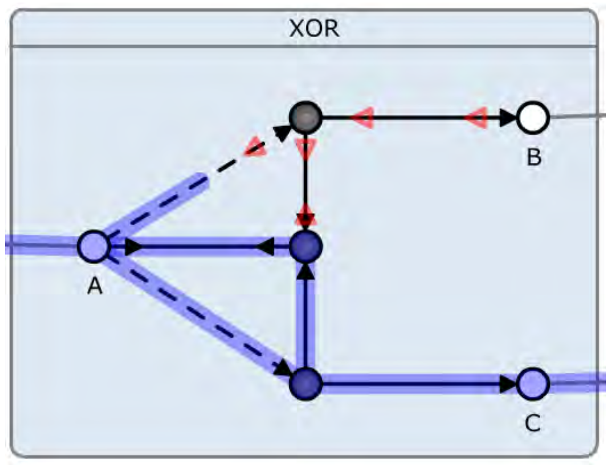
Master thesis

# QoS analysis by simulation in Reo



Author: Oscar Kanters

March, 2010

*Supervisors CWI:*
Chrétien Verhoef
Rob van der Mei

*Supervisors VU:*
Martijn Schut
Dennis Roubos

# Preface

The end of the BMI (Business Mathematics and Informatics) Master program consists of an internship of six months that has to be carried out within a business, industry, or research facility outside the Faculty of Sciences of the VU. My Master project was at the CWI at the PNA2 (Probability and Stochastic Networks) research group.

I would like to thank Chrétien Verhoef, my internship supervisor at CWI for all the discussion, help and support with my thesis. I also would like to thank Rob van der Mei for the opportunity to do my internship at CWI and for the tips and meetings during the internship. I would also like to thank Martijn Schut for being my supervisor at the VU, and Dennis Roubos for being the second reader at the VU. Both of them gave some very useful tips during the research. I would like to thank Farhad Arbab and Christian Krause from SEN3 (Coordination Languages) at the CWI also. Christian Krause helped me with explaining the source code of Reo and helping me with any problem I had during implementation. Farhad Arbab, with all his knowledge about Reo, was very useful during the meetings we had about the set-up and the output of the simulation. Finally I want to thank all other colleagues at CWI for the nice talks during lunch breaks and the nice time I had at CWI.

Oscar Kanters

March 2010

# Summary

The Reo coordination language is designed to model synchronization between different parts of complex systems. Originally, Reo was designed without a notion of time. Later, additional models made it possible to make Reo stochastic by defining delays on channels and arrival rates on the boundary nodes of the system. With this approach it is possible to model such systems as continuous time Markov chains to perform quality of service analysis. However, this conversion is only possible if the Markov property has been met, which is only the case when using exponential distributions to model the arrival rates and delays.

When using other general distributions, the Markov property is not satisfied, so the Markov chains could not be used any more for QoS analysis. Analytically, it is also very hard to solve complex systems with synchronization and various general distributions for the delays and arrival rates. For this reason, we created a simulation tool to do quality of service analysis on systems modelled in Reo.

This simulation model has been implemented within the Eclipse Coordination Tools framework, which already includes the graphical user interface and other plug-ins for Reo. The simulator uses discrete event simulation, and will keep track of all sorts of quality of service statistics which might be of interest for the systems the user wants to model in Reo.

For the simulation, we used the colouring semantics as driver, which indicates which parts of a connector can have data flow in a given configuration of the system. Because this semantics respect the synchronization principle of Reo, we needed some workarounds to evaluate asynchronous systems. Although we were able to model all systems with these workarounds, this still had some disadvantages regarding memory usage and simulation speed. In the future, we could probably use other drivers for the simulator, which makes the workarounds unnecessary.

We validated the simulator using continuous time Markov chains on some systems. We also modelled some queueing models, and validated the simulation results with the results known in queuing theory. The results were also verified using two other simulators. Finally, we used the simulator to evaluate systems with general stochastic distributions which could not be evaluated before.

So with this simulator in Reo we are able to perform quality of service analysis on almost any system modelled in Reo with general stochastic distributions on the delays of the channels and the arrival rate on the boundary nodes.

# Contents

# Chapter 1

# Introduction

In this chapter we will give a general introduction about CWI in section 1.1. In section 1.2 we will introduce the research topic and define our research objective and questions. Finally, section 1.3 gives an overview of the structure of the thesis.

## 1.1 About CWI

Founded in 1946, CWI is the national research center for mathematics and computer science in the Netherlands. More than 170 full professors have come from CWI, of whom 120 still are active. CWI's strength is the discovery and development of new ideas, and the transfer of knowledge to academia and to Dutch and European industry. This results in importance for our economy, from payment systems and cryptography to telecommunication and the stock market, from public transport and internet to water management and meteorology.

**An international network**
With its 55 permanent research staff, 40 postdocs and 65 PhD students, CWI lies at the heart of European research in mathematics and computer science. Researchers at CWI are able to fully concentrate their efforts on their scientific work, and to build an international network of peers. More than half of the permanent research staff maintains close contact with universities as part-time professors. The personal and institutional research networks strengthen CWI's positions and serve as a magnet for attracting talent. The CWI researchers come from more than 25 countries world-wide.

**A source of pride**
CWI was a birthplace of the world-wide internet. Cwi.nl was the first national domain name ever issued anywhere. CWI helped develop the wing of the Fokker Friendship - chosen the most beautiful Dutch design of the 20th century. The popular language Python was invented at CWI, the language in which Google was developed. CWI applied combinatorial algorithms to the scheduling of the Dutch railway system. XML-databases were build to the needs of the Netherlands Forensic Institute and 3D visualization techniques to better detect cancer tumors.

## 1.2 Research questions

Within CWI, a model has been developed (Reo Coordination Language [5]) to model complex systems like software, communication systems or websites such as holiday reservation sites and PayPal. New to this approach is to model synchronization between different parts of the system. The real-time end-to-end delay for the customer is a big factor for these systems. The response times should not be too large, else customers will leave the system. A system should also not be blocked because parts of the system will wait on each other. It will become difficult when different parts of the system are dependent of each other and synchronization should take place inside the system.

Reo has been developed to evaluate systems with synchronization, among other things. Reo has a graphical user interface in Eclipse, called Eclipse Coordination Tools (ECT) which is a set of plug-ins for the Eclipse platform. At the beginning Reo was designed without a notion of time, so all transitions happens instantly. Later, new models were invented to add delays and arrival rates to the system. These models can be transformed into continuous time Markov chains when modelling systems with exponential delays and arrival rates. These Markov chains can be used to get the steady state behaviour of the system and with some work also to get blocking probabilities, end-to-end delay and other quality of service (QoS) metrics.

When we want to use other probability distributions, we can not use the Markov chains any more causing that we can not derive statistics about the Reo models. For example, when we want to model systems with discrete arrivals or arrivals based on a trace file. It is still possible to detect if the system will be blocked or to see if certain chains of events can occur, but giving QoS statistics is not possible.

Analytically, synchronization is very difficult when combining all kinds of distributions in complex systems. With Reo, we are able to build almost any system we want. In this system we should be able to define various continuous and discrete distributions. Analytically, it is very difficult or maybe impossible to get quality of service information for these systems. For this reason we need a simulation model to evaluate systems with general probability distributions. With this simulation model we should be able to get all quality of service statistics which might be interesting for a system modelled in Reo. With a simulator it is also important to know if the system is stable, because if the system is not stable, the results are not reliable. For this reason we should be able to show the behaviour of the model over time.

For this research we define the following research objective:

- **Build a simulation tool for Reo integrated in ECT to perform quality of service analysis on Reo models.**

For this objective we define the following research questions:

1. What are the current limitations of Reo?
2. Which output statistics are relevant for the systems we want to model?

3. Does the Reo simulator produce the same results as other simulators?
4. What kind of systems can we model with Reo?
5. What are the limitations of the simulator or the approach used for the simulator?

## 1.3 Thesis structure

This thesis addresses the simulation approach in Reo to perform QoS. Chapter 2 introduces Reo and the connector colouring semantics used as the driver for the simulator. It will also introduce CA models as a formalism to capture the operational semantics of Reo, QIA is an extension to CA which implements stochastic Reo to produce models which can be converted to continuous time Markov chains.

Chapter 3 introduces the simulation model as an extension to Reo for general distributions. It will describe all output statistics which will be displayed after the simulation. The chapter will also give a general description of the set-up and the limitations and workarounds which are the results of choices made in the set-up.

Chapter 4 describes the details of the implementation of the simulator in Reo.

Chapter 5 will validate the simulator by comparing the simulation results with results produced by QIA, queueing theory and other simulators.

Chapter 6 will discuss Reo systems which could not be evaluated before by using general stochastic distributions.

