the arrows behave as unbounded FIFO queues. This is not hard to do, the most important part is to get the interaction right when we have no output from the splitter (or Drop/Take) and how to make that properly interact with the other nodes. In the case of all nodes if we get an input of nothing (which we'll call ε) the node will also output ε. The node we will discuss in detail is the register, as we have to do backtracking if we get ε as an input. To show the algorithm we use the example in Figure 7.3.

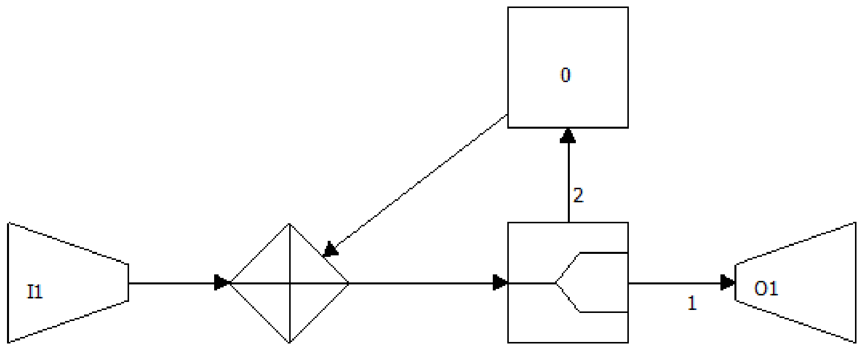


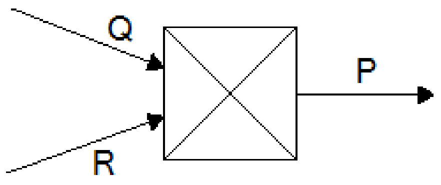
Figure 7.3: An example circuit indicating the behaviour between splitters and registers.

As an input for this circuit we use I1 $= (1, 2, 3, 4, \ldots)$. When the first value from the input node arrives add the adder we need to kickstart the register. The addition will then happen, and the output would have been passed to the output node. However because of the splitter the register will get ε as input and therefore we were not allowed to kickstart the register and we need to correct our mistake. We have taken a precaution, we have marked the value we kickstarted as being kickstarted. This tag will pass on to any results the tagged value is involved in. Therefore the result of the addition will also be tagged as well as the value passed to the output node. As soon as we get ε as an input we use a second algorithm to correct our mistake. The pseudocode for this algorithm is given in Listing 7.4.

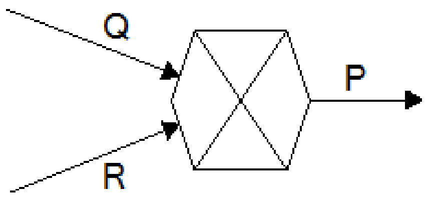
```
followArrows(node){
    node.passed = true
    foreach(arrow in node.outgoingArrows){
        if(!node.passed)
            if(arrow.node.type == outputNode)
                node.value = epsilon;
            //If we find a register let the old value stay in that register
            else if(arrow.node.type == buffer){
                node.newValue - node.oldValue;
                followArrows(arrow.node);
            }
            else
                followArrow(arrow.node);
    }
}
```

Listing 7.4: Pseudocode for correcting the circuit after incorrectly kickstarting a register.

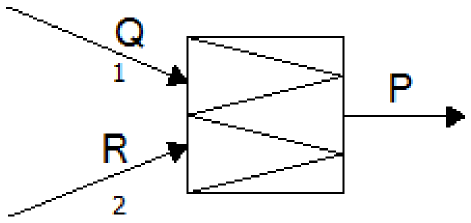
What we do in the algorithm is start at the register we have gotten ε as an input and continue following the arrows from there. None of the nodes have internal memory except for the register. This means that for almost all nodes we can just call `followArrows()` again and continue recursively. For any other register we pass we need to get the old value (which we still have) and make that the current value. If we reach an output node we must change whatever value we had there into ε.



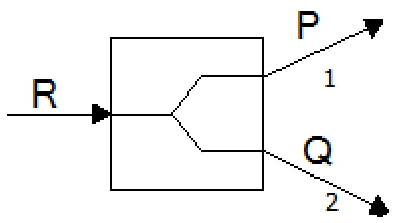
(a) Hadamard products's equation: $P = Hadamard(Q, R)$.



(b) Convolution product's equation: $P = convolution(Q, R)$.



(c) Merger's equation: $P = merge(Q, R)$.



(d) Splitter's equations: $P = split(0, 2, R)$ and $Q = split(1, 2, R)$.

Figure 7.4: An overview for making the equations for all the synchronous nodes.

7.4 Conversion from Asynchronous Circuits

Conversion from a stream circuit with asynchronous nodes is again done via MATLAB. The main thing we are going to use are so called symbolic functions. This allows us to define functions for all the nodes in the extended stream circuits. An overview for the conversion is given in Figure 7.4.

