Universiteit Leiden
Opleiding Informatica

Constructing Monitors for Reo Circuits

Name: Bart Hijmans

Date: 05/02/2019

1st supervisor: Marcello Bonsangue
2nd supervisor: Farhad Arbab

MASTER'S THESIS

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

Abstract

In this thesis we discuss the construction of monitors for the Reo language. The monitors are specified in a new type of expression we call Reo expressions, that are based on regular expressions. Several algorithms that build automata for regular expressions are extended and modified to build Reo automata for Reo expressions, which can be used to monitor implementations of Reo circuits.

1 Introduction

The Reo coordination language is a visual programming language designed to glue together other pieces of software and hardware, and coordinate the communication and other interactions among those pieces. Reo programs, called Reo circuits, connect these pieces together using connectors that impose data and synchronization constraints, and define when communication can take place. It is vital that that communication works correctly and as specified.

The correctness of Reo circuits can be verified by input-output conformance (ioco) testing as has been done in Kokash et al. [6]. This kind of testing can discover that the circuit usually works correctly, and edge cases can be investigated. However, in a scenario where there are infinitely many possible input sequences, such as most Reo circuits, it is not possible to make a test case for all possible inputs. In order to determine if a Reo circuit works as intended, it is necessary to continue testing while it is deployed. This type of testing is done using monitors.

Monitors run in tangent with the implementation and check whether or not an execution is correct at all times. If an error is detected, the monitor may force the program to stop so the user can investigate it. In practice it is not always possible to know the instant something goes wrong. This is black box testing, so it is not possible to observe the internal behavior of an implementation and it may take some time for such an error to propagate into an error that can be detected. If an error occurs it must eventually be observable by a monitor, otherwise the circuit would be working as intended.

The problem we set out to solve in this thesis is how best to construct monitors for Reo circuits. We do so by making specifications for the Reo circuit in modified regular expressions we call Reo expressions. Then we modify two existing methods that convert regular expressions into finite automata, to turn Reo expressions into Reo automata, which can in turn be used for monitoring.

After some initial definitions in Section 2 we will spend Section 3 discussing what the specifications for monitors look like, by first defining a single action and then stringing those actions together into Reo expressions. Then, in Section 4, we will discuss several methods to build monitors out of these expressions and compare them. We will finally summarize in Section 5 and discuss future work.

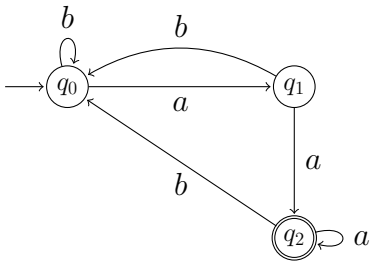


Figure 2.1: A finite automaton

2 Preliminaries

Definition 2.1. *Given a predefined set of symbols Σ, called an alphabet; a* ***string*** *is an element of Σ^*, the set of all sequences of symbols in Σ.*

A string of no symbols, also called the empty string, is denoted ϵ.

Definition 2.2. *A finite automaton is a 5-tuple consisting of*

- *Q, a finite set of states,*
- *Σ, a finite set of symbols called the alphabet,*
- *$\delta : Q \times \Sigma \to Q$, a transition function,*
- *$q_0 \in Q$ an initial state, and*
- *$F \subseteq Q$, a set of accepting states.*

These automata start in their initial state, read strings of symbols one at a time and move for each symbol from the current state to a new one as prescribed by the transition function. If the automaton is in an accepting state after reading the entire string, the string is accepted. In the finite automaton in Figure 2.1 all strings that end in 'aa' are accepted.

Definition 2.3. *Given an alphabet Σ, a regular expression denotes a set of strings over Σ and is defined recursively as follows.*

- *$\emptyset$, the empty set, is a regular expression.*
- *ϵ, denoting the set containing only the empty string, is a regular expression.*
- *$a \in \Sigma$, denoting the set containing only the string a, is a regular expression.*
- *If α and β are regular expressions, $\alpha + \beta$ is a regular expression denoting the union $\alpha \cup \beta$.*
- *If α and β are regular expressions, $\alpha \cdot \beta$ (often shortened to $\alpha\beta$) is a regular expression denoting the set of all strings xy with $x \in \alpha$ and $y \in \beta$.*
- *If α is a regular expression, α^* is a regular expression denoting the set of strings that are 0 or more elements of α in sequence.*
- *Nothing else is a regular expression.*

Definition 2.4. *Given a regular expression τ over the alphabet Σ, the language of that expression $\mathcal{L}(\tau)$ is recursively defined as follows.*

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\epsilon) = \{\epsilon\}$

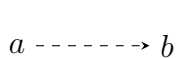
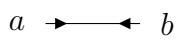
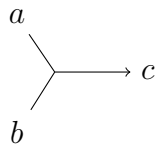
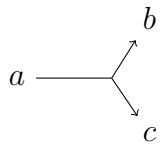
$Sync(a, b)$		Reads from a and writes to b atomically.
$LossySync(a, b)$		Same as Sync, except it can also read from a and discard the data if b cannot accept a write.
$SyncDrain(a, b)$		Reads from both a and b atomically, discarding the data from both.
$ASyncDrain(a, b)$		Reads from either a or b, but never atomically reads from both.
$Fifo1(a, b)$		Reads from a and stores the value in the buffer, then writes to b and forgets the value, resetting the channel.
$Merger(ab, c)$		Reads from a and writes to c atomically, or reads from b and writes to c atomically.
$Rep(a, bc)$		Reads from a and writes to both b and c atomically.

Table 1: Basic Reo Connectors

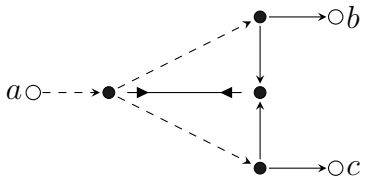


Figure 2.2: Lossy Exclusive Router

- $\mathcal{L}(a) = \{a\}, a \in \Sigma$
- $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha \cdot \beta) = \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

Where $\mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$ is the set of all strings ab with $a \in \mathcal{L}(\alpha)$ and $b \in \mathcal{L}(\beta)$.

Reo (Arbab [1]) is a programming language designed to model the communication between different software and hardware components. It is made up of a small set of basic connectors that can be glued together to create more complex connectors. Some of these basic connectors are shown in Table 1. These basic connectors are combined together through nodes, represented graphically as dots. Mergers and replicators are usually not explicitly shown. Nodes with multiple incoming and outgoing arrows are treated as a series of mergers to merge all the incoming arrows, and then a series of replicators to replicate to all outgoing arrows.

Some nodes in a Reo circuit connect to the outside world through ports. A port is essen-

tially a sync channel from an outside component to the circuit or vice versa. Data can flow through each port in one predefined direction. The ports are marked with lowercase letters $a, b, \ldots$. A port is said to *fire* when data passes through it. It can only fire if the outside component *triggers* it, and the circuit allows it. A port is considered triggered when the component tries to read from or write to the port. Often, ports can fire only if some other port also fires, or if some buffer(s) are full or empty. An example of a Reo circuit is shown in Figure 2.2. It is a lossy exclusive router that receives data from port a and sends it to either b or c, or discards it if neither b nor c can accept it.

Modelling Reo circuits and the firing of the ports has traditionally been done using a type of automaton called a constraint automaton. However here we use a different model called a Reo automaton, introduced in (Bonsangue et al. [2]), that enables us to use the information of which ports are triggered to determine whether a firing is allowed to occur. In the basic connectors described here, this is particularly relevant for the lossy sync channel. Using constraint automata as monitors it is impossible to check if b could accept a write if a fires. Reo automata can detect (or prevent) that error.

Reo automata are guarded automata, meaning that every transition has a guard; a Boolean expression that has to be true in order for a transition to be taken.

Definition 2.5. *Let Σ be an alphabet. The set of possible guards over that alphabet, $\mathcal{B}_\Sigma$ is defined to be the free Boolean algebra generated by the following grammar:*

$$g ::= a \in \Sigma | 1 | 0 | g \vee g | g \wedge g | \neg g$$

A guard g is an element of $\mathcal{B}_\Sigma$.

Definition 2.6. *Let Σ be an alphabet, and let g be an element of $\mathcal{B}_\Sigma$. Let $\alpha \in 2^{|\Sigma|}$ be a Boolean assignment containing a Boolean value a or $\neg(a)$ for each element a of Σ. α satisfying g, or $\alpha \vDash g$, is recursively defined as follows:*

$\alpha \vDash 1$	*always*
$\alpha \vDash 0$	*never*
$\alpha \vDash a$	*if* $a \in \alpha$
$\alpha \vDash x \vee y$	*if* $\alpha \vDash x$ *or* $\alpha \vDash y$
$\alpha \vDash x \wedge y$	*if* $\alpha \vDash x$ *and* $\alpha \vDash y$
$\alpha \vDash \neg x$	*if* $\alpha \nvDash x$

When g is only one symbol, we use $\bar{g}$ instead of $\neg(g)$ for brevity. In a Reo automaton a is true in α if and only if the port a is triggered.

Definition 2.7. *Given a set of ports P, a **Reo automaton** is a 4-tuple consisting of the following components:*

- *Q, a finite set of states,*
- *P, a finite set of ports,*
- *$\delta : Q \times (\mathcal{P}(P) \setminus \emptyset) \times \mathcal{B}_P \to \mathcal{P}(Q)$, a transition function, and*
- *$q_0 \in Q$ an initial state.*

Reo automata for each of the Reo connectors in Table 1 are given in Table 2. For example, the lossy sync has one state and can fire the set of ports a, b if a and b are triggered, or a if a is triggered and b is not.