Finally, chapter 7 discusses possible future work on the simulator.

# Chapter 2

# Background & Motivation

This chapter gives an introduction to the research topic. Section 2.1 introduces Reo as a coordination language to model complex systems with synchronization. Section 2.2 introduces the colouring semantics to distinguish parts of the system with and without data flow. This colouring semantics will be used as the driver for the simulation which will be explained in chapter 3. Section 2.3 introduces CA models to as a formalism to capture the operational semantics of Reo, QIA is an extension to CA which implements stochastic Reo to produce models which can be converted to continuous time Markov chains. Finally, section 2.4 summarizes the first sections and states why we would need a simulation model. This section answers the research question: What are the current limitations of Reo?

## 2.1 Reo

Within CWI, a model has been developed (Reo Coordination Language [5]) to model complex systems like software, communication systems or websites such as holiday reservation sites and PayPal. New to this approach is to model this synchronization between different parts of the system. In this section we present an overview of Reo, and is an adapted version of chapter 2 in [14]. For a full account of Reo, see [8, 9].

The emphasis in Reo is on *connectors* (a system modelled in Reo) which act as exogenous coordinators to orchestrate the components that they interconnect in a composed system. *Channels* constitute the only primitive connectors in Reo, each of which is a point-to-point communication medium with two distinct primitive ends. Reo uses a generalized notion of channels. In addition to the common channel types of synchronous and asynchronous, with bounded or unbounded buffers, and with FIFO and other ordering schemes, Reo allows an open-ended set of channels, each with its own, sometimes exotic, behaviour. For instance, a channel in Reo does not need to have both an source end which accepts data into the channel, and an sink end which dispenses data out of the channel; it can instead have two source or sink ends.

More complex connectors can be constructed out of simpler ones through connector composition. In Reo, channels are composed by conjoining their ends to form *nodes*. A node may contain any number of channel ends.We classify nodes into three different types depending on the types of their coincident ends: a *input (source) node* contains only source ends; a *output (sink) node* contains only sink ends; and a *mixed node* contains both kinds of channel ends.

Components interact with a Reo connector using a simple interface. A component will have access to a number of input and output nodes. Components perform I/O operations on input and output nodes only. The only way a component may interact with a connector is by issuing I/O operations (*write* and *take*) on these ends. A connector can perform a *write* with some data on an input end or a *take* on an output end. The *write/take* will succeed when the connector either accepts the data of the *write* or produces data for the *take*. It is by delaying these operations that coordination is achieved. We refer to an I/O operation that is being delayed as a *pending* operation. In addition, there are various operations for constructing and reconfiguring Reo connectors, but these are irrelevant for this paper.

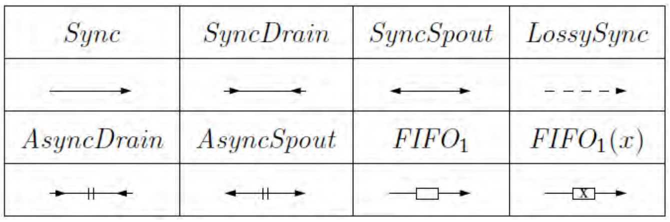


Figure 2.1: Some basic channel types in Reo

Figure 2.1 shows some example channels. *Sync* denotes a synchronous channel. Data flows through this channel if and only if it is possible to simultaneously accept data on one end and pass it out the other end. *SyncDrain* denotes a synchronous drain. Data flows into both ends of this channel only if it possible to simultaneously accept the data on both ends. *SyncSpout* denotes a synchronous spout. Data flows out of both ends of this channel only if it possible to simultaneously take the data from both ends. *LossySync* denotes a lossy synchronous channel. If a take is pending on the output end of this channel and a write is issued on the input end, then the channel behaves as a synchronous channel. However, if no take is pending, the write can succeed, but the data is lost.

Observe that this channel has *context dependent behaviour*, as it behaves differently depending upon the context. If it were context independent, the data could be lost even if a take was present. *AsyncDrain* denotes an asynchronous drain. Data can flow into only one end of this channel at the exclusion of data flow at the other end. *AsyncSpout* denotes an asynchronous spout. Data can flow out of only one end of this channel at the exclusion of data flow at the other end. *FIFO*$_1$ denotes an empty FIFO buffer. Data can flow into the input end of this buffer, but no flow is possible at the output end. After data flows into the buffer, it becomes a full FIFO buffer. *FIFO*$_1$*(x)* denotes a full FIFO buffer. Data can flow out of the output end of this buffer, but no flow is possible at the input end. After data flows out of the buffer, it becomes an empty FIFO buffer.

A write operation to a source node succeeds only if all source ends coincident on the node accept the data item, in which case the data item is written to every source end coincident on the node. An input node thus acts as a replicator. A take operation on an sink node succeeds only if at least one of the sink ends coincident on the node offers a data item; if more than one coincident channel end offers data, one is selected non-deterministically, at the *exclusion* of all others. A sink node, thus, acts as a merger. A mixed node combines the behaviour of a source (merger) and a sink node (replicator).

Although Reo connector may look like electrical circuits and the synchronous channels may lead the reader to think of Reo connectors as synchronous systems, it would be wrong to equate Reo with either model. Although the precise implementation details are more involved, a Reo connector is executed essentially in two steps: (1) based on pending write/take, solve the synchronisation/exclusion constraints imposed by the channels of a connector to determine where data can flow; and (2) send data in accordance with the solution in step (1). The second step may not occur if no data flow is possible. In between steps (2) and (1), new write/take operations may be performed on the channel ends, or existing ones may be retracted. Not all of the connector needs to be involved in step (1) at the same time: FIFO buffers, for example, serve to divide connectors into *synchronous regions* which operate more or less independently.

## 2.2 Connector colouring

In the following section we will provide an overview of the connector colouring semantics as a way to mark which parts of the connector has data flow. This section uses the relevant parts of the connector colouring paper [14], for a full description about connector colouring, see [14].

The semantics of a Reo connector is defined as a composition of the semantics of its constituent channels and nodes. We illustrate Reo's semantics through an example, in part to give an understanding of how Reo works, but also to motivate the upcoming notion of connector colouring.

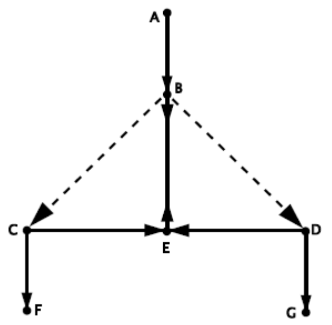


Figure 2.2: Exclusive router connector

The connector in figure 2.2 is an exclusive router (XOR) built by composing five Syncs, two LossySyncs and one SyncDrain. The intuitive behaviour of this connector is that data obtained through its input node A is delivered to exactly one of its output nodes F or G. If both F and G are willing to accept data, then the node E non-deterministically selects which side of the connector will succeed in passing data. The SyncDrain and the two Syncs in the node E conspire to ensure that data flows at precisely one of C and D, and hence F and G, whenever data flows at B. An informal, graphical way of depicting the possible data flow through the exclusive router is by colouring where data flows, as illustrated in figure 2.3, where the thick solid line marks the parts of the connector where data flows and unmarked parts correspond to the parts where no data flows. This idea of colouring underlies our model. Note that we abstract away from the direction of data flow, as the channels themselves determine this.

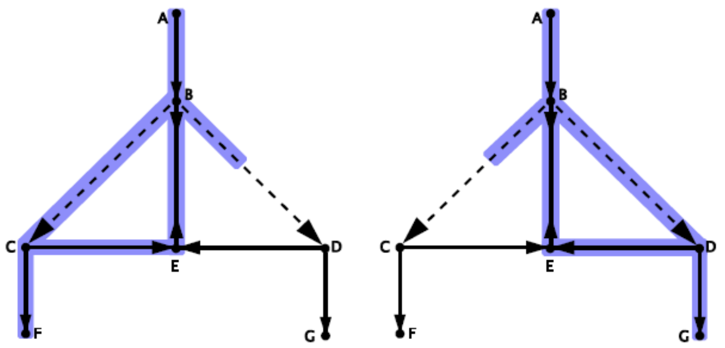


Figure 2.3: Possible colourings for XOR connector

### 2.2.1 2-Colouring

The colouring model is based on the idea of marking data flow and its absence by colours. Each colouring of a connector is a solution to the synchronisation constraints imposed by its channels and nodes. Let *Colour* denote the set of colours. A reasonable minimal set of colours is *Colour* = {—, ---}, where the colour '—' marks places in the connector where data flows, and the colour '---' marks the absence of data flow.