Most of these are straightforward, the splitter however requires a little bit of explanation. The first two numbers shown in the function *split*() indicate the outgoing arrow number first and secondly the total number of outgoing arrows. Note that like with the take and drop nodes we start counting from 0 and not 1. Also we do not yet have an implementation for the take and drop nodes in terms of conversion to an equation. Furthermore as explained previously we can get non-rational streams which we cannot solve automatically.

The conversion the other way around is much more complicated and a direct solution is not yet implemented. However a solution for this is present in some form. When a user wants to input a synchronous polynomial we allow the user to input more than one at the same time. These multiple synchronous circuits can then be combined with one of the following: adder, Hadamard product, convolution product, merger or splitter. This allows for the user to input two streams σ, τ and make for example $split(\sigma, \tau)$.

To summarize TASCI has the following functionality:

- It allows the user to draw circuits with 10 different kinds of nodes.
- Any, well formed, circuit can be simulated.
- Rational streams can be transformed into an equivalent stream.
- A rational stream's output can be directly calculated.
- Rational circuits can be transformed into an equivalent rational stream.

8 | Conclusion and Discussion

In this thesis we have presented an implementation which can simulate stream circuits designed by the user. This is done in order to solve the problem that stream circuits are difficult to understand. Simulating a stream circuit makes it possible to easily get the output for any circuit. Furthermore most circuits can be converted into an equivalent mathematical form. This mathematical structure is sometimes easier to solve than the actual circuit. We have also presented the first tool that allows for the manipulation of asynchronous circuits.

We have decided to extend on the known stream circuits by adding additional nodes. These nodes allow for additional functionality as well as make certain streams easier to make. Of course we could have added many more nodes as long as we can define its behaviour. The question then arises what nodes to add and which node can be ignored.

TASCI is powerful and all its functionality has been explained in the previous chapters. It is easy to use, although the user is required to at least have a basic understanding of how stream circuits work. Because TASCI is programming in Qt we can distribute it on multiple operating systems, this is an advantage.

The entire program contains 12 classes, these classes (only the `.cpp` files) contain a total of ∼4500 lines of code, ∼3500 of these have been written by us. Adding the header (`.h`) files will add another ∼750 lines of code. The lines of code with which we started mostly contained the GUI as well as an option to connect figures. This means all of the functionality has been written by us while most of the GUI came given.

There are of course limitations to TASCI's functionality. First of all we cannot convert non-rational circuits to an equivalent mathematical form automatically. This is unfortunate but there is currently no known method of doing this automatically. Secondly we do not allow a user to input a stream with split and merge functions. We have however allow the user to input multiple rational streams and combine them using these functions.

8.1 Future Work

A key thing to do in future is to change the tool to not use MATLAB anymore. While MATLAB is very powerful and definitely has provided the functionality we wanted from it there is one issue and that is that it is an expensive piece of software. While it is commonly used in the field of computer science we ideally have TASCI to be stand alone without any third party software.

Furthermore more theoretical research needs to be done in the area of non-rational streams in order to make their mathematical structure more clear. If this can be achieved then we can extend our application with them in order to make it more complete.

A | How to use the Software

This appendix contains a detailed explaination on how to use the tool. First we refer to Figure A.2 in which we have marked several parts of the tool and we will explain them here.

1. This button is the delete button. It will remove an arrow or node from the scene. One may also press the `delete`-key on the keyboard to remove an arrow or node.
2. The buttons in group 2 allow the user to alter the fontsize and style, as well as change the colour of the text and lines drawn.
3. These are the main functionality buttons, we explain them in detail a little bit later.
4. In the toolbox on the left the nodes for creating a stream circuit are present.
5. The main scene where to the user can drag and drop the nodes from the toolbox.
6. If the user decided to input a polynomial stream to draw or solve the inputted polynomial will be shown here.
7. In this textbox we show the values for all the input nodes in the circuit. The values that are bold have been inputted into the circuit the others are still waiting in the queue.
8. This textbox shows the values that have been output for each of the output nodes.

In Figure A.1 the buttons containing the main functionality of the tool are presented. The functionality for each of these buttons are (from left to right):

- The first two buttons allow the user to switch between the mouse cursor, which allows for placing and moving items, and a pointer which allows the user to draw arrows between nodes.
- Next is a dropdown box which can be used to change the scale of the circuit.
- The green arrow will do a single clockcycle and will let the diagram behave accordingly.
- The double green arrow allows the user to do n clockcycles in quick succession.
- Finally in terms of animation we have the third arrow which will do as many clockcycles as needed untill an output is produced on the output node. This is usefull especially for circuit with a splitter.
- The blue cross will clear the outputs from the textbox.
- The red cross will remove all nodes and arrows, to get a clean slate again.
- $\frac{ax}{bx}$ allows the user to input a fraction of polynomials and the corresponding circuit will then be drawn.
- $\frac{ax}{bx} \overset{?}{=}$ allows the user to input a fraction of polynomials, which will then be solved. No circuit is drawn.