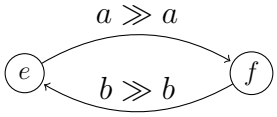
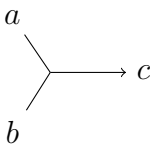
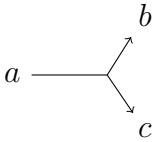
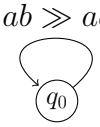
$Sync(a,b)$	$a \longrightarrow b$	$ab \gg ab$ 
$LossySync(a,b)$	$a \dashrightarrow b$	$ab \gg ab$ $a\bar{b} \gg a$ 
$SyncDrain(a,b)$	$a \rightarrow\!\!\leftarrow b$	$ab \gg ab$ 
$ASyncDrain(a,b)$	$a \rightarrow \parallel \leftarrow b$	$a \gg a$ $b \gg b$ 
$Fifo1(a,b)$	$a \longrightarrow\!\!\square\!\!\longrightarrow b$	
$Merger(ab,c)$		$ac \gg ac$ $bc \gg bc$ 
$Rep(a,bc)$		$ab \gg ab$ 

Table 2: Reo Automata for Basic Reo Connectors

$$ab \gg ab$$
$$ac \gg ac$$
$$a\bar{b}\bar{c} \gg a$$

Figure 2.3: Reo Automaton for Lossy Exclusive Router

A Reo automaton describes the behavior of a Reo circuit based on the triggered ports and the state of the automaton. They can be generated directly from a Reo circuit, and only have constraints on the ports that can fire, not the data transmitted over the Reo circuit. Reo automata have no explicit accepting states, rather they reject any string that contains a symbol that does not lead to a valid transition. Reo automata are therefore prefix-closed. A transition from q to q' with ports f and guard g can be written as follows.

$$q \xrightarrow{g \gg f} q' \iff q' \in \delta(q, f, g)$$

The transitions in a Reo automaton have a non-empty set of ports that fire and a guard over the ports that are triggered. When a set of ports S is triggered, a transition $q \xrightarrow{g \gg f} q'$ can fire only if $f \subseteq S$ and $S \vDash g$, and the automaton is in state q. There may be multiple transitions that meet those criteria, in which case one is taken nondeterministically.

A Reo automaton accepts strings of pairs, consisting of a set of ports that fire ($N \subseteq P$) and a Boolean assignment for each port that is true if and only if that port is triggered ($\alpha \in 2^{|P|}$).

Definition 2.8. *The language of a Reo automaton $M = (Q, P, \delta, q_0)$ or $\mathcal{L}(M)$ is defined to be the language of the initial state of M, $\mathcal{L}(M)q_0$. The language of a state of a Reo automaton is inductively defined as follows, where (α, N) is a pair with $N \subseteq P$ and α as an assignment of Boolean values for each $p \in P$. With $q \in Q$, $q' \in Q$, $f \subseteq P$ and $g \in B_P$.*

- $\epsilon \in \mathcal{L}(M)q$
- $(\alpha, N)x \in \mathcal{L}(M)q$ *if* $q' \in \delta(q, f, g)$ *and* $\alpha \vDash g$ *and* $x \in \mathcal{L}(M)q'$

Essentially for each symbol (α, N) in a string, a transition is taken from the current state q to a state q' if a transition $q \xrightarrow{g \gg f} q'$ exists such that $N = f$ and $\alpha \vDash g$. Then the process is repeated for the next symbol in the current state q'. If multiple transitions exist for the same symbol and one of them leads to the string being accepted and the other does not, the string is accepted. If no such transitions exist, the string is rejected.

Reo automata for Reo circuits are made by taking the product of these basic connectors to generate larger automata. The Reo automaton for the circuit in Figure 2.2 is given in Figure 2.3. For more specifics on constructing Reo automata from Reo circuits we will refer back to the Bonsangue et al. paper.

3 Reo Expressions

Before we can begin to build monitors, we need to identify which actions on the Reo-like circuit we can observe, which we do in Section 3.1. When we have an understanding of what actions the system can take, we can define a structure to dictate which actions the system is allowed to take. Section 3.2 will discuss firings and guards, which define a single such action. In Section 3.3 we will string these single actions together into what we will call Reo expressions, which can be used to model the entire behavior of a circuit insofar as we can observe it. We will also introduce operators to handle the order-agnostic and parallel nature of some Reo circuits.

Figure 3.1: Lossy Sync Channel

3.1 Monitorable Data

There are three levels of information we can sensibly look at. At the very least we need to know which ports fire. If we don't know that, there is nothing to monitor. The second level of information, the one we use here in this thesis, includes which ports are triggered. This distinction is important because of the lossy sync channel. The lossy sync channel in 3.1 is not allowed to fire just a if both a and b are triggered, and without knowing if b is triggered there is no way to catch that error if it occurs. It will also be a trivial adjustment to apply the methods in this thesis to a system where we only know the ports that fire, by simply removing all guards.

The third level of information also includes the value of any data that passes through. This would allow a monitor to make sure that the correct data is sent to the right ports at any given time. While this would be an interesting and useful addition to the methods described in this thesis, we do not explore it here. Adding data correctness to the monitor would add an extra layer of complexity, and it makes sense to leave that out for the purposes of this first attempt at building monitors for Reo. So we will leave that for future work, while focusing on the correctness of the more basic protocol here.

So for the purposes of this thesis we assume we have access to the following information at all times.

- A set of ports P
- For every firing that occurs in the application, the nonempty set $N \subseteq P$ of ports that fire.
- For every firing that occurs, the Boolean valuation $\alpha \in 2^{|P|}$ that is true for each port that was triggered, and false for each port that was not.
- The order in which the firings occur.

In essence, the monitor receives a string of pairs (α, N) and it must determine whether or not that string is accepted. It suffices to receive updates whenever a port is triggered or no longer triggered, and the full set when a firing occurs, as long as those updates are done in order. For instance we could receive "a is triggered", "$\{a\}$ fires", "a no longer triggered" and we would have the information we need.

3.2 Firings and Guards

In this section we will discuss the smallest steps a circuit can take, which we will use as literals for the expressions we make later. A firing in Reo occurs atomically, meaning every port that fires, fires simultaneously and inseparably. So our literals must be able to say a specific set of ports fires as a single inseparable block.

We also have access to which ports were triggered at the time a firing occurs. There are two ways in which this can preclude a firing. First, a firing can only occur if all ports involved in the firing are triggered. We leave whether or not to test for this up

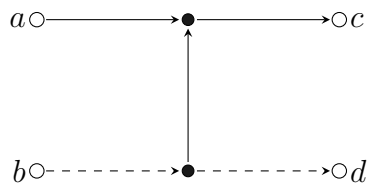


Figure 3.2: Reo Example for Section 3.2

to the implementer; it may already be enforced by the way code connects to the Reo implementation. The other way triggered ports are important is in Reo's lossy channels. In Figure 3.1 we see a very simple connector. If a is triggered and b is not, it can fire a and the data will be lost. If a and b are both triggered, it can't fire a, it can only fire ab. In the version of Reo we use here this is the only channel that causes this behavior, but the methods we use here can also be used in any expansion of Reo where one firing being triggered precludes another from firing with the right guards.

We will use the example in Figure 3.2. We will make a table of all possible permutations of ports being triggered, and all firings allowed by that permutation. The format we use is the set of ports that are triggered, followed by $\gg$ as a separator, and finally the firing that can occur.

$abcd \gg ac$	$abcd \gg bcd$	$abc\bar{d} \gg ac$	$abc\bar{d} \gg bc$
$\bar{a}bcd \gg bcd$		$\bar{a}bc\bar{d} \gg bc$	
$a\bar{b}cd \gg ac$		$a\bar{b}c\bar{d} \gg ac$	
$\bar{a}\bar{b}cd \gg -$		$\bar{a}\bar{b}c\bar{d} \gg -$	
$ab\bar{c}d \gg b$		$ab\bar{c}\bar{d} \gg b$	
$\bar{a}b\bar{c}d \gg b$		$\bar{a}b\bar{c}\bar{d} \gg b$	
$a\bar{b}\bar{c}d \gg -$		$a\bar{b}\bar{c}\bar{d} \gg -$	
$\bar{a}\bar{b}\bar{c}d \gg -$		$\bar{a}\bar{b}\bar{c}\bar{d} \gg -$	

Since firings must involve at least one port, we can remove all those that don't fire, but for now they're there to make it easier to see patterns. Observe that there are only 4 different firing sets in the table, b, ac, bc and bcd, and all of them are listed multiple times. Take a look at the four instances of b firing. In all four cases b is triggered and c is not; and there are four cases because every permutation of a and d being triggered is in there as well. Clearly it doesn't matter whether or not a and d are triggered. So instead of writing out $|P|^2$ permutations of triggered ports, or triggered sets, we will make Boolean expressions over P. These expressions are called guards and we will put them in the place of the triggered sets. They are identical to the guards used in Reo automata. The Boolean variable a (seperated from the firing set element a by being on the left of the $\gg$ symbol) is considered true if $a \in t$, and false otherwise. We will also add brackets which will be helpful when we make expressions. This leaves us with the following firings:

$$[(a \wedge b \wedge \bar{c} \wedge d) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge d) \vee (a \wedge b \wedge \bar{c} \wedge \bar{d}) \vee (\bar{a} \wedge b \wedge \bar{c} \wedge \bar{d}) \gg b]$$

$$[(a \wedge b \wedge c \wedge d) \vee (a \wedge \bar{b} \wedge c \wedge d) \vee (a \wedge b \wedge c \wedge \bar{d}) \vee (a \wedge \bar{b} \wedge c \wedge \bar{d}) \gg ac]$$

$$[(a \wedge b \wedge c \wedge \bar{d}) \vee (\bar{a} \wedge b \wedge c \wedge \bar{d}) \gg bc]$$

$$[(a \wedge b \wedge c \wedge d) \vee (\bar{a} \wedge b \wedge c \wedge d) \gg bcd]$$