Reo semantics dictates that data is never stored or lost at nodes [8]. Thus, the data flow at one end attached to a node must be the same as at the other end attached to the node. Either data will flow out of one end, through the node, and into the other end, or there will be no flow at all. Hence, the two ends plugged together will be given the same colour, and thus we just colour the node. Colouring nodes determines the colouring of their attached ends, which in turn determines the colouring of the connector, and thus the data flow through the entire connector. Colouring all the nodes of a connector, in a manner consistent with the colourings of its constituents, produces a valid description of data flow through the connector. Channels and other primitive connectors then determine the actual data flow based on the colouring of their ends.

The following definition formalizes the notion of a colouring. Let *Node* be a denumerable set of node names.

**Definition 2.1.** ***Colouring***
*A colouring* $c \colon N \to Colour$ *for* $N \subseteq Node$ *is a function that assigns a colour to every node of a connector.*

Let us consider a $FIFO_1$ with input end $n_1$ and output end $n_2$. One of its possible colourings is the function $c_1 : \{n_1 \mapsto —, n_2 \mapsto \text{---}\}$, which describes the situation where data flows through the input end $n_1$ and no data flows through the output end $n_2$.

Channels, nodes, and connectors typically have multiple possible colourings to model the alternative ways that they can behave in the different contexts in which they can be used. The collection of possible colourings of a connector is represented by its colouring table.

**Definition 2.2.** ***Colouring table***
*A colouring table $T$, over nodes $N \subseteq Node$ is a set of colourings with domain $N$.*

A colouring table for a Reo connector describes the possible behaviour in a particular configuration (or snapshot) of the connector, which includes the states of channels, plus the presence or absence of I/O requests. A colouring corresponds to a possible next step based on that configuration.

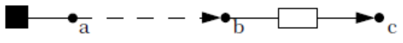
I/O operations need to be modelled in colouring tables so that we can determine the context dependent behaviour of a connector. It is the presence and absence of I/O operations on the boundary of a connector which gives the context.

When we refer back to the exclusive router connector, we have one colouring table with three possible colourings. The two colourings given in figure 2.3 and the no flow colouring when either the source node A has no I/O operation or both F and G have no I/O operation.

### 2.2.2 3-Colouring

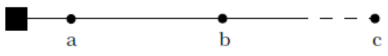
In this section we address the issue of context dependent behaviour. We demonstrate that the 2-colouring scheme of the previous section applied to a connector involving a LossySync fails to give the expected data flow behaviour. We argue that this occurs because context information is not propagated to enable channels to choose their own correct context dependent behaviour. Previous semantic models of Reo connectors [13, 12] remain at a coarser level of abstraction and fail to address this issue.

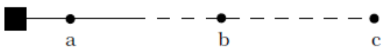
A LossySync has the following context dependent behaviour, if both a write is pending on its input end and a take is pending on its output end, then it behaves as a Sync, the write and take simultaneously succeed, and the data flows through the channel. If, on the other hand, no pending take is present, then the write succeeds but the data is lost. Problems with the 2-colouring scheme reveal themselves when we compose a LossySync, an empty $\text{FIFO}_1$, and an I/O request on the input end of the LossySync, as follows:



This connector has the following two alternative 2-colourings:

The first colouring indicates that the I/O operation succeeds, the data flows through a and that the LossySync acts as a Sync sending the data through b into the $\text{FIFO}_1$. This is the expected behaviour in this configuration.





The second colouring indicates that data flows through node a, but not at node b, indicating that it is lost in the LossySync. An empty $FIFO_1$ is, however, input enabled, meaning that it should always be able to accept data. Another way of seeing this is that an empty $FIFO_1$ always issues a take to whatever channels it is connect to. Indeed, the only reason that it should not succeed in receiving data is if the connector gives it a reason not to, such as by not sending it any data. One can therefore interpret the situation as a violation of the intended semantics of the LossySync channel, because the information that the data can be accepted on its output end is not appropriately propagated to it. The LossySync cannot detect the presence of the pending take issued by the input-enabled, empty $FIFO_1$ buffer. Similar situations arise when dealing with a LossySync in isolation or in the context of any connector.

The behaviour of a context dependent primitive depends upon the presence or absence of I/O requests on its ends. For mixed nodes, however, no I/O request information is present, so it is not obvious what the context is. The key to resolving this is to determine what context information can be consistently propagated while addressing synchronisation constraints. Rather than propagating the presence of an I/O request, the approach focuses on propagating their absence, or more generally, on any reason to delay data flow, such as unsatisfiable synchronisation constraints or due to choices made by a merger.

To address the problem just described, we modify our set of colours. Since we wish to trace the reason to delay, we replace the no-data-flow colour by two colours which both use a dashed line marked with an arrow. This colouring scheme is referred to as 3-colouring. The arrow indicates the direction that a reason to delay comes from, that is, it points away from the reason in the direction that the reason propagates. Thus we now work with colours, *Colour*={—, -◁-, -▷-}. In fact, the colours depend upon how the arrow lies in relation to the channel end being coloured. A no-flow colouring with the arrow pointing towards the end, -▷-●, means give a reason to delay, and a colouring with the arrow pointing the opposite way, -◁-●, means require a reason to delay.

We can compose two colourings at a given node if at least one of the colours involved gives a reason to justify no flow. Of the four possible combinations of end colourings at a node, three can be composed, as given in figure 2.4. The last case is not permitted as it joins two colouring which require a reason to delay, without actually giving a reason.

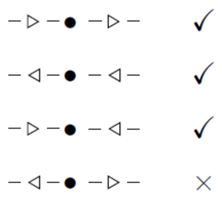


Figure 2.4: Composition of 3 colourings at mixed node

Note that after composition has been performed, the direction of the arrow on mixed nodes no longer matters: the colouring simply represents no data flow. (This fact is used to reduce table sizes.)

When looking back at the example of the LossySync followed by a $\text{FIFO}_1$. We will see in figure 2.5 that the wrong context dependent colouring given by 2-colouring is rejected by the 3-colouring because the colours at mixed node b do not match.

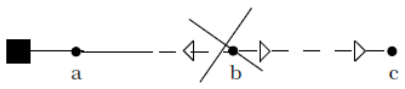


Figure 2.5: LossySync-$\text{FIFO}_1$ connector with 3-colouring

## 2.3 CA, Stochastic Reo and QIA

This section introduces Constraint Automata (CA), Stochastic and Quantitative Intentional Automata (QIA). This section contains a selection of the most important parts of the QIA paper [11].

CA [13] were introduced to express the operational semantics of Reo. Indeed, CA provide a unified model to capture the semantics of components and services, as well as Reo connectors and their composition. Quantitative Reo and Quantitative Constraint Automata (QCA) [10] extend Reo and CA with the means to describe and combine the QoS aspects of composed systems. The QCA model integrates the QoS aspects of components/services and connectors that comprise an application to yield the QoS properties of that application, ignoring the impact of the environment on its performance such as throughput and delays. While QCA provide a useful model for service selection and composition [16], the performance of a system can crucially depend not only on its internal details, but also on how it is used in an environment, as determined, for instance, by the frequencies and distributions of the arrivals of I/O requests which belong to stochastic aspects. However, such stochastic aspects are not investigated in [16]. Intentional Automata (IA) [15] take into account the influence of the environment as well as internal details of a system by describing the pending status of I/O operators interacting with the environment. A particular class of IA models, called the Reo Automata class, is defined in [15], which provides precise characterization of context-dependent connectors [13].

QIA is introduced as an extension of IA that allows for incorporating the influence of a system's environment on its performance. The QIA model extends the semantics of Reo by admitting annotations on its channel ends and the channels to represent the stochastic properties of request arrivals at those ends, dataflows, and data processing and transportation delays through those channels. The resulting *Stochastic Reo* model retains its compositional semantics through QIA: the QIA of a composed system is the product (composition) of the QIA of the individual channels and components/services used in its construction.

### 2.3.1 Constraint Automata

Constraint Automata (CA) were introduced [13] as a formalism to capture the operational semantics of Reo, based on timed data streams, which also constitute the foundation of the coalgebraic semantics of Reo [12].

We assume a finite set $\mathscr{N}$ of nodes, and denote by *Data* a fixed, non-empty set of data that can be sent and received through these nodes via channels. CA use a symbolic representation of data assignments by data constraints, which are propositional formulas built from the atoms '$d_A \in P$', '$d_A = d_B$' and '$d_A = d$' using standard Boolean operators. Here, $A, B \in \mathscr{N}$, $d_A$ is a symbol for the observed data item at node $A$ and $d \in Data$. $DC(N)$ denotes the set of data constraints that at most refer to the observed data items $d_A$ at node $A \in N$. Logical implication induces a partial order $\leq$ on $DC$: $g \leq g'$ iff $g \Rightarrow g'$.