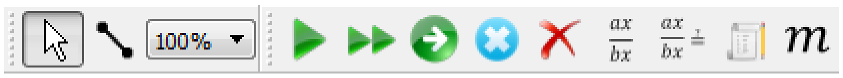


Figure A.1: The buttons used for the main functions in the tool.

- The scroll symbol will take the current circuit and produce the corresponding polynomial.
- Finally m allows the user to produce the Moessner numbers. See Section B.2 for the details.

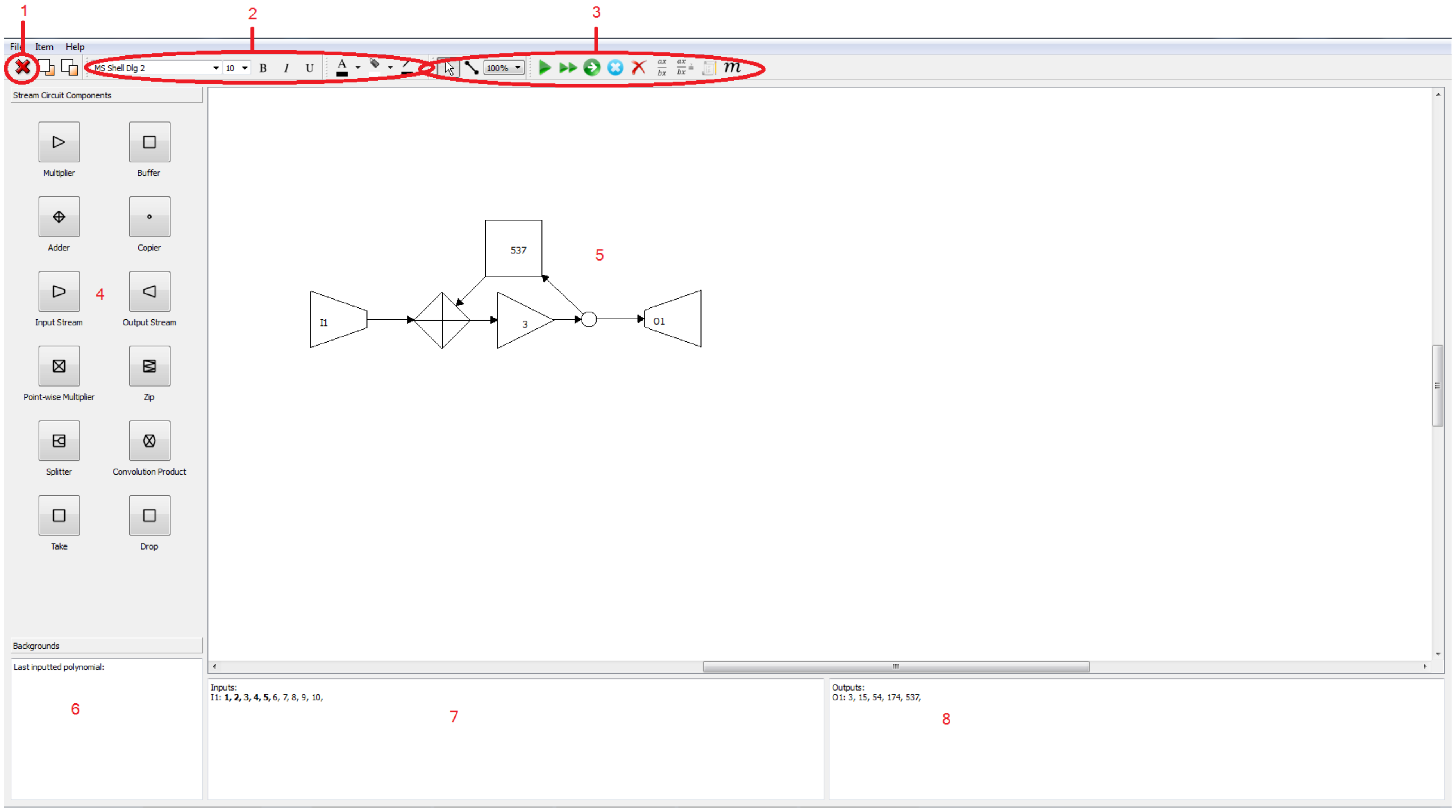
When drawing arrows between nodes move the cursor from the starting node to the end node, in order to get the arrow pointing in the right direction. When inputting registers, multipliers etc. the user will be asked what value it needs to start at. One can always manually change this by right clicking on the node and selecting `chenge value`. This method also works for changing the values for the input node. Like the values for the input and output nodes one can also change the name in a similar fashion, only now click `change name`.