We can group common terms and simplify by using the idempotency $a \vee a = a$.

$$[(a \vee \bar{a}) \wedge b \wedge \bar{c} \wedge (d \vee \bar{d}) \gg b]$$
$$[a \wedge (b \vee \bar{b}) \wedge c \wedge (d \vee \bar{d}) \gg ac]$$
$$[(a \vee \bar{a}) \wedge b \wedge c \wedge \bar{d} \gg bc]$$
$$[(a \vee \bar{a})b \wedge c \wedge d \gg bcd]$$

And we can simplify instances of $a \vee \bar{a}$ to 1 and then remove them because 1 is the identity of $\wedge$.

$$[b \wedge \bar{c} \gg b]$$
$$[a \wedge c \gg ac]$$
$$[b \wedge c \wedge \bar{d} \gg bc]$$
$$[b \wedge c \wedge d \gg bcd]$$

We can simplify these in a few additional steps. Observe that in order for a firing to occur, the ports that fire must always be triggered. Knowing that, we don't have to explicitly state it in the guards. Note that while we won't explicitly state them in guards anymore, they are still implicitly there and must be taken into account when performing operations on the guards.

$$[\bar{c} \gg b]$$
$$[\gg ac]$$
$$[\bar{d} \gg bc]$$
$$[\gg bcd]$$

Now that we have made those guards implicit and with the knowledge that a firing set can't be empty, we can remove the $\gg$ symbol if the firing doesn't have an explicit guard anymore. Also if there is only one port in the firing set, we can remove the brackets as well ($[\gg a]$ becomes a).

$$[\bar{c} \gg b]$$
$$[ac]$$
$$[\bar{d} \gg bc]$$
$$[bcd]$$

Now we will formally define what we made here.

Definition 3.1. *Let P be a set of ports. A **firing set** is a nonempty subset of P.*

Definition 3.2. *Let P be a set of ports, and $\mathcal{B}_P$ be the set of possible guards over P. A **firing** is a pair (g, f) with $g \in \mathcal{B}_P$ a guard, and f a firing set.*

We will continue writing firings as $[g \gg f]$. We will now define an operator to combine firings that will be used in later sections.

Definition 3.3. *Let $t = (g, f)$ be a firing. t is **satisfiable** if and only if $f \vDash g$.*

The guard always includes an *and* over the firing set. So if $f \nvDash g$ there are two options. The first is that g is unsatisfiable. If g contains a subexpression like $a \wedge \bar{a}$ it can never hold. The second is that some additional port or ports must be triggered in order to satisfy g. However, there is no mechanism in Reo where a port being triggered could enable a firing on other ports. That port can only block that firing by enabling some other firing through the lossy sync or a similar channel. So a guard like this can't be implemented in Reo, and a specification including it is not a valid specification for a Reo circuit. We assume here that all specifications are valid, and conclude that a firing that is not satisfiable can never fire.

Definition 3.4. *Let $t_1 = (g_1, f_1)$ and $t_2 = (g_2, f_2)$ be two firings from the set of all firings F. Their **combination** $t = t_1 \triangle t_2$ is a partial function $F \times F \rightharpoonup F$ defined as follows: $t = (g, f)$ is a firing with firing set $f = f_1 \cup f_2$ and guard $g = g_1 \wedge g_2$. It is defined only if $f_1 \cap f_2 = \emptyset$ and t is satisfiable.*

For example:

$$a \triangle b = [ab]$$
$$[\bar{a} \gg b] \triangle [\bar{c} \gg d] = [\bar{a} \wedge \bar{c} \gg bd]$$
$$[\bar{a} \gg b] \triangle a = \text{undefined}$$

Observe that in the last example the combination would be $[\bar{a} \wedge b \wedge a \gg ab]$. However, that result is unsatisfiable because $a \wedge \bar{a} = \text{false}$, so the combination is undefined. The combination operator will be used for letting two firings occur in parallel in Section 3.3.

3.3 Expressions

With a definition for firings in hand, we will now use them to define Reo expressions. We will then identify a few problems with the expressions and define new operators to improve them. A Reo expression is similar to a regular expressions, with some subtle changes and two new operators which we define here. A full definition of Reo expressions and the languages generated by Reo expressions are at the end of this section.

If we go back to our example in Figure 3.2, we can take the firings we made there and make the following expression:

$$([\bar{c} \gg b] + [ac] + [\bar{d} \gg bc] + [bcd])^*$$

For any circuit without internal storage we can simply take all possible firings, put + symbols between them, and take the Kleene star of the result. However, for circuits with

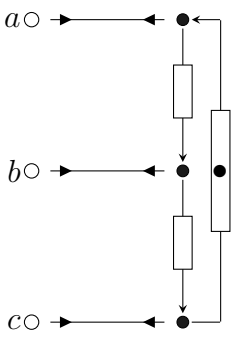


Figure 3.3: 3-way Sequencer

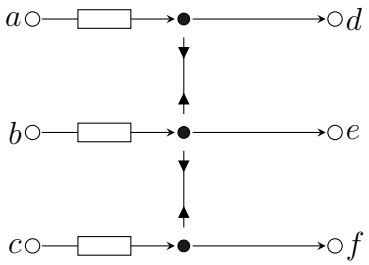


Figure 3.4: 3 Input Buffers, Synchronous Output

internal storage, some firings will be enabled or disabled by the presence of data in that storage. If we look at the sequencer in Figure 3.3, we can see that a, b and c must occur in that order before the pattern repeats. So the expression for this circuit is $([a] \cdot [b] \cdot [c])^*$, which we can shorten to $(abc)^*$.

So our expressions work well for choice and sequences. Where we start to notice some problems is in the example in Figure 3.4. It is not a very complicated circuit. It just reads inputs from a, b and c in any order and then outputs the data to d, e and f simultaneously. However, if we put that in a Reo expression we have to state every order in which it can read from a, b and c. This gives us the expression $((abc+acb+bac+bca+cab+cba)[def])^*$, or alternatively $((a(bc+cb)+b(ac+ca)+c(ab+ba))[def])^*$.

If we have two FIFO-buffers in a circuit that are completely unrelated to each other as in Figure 3.5, we have to always remember the state of both buffers in our expression. This leads to the overly complicated expression $(a(b+(c(ba)^*d)^*b+(c(ba)^*bd))+c(d+(a(dc)^*b)^*d+(a(dc)^*db)))^*$, and this example has only 4 states the buffers can be in. We're not even going to attempt to make an expression for two copies of Figure 3.4. Making monitors should not be more complicated than making an implementation. Thankfully we don't have to worry about that, because we will make the expressions much simpler by introducing the shuffle operator as discussed in (Broda et al. [3]).

Definition 3.5. *Let Σ be an alphabet, the shuffle over two strings in Σ^* is a set of*

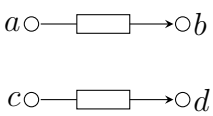


Figure 3.5: 2 Unconnected Buffers

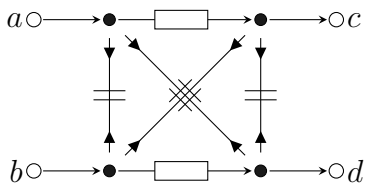


Figure 3.6: 2 Asynchronous Buffers

strings in Σ^**, and it is recursively defined as follows; with* $x, y \in \Sigma^*$ *and* $a, b \in \Sigma$*.*
$x \sqcup\!\sqcup \epsilon = \epsilon \sqcup\!\sqcup x = \{x\}$
$ax \sqcup\!\sqcup by = \{az | z \in x \sqcup\!\sqcup by\} \cup \{bz | z \in ax \sqcup\!\sqcup y\}$

This definition can be trivially expanded to Reo expressions by letting a and b refer to firings and x and y refer to Reo expressions.

Using the shuffle operator we can make much shorter and simpler expressions for the examples we discussed. An expression for Figure 3.4 is now $((a \sqcup\!\sqcup b \sqcup\!\sqcup c)[def])^*$; and for Figure 3.5 we'll only need $(ab)^* \sqcup\!\sqcup (cd)^*$. However, we are not done yet, because these expressions are not entirely correct. Reo allows for ports to fire simultaneously in some cases, when there is no overlap between them. Since the two buffers in Figure 3.5 are completely independent, if a firing occurs on the one, a firing can occur on the other at the same time. We will need another new operator for this, the parallel operator. This operator is inspired by a similar operator used in process algebra. It only works on strings of sets.

Definition 3.6. *Let* Σ *be an alphabet, and* $\mathcal{P}(\Sigma)$ *be the powerset of* Σ*. The parallel of two strings of sets in* $\mathcal{P}(\Sigma)^*$ *is a set of strings in* $\mathcal{P}(\Sigma)^*$ *that is inductively defined as follows; with* $x, y \in \mathcal{P}(\Sigma)^*$ *and* $a, b \in \mathcal{P}(\Sigma)$*.*
$x||\epsilon = \epsilon||x = \{x\}$

$$ax||by = \begin{cases} \{az | z \in x||by\} \cup \{bz | z \in ax||y\} \cup \{(a \cup b)z | z \in x||y\} & \text{if } a \cap b = \emptyset \\ \{az | z \in x||by\} \cup \{bz | z \in ax||y\} & \text{otherwise} \end{cases}$$

This can be expanded from strings of sets of alphabet symbols to firings as follows.