A CA over the data domain *Data* is a tuple $\mathscr{A} = (S, S_0, \mathscr{N}, \rightarrow)$ where $S$ is a set of states, also called configurations, $S_0 \subseteq S$ is the set of its initial states, $\mathscr{N}$ is a finite set of nodes, $\rightarrow$ is a finite subset of $S \times \{N\} \times DC(N) \times S$ with $N \in 2^{\mathscr{N}}$, called the transition relation. A transition fires if it observes data items in its respective ports/nodes of the component that satisfy the data constraint of the transition, and this firing may consequently change the state of the automaton.

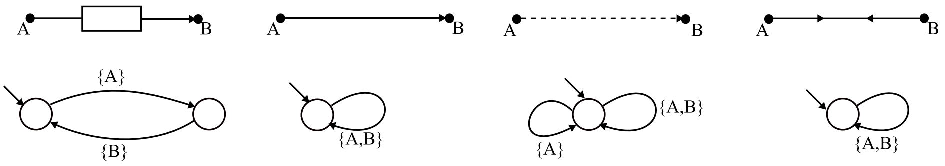


Figure 2.6: Constraint Automata for basic Reo channels

Figure 2.6 shows the CA for some primitive Reo channels. In this figure, for simplicity, we assume the data constraints of all transitions are true (which simply imposes no constraints on the contents of the data-flows) and omit them to avoid clutter. For proper full treatment of data constraints in CA, see [13].

### 2.3.2 Stochastic Reo

Stochastic Reo is an extension of Reo annotated with stochastic properties, such as processing delays on channels and arrival rates of data/requests at the channel ends, allowing general distributions. Figure 2.7 shows some primitive channels of Stochastic Reo. In this figure and the remainder of this paper, for simplicity, we delete node names, but these names can be inferred from the names of their respective arrival processes: for instance, 'dA' means an arrival process at node 'A'.

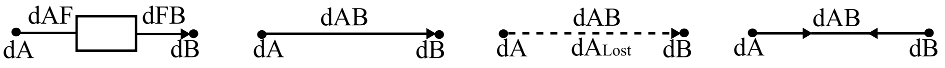


Figure 2.7: Basic Stochastic Reo channels

The labels annotating Stochastic Reo channels can be separated into the following two categories:

- **Channel delays**: To model the stochastic behaviour of Reo channels, we assume every Reo channel has one or more associated delays represented by their corresponding random variables. Such a delay represents how long it takes for a channel to deliver or throw away its data. For instance, a LossySync has two associated variables 'dAB' and 'dALost' for stochastic delays of, respectively, successful dataflow through the nodes 'A' and 'B' and losing data at node 'A' when a read request is absent at node 'B'. In a FIFO$_1$ 'dAF' means the delay for data-flow from its source 'A' into the buffer, and 'dFB' for sending the data from the buffer to the sink 'B'. Similarly, the random variable of a Sync (and a SyncDrain) indicates the delay for data-flow from its source node 'A' to its sink node 'B' (and losing data at both ends, respectively).
- **Arrivals at nodes**: I/O operations are performed on the source and sink nodes of a Reo circuit through which it interacts with its environment. We assume the time between consecutive arrivals of read and write requests at the sink and source nodes of Reo connectors depends on their associated stochastic processes. For instance, 'dA' and 'dB' in Figure 6 represent the associated arrival processes at nodes 'A' and 'B'. Furthermore, at most one request at each boundary node can wait for acceptance. If a boundary node is occupied by a pending request, then the node is blocked and consequently all further arrivals at that node are lost.

Stochastic Reo supports the same compositional framework of joining nodes as Reo. Most of the technical details of this join operation are identical to that of Reo. The nodes in Stochastic Reo have certain QoS information on them, hence joining nodes must accommodate their composition.

Nodes are categorized into mixed, source, and sink nodes. Boundary nodes receive data/requests from the environment, after that mixed nodes are synchronized for data-flow and then merely pump data in the circuit, i.e., mixed nodes do not interact with the environment. This account shows the causality of the events happening in the circuit, such as arrivals of data/requests at its boundary nodes, synchronizing its mixed nodes, and occurrences of data-flow, sequentially. Besides, we assume that pumping data by mixed nodes is an immediate action and therefore mixed nodes have no associated stochastic variables[1].

In order to describe stochastic delays of a channel explicitly, we name the delay by the combination of a pair of (source, sink) nodes and the buffer of the channel. For example, the stochastic property dAF of FIFO$_1$ in Figure 2.7 stands for the data-flow from the source end 'A' into the buffer of the FIFO$_1$. However, in cases where, for instance, a source node (as a replicator) A is connected to two different FIFO$_1$ buffers, then the corresponding stochastic processes have the same name, e.g., dAF. To avoid such an ambiguous situation, we rename the stochastic processes by adding a number after its node name like dA1F and dA2F when

[1]This assumption is not a real restriction. A mixed node with delay can be modelled by replacing this mixed node with a Sync channel with the delay. Moreover, according to the required level of specification detail, each input and output of the mixed node can be modelled by adding corresponding Sync channels with their stochastic values.

the node has more than one outgoing channel or one incoming channel. As an example of composed Stochastic Reo, figure 2.8 shows the ordering circuit with the annotation of its stochastic variables.

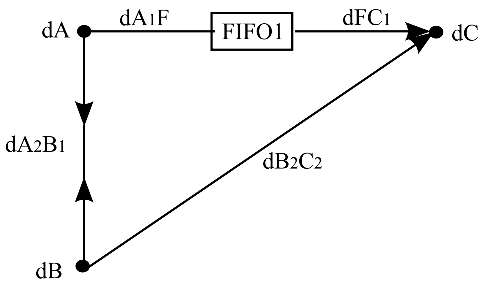


Figure 2.8: Ordering circuit in Stochastic Reo

### 2.3.3 Quantitative Intentional Automata

In this section we introduce the notion of Quantitative Intentional Automata (QIA) which is an extension of CA and provides operational semantics for Stochastic Reo. Whereas CA transitions describe system configuration changes, QIA transitions describe the changes of not only the system configuration but also the status of its pending I/O operations.

In CA, configurations are shown as states, and processes causing state changes are shown in transition labels as a set of nodes where data are observed. Similarly, in QIA, system configurations and the status of pending I/O operations are shown as states. Data-flow or firing through nodes causes changes in the system configuration, and arrivals of data/requests at the nodes or synchronization of nodes changes the status of pending data/requests. These two different types of changes are shown in the transition labels by two different sets of nodes. Moreover, QIA transitions carry their relevant stochastic properties in their labels. We use such QIA as an intermediate model for translation Stochastic Reo into a homogeneous continuous time Markov Chain (CTMC).

**Definition 2.3.** ***QIA***
*A Quantitative Intentional Automaton is a tuple* $\mathscr{A}=(S, S_0, \mathscr{N}, \rightarrow)$ *where*

- $S \subseteq L \times 2^{\mathscr{N}}$ *is a finite set of states.*
  - $L$ *is a set of system configurations.*
  - $R \in 2^{\mathscr{N}}$ *is a set of pending nodes, that describes the pending status in the current state.*
- $S_0 \subseteq S$ *is a set of initial states.*
- $\mathscr{N}$ *is a finite set of nodes.*
- $\rightarrow \subseteq \bigcup_{M,N \subseteq \mathscr{N}} S \times \{M\} \times \{N\} \times DC(N) \times 2^{DI} \times S$ *is the transition relation.*
  - $DI \subseteq 2^{\mathscr{N}} \times 2^{\mathscr{N}} \times \mathbb{R}^+$.

A transition in a QIA is represented as $\langle l, R\rangle \xrightarrow{M,N,g,D} \langle l^{'}, R^{'}\rangle$, where $M$ is the set of nodes that exchange data or synchronize for data-flow through the transition, $N$ is the set of nodes to be released by the firing of the transition, and $D \subseteq DI$ is the set of delay information tuples $(I, O, r)$ where I and O are sets of, respectively, source and sink nodes, and $r$ indicates the stochastic delay rate for the data-flow from $I$ to $O$ or the arrival rate of data/request from the environment at nodes in $I \cup O$. Furthermore, let $D = \{(I_j, O_j, r_j)|1 \leq j \leq n\}$, then $\bigcup_{1 \leq j \leq n} (I_j \cup O_j) = N \cup M$.

## 2.4 Current limitations of Reo

This section will give an answer to our first research question: What are the current limitations of Reo?