Next we explain the method for inputting fraction of polynomial streams. The user will always be prevented with two input boxes, in the top one put the numerator and in the bottom one put the denominator. As an example we use the fraction:

$$\frac{4 + 3X + \frac{1}{3}X^3 - 2X^4}{-2 - 9X + X^2 + 2X^4}.$$

The input for the numerator would then be `4+3x+1/3x^3-2x^4` and for the denominator it would be `-2-9x+x^2+2x^4`. Note that we do not use a capital `X` but a small `x`. Also be carefull on certain operating systems it may change `x^2` into $\mathtt{x}^2$. Be sure to correct this as this causes the program to fail, one can simply press space before the number.

TASCI is available for download from the following link:
`https://www.dropbox.com/sh/1odsm5iy4gg01h1/AAC-wVYJ37j5g_YQ2lPp8jvka`
In the folder named "release" you can find an executable called `SFGTool.exe`, executing this file will launch TASCI. Currently only a Windows version is available.



Last inputted polynomial:

Inputs:
I1: **1, 2, 3, 4, 5,** 6, 7, 8, 9, 10,

Outputs:
O1: 3, 15, 54, 174, 537,

Figure A.2: A screenshot of the created tool with marked sections.

B | Examples

B.1 Example Streams

In this appendix we present several polynomial streams which can be used to check the results from the tool. The first polynomial stream will be worked out in detail while the others are left to the reader. All the results are based on the input stream $[1] = (1, 0, 0, 0, \ldots)$. The first fraction of polynomial streams is $\frac{X}{1-X-X^2}$. The result signal flow graph generated by PolySFG for this fracion of polynomial streams is shown in Figure B.1.

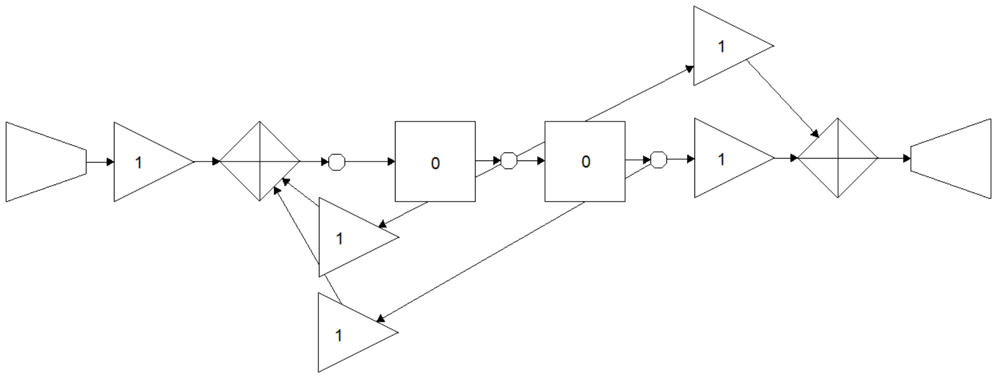


Figure B.1: The automatically generated SFG for the fraction of polynomial streams $\frac{X}{1-X-X^2}$

The corresponding output for the fraction of polynomial streams $\frac{X}{1-X-X^2}$ with input $1, 0, 0, 0, \ldots$ is $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ or simply the Fibonacci numbers. The following fractions of polynomial streams and their results can also be used to check the tool's output.

- $\dfrac{r}{1-X} = r, r, r, \ldots$
- $\dfrac{1}{1-2X+X^2} = 1, 2, 3, 4, \ldots$
- $\dfrac{1}{1-X^2} = 1, 0, 1, 0, 1, 0, \ldots$
- $\dfrac{1}{1+X} = 1, -1, 1, -1, 1, -1, \ldots$
- $\dfrac{1}{1-rX} = 1, r, r^2, r^3, \ldots$
- $\dfrac{1}{1+X^2} = 1, 0, -1, 0, 1, -1, 0, 1, \ldots$
- $\dfrac{3-8X}{1-7X+10X^2} = 3, 13, 91, \ldots$

To calculate the elements from the last fraction of polynomial streams we use the formula $\frac{2}{3}2^n + \frac{7}{3}5^n$ to calculate the n-th value.

One thing that can be done to check correctness is to input a rational stream convert it to its circuit and then convert it back to a stream again. This will allows the user to check if TASCI is wroking properly. Note that when doing this the result is most likely in a different form. For example the result will have the highest power in front instead of at the end.

B.2 Moessner's theorem

Moessner's theorem [7] is a method for calculating the power (k) of all integers without using multiplication. The theorem is given below.