Definition 3.7. *Let* P *be a set of ports, and* T *be the set of all firings over* P*. The parallel of two strings of firings in* T^* *is a set of strings in* T^* *that is inductively defined as follows; with* $x, y \in T^*$ *and* $a, b \in T$*.*
$x||\epsilon = \epsilon||x = \{x\}$

$$ax||by = \begin{cases} \{az | z \in x||by\} \cup \{bz | z \in ax||y\} \cup \{(a \triangle b)z | z \in x||y\} & \text{if } a \triangle b \text{ exists} \\ \{az | z \in x||by\} \cup \{bz | z \in ax||y\} & \text{otherwise} \end{cases}$$

The parallel operator is similar to the shuffle operator, except it explicitly says two firings, a from one side and b from the other, can fire simultaneously if $a \triangle b$ exists and both a and b can fire. So the correct expressions for the two examples are $((a||b||c)[def])^*$ and $(ab)^*||(cd)^*$ respectively. We will also continue to use the shuffle operator, which may be useful in case someone wants to specifically disallow parallel behavior, such as in Figure 3.6.

Now that we have defined the new operators we can formally define Reo expressions.

Definition 3.8. *Given a set of ports P, a Reo expression denotes a set of strings of firings $t = (g_t, f_t)$ in T, the set of all firings over P. It is recursively defined as follows.*

- $\emptyset$*, the empty set, is a Reo expression*
- ϵ*, denoting the set containing only the empty string, is a Reo expression*
- $t \in T$*, denoting the set containing only the firing t, is a Reo expression.*
- *If α and β are Reo expressions, $\alpha + \beta$ is a Reo expression denoting the union $\alpha \cup \beta$.*
- *If α and β are Reo expressions, $\alpha \cdot \beta$ (often shortened to $\alpha\beta$) is a Reo expression denoting the set of all strings xy with $x \in \alpha$ and $y \in \beta$.*
- *If α is a Reo expression, α^* is a Reo expression denoting the set of strings that are 0 or more elements of α in sequence.*
- *If α and β are Reo expressions, $\alpha \sqcup\!\sqcup \beta$ is a Reo expression denoting the shuffle of α and β.*
- *If α and β are Reo expressions, $\alpha||\beta$ is a Reo expression denoting the parallel run of α and β.*
- *Nothing else is a Reo expression.*

Definition 3.9. *Given a Reo expression τ over the set of ports P. The language of τ, or $\mathcal{L}(\tau)$, is the prefix closure of $L(\tau)$ which is recursively defined as follows, where t is a firing (g_t, f_t).*

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L(t) = \{(\alpha, N) : N = f_t, \alpha \vDash g_t\}$
- $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
- $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
- $L(\alpha^*) = L(\alpha)^*$
- $L(\alpha \sqcup\!\sqcup \beta) = L(\alpha) \sqcup\!\sqcup L(\beta)$
- $L(\alpha||\beta) = L(\alpha)||L(\beta)$

Where $L(\alpha) \cdot L(\beta)$ is the set of all strings ab with $a \in L(\alpha)$ and $b \in L(\beta)$.
Where $L(\alpha) \sqcup\!\sqcup L(\beta)$ is the set of all strings $a \sqcup\!\sqcup b$ with $a \in L(\alpha)$ and $b \in L(\beta)$.
And where $L(\alpha)||L(\beta)$ is the set of all strings $a||b$ with $a \in L(\alpha)$ and $b \in L(\beta)$.

The definition of $\mathcal{L}(\tau)$ as the prefix closure of $L(\tau)$ means that for any string $xy \in L(\tau)$, $x \in \mathcal{L}(\tau)$. Any prefix of a string that is accepted by the Reo expression, is also accepted by that Reo expression.

With Reo expressions we are able to make any specification for the behavior of a Reo-like circuit. The parallel and shuffle operators allow us to do so while keeping the expressions at a reasonable size and human-readable. Now it's time to use the expressions to build monitors.

4 Building Monitors

In order to use Reo expressions to monitor an implementation, we are going to convert the expressions into automata. We will discuss two different methods. The first is a modified Thompson Construction in Section 4.1. In Section 4.2 we will discuss a method using

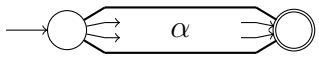


Figure 4.1: Construction Invariant Automaton

derivatives of Reo expressions. Finally, in Section 4.3 we will compare the two methods and look at why you would choose one over the other.

4.1 Improved Thompson Construction

We make two versions of this construction, the full automaton construction and the splits and joins construction.

4.1.1 Full Automaton Construction

In this section we construct Reo automata by adapting an improved version of the Thompson Construction from (Ilie and Yu [5]). This version uses an invariant that has one initial state with no incoming transitions and one accepting state with no outgoing transitions. It will be non-deterministic and contain ϵ-transitions, but the invariant limits both, because it is safe to combine initial and accepting states of different (sub)automata without unintentionally allowing incorrect behavior. Reo automata don't have accepting states, but they are used in the construction to indicate which states should be merged and are used only for that purpose. Once construction is complete any states marked as accepting will be treated as an ordinary state.

The construction is done depth-first on the parse tree of the Reo expression. For the expression $\alpha + \beta$, the automata for α and β are constructed first, and then combined into an automaton for $\alpha + \beta$. For that construction it is irrelevant what α and β do, as long as they match the invariant. So regardless of what it looks like internally, for the purposes of the next step of the construction the automaton for some subexpression α always looks like Figure 4.1.

The construction rules are given in Table 3. The steps are applied depth-first. This table does not include rules for shuffle and parallel. The visual descriptors break down when we add constructions for these operators, because we have to work with internal states.

For $\alpha \sqcup\!\sqcup \beta$ we will construct a composite of the automata α and β by making a new automaton γ. For every state a in α and b in β we create a state (a, b) in γ. For every transition from a to a' in α we create a transition from each state (a, b) to (a', b) in γ with the same transition symbol. Symmetrically, for every transition from b to b' in β we create a transition from each state (a, b) to (a, b') in γ. If the initial states of α and β are a_0 and b_0 respectively, the initial state of γ is (a_0, b_0). Similarly if the accepting states are a_f and b_f respectively, the accepting state γ will be (a_f, b_f).

Observe that since neither a_0 nor b_0 had incoming transitions, and our construction doesn't add them, (a_0, b_0), the initial state of γ, doesn't have any either. Similarly the accepting state (a_f, b_f) doesn't have any outgoing transitions. Therefore the invariant holds.

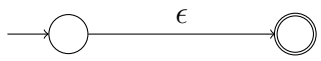
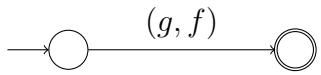
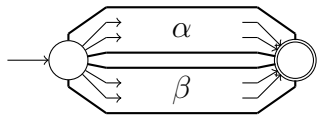
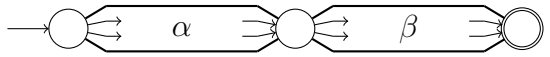
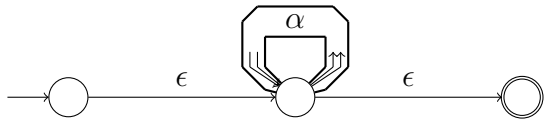
	ϵ	
	(g, f)	
Union	$\alpha + \beta$	
Concatenation	$\alpha \cdot \beta$	
Iteration	α^*	

Table 3: Improved Thompson Construction

The construction for the expression $\alpha||\beta$ is identical to the construction for $\alpha \sqcup\!\sqcup \beta$, except it adds another step. For every state (a, b) in γ and every transition t_1 from (a, b) to (a', b) and t_2 from (a, b'), if neither t_1 nor t_2 are ϵ-transitions and if $t_1 \triangle t_2$ exists, add a transition from (a, b) to (a', b') with transition symbol $t_1 \triangle t_2$.

Note that we do not combine ϵ-transitions. They can already be taken at any time, so adding combinations with ϵ-transitions would add unnecessary transitions.

Theorem 4.1. *Let E be a Reo expression, and let M be the Reo automaton generated from E by the construction.*
$\mathcal{L}(E) = \mathcal{L}(M)$.

Proof. Definition 3.9 lists the language of a Reo expression for each base case and operator. For $L(\emptyset)$ the construction makes no automaton, and so the automaton can't accept any language.
For $L(t)$ with $t = (g_t, f_t)$ the Reo expression accepts any (α, N) such that $N = f_t$ and $\alpha \vDash g_t$. After construction the transition generated from t has guard g_t and firing set f_t and originates from q_0. Per Definition 2.8 it accepts the same (α, N).
The correctness for ϵ, union, concatenation and iteration are clear from the construction.
Let E be the Reo expression $\alpha \sqcup\!\sqcup \beta$ and let M be the Reo automaton generated from E. Let x be a string accepted by α, and y be a string accepted by β. Let M_α and M_β be the automata for α and β. There must be some path of states $q_0 q_1 q_2 \ldots$ in α that accepts x with the transitions t_0 from q_0 to q_1, t_1 from q_1 to q_2 etc. In the composite automaton M the initial state is (q_0, q_β) with q_β the initial state of M_β. By construction, from any (q_x, q_β) there is a transition t_x to q_{x+1}. Processing symbols from y triggers transitions from (q_x, q_β) to (q_x, q'_β). From there a transition t_x to (q_{x+1}, q'_β) also exists by construction. So any interleaved string $x \sqcup\!\sqcup y$, where x is accepted by M_α and y by M_β, will be accepted by M.
The parallel case is identical to the shuffle case except that it allows combinations of firings $a \triangle b$. Observe that in both the expression and the automaton that is only allowed if $a \triangle b$ exists. If a combination exists, by definition the expression $ax||by$ accepts any string

$(a \triangle b)z$ where z is accepted by $x||y$. If α and β accept az_α and bz_β respectively such that $z_\alpha \triangle z_\beta = z$, then reading a from q_a, the initial state of M_α, leads to a state q'_α that accepts z_α and reading b from q_β leads to a state q'_β that accepts z_β. By construction M has a transition for $a \triangle b$ from (q_α, q_β) to (q'_α, q'_β), and inductively (q'_α, q'_β) accepts $z_\alpha \triangle z_\beta = z$.
Lastly, both Reo automata and Reo expressions are prefix-closed.
Therefore $\mathcal{L}(E) = \mathcal{L}(M)$ holds in all individual cases and inductively holds in all cases. □

There are four improvements that can be made to these automata as suggested by Ilie and Yu [5], which are repeated below. For the sake of clarity and simplicity we won't be actively using them in the construction of automata with the exception of rule (d), but we may refer to them later. Improvement (d) was modified because we can merge transitions with the same source, destination and firing set even when their guards differ.