QIA was introduced as an extension of CA and provides operational semantics for Stochastic Reo. Stochastic Reo enables us to annotate stochastic properties in Reo including processing delays on channels and arrival rates of data/requests at channel ends, allowing general distributions. Although Stochastic Reo allows general distributions, when converting this to a Continuous Time Markov Chain (CTMC) using QIA we can only use exponential distributions because these have the property that they are memoryless. This memoryless property means that if a random variable T is exponentially distributed, then the following property holds:

$$P(T > s + t|T > s) = P(T > t) \text{ for all } s, t \geq 0. \tag{2.1}$$

This says that the conditional probability that we need to wait, for example, more than another 10 seconds before the first arrival, given that the first arrival has not yet happened after 30 seconds, is equal to the initial probability that we need to wait more than 10 seconds for the first arrival.

In a Markov Chain, the current state of the system should be completely independent of the past state and transitions, this is referred to as the Markov property. In a CTMC, this property can only hold if the distributions are memoryless. If we would have distributions without the memoryless property, this is not met any more so there will be dependency.

With the generated CTMC we can do steady state analysis, but the states in the CTMC are just numbered. For this reason, it is not obvious what every state represents. For example, one state represents the presence of a request at boundary node A and another represents the presence of a request at node A and B, but this can not be seen at first. Normally we also have to sum up a lot of different states to get the right statistic. For example, in the relative easy ordering connector of figure 2.8 discussed in the QIA paper, we need to sum up 8 different states to get the blocking probability at boundary node A.

When dealing with larger connectors, the number of states increases rapidly, which will make it much harder to derive the right statistics. With these Markov chains it is possible to derive all kind of statistics, but some of them are very hard to derive. For example, calculating the end-to-end delay will be very difficult for large connectors.

If we want to use non-exponential distributions, we can also not use the CTMC any more and we can not do any quality of service (QoS) analysis. With the provided automata model it is still possible to do some analysis like detecting deadlocks or checking if certain paths through the model are possible, but QoS analysis is not possible. Analytically it is also very hard to solve complex systems with synchronization and many different general stochastic distributions.

For this reason we need a new approach by simulating the Reo connector and deriving all sorts of results. In this simulator we are able to use the colouring algorithm described in section 2.2 as the driver of the simulator. With this simulator we will always use the 3-colouring algorithm and not the 2-colouring algorithm because, as we discussed in section 2.2.2 that colouring algorithm did not have the right context dependent behaviour. The simulation approach will be discussed in detail in chapter 3.

# Chapter 3

# Simulation

In the previous chapter we stated that we needed a simulator in Reo to perform quality of service analysis on Reo systems with general stochastic distributions. For this, we will use discrete event simulation. In discrete event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. In addition to the representation of system state variables and the logic of what happens when system events occur, discrete-event simulations normally includes the following components:

- **Clock**: The simulation must keep track of the current simulation time, in whatever measurement units are suitable for the system being modelled. In discrete-event simulations, as opposed to real time simulations, time hops because events are instantaneous  the clock skips to the next event start time as the simulation proceeds.
- **Event list**: The simulation maintains at least one list of simulation events. This is sometimes called the pending event set because it lists events that are pending as a result of previously simulated event but have yet to be simulated themselves. An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. It is common for the event code to be parameterised, in which case, the event description also contains parameters to the event code.
- **Random-number generators**: The simulation needs to generate random variables of various kinds, depending on the system model. This is accomplished by one or more Pseudorandom number generators. The use of pseudorandom numbers as opposed to true random numbers is a benefit because rerunning a simulation produces exactly the same behaviour.
- **Statistics**: The simulation typically keeps track of the system's statistics, which quantify the aspects of interest.
- **Ending condition**: Because events are bootstrapped, theoretically a discrete event simulation could run forever. So the simulation designer must decide when the simulation will end. Typical choices are 'at time t' or 'after

processing n number of events' or, more generally, 'when statistical measure X reaches the value x'.

In this chapter we describe the simulator of Reo. Section 3.1 gives an answer to our research question: Which output statistics are relevant for the systems we want to model? These statistics will be outputted after the simulation has been finished. Section 3.2 describes the general set-up and explain the choices we made for the simulation. Section 3.3 gives an answer to another research question: What are the limitations of the simulator or the approach used for the simulator? It describes the limitations we experienced because of the choices made in section 3.2, and it will explain how to work around these limitations and when the workarounds are needed.

## 3.1 Output of the simulation

In this section we will give an answer to our research question: Which output statistics are relevant for the systems we want to model?. Because Reo can model all kind of systems it is not always clear what the output should be. For all output statistics[1] we give an average, a standard deviation (by using the batch-means method for the simulation), a coefficient of variation and the number of observations of the statistic. We will also give a confidence interval for the mean of the statistic. For this confidence interval, we assume that the results of the different batches in the simulation are normally distributed. To see if this is the case, we added a histogram of the batch results, if this histogram does not look like the 'bell curve' shape of the normal distribution, the confidence interval will be wrong.

One statistic might be based on thousands of observations while others are based on only one observation, so the meaning of the statistic can be very different. So if the user observes that one statistic is based on only one observation, he knows that it is because of a very rare event.

With the simulation one may expect that the results and the number of observation in every batch is approximately the same, but if this is not the case this gives an indication that something might be wrong. For example, a deadlock will have highly different results for the time before the deadlock and the time after the deadlock. Also, if the results in the first batch are very different from the rest, you might have to use a longer warm-up period.

### 3.1.1 Long-term simulation

Long-term simulation can be used to get a lot of results. Long-term simulation has the advantage that only one warm-up period have to be used. The disadvantage of long-term simulation is that batches are dependent of each other. Because the state of the system will not be reset after every batch, the current batch will be dependent of the previous batch. This is especially noticeable when using a short

[1]An overview of the results of all statistics will be provided after the simulation in the folder 'Results' of the workspace directory in Eclipse.

simulation period, when using a long enough simulation period the batches will be approximately independent.

### System state

One thing we have as output is a system state for steady state analysis, so if you observe the system at a random moment in time, what is the probability that you observe a certain state. This will not mean that the system is actually stable after a certain amount of time, but more like a long-term average. For this output measure we have to know how a 'state' is defined.

We use the following definition for the system state:

**Definition 3.1.** ***System state***
*The system state is the conjunction of the state of the boundary nodes followed by the state of the FIFO buffers, where all individual parts will be ordered in an alphabetical order.*

The boundary nodes can be empty meaning that there is no waiting request, waiting meaning that there is a request waiting at the boundary node (port) or busy meaning it is busy sending data. The FIFO buffers can have only two states, full or empty. Because the system state will have the states of the individual parts in an alphabetical order, it is important to give names to the nodes in the connector. For example, if we have a connector with nodes A, B and C and 2 buffer the state can be 'ewbfe', where port A is empty, port B waiting, port C busy, buffer 1 full and buffer 2 empty.

When we have M ports and F FIFO buffers, this means we can have a total of $3^M \times 2^F$ possible states. This number can increase pretty fast when evaluating large system. However, normally not all states are possible, because a lot of states do not correspond to a colouring. For this reason we will only output the observed states.

This system state is also used for two extra options in the simulation. The first one is defining a system state to stop the simulation. In this specification it is possible to add the wild-card '?' to the state to indicate that certain parts of the state can be any value. For example, if you specify the value 'ww?' the simulation will stop if the simulation will reach the state 'wwe', 'www' or 'wwb'. Multiple states can be defined by separating them with a comma sign. Then the simulation will stop if it reaches any of the specified states.

The same specification can also be used to define a 'special state'. This 'special state' can be used to get statistics of multiple states together, so this will output the ratio of time the system is in one of the specified states. The mean of this statistic should be the same as the sum of the means of the individual states, but the standard deviation can not be calculated out of the individual statistics because these are dependent of each other.

### Channel utilization

We can also give the utilization of all channels, which will be defined as follows:

**Definition 3.2.** ***Channel utilization***
*The channel utilization is the time a channel is busy handling requests divided by the total simulation time. For the FIFO, Lossy, AsyncDrain and AsyncSpout, this channel utilization will be split into two parts with one utilization for every end.*

We also have to mention how to define the time a channel is busy. For example, if we take the connector of figure 3.1, we have three channels in series. When a request is available at A and at D, all three channels will be locked at the same time. If every channel takes 10 seconds to process, it will take a total time of 30 seconds for the item to go from A to D, so every channel is locked for 30 seconds. However, the separate channels are only busy for 10 seconds. For the channel utilization defined before we use the 10 seconds of the channel, for the 30 seconds we introduce another output statistic.

**Definition 3.3.** ***Channel locked***
*The channel locked utilization is the total time a channel is locked divided by the total simulation time. A channel is locked when a colouring is active involving the channel.*

The channel locked utilization uses the colourings to know when a channel is locked, for this reason we also need an output statistic which indicates the utilization of a certain colouring.