Theorem B.2.1 *(Moessner). Assume $k > 0$. Start from the infinite sequence of ones: $(1, 1, \ldots)$, drop every $k + 1$st element, and form the sequence of partial sums of the resulting subsequence. Next drop every kth element and form the sequence of partial sums again. Next drop every $k - 1$st element and form the sequence of partial sums again. Repeating this procedure k times one obtains the sequence of the kth powers of the integers.*

As an example we will explain the case whre $k = 3$, i.e., all integers to the power 3.

$(1, 1, 1, 1, \ldots)$	
$(1, 1, 1, 1, \ldots)$	every 4th element is dropped,
$(1, 2, 3, 4, \ldots)$	partial sums of the above stream,
$(1, 2, 4, 5, \ldots)$	every 3rd element is dropped,
$(1, 3, 7, 12, \ldots)$	partial sums of the above stream,
$(1, 7, 19, 37, \ldots)$	every 2nd element is dropped,
$(1, 8, 27, 64, \ldots)$	partial sums of the above stream,

Moessner's theorem can also be implemented using stream circuits. To do the we first need a simple circuit to calculate the partial sum. This circuit is shown in Figure B.2. The other part we need is a drop operator. We already have a node for this in the drop node. We can cut a corner in Moessner's theorem by starting with the natural numbers instead of a stream of only ones. In the previous section we have shown a fraction of stream polynomials which can do this. Now it is a simple case of attaching the drop operators (with the correct period and block size) to the partial sums.

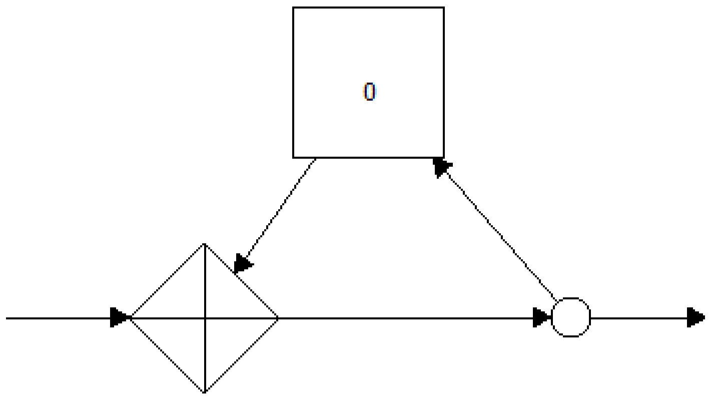


Figure B.2: A simple circuit for calculating the partial sum.

C | Code

This chapter contains part of the code for the tool. We did not provide all code, but made a selection of all the code we have.