(a) After concatenation, if the merged state p has exactly one outgoing transition $p \xrightarrow{\epsilon} q$, the states p and q can be merged and the transition removed. Otherwise if p has exactly one incoming transition $q \xrightarrow{\epsilon} p$, p and q can be merged and the transition removed.
(b) After building an iteration, if there are any loops of ϵ-transitions from the middle state back to the middle state, merge all states in those loops and remove the ϵ-transitions.
(c) After the construction is complete, if the initial state has exactly one outgoing transition labeled ϵ, remove that transition and merge the initial state with the destination of the transition.
(d) When there are multiple transitions with the same source, destination and firing set, replace them with a single transition with that source, destination and firing set and the union over the guards of the transitions.

Let $M = (Q, P, \delta, q_0$ be a Reo automaton built by the construction. In order to monitor an implementation we keep a set of states $S \subseteq Q$ that the system could be in. Initially we will add to that set the initial state, and any state reachable from there using ϵ-transitions. We can then remove any states from the set for which the only outgoing transitions are ϵ-transitions, with the exception of a state with no outgoing transitions. By construction, there is exactly one such state. Whenever we observe a firing f, we make a new set of states S'. For every $s \in S$, for every outgoing transition from s to some s' that accepts f, we add s' to S'. Then we also add every state reachable from s' using only ϵ-transitions to S', and we once again remove all states with only outgoing ϵ-transitions (that have outgoing transitions). Then we delete S and replace it with S' and continue. If at any point S is empty, we have detected an error. This is what we're monitoring for.

Observe that the shuffle and parallel operators add a lot of states and especially transitions to the automaton. That is because the shuffle and parallel operators very effectively reduce the size of Reo expressions, and we don't have a similar structure for the automata. For example, if we have a checklist with 26 points; a circuit that might be used in a system with 26 components where we have to wait for each component to be ready before we can start. We do not care about the order in which they are ready and multiple components may report ready at the same time. The expression for that system is $a||b||c||\ldots||y||z$. Each of the one-letter subautomata has 2 states. Therefore the total automaton will have 2^{26} states. Observe that from the initial state it is possible to reach every single other

state in a single transition, so there are $2^{26} - 1$ transitions from that state alone. The total number of transitions is as follows.

$$\sum_{n=0}^{26} \binom{26}{n} (2^{26-n} - 1) = 2,541,798,719,465$$

Note that n is the number of letters checked, the binomial coefficient is how many possibilities there are for having checked n letters, and $(2^{26-n} - 1)$ is all the possible combinations of letters that could still be read if n letters have already been read. Storing all those transitions would take terabytes of data. Asking someone to run something that big just to monitor their much smaller program is unreasonable. What we need is some structure on automata that serves the same function as the shuffle and parallel operators do in expressions. We will call that structure splits and joins.

4.1.2 Splits and Joins

If we don't want to break open the subautomata α and β to combine them, we have to connect them on the edges instead. When we get to the initial state of our automaton for $\alpha \sqcup\!\sqcup \beta$ we will split our attention and look at both sides simultaneously. So what we'll do is combine the initial states of α and β into a split state, essentially splitting the automaton into two automata. The automaton will be in a state in both subautomata. So it is simultaneously in the initial state of α and the initial state of β. When a firing occurs we take a transition from one of the two states to whatever state the transition leads to, and remain in the state we were in in the other.

For the parallel operator, we will consider splitting the firing, taking part of it as a transition on one side and another part on the other.

We also combine the accepting states of the two subautomata into a join state. If the join state is accepting, it only accepts if both subautomata are in the join state, and a join state only allows outward transitions if both subautomata are in the join state.

Definition 4.1. *A **split state** is a triple of three states, the top state and two branch states. If a system is in the top state, it is considered to be in both branch states as well. The branch states may not have any incoming transitions. The split state is also either a shuffle-split state or a parallel-split state.*

Definition 4.2. *A **join state** is a triple of three states, the top state and two branch states. If the system is in both of the branch states, it is also in the top state. The branch states may not have any outgoing transitions.*

Note that the split states and join states are opposites of each other. While we will continue calling split states and join states *states*, it makes sense for the construction to treat them as three separate states. With the splits and joins, we can define graphically what the construction is for the shuffle and parallel operators in Table 4. The big states are the top states and the two smaller states connected to each top state are the matching branch states. Observe that our graphical representation breaks down somewhat when combining multiple shuffle or parallel operators together, such as in $((a \sqcup\!\sqcup b) + (c \sqcup\!\sqcup d)) \cdot ((a \sqcup\!\sqcup b) + (c \sqcup\!\sqcup d))$ where the top states of two joins and two splits are all the same

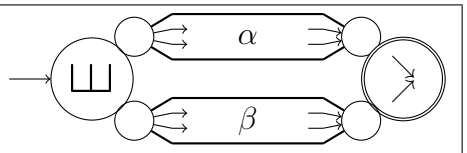
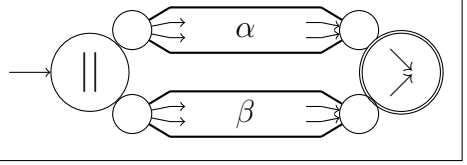
Shuffle	$\alpha \sqcup\!\sqcup \beta$	
Parallel	$\alpha \mid\mid \beta$	

Table 4: Shuffle and Parallel Additions to Improved Thompson Construction

state, but that only affects the graphical representation. Note that in the construction, improvement (a) from the previous section does not apply to branch states.

There is one downside to this construction over the larger one from Section 4.1.1, which is that we no longer have a transition for every valid input. This is especially a problem for the parallel operator. If we are in a pair of states under a parallel operator, and there is a firing where 10 ports fire, we have to check each of the 2^{10} possible ways to divide the ports over the two branches to find which ones lead to a valid transition and which ones don't. In the checklist example at the end of Section 4.1.1, if we fired 10 ports, we would have to check every way those 10 ports could be divided over 26 branches, which leads to $26^{10} = 141,167,095,653,376$ possibilities, from which exactly one will lead to a transition. We need to be able to tell which way to go, so we will introduce a rule on the use of parallel operators.

Definition 4.3. *In the expression $\alpha||\beta$, the union of all firing sets from all firings in α must be disjoint from the union of all firing sets from all firings in β.*

In essence, the parallel operator may only be used on two subexpressions that cover a completely separate subset of ports. Note that it isn't nearly as important to have this rule for the shuffle operator. The same checklist example with shuffle instead of parallel operators would only have 26 different possibilities. Therefore we do not extend it to shuffle. However, the meaning and use cases behind $a \sqcup\!\sqcup a$ are questionable and one might easily choose differently.

With the rule in place we can, for every parallel-split, have two sets of ports. One set has the ports which are used in one branch, and one has the ports which are used in the other. If we are in a pair of states under a parallel operator now, and there is a firing where 10 ports fire again, all we have to do is check, for each of the ports, which of those two sets they are in. When we have divided them over the two branches, we can look for transitions from one of the states with one subset, and from the other state with the other subset.

With the splits and joins and the new rule, we can make monitors. In Section 4.1.1 we made monitors that kept a set of states the automaton could be in. This will no longer be enough, since we can be in multiple states at once. We could make sets of sets of states and that would contain all the information we need, but it would necessitate checking which two states belong to the same pair of branches. It would be much more efficient to store that information as well, and to that end we will make sets of binary trees instead.

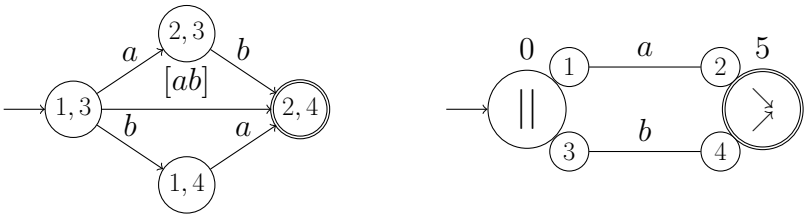


Figure 4.2: Comparison Full Construction and Splits and Joins

The trees have two types of nodes. The first type is the split node. Each split node is either a shuffle-split node or a parallel-split node, and, in the case of parallel-split, contains (or links to) the sets of ports each branch of the split covers. The other type of node is the state-node. It contains exactly one state from the automaton. All leaf nodes are state nodes and all other nodes are split nodes. Every split node has exactly two children, one for the left branch and one for the right branch.

We make a set of trees S, and first insert a tree with one state node with the initial state of the automaton.

Whenever a firing occurs the monitor calls a recursive function called FindTransitions on the root of all trees in S. FindTransitions has three arguments; a tree x, a set of ports p and a set of ports that were triggered p_t, and it returns a set of trees. We send the trees from S to x, the ports that fire to p and the ports that were triggered to p_t. We replace S with the union of the return values. Pseudocode for the FindTransitions function is found in Algorithm 1.

The gist of it is that it finds all transitions that can be taken from trees in S and returns all the states or trees of states they lead to. It does so depth-first on the trees, and it takes splits and joins whenever needed, growing or shrinking the trees in the process.