**Definition 3.4.** ***Colouring***
*The colouring statistic is the the total time colouring c is active divided by the total simulation time.*

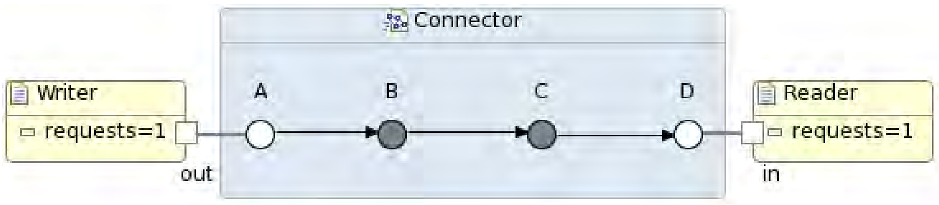


Figure 3.1: Channels in series

## Arriving requests

The next statistic we can give is about the arriving requests at an I/O operator (Read/Write). When we have an arbitrary request at a boundary node there are three possibilities:

1. The boundary node is in the state 'empty'
2. The boundary node is in the state 'waiting'
3. The boundary node is in the state 'busy'

In the first case, the request will be accepted at the boundary node and will be processed immediately if all other ends are available or it will wait if it has to wait for a sync at one of the other ends. If a request observes the boundary node in the state 'waiting' or 'busy', the request will be blocked. We can have two types of statistics based on the arriving requests which will be defined as follows:

**Definition 3.5.** ***Request observations***
*The request observations is the ratio of requests observing state $j$ at boundary node $i$ divided by the total number of arrived requests at boundary node $i$.*

**Definition 3.6.** ***Node state***
*The node state is the total time a boundary node $i$ is in state $j$ divided by the total simulation time.*

The node state indicates the probability that a request arriving at a random moment in time will observe one of the three states, while the request observations indicates the actual observations of requests. When the arrival process has an exponential distribution, both statistics should be equal because of the memoryless property.

When a request will be delayed until all sync ends are available, we also want to know the average waiting time. For this reason we have two extra definitions.

**Definition 3.7.** ***Average waiting time***
*The average waiting time is the total waiting time of all requests at boundary node $i$ divided by the total number of requests.*

**Definition 3.8.** ***Average conditional waiting time***
*The average conditional waiting time is the total waiting time of all requests at boundary node $i$ divided by the total number of non-blocked requests.*

### FIFO buffer full

The previous statistics are statistics you can have for any simulation program, but we can also have some statistics which are specific for the Reo simulation. An example of such a statistic is the FIFO buffer utilization which will be defined as follows.

**Definition 3.9.** ***FIFO buffer utilization***
*The FIFO buffer utilization is the total time a FIFO buffer is full divided by the total simulation time.*

### Loss ratio at lossy sync

The next Reo specific statistic is the behaviour of a lossy sync. The lossy sync can have two types of behaviour, it either behaves as a normal sync channel when there is no reason for delay, or it will lose its data when there is a reason for delay at the sink end. We will have the following output statistic for the lossy sync channel.

**Definition 3.10.** ***Actual loss ratio***
*The actual loss ratio is the number of requests lost in a lossy sync channel divided by the total number of requests arriving at the lossy sync channel.*

### Merger

Another relevant measure to look at, are the mergers. We can look at the ratio of times the data arrives from one of the directions. This measure can not be derived from the steady state analysis, because the steady state analysis indicates the percentage of time the system is in a certain state. For example, if we have the system as in figure 3.2 (where the numbers at the readers/writers means the number of non-blocked requests), you can see that the probability that an arriving item at C is from A is 99%. But when you look at the percentage of time A is busy sending to C this is only 10%. So we can define the statistic for the merger as follows:

**Definition 3.11.** ***Merger direction***
*The merger direction is the number of requests arriving at node $N$ from sink end $e$ divided by the total number of requests arriving at node $N$.*

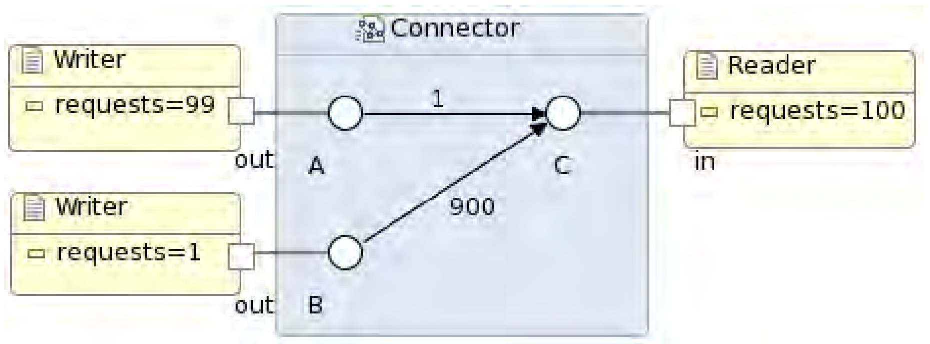


Figure 3.2: Merger

### End-to-end delay

Another very important measure in every system is the end-to-end delay and inter-arrival times, which we define as follows:

**Definition 3.12.** ***End-to-end delay***
*The end-to-end delay is the average delay from starting point $s$ to ending point $e$.*

**Definition 3.13.** ***Inter-arrival times***
*The inter-arrival times are the average time between two arrivals at ending point $e$, coming from starting point $s$.*

With this definition we also have to know what the possible starting and ending points could be. This is not always obvious when we have a system where every data item will be duplicated and arrive at the same endpoint. In figure 3.3 with the delay times displayed near the channels, you will see an example of such a system. When an item is duplicated, we can take the duplicator node as the starting point, or the original writer. We will choose the original writer, to be able to distinguish between different origins before the duplicator node. But a writer is not the only way the data could start, we can also start the system with a full FIFO buffer or the data can begin at a spout channel.

**Definition 3.14.** ***Starting point***
*A starting point for the end-to-end delay or the inter-arrival times is either a boundary node, full FIFO buffer or a spout channel.*

The data can also leave the system on other places than the readers; in lossy sync channels and in drains. The item can also become stuck in a FIFO channel, but then it will never get an ending time.

**Definition 3.15.** ***Ending point***
*An ending point for the end-to-end delay or the inter-arrival times is either a boundary node, LossySync, SyncDrain or AsyncDrain.*

If we evaluate the system of figure 3.3 we will see that one item arrives after six time units, while the other item arrives after eight time units. So in this case the average end-to-end delay for items from A to E will be seven, while the average inter-arrival time will be four.

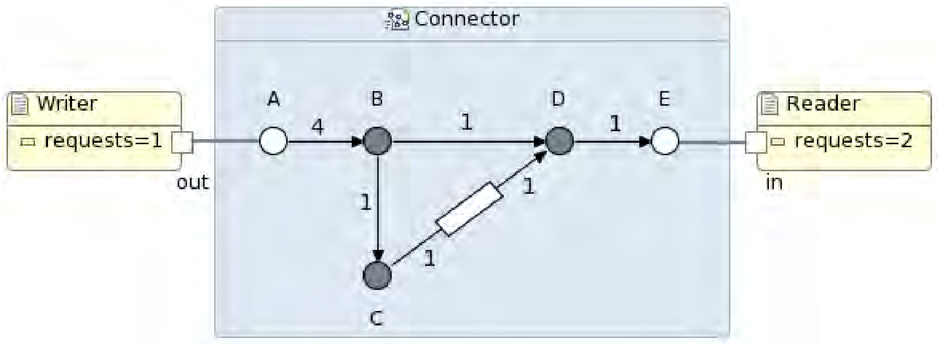


Figure 3.3: Duplicator connector

When a full FIFO buffer is the starting point of a token, the end-to-end delay is not relevant any more for that token. For example, when we look at the connector of figure 3.4. The token which starts in the FIFO will be replicated every time and will arrive at the reader after for example 1, 3, 5, .. time units. Because the starting time of the token will stay at zero the end-to-end delay will go to infinity when using a long simulation period. However, we have to keep it as a starting position because it can be relevant for the inter-arrival times.

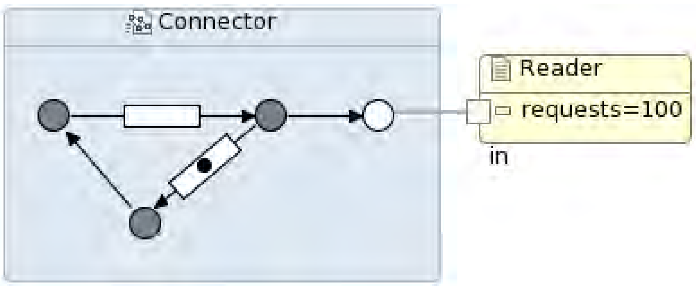


Figure 3.4: Replicating FIFO loop

### 3.1.2 Short-term simulation

By using short-term simulation we can also do some relevant analysis. The advantage of this type of simulation is that you can do transient analysis, deadlock and livelock analysis. The disadvantage of short-term simulation compared to long-term simulation is that you will have a warm up period in every batch.