C.1 Animation

Not all the nodes are present in the listing below. We have removed the Hadamard product (as it is similar to the adder), splitter (as it is simialr to the merger) and take (as it is similar to the drop).

```
// Updates all possible items in the diagram.
void Animation::handleItem(DiagramItem *item){
    if(item->myDiagramType == DiagramItem::InputStream && !item->valueUpdated){
        foreach(Arrow *arrow, item->arrows){
            if(!item->passedValue){
                arrow->queue.append(item->myValue);
                item->passedValue = true;
            }
        }
        item->valueUpdated = true;
    }
    else if(item->myDiagramType == DiagramItem::Buffer && !item->valueUpdated){
        foreach(Arrow *arrow, item->arrows){
            if(arrow->endItem() != item && !item->passedValue){
                item->myValue.fromBuffer = true;
                arrow->queue.append(item->myValue);
                item->passedValue = true;
            }
            else if (arrow->endItem() == item && !arrow->queue.isEmpty()){
                if(arrow->queue.first().epsilon == false){
                    item->myCalculation = arrow->queue.takeFirst();
                    item->valueUpdated = true;
                }
                else{
                    arrow->queue.removeFirst();
                    item->myCalculation = item->myValue;
                    followArrows(item);
                    item->valueUpdated = true;
                }
            }
        }
    }

    else if(item->myDiagramType == DiagramItem::Copier && !item->valueUpdated){
        bool available = false;
        foreach(Arrow *arrow, item->arrows){
            if(arrow->endItem() == item && !arrow->queue.isEmpty()){
                available = true;
                item->myValue = arrow->queue.takeFirst();
            }
        }
        if(available){
            foreach(Arrow *arrow, item->arrows){
                if(arrow->endItem() != item)
                arrow->queue.append(item->myValue);
            }
            item->valueUpdated = true;
        }
    }

    else if(item->myDiagramType == DiagramItem::Multiplier && !item->valueUpdated){
        bool available = false;
        foreach(Arrow *arrow, item->arrows){
            if(arrow->endItem() == item && !arrow->queue.isEmpty()){
                available = true;
                item->myCalculation = arrow->queue.takeFirst();
            }
        }
        if(available){
            foreach(Arrow *arrow, item->arrows){
                if(arrow->endItem() != item){
                    if(item->myCalculation.epsilon == true){
                        fraction f;
                        f.set(0);
                        f.epsilon = true;
                        arrow->queue.append(f);
                    }
                    else{
                        arrow->queue.append(item->myValue * item->myCalculation);
                        if(item->myValue.fromBuffer == true)
                            arrow->queue[0].fromBuffer = true;
                    }
                }
            }
            item->valueUpdated = true;
        }
    }

    else if(item->myDiagramType == DiagramItem::Adder && !item->valueUpdated){
        bool available = true;
        item->myValue.set(0);
        Arrow* myArrow;
        foreach(Arrow *arrow, item->arrows){
            if(arrow->endItem() == item && arrow->queue.isEmpty())
                available = false;
            else if(arrow->endItem() != item)
                myArrow = arrow;
        }

        if(available){
            if(findEpsilon(item)){
                fraction f;
                f.set(0);
                f.epsilon = true;
                myArrow->queue.append(f);
                item->valueUpdated = true;
            }
            else{
                foreach(Arrow *arrow, item->arrows){
                    if(arrow->endItem() == item){
                        if(arrow->queue.first().fromBuffer == true)
                            item->myValue.fromBuffer = true;
                        item->myValue.sum(arrow->queue.takeFirst());
                    }
                }
                myArrow->queue.append(item->myValue);
                item->valueUpdated = true;
            }
        }
    }

    else if(item->myDiagramType == DiagramItem::Product && !item->valueUpdated){
        bool available = true;
        item->myValue.set(1);
        Arrow* myArrow;
        foreach(Arrow *arrow, item->arrows){
            if(arrow->endItem() == item && arrow->queue.isEmpty())
                available = false;
            else if(arrow->endItem() != item)
                myArrow = arrow;
        }
        if(available){
            if(findEpsilon(item)){
                fraction f;
                f.set(0);
                f.epsilon = true;
                myArrow->queue.append(f);
                item->valueUpdated = true;
            }
            else{
                bool first = true;
                foreach(Arrow *arrow, item->arrows){
                    if(arrow->endItem() == item && first){
                        item->L1.append(arrow->queue.takeFirst());
                        first = false;
                    }
                    else if (arrow->endItem() == item && !first)
                        item->L2.append(arrow->queue.takeFirst());
                }
                fraction result;
                result.set(calcProduct(item->L1,item->L2));
                myArrow->queue.append(result);
                item->valueUpdated = true;
            }
        }

    }

    else if(item->myDiagramType == DiagramItem::Zip && !item->valueUpdated){
        //First time we handle the zip.
        if(item->currentArrow == -1){
            Arrow *myArrow;
            //First idenitify the single outgoing arrow, which we need to ignore.
            for(int i = 0; i < item->arrows.size(); i++){
                if(item->arrows.at(i)->startItem() == item){
                    item->outgoing = i;
                }
            }
            if(item->outgoing != 0){
                myArrow = item->arrows[0];
                item->currentArrow = 0;
            }
            else{
                myArrow = item->arrows[1];
                item->currentArrow = 1;
            }

            if(!myArrow->queue.isEmpty()){
                item->arrows[item->outgoing]->queue.append(myArrow->queue.takeFirst());
                item->valueUpdated = true;
                if(item->arrows[item->outgoing]->queue.first().epsilon == false){
                    if(item->currentArrow + 1 != item->outgoing &&
                        item->currentArrow + 1 < item->arrows.size())
                        item->currentArrow++;
                    else if(item->currentArrow + 1 == item->outgoing &&
                        item->currentArrow + 2 < item->arrows.size())
                        item->currentArrow = item->currentArrow + 2;
                    else if(item->currentArrow + 1 >= item->arrows.size() &&
                        item->outgoing != 0)
                        item->currentArrow = 0;
                    else if(item->currentArrow + 1 == item->outgoing &&
                        item->currentArrow + 2 >= item->arrows.size())
                        item->currentArrow = 0;
                    else
                        item->currentArrow = 1;
                }
            }
        }
        //All other times.
        else{
            Arrow* myArrow = item->arrows.at(item->currentArrow);
            if(!myArrow->queue.isEmpty()){
                item->arrows[item->outgoing]->queue.append(myArrow->queue.takeFirst());
                item->valueUpdated = true;
                if(item->arrows[item->outgoing]->queue.first().epsilon == false){
                    if(item->currentArrow + 1 != item->outgoing &&
                        item->currentArrow + 1 < item->arrows.size())
                        item->currentArrow++;
                    else if(item->currentArrow + 1 == item->outgoing &&
                        item->currentArrow + 2 < item->arrows.size())
                        item->currentArrow = item->currentArrow + 2;
                    else if(item->currentArrow + 1 >= item->arrows.size() &&
                        item->outgoing != 0)
                        item->currentArrow = 0;
                    else if(item->currentArrow + 1 == item->outgoing &&
                        item->currentArrow + 2 >= item->arrows.size())
                        item->currentArrow = 0;
                    else
                        item->currentArrow = 1;
                }
            }
        }
    }

    else if (item->myDiagramType == DiagramItem::Drop && !item->valueUpdated){
        Arrow* in, *out;
        foreach(Arrow* arrow, item->arrows){
            if(arrow->endItem() == item)
                in = arrow;
            else
                out = arrow;
        }
        if(!in->queue.isEmpty()){
            if(in->queue.first().epsilon == true)
                out->queue.append(in->queue.takeFirst());
            else{
                if(item->currrentBlocksize == item->blocksize){
                    in->queue.removeFirst();
                    fraction mock;
                    mock.set(1);
                    mock.epsilon = true;
                    out->queue.append(mock);
                    item->currrentBlocksize++;
                }
                else{
                    if(in->queue.first().epsilon == true)
                        out->queue.append(in->queue.takeFirst());
                    else{
                        out->queue.append(in->queue.takeFirst());
                        item->currrentBlocksize++;
                    }
                }
                if(item->currrentBlocksize >= item->period)
                    item->currrentBlocksize = 0;
            }
            item->valueUpdated = true;
        }
    }

    else if(item->myDiagramType == DiagramItem::OutputStream && !item->valueUpdated){
        //The output has only a single incoming arrow and no outgoing arrows.
        foreach(Arrow *arrow, item->arrows){
            if(!arrow->queue.isEmpty()){
                item->myCalculation = arrow->queue.takeFirst();
                item->valueUpdated = true;
            }
        }
    }
}
```