Note that the pseudocode does not deal with ϵ-transitions at all. ϵ-Transitions should be taken from every a and b in the if-statement on line 6; every d in the else clause of that if-statement, and every top state c found on line 42, adding all reachable states or pairs of states to the relevant sets, or making the necessary recursive calls. It also doesn't deal with the case where a state is the top state for multiple splits. ϵ-transitions should also be taken from the branch states of a split state as possible. FindTransitions should never return the tree with both branch states of a split, since it could only do that if the top state of that split is also returned, and the tree would add no new information.

If S is ever empty, the monitor has detected an error and should act accordingly. This is the value that is monitored and as long as the set is not empty, the implementation is working correctly to the best of our knowledge.

Automata made with splits and joins don't much look like the Reo automata that we were aiming for. However, if we look at the comparison in Figure 4.2, we can clearly see that when we are in state $(1, 3)$ in the full construction, we are in a tree with leaves 1 and 3 in the splits and joins construction. From there we can take transitions with a, b and $[ab]$. These are exactly the same transitions that can be taken from $(1, 3)$. The automaton with splits and joins, together with the trees we introduced to identify states, are simply a different representation of the exact same automaton. So technically we still have all the same states, but by using splits and joins and storing them as trees we only have to store

Algorithm 1 FindTransitions

```
1: function FindTransitions(x, p, p_t)
2:     if p = ∅ then                                  ▷ If no ports are sent to this branch,
3:         return x ,                                                  ▷ take no transitions
4:     else if x is a state node then
5:         R ← ∅
6:         if x is a split state with operator o and branch states a and b then
7:             make a new tree y with o as root
8:             add a and b as children of o
9:             R = FindTransitions(y, p, p_t)
10:        end if                          ▷ If x is a split, also explore the branches
11:        for all transitions t from x to d do                  ▷ Find transitions from x
12:            if t has firing set p and is enabled by p_t then
13:                make a new tree y with root d
14:                add y to R
15:            end if
16:        end for
17:        return R                                              ▷ Return all trees found
18:    else                                          ▷ t is the top node of a (sub)tree
19:        R ← ∅
20:        if t is shuffle-split then                          ▷ Take transitions in 1 branch
21:            a ← FindTransitions(t_left, p, p_t)
22:            b ← FindTransitions(t_right, p, p_t)
23:            R ← all trees with a_i ∈ a as left child and t_right as right child
24:            R ← R∪ all trees with b_i ∈ b as right child and t_left as left child
25:        else                                                      ▷ t is parellel=split
26:            p_a ← subset of p associated with left branch of operator in root of t
27:            p_b ← subset of p associated with right branch of operator in root of t
28:            if p_a ∪ p_b = p then          ▷ if a port is in neither branch, stop looking
29:                a ← FindTransitions(t_left, p_a, p_t)
30:                if a ≠ ∅ then                          ▷ otherwise we know we're done
31:                    b ← FindTransitions(t_right, p_b, p_t)
32:                end if
33:                for all a_i ∈ a and b_j ∈ b do
34:                    Make a new tree y with the same root as t
35:                    t'_left ← a_i                ▷ combine all options for the left branch
36:                    t'_right ← b_j                          ▷ with all options for the right
37:                    add y to R
38:                end for
39:            end if
40:        end if
41:        for all Trees in R with operator x as root with two leaf nodes a and b do
42:            if a and b are branch states of a join with top state c then
43:                Remove the subtree and replace it with c
44:            end if
45:        end for
46:        return R
47:    end if
48: end function
```

the ones the automaton could actually be in, and not all of them. Instances where we can be in so many different possible states that it becomes problematic in computation time or storage space should be exceedingly rare.

It is clear from the construction that $\mathcal{L}(E) = \mathcal{L}(M)$ from Theorem 4.1 also holds for the splits and joins construction.

4.2 Derivative Automata

In this section we will make automata on the fly instead of making them beforehand. We will be doing that using Brzozowski derivatives of our Reo expressions, first introduced in (Brzozowski [4]). The derivative is what is left of the expression after reading a symbol (or in this case a firing). For example the derivative after reading a in ab, or $\delta_a(ab)$, is b.

There are two Reo expressions that we haven't used before, but will now. They are 0 and 1, and they are synonyms of $\emptyset$ and ϵ respectively. Using 0 and 1 makes the definitions below easier to understand.

The function ϵ over Reo expressions is defined below. The function determines if the empty string is accepted by the expression E. For this specific function, the expression is not prefix-closed and ϵ only returns 1 if $\epsilon \in L(E)$ (see Definition 3.9). We have added entries for the shuffle and parallel operators.

Definition 4.4. *Given a Reo expression E the function $\epsilon(E)$ is recursively defined as follows:*
$\epsilon(0) = 0$
$\epsilon(1) = 1$
$\epsilon(a) = 0$
$\epsilon(\alpha + \beta) = Max(\epsilon(\alpha), \epsilon(\beta))$
$\epsilon(\alpha \cdot \beta) = Min(\epsilon(\alpha), \epsilon(\beta))$
$\epsilon(\alpha^*) = 1$
$\epsilon(\alpha \sqcup\!\sqcup \beta) = Min(\epsilon(\alpha), \epsilon(\beta))$
$\epsilon(\alpha || \beta) = Min(\epsilon(\alpha), \epsilon(\beta))$

In the definition of derivatives below, we have added a check for the guards and entries for the shuffle and parallel operators.

Definition 4.5. *Given a Reo expression E over ports P, the derivative of E with regards to some firing $a = (v, N)$ with v (usually called α) a Boolean valuation for each port in P and $N \subseteq P$, $\delta_a(E)$, is recursively defined as follows:*
$\delta_a(E) = E$ *for any expression E if $N = \emptyset$*
$\delta_a(0) = 0$
$\delta_a(1) = 0$
$$\delta_a(b) = \begin{cases} 1 & \text{if } a = (\alpha, N) \text{ and } f_b = N \text{ and } \alpha \vDash g_b \\ 0 & \text{otherwise} \end{cases}$$
$\delta_a(\alpha + \beta) = \delta_a(\alpha) + \delta_a(\beta)$
$\delta_a(\alpha \cdot \beta) = \delta_a(\alpha) \cdot \beta + \epsilon(\alpha)\delta_a(\beta)$
$\delta_a(\alpha^*) = \delta_a(\alpha\alpha^*)$

$$\delta_a(\alpha \sqcup\!\sqcup \beta) = \delta_a(\alpha) \sqcup\!\sqcup \beta + \alpha \sqcup\!\sqcup \delta_a(\beta)$$
$$\delta_a(\alpha||\beta) = \sum_{b \cup c = a, b \cap c = \emptyset} \delta_b(\alpha)||\delta_c(\beta)$$

The $\delta_a(E) = E$ case can only occur after a parallel case where $b = a$ and therefore $c = \emptyset$, or $c = a$ and $b = \emptyset$.

Theorem 4.2. *Let E be a Reo expression and $ax \in \mathcal{L}(E)$: $x \in \mathcal{L}(\delta_a(E))$.*

Proof. The proof follows from Definition 3.9. The construction of the derivatives precisely follows the rules set out in that definition, and in Definitions 3.5 and 3.7. □

The monitor will need to simplify the resulting expression to detect if it has found an error. This is done recursively and depth first, and can be done while calculating the derivative or separately afterwards. The simplifications are as follows:

$$0 + \alpha = \alpha + 0 = \alpha$$
$$0 \cdot \alpha = \alpha \cdot 0 = 0 \qquad 1 \cdot \alpha = \alpha \cdot 1 = \alpha$$
$$0 \sqcup\!\sqcup \alpha = \alpha \sqcup\!\sqcup 0 = 0 \qquad 1 \sqcup\!\sqcup \alpha = \alpha \sqcup\!\sqcup 1 = \alpha$$
$$0||\alpha = \alpha||0 = 0 \qquad 1||\alpha = \alpha||1 = \alpha$$
$$0^* = 1 \qquad 1^* = 1$$

In all other cases, no changes are made. Observe that the one simplification that appears to be missing is $1 + \alpha$. This can't be simplified because $\epsilon(1 + \alpha) = 1$ regardless of the value of α; but the derivative $\delta_a(1 + \alpha) = \delta_a(\alpha)$. Simplifying $1 + \alpha$ to 1 would break the derivative and simplifying it to α would break the ϵ-function. In all cases listed above the simplification works for both the derivative and ϵ.

After the expression has been simplified, if the resulting expression is 0, an error has been detected in the implementation, and monitoring stops.

There are several places we can optimize this algorithm. Observe that, as in Section 4.1.2 a parallel operator on a firing with n ports generates 2^n unique subexpressions. This can be avoided again by enforcing the rule in Definition 4.3. With that rule and the necessary metadata about which ports belong on which side of the operator, only one subexpression will have to be made.

Another optimization we can make is duplicate detection. Take the example $(a^*)^*$. We will take the derivative twice.

$$\delta_a(a^*)^* = \delta_a(a^*)(a^*)^* = \delta_a(a)a^*(a^*)^* = 1a^*(a^*)^* = a^*(a^*)^*$$

$$\delta_a(a^*(a^*)^*) = \delta_a(a^*)(a^*)^* + \epsilon(a^*)\delta_a(a^*)^* = 1a^*(a^*)^* + 1a^*(a^*)^* = a^*(a^*)^* + a^*(a^*)^*$$

The problem here is that every time we take that derivative we double the size of the expression. Obviously $a^*(a^*)^* + a^*(a^*)^* = a^*(a^*)^*$. We can detect duplicates over $+$ by comparing either parse trees or the strings that represent the subexpression, and remove duplicates from there. It may be necessary to detect duplicates over associativity as well, as in $a + b + a = a + b$. Together these should deal with most instances of expressions growing in size unnecessarily.