#### Transient analysis

With short-term simulation we are able to do transient analysis. We can run the simulation to time X multiple times to see what the probability is that the system is in state Y after time X. With short term simulation it is still possible to define a warm up period, but if you create a particular starting situation you should probably set the warm up period to zero, because you will lose the starting position otherwise.

#### Deadlocks

Some systems might end up in a deadlock, meaning that the system is stuck and all flow is blocked. An example of such a system can be seen in figure 3.5, when both FIFO buffers are full the system is locked. This can happen after 3, 5, 7, 9, ... steps, or it can also be that it will never happen. So by using short-time simulation you can give some figures about (possible) deadlocks. For example, you can simulate over and over again until a deadlock occurs, then we can give the average time (or number of steps) it will take before the deadlock occurs. However, the problem with this approach is that a deadlock might be very rare and it could take an infinite time before the deadlock occurs. This will ruin the results generated by the simulation.

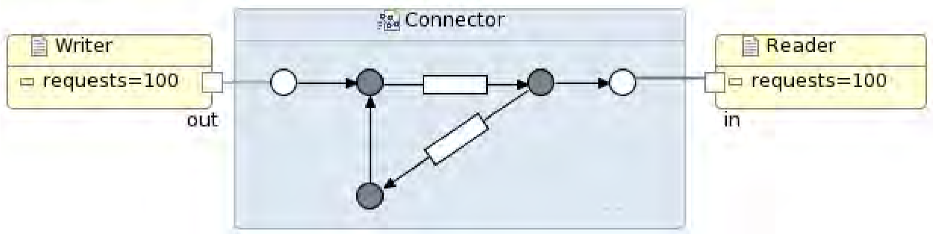


Figure 3.5: Connector with possible deadlock

This can be solved by using a maximum time (or steps) for the simulation, this can give you the probability that a deadlock will occur within X time units. This way you can also give an average time to the deadlock by using only the simulation runs where a deadlock actually occurs.

We also have to know when we are in a deadlock, but this is easy to detect. When we end up in a deadlock, the current colouring table will be empty. Normally you would still have a no-flow colouring, but these are filtered out before the simulation. One might think that a deadlock can also be detected by an empty event list, but that is not the case, because there are still requests coming in. So a deadlock is defined as follows:

**Definition 3.16.** ***Deadlock***
*A deadlock indicates that a system is stuck and all flow is blocked, a deadlock can be detected by a colouring table with only no flow colourings.*

We can also define another kind of deadlock, when we have an internal loop where the flow circles around but all requests at the ports collides, this is called a livelock. An example of such a system is in figure 3.6. This connector will have a constant change of colouring, but every time the data will be lost in the lossy sync because the data in the other loop is in the other buffer. For this reason the data will never reach the readers.

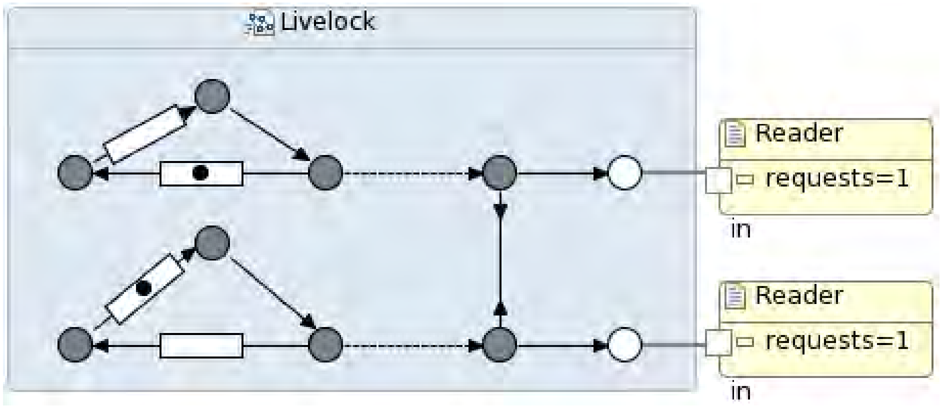


Figure 3.6: Connector with livelock

We will define a livelock as follows:

**Definition 3.17.** ***Livelock***
*A livelock is a special kind of deadlock, where there is internal flow in the connector without any action at the boundary nodes.*

In general, it is hard to say when we actually have a livelock, it might happen that we do not have any action on a boundary node for a long time, but once a request arrives at a boundary node with a large inter-arrival time the boundary nodes will have action again. So the way we will detect a livelock is by counting the number of colourings chosen without any involved boundary nodes. Whenever we choose a colouring with an involved boundary node, this counter will be reset again. Another restriction is that we start counting when all boundary nodes have a waiting request.

## 3.2 Simulation set-up

This section describes the general set-up of the simulator in Reo. Figure 3.7 shows the flow of the long-term discrete event simulation, and figure 3.8 shows the flow from the short-term simulation. In the next paragraphs parts of these flow charts will be explained in more detail.