C.2 Stream Calculus

```
fraction StreamCalculus::divide(QList<fraction> d, int i){
    fraction value;
    value.set(1,1);
    value.division(d[0]);
    if(i == 0){
        return value;
    }
    else{
        fraction sum;
        sum.set(0,1);
        for(int k = 0; k <= i-1; k++){
            fraction x = divide(d,k);
            fraction help = d[i-k];
            help.product(x);
            sum.sum(help);
        }
        sum.product(value);
        return sum;
    }
}

fraction StreamCalculus::solve(QList<fraction> I, QList<fraction> n,
                               QList<fraction> d, int i){
    fraction msum, sum;
    sum.set(0,1);
    if(!simpleInput(I)){
        for(int k = 0; k <= i; k++){
            msum.set(0,1);
            for(int l = 0; l <= i-k; l++){
                fraction x = divide(d,i-k-l);
                fraction help = n[l];
                help.product(x);
                msum.sum(help);
            }
            fraction hulp = I[k];
            hulp.product(msum);
            sum.sum(hulp);
        }
    }
    else{
        msum.set(0,1);
        for(int l = 0; l <= i; l++){
            fraction x = divide(d,i-l);
            fraction help = n[l];
            help.product(x);
            msum.sum(help);
        }
        sum = msum;
    }
    return sum;
}

bool StreamCalculus::simpleInput(QList<fraction> I){
    fraction one, zero;
    one.set(1);
    zero.set(0);
    if(I[0] != one){
        return false;
    }
    else{
        for(int i = 1; i < I.size(); i++){
            if(I[i] != zero)
                return false;
        }
    }
    return true;
}
```