We have now made a working monitor, but we can choose to build a Reo automaton by remembering all the expressions the monitor has already seen as states and all the firings that have been processed as transitions. The construction isn't very complicated. The initial expression becomes the initial state. When a new firing is observed the monitor checks to see if a transition exists for it and takes that transition. If it doesn't exist, it calculates the derivative. If that derivative matches a state that already exists, the monitor makes a transition from the current state to that state and takes it. Otherwise it makes a new state with the derivative and makes a transition there.

There are two things we glossed over in describing the construction, the first of which is what it means for two expressions e_1 and e_2 to match. There is some flexibility in how we define expressions matching, the strongest of which is equivalence. The expressions e_1 and e_2 are equivalent if and only if $\mathcal{L}(e_1) = \mathcal{L}(e_2)$. The trivial way to test if two expressions are equivalent is to construct the minimal (Reo) automaton for both expressions and compare them, but using that method we would need to construct the entire automaton in the first step, so any time and memory saved by not doing that is gone. There are many other, more efficient methods for finding equivalence between regular expressions and any of them can be used in this construction. However we will not focus on those methods here.

Instead of expression-equivalence we'll use lexicographical equivalence. Two expressions are only equivalent if they are exactly the same. For our purposes in many cases that will be enough. The derivatives generated are generally well-behaved and while they will absolutely spawn copies of subexpressions, they won't rearrange subexpressions and they will spawn copies in a predictable and consistent manner. If there is a subexpression $a + b$, and no other instances of a or b, there is no combination of derivatives that can be taken that will generate the subexpression $b + a$. Experiments will have to show if, and under what circumstances, it is necessary to reorder expressions into a canonical form to determine if they are equivalent by associativity or commutativity. It is unclear how the cost of doing so compares to the benefit of possibly having fewer states in the final automaton.

We also glossed over making transitions. Obviously the firing set will be the set of ports that fired, and the source and destination are clear, but the guard is more complicated. The simplest way to make the guard is to take the *and* over the states of all the ports. So if we have ports $\{a, b, c\}$ and a fires when a and b are enabled, the guard would be $a \wedge b \wedge \bar{c}$ because that was what was observed. Then if we have the same firing from the same state with a, b and c enabled, instead of adding a new transition, we would do an *or* over the two guards. So we would get $(a \wedge b \wedge \bar{c}) \vee (a \wedge b \wedge c)$ which we can simplify to $a \wedge b \wedge (\bar{c} \vee c)$ and further to $a \wedge b$. However in circuits where a typical firing only relies on a small subset of ports, it can take a very long time to find that information.

We can do better by combining the guards of every derivative we take. When a firing occurs we make a new empty Boolean expression $g = 1$. Whenever we encounter a $\delta_a(b)$ case, and $f_b = a$ and the guard g_b holds, g becomes $g \wedge g_b$. If $f_b = a$ and g_b does not hold, g becomes $g \wedge \neg g_b$. The final g covers every instance where all of the subexpressions return the same result. This doesn't necessarily mean this is the exact right guard for this transition, but it is guaranteed to be correct and it will be much closer than before.

We can also do better comparing expressions to the expressions already in the automaton.

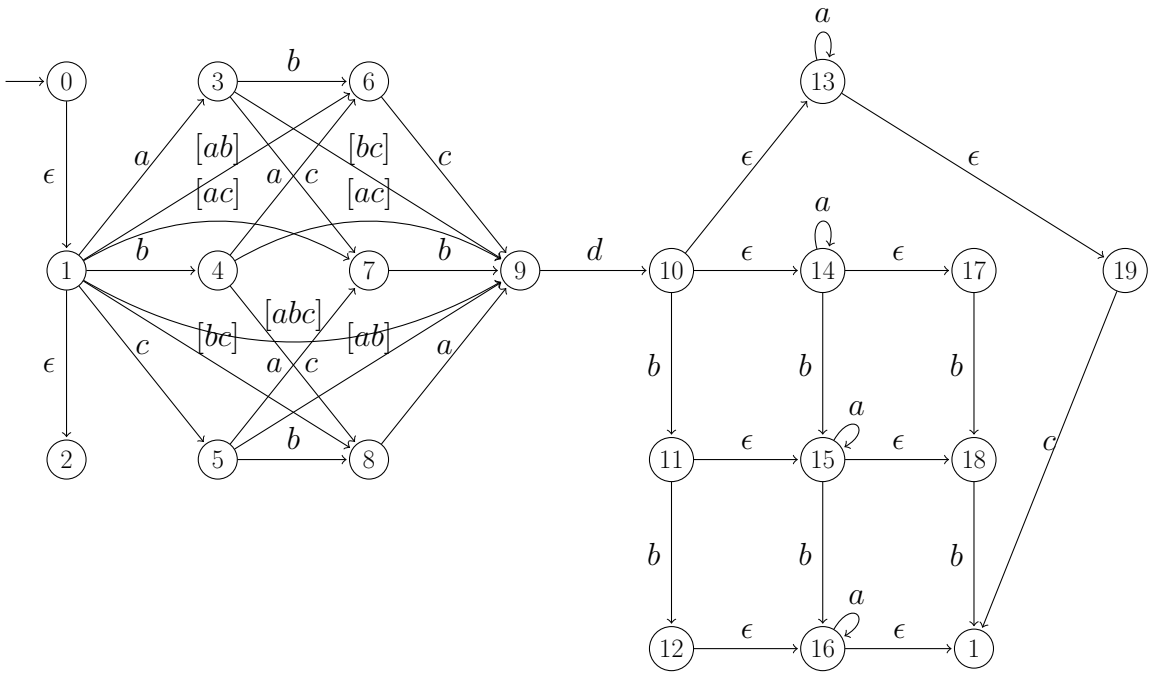


Figure 4.3: Full automaton construction

Instead of simply comparing the new expression to all the expressions we've had before, we can use a hashing function or a binary search tree or a similar data structure to store the states and find the right one, or the non-existance thereof, much more efficiently.

4.3 Method Comparisons by an Example

So now we have four methods to make automata out of expressions: the full automaton construction, the automaton construction with splits and joins, the derivative construction and the derivative construction that builds an automaton. We will start with a big example and do the construction for each method. We will simply mark which expressions are identical for the derivative construction with arrows, because the actual derivatives are the same whether we build an automaton or not. The example is the specification $((a||b||c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$, and the string $a[bc]daabbad$.

The constructed full automaton is shown in Figure 4.3. The area between states 1 and 9 is a little hard to read, but it may help to know that all labels are above their respective transitions. The automaton with splits and joins is shown in Figure 4.4. Note that both automata have two states 1. They are the same state, but overlapping them would make the picture less clear, and we don't have graphical notation for a state being simultaneously the top state of a join and a split. Their sets of states respectively sets of trees are in Table 5.

The derivatives are in Table 6. Whenever a symbol is read, the table shows an unsimplified derivative and a simplified one on the next line. Subexpressions that simplify to $+0$ are omitted. Arrows mark the three simplified expressions that are identical and would be the same state if we built an automaton. The last column is part of the expression, but since that part never changes it is separated out for clarity.

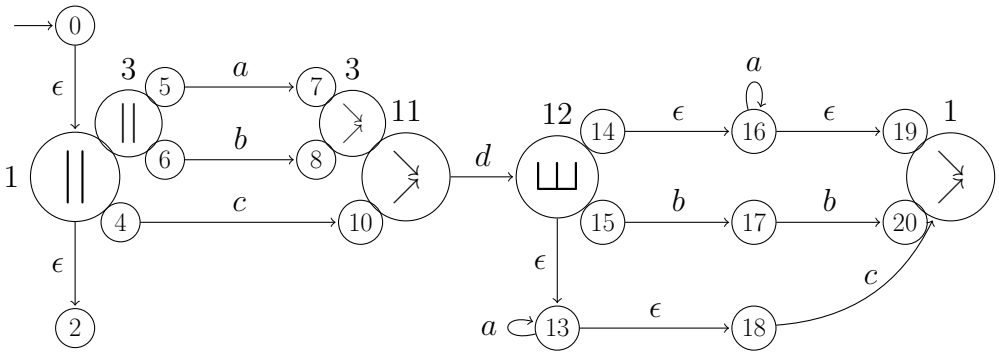


Figure 4.4: Splits and joins construction

	$\{0, 1, 2\}$	$\{0, 1, 2\}$
a	$\{3\}$	$\left\{ \begin{array}{c} \| \\ 4 \quad \| \\ 6 \quad 7 \end{array} \right\}$
$[bc]$	$\{9\}$	$\{11\}$
d	$\{10, 13, 14, 17, 19\}$	$\{12, 13, 18\} \cup \left\{ \begin{array}{cc} \sqcup & \sqcup \\ 15 \quad 16 & 15 \quad 19 \end{array} \right\}$
a	$\{13, 14, 17, 19\}$	$\{13, 18\} \cup \left\{ \begin{array}{cc} \sqcup & \sqcup \\ 15 \quad 16 & 15 \quad 19 \end{array} \right\}$
a	$\{13, 14, 17, 19\}$	$\{13, 18\} \cup \left\{ \begin{array}{cc} \sqcup & \sqcup \\ 15 \quad 16 & 15 \quad 19 \end{array} \right\}$
b	$\{15, 18\}$	$\left\{ \begin{array}{cc} \sqcup & \sqcup \\ 17 \quad 16 & 17 \quad 19 \end{array} \right\}$
b	$\{1, 2, 16\}$	$\{1, 2\} \cup \left\{ \begin{array}{c} \sqcup \\ 20 \quad 16 \end{array} \right\}$
a	$\{1, 2, 3, 16\}$	$\{1, 2\} \left\{ \begin{array}{ccc} \sqcup & \| & \\ 20 \quad 16 & 4 & \| \\ & & 6 \quad 7 \end{array} \right\}$
d	$\emptyset$	$\emptyset$