C.3 Stream to polynomial conversion

In the function `askMatlab()` in the listing below we input four strings into MATLAB, `syms`, `solve`,`vector` and `value`. These are based on the system of equation we make in function `getEquations`. The actual conversion from the system of equations to the strings is ommited.

```
// Given a (valid) SFG we get the equations at each point in the diagram.
void ConvertToPolynomial::getEquations(DiagramScene *scene){
    initializeNames(scene);
    uitvoer.open("uitvoer.txt");

    foreach (QGraphicsItem *item, scene->items()){
        if (item->type() == Arrow::Type){
            Arrow *arrow = qgraphicsitem_cast<Arrow *>(item);
            if(arrow->startItem()->myDiagramType == DiagramItem::Buffer){
                uitvoer << arrow->name;
                uitvoer << " == " << arrow->startItem()->myValue.toString()
                    .toUtf8().constData() << " + X * ";
                uitvoer << handleBMC(arrow);
                uitvoer.put('\n');
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Multiplier){
                uitvoer << arrow->name;
                uitvoer << " == " << arrow->startItem()->myValue.toString()
                    .toStdString().c_str() << " * ";
                uitvoer << handleBMC(arrow);
                uitvoer.put('\n');
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Copier){
                uitvoer << arrow->name;
                uitvoer << " == ";
                uitvoer << handleBMC(arrow);
                uitvoer.put('\n');
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Adder){
                uitvoer << arrow->name;
                uitvoer << " == ";
                handleAdder(arrow);
                uitvoer << "\n";
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Zip){
                uitvoer << arrow->name;
                uitvoer << " == zip(";
                handleZip(arrow);
                uitvoer << ")\n";
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Splitter){
                uitvoer << arrow->name;
                uitvoer << " == split(";
                handleSplitter(arrow);
                uitvoer << ")\n";
                uitvoer << helpString1.toUtf8().constData() << "\n";
                uitvoer << helpString2.toUtf8().constData() << "\n";
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::PointMultiplier){
                uitvoer << arrow->name;
                uitvoer << " == hadamard(";
                handleZip(arrow);
                uitvoer << ")\n";
            }
            else if (arrow->startItem()->myDiagramType == DiagramItem::Product){
                uitvoer << arrow->name;
                uitvoer << " == convolution(";
                handleZip(arrow);
                uitvoer << ")\n";
            }
        }
    }
    uitvoer.close();
}

//Put our system of linear equations into Matlab for solving
void ConvertToPolynomial::askMatlab(){
    Engine *ep;
    mxArray *S = NULL;
    QMessageBox msgBox;
    // Launch Matlab, return an error if impossible.
    if (!(ep = engOpen(NULL))) {
        msgBox.setText("ERROR: Matlab could not be launched.");
        msgBox.exec();
        return;
    }

    ep = engOpen(NULL);

    engEvalString(ep, syms);
    engEvalString(ep, solve);
    engEvalString(ep, vector);

    char *t;
    char j[10];
    QString output;
    strcpy(j, "j = 1;");
    engEvalString(ep, j);
    bool failed = true;

    for(int i = 1; i <= outputs; i++){
        engEvalString(ep, value);

        S = engGetVariable(ep, "t");
        t = mxArrayToString(S);
        if(t == NULL){
            output.append("No solution for the stream circuit could be found."
                "\nThe inputted circuit is most likely non-rational.");
            failed = true;
            break;
        }
        output.append(QString::fromStdString(outputNames[i-1]) + " = " +
            QString::fromStdString(t));
        output.append('\n');

        engEvalString(ep, "j = j+1");
    }
    if(failed)
        msgBox.setText(output);
    else
        msgBox.setText("The resulting polynomial(s) is/are:\n" + output);
    msgBox.exec();

    engEvalString(ep, "quit");
}
```

Bibliography

[1] H. Basold, M.M. Bonsangue, H. Hansen, and J.J.M.M. Rutten. “Algebraic Characterisations of Linear Circuits”. In: *ranck van Breugel et al (eds.), Horizons of the mind: a tribute to Prakash Panangaden.* LNCS 8464 (2014), pp. 124–145.

[2] C.L. van Delft. *Simulating Signal Flow Graphs via Polynomials.* Leiden University. 2013.

[3] Digia. *Qt Diagram Scene example.* June 2014. URL: `http://qt-project.org/doc/qt-4.8/graphicsview-diagramscene.html`.

[4] Digia. *The Qt Project.* June 2014. URL: `http://qt-project.org/`.

[5] R. Hinze. “Concrete Stream Calculus”. In: *Journal of Functional Programming* 20 (5–6 2010), pp. 463–535.

[6] MathWorks. *MATLAB.* June 2014. URL: `http://www.mathworks.nl/products/matlab/`.

[7] A. Moessner. “Eine Bemerkung über die Potenzen der natürlichen Zahlen, Sitzungsber”. In: *Math.-Naturw. Kl. Bayer. Akad. Wiss. München* 1951 (1952), p. 29.

[8] M. Niqui and J.J.M.M. Rutten. “Stream processing coalgebraically”. In: *Science of Computer Programming* 78 (2013), pp. 2192–2215.

[9] K. Ogata. *Modern Control Engineering.* 4th ed. Sec 3-9: Signal Flow Graphs. Prentice Hall, 2002.

[10] J.J.M.M. Rutten. “A coinductive calculus of streams”. In: *Math. Struct. in Comp. Science* 15 (2005), pp. 93–147.

[11] J.J.M.M Rutten. “A tutorial on coinductive stream calculus and signal flow graphs”. In: *Theoret. Comput. Sci.* 343 (2005), pp. 1–53.

[12] J.J.M.M Rutten. “An Application of Stream Calculus to Signal Flow Graphs”. In: *Formal Methods for Components and Objects, Lecture Notes in Computer Science* 3188 (2004), pp. 276–291.

[13] J.J.M.M Rutten. “Behavioural differential equations: a coinductive calculus of streams, automata, and power series”. In: *Theoret. Comput. Sci.* 308 (2003), pp. 1–53.