Table 5: Sets of States and Sets of Trees Example

0		$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$	
	a	$(1\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
1		$(b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	$[bc]$	$(1\|\|1)d((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
2		$d((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	d	$1((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
3	$\rightarrow$	$((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	a	$((1a^* \sqcup\!\sqcup bb) + 1a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
4	$\rightarrow$	$((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	a	$((1a^* \sqcup\!\sqcup bb) + 1a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
5	$\rightarrow$	$((a^* \sqcup\!\sqcup bb) + a^*c)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	b	$((a^* \sqcup\!\sqcup 1b) + 0)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
6		$(a^* \sqcup\!\sqcup b)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	b	$(a^* \sqcup\!\sqcup 1)$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
7		(a^*)	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
	a	$((a^*) + (1\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
8		$((a^*) + (b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))$	$((a\|\|b\|\|c)d((a^* \sqcup\!\sqcup bb) + a^*c))^*$
9	c	0	

Table 6: Derivatives Example

So what can we learn from the constructions? We can see the differences between the full automaton and the splits and joins version by seeing that the splits and joins version has a lot fewer transitions. Specifically it reduced the number of transitions in the $(a||b||c)$-section from 19 to 3. We can also see in the subautomaton for $(a^* \sqcup\!\sqcup bb)$ that there is repetition. There are three copies of the subautomaton for a^* horizontally and three copies of bb vertically. No such repetition exists in the splits and joins version.

Looking at Table 5 we can see that at every stage both construction methods have the same number of possible states or trees. This is partly because we do not allow a tree with the two branch states of a split to be in a set together with the top state of that split (or recursively as subtrees of otherwise identical trees), because they are redundant. Because of that every tree in the set uniquely matches one state in the full automaton. So whenever state 13 is in the set, the tree $15 \sqcup\!\sqcup 16$ is in the other set.

Take a look at the third row of Table 5. Both automata can only be in a single state, 9 and 11 respectably. If we look at those states and read the letter d, in the full automata we explore the transitions $9 \rightarrow 10$, $10 \rightarrow 13$, $13 \rightarrow 19$, $10 \rightarrow 14$ and $14 \rightarrow 17$. In the automaton with splits and joins we explore $11 \rightarrow 12$, $12 \rightarrow 13$, $13 \rightarrow 18$, $14 \rightarrow 16$, and $16 \rightarrow 19$. We also look at, but don't take, the same four transitions. So even the transitions are essentially the same. The only difference betweeen the two automata constructions is where we store the information about where each side of a shuffle or parallel operator is, and what transitions or combinations of transitions can be taken. In the full construction we store everything in the automaton, which is why we end up with many more transitions and more states (when the branches are larger or more numerous). In the splits and joins

method we store some of that information in the trees instead.

We define the **split-depth** d of an expression to be the largest number of parallel or shuffle branches that can be explored at the same time. For example a has a split-depth of 1, $(a||b)$ has a split-depth of 2, our example has a split-depth of 3 and $((a \sqcup\!\sqcup b)||(c \sqcup\!\sqcup d))$ has a split-depth of 4. In the splits and joins construction, the number of nodes in a tree is at most $2d - 1$, where the set of states stores exactly one state. If we look at the number of states in the automaton, the number of states in the splits and joins construction is exactly $2d$ plus the number of states in subexpressions, assuming there are exactly d splits. In the full automaton construction it is the product of the number of states in each subexpression. Since empty subexpressions are meaningless, each subexpression must have at least two states and thus the total number of states is at least 2^d. So the splits and joins method scales linearly with split-depth, and the full construction scales exponentially, making the full construction substantially less efficient for expressions with high split-depth.

When we use derivatives the first question we should ask ourselves is if we should construct the automaton or not. The upside of building the automata is that eventually we won't have to calculate derivatives anymore, and the resulting automaton will be very efficient, because it is deterministic. The downside is that it takes resources to build and store the automaton.

It should be clear that if there are no states visited repeatedly, it makes no sense to store them. Let x be an expression without a Kleene closure. Observe that if we make an automaton for x, using any of our construction methods, the automaton will have no loops. Meaning that it is never possible to reach a state we have already been. So if there is no Kleene closure in the expression and we are taking derivatives, it is pointless to build an automaton from them.

If we look at the checklist example (with a Kleene star around it) $(a||b||\ldots||z)^*$ we can easily see that there is an expression for all s^{26} subsets of $\{a, b, \ldots, z\}$, and all the 2.5 trillion transitions are there to be found as well. Additionally, instead of storing just the states and transitions, we also have to store a unique expression for each state. So in the worst case building a DFA using derivatives uses the most storage. However in practice only a small subset of these states will ever be visited, and thereby constructed. The derivative automaton construction only builds the parts of the automaton that are actually used, which is why, in practice, storage space shouldn't be a big concern.

The actual speeds of the four algorithms and the sizes of the automata they generate both vary heavily depending on the expression, so it isn't possible to say which one is the smallest and which one is the fastest, but we will give some general heuristics based on expectations and examples, but there are counterexamples to be found for almost all of them.

For size we can be pretty confident that the derivatives without automaton will usually be the smallest. The splits and joins construction will generally be smaller than the full automaton construction which can grow to ludicrous sizes if there are many shuffle or parallel operators in an expression. Using derivatives to build a DFA will eventually lead to the largest automaton in the worst case, but for many automata it will be substantially smaller than the full construction.

If we look at efficiency the full automaton construction will generally be the most efficient. Splits and joins is a more complicated version of the same construction that saves space at the cost of some efficiency. The derivatives will typically be slower. Using derivatives to construct a DFA will start off as the slowest method, but when the automaton is constructed it will easily be the fastest. How long it takes to get to that point will vary wildly.

Over all, the splits and joins method seems a good general choice. It is not the fastest and it is not the smallest, but it is consistent and predictable. Further experiments should be done to see which method works best in practice.

5 Conclusions

In this thesis we have described methods that construct monitors for Reo-like circuits. We adapted regular expressions into Reo expressions, based on the ideas of Reo automata, to serve as specifications. Like Reo automata, the Reo expressions are guarded. We also added two operators; the shuffle operator which allows the behavior of two sub-expressions to be interleaved, and the parallel operator that allows independent sections of a circuit to act simultaneously.

With the Reo expressions in hand, we described four methods for constructing monitors based on these specifications. The first method constructed non-deterministic automata using an improved version of the Thompson construction, modified to add constructions for the shuffle and parallel operators. Some investigation revealed that in some cases these automata could grow to unreasonable sizes. A solution was found in a second method, which introduced splits and joins on automata. The essence of this improvement is that if two subautomata were put in parallel, the full automaton would have a state for every possible combination of states in those subautomata. Splits and joins calculates those combinations if and when they're needed, potentially saving a lot of storage space at the cost of more processing time.

The other two methods both take the derivatives of Reo expressions. To that end Brzozowski derivatives of regular expressions were expanded to deal with firings, and the shuffle and parallel operators. The difference between these methods is that one constructs a deterministic automaton based on every derivative it has taken and the other doesn't. Eventually the automaton is going to be faster, but it may not be worth the effort if there are a very large number of states that are rarely visited.

We have also tried to compare the different methods, but the results from pure theory are inconclusive. There seems to be a niche for most, if not all, of the methods. Determining which method works best in general, real-world scenarios would require more experiments.

The obvious future work would be actually implementing one or more of these methods and using them to monitor Reo-like circuits. That would also allow the methods to be thoroughly tested to see which one works best in practice. It would also be useful if that implementation connected easily to the Extensible Coordination Tools that are currently used to build and generate code for Reo circuits.

Another piece of future work would be to try and build a monitor that doesn't just

check if a firing was allowed to take place. With the methods presented here we can't determine if a firing was *supposed* to take place based on the ports that were triggered. This information could potentially be used to detect an error if a firing was supposed to happen, and didn't. Also, in an implementation that could either be in a state that could fire a or a state that could fire b, if a is triggered and no firing occurs, we could conclude that it must be in the state that can fire b.

If the splits and joins-automaton is used in future work and papers, it would be good to have a robust definition of it, as well as a graphical representation that can present having one state be the top state of multiple splits and/or joins in an understandable graphical manner.

It will also be worth exploring monitors that can also check that the data being transferred is correct. We decided not to do that in this thesis for the sake of simplicity in new methods, but it should be worth a closer look. It isn't hard to imagine a circuit that would be undeniably wrong and would still pass the test here. Imagine a circuit that takes data from two input ports and sends it to two output ports atomically. The Reo expression can't have information on which input is supposed to be connected to which output, but that is definitely important to how the circuit functions. It seems possible to monitor that data flow, but it may be that traditional test cases are better for detecting these sorts of errors.

References

[1] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. doi: 10.1017/S0960129504004153.

[2] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. Automata for context-dependent connectors. In John Field and Vasco T. Vasconcelos, editors, *Coordination Models and Languages*, pages 184–203, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02053-7. doi: 10.1007/978-3-642-02053-7_10.

[3] Sabine Broda, António Machiavelo, Nelma Moreira, and Rogério Reis. Automata for regular expressions with shuffle. *Information and Computation*, 09 2017. doi: 10.1016/j.ic.2017.08.013.

[4] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL `http://doi.acm.org/10.1145/321239.321249`.

[5] Lucian Ilie and Sheng Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, October 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00090-7. URL `http://dx.doi.org/10.1016/S0890-5401(03)00090-7`.

[6] Natallia Kokash, Farhad Arbab, Behnaz Changizi, and Leonid Makhnist. Input-output conformance testing for channel-based service connectors. *Electronic Proceedings in Theoretical Computer Science*, 60, 08 2011. doi: 10.4204/EPTCS.60.2.