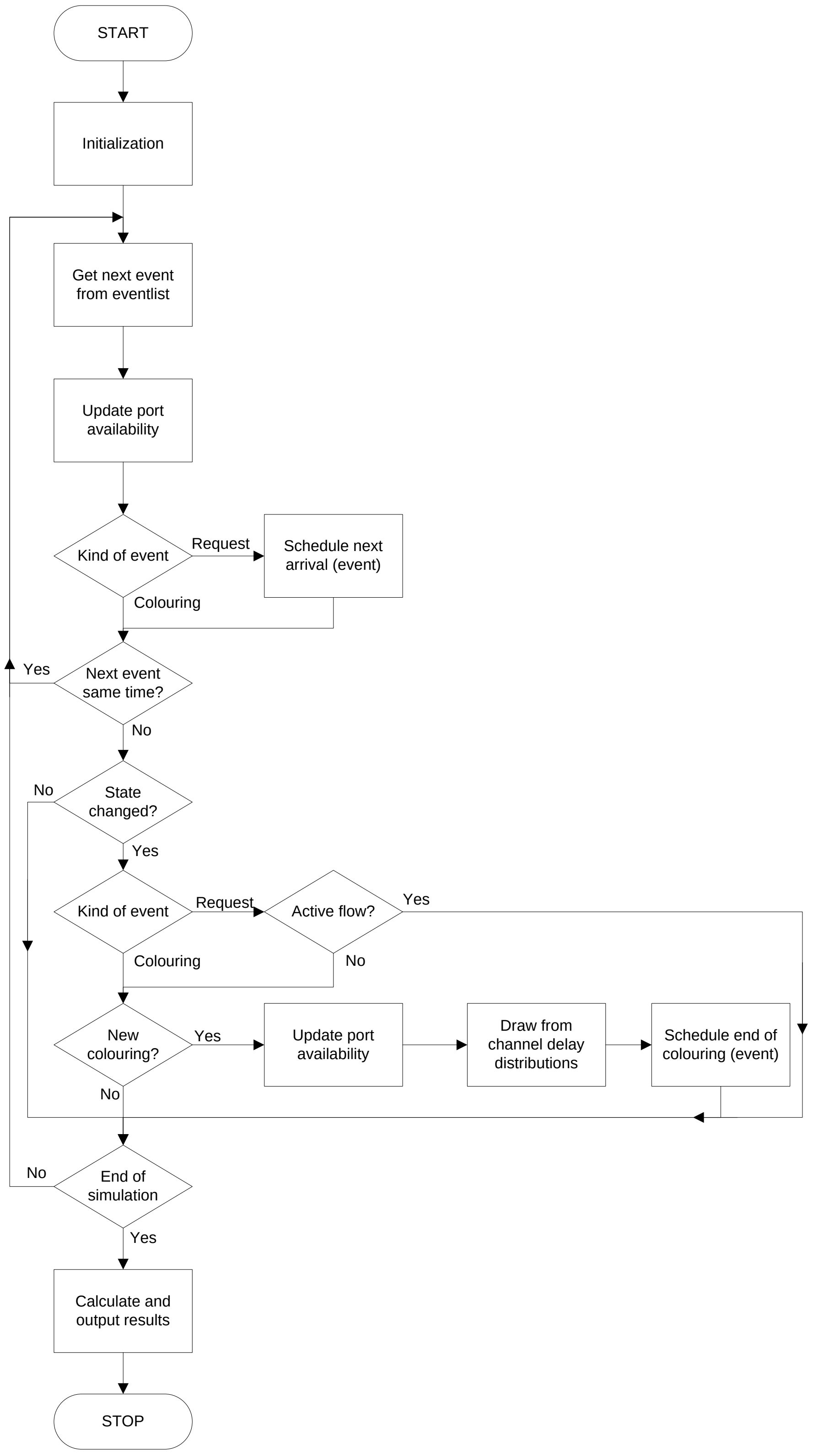


Figure 3.7: Flow chart for the long-term simulation

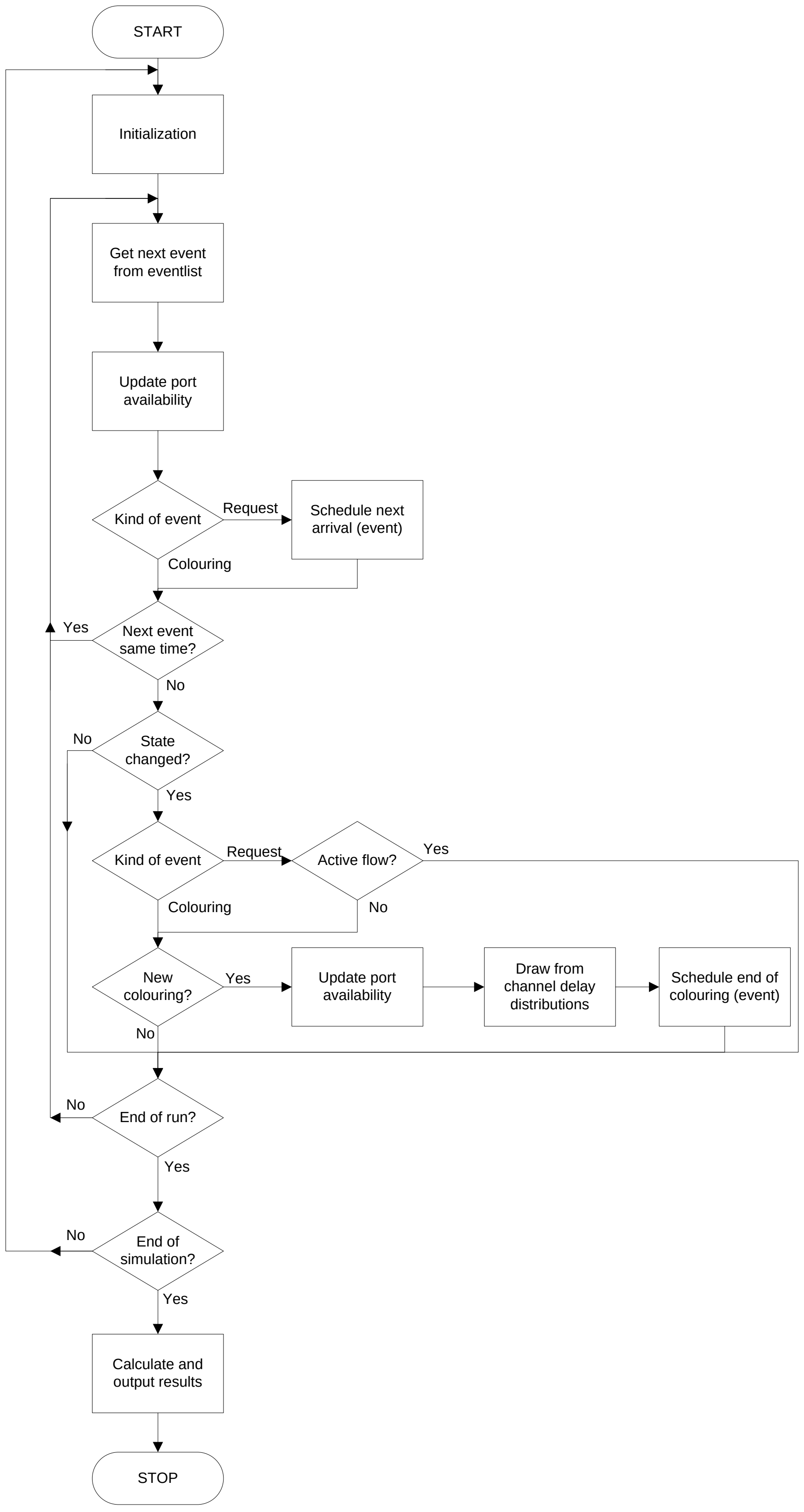


Figure 3.8: Flow chart for the short-term simulation

### 3.2.1 Input

The input of the simulation can be split into two parts, one part is setting up the Reo model and the other part is setting the parameters for the simulation.

#### Reo model

The first type of input is the Reo model itself, the user has to build the Reo model he wants to simulate. Next to the usual Reo model, the user will also have to specify some stochastic properties in the model. The delay of all the channels could be set, but if the user does not specify a specific distribution the simulation model will use a delay of zero. Four of the channels will also have two separate delays, for the FIFO channel we have the delay into the buffer and out of the buffer. For the lossy sync channel we have the delay when it behaves as a sync channel and the delay when it loses its data. The AsyncDrain and AsyncSpout both have two delays, one delay for every end of the channel.

Another thing the user has to specify is the distribution of the inter-arrival times of requests at the boundary nodes. It is also possible to specify if the boundary node should start with a request. This ability gives the user the possibility to create any starting situation. This should not have a (large) impact on any long-term simulation because of the warm-up period, but for short-term simulation this could be very handy. Setting this starting state of the ports can be done by specifying a boolean value.

#### Settings

Before the user starts the simulation the user has to set up his simulation run. The following parameters have to be set:

- Type of simulation: long- or short-term simulation
- Base simulation end on: events or time
- Warm-up period: time or number of events till the simulation starts gathering statistics
- Simulation length: time or events till the simulation stops
- Number of batches: the simulation length will be split into multiple batches to be able to give a confidence interval for the statistics. The number of batches is normally chosen between 25 and 50 [17]
- Confidence interval: how accurate the confidence interval should be
- Detect deadlock: if enabled, the simulation will stop whenever we are in a deadlock. If disabled, the simulation will go on so the user can see what happens with the statistics after the deadlock.
- Detect livelock: ability to specify if a livelock should be detected
- Internal colourings for livelock: specify how many colourings in a row without any involved boundary nodes should be chosen to indicate a livelock. Also see section 3.1.2
- State to stop simulation (optional): possibility to define a certain system state in which the simulation should stop. Also see section 4.5

- Special state (optional): possibility to define a system state to get statistics from. Also see section 3.1.1
- Seed: define a seed if you want to produce the same results in every consecutive simulation with the same parameters
- Max chart points: maximum number of chart points for the charts. Also see section 4.6.1.

Choosing an appropriate warm-up period and simulation length might be very hard to do. When a certain simulation time has been chosen it is very hard to predict the number of events that will happen in the time period because the kind of systems used can vary a lot. For this reason it might be better to use events.

### 3.2.2 Statistics

During the simulation we have to keep track of all kind of statistics to be able to derive all output defined in section 3.1. The variables and their description can be found in table 3.1, with these primary statistics we are able to derive all other statistics. In this table, we can replace run r by a batch number if we are using long-term simulation. Every statistic will also have a count with the number of observations in every batch or run.

| *Variable* | *Description* |
|---|---|
| $B_f(r)$ | Total time FIFO buffer $f$ is full in run $r$ |
| $C_c(r)$ | Total time channel $c$ is busy with transferring data in run $r$ |
| $A_a(r)$ | Total time colouring $a$ is used in run $r$ |
| $D_{se}(r)$ | Total end-to-end delay from starting point $s$ to ending point $e$ in run $r$ |
| $S_{ij}(r)$ | Total time port $i$ is in state $j$ in run $r$ |
| $Sys_j(r)$ | Total time the system is in state $j$ in run $r$ |

Table 3.1: Variables

Note that for some channels (FIFO, lossy sync and Async channels) the variable $C_c(r)$ will be split into two parts. For the FIFO buffer we have the flow from the sink end to the buffer and from the buffer to the source end. In the lossy sync we have the time it behaves as a normal sync channel and the time it loses its data.

### 3.2.3 Initialization

During the initialization step we have to initialize all the variables and schedule the first events. Because the user can define if there is a waiting request at a port at the start of the simulation we have two possibilities. If the user has defined that there is a request waiting at a port, we will schedule the arrival of the request at time zero. If the port is empty at the start of the simulation we will schedule the first arrival at a time which will be sampled from the arrival distribution of the port.

Note that the initialization will not include the warm-up period, because during the warm-up period we have to use the normal flow as defined in the flow chart. However, during the warm-up period the statistics will not be updated yet.

For the short-term simulation we will arrive in the initialization process after every run. After every run the state of the system might be different from the specified state in the Reo model, so this state has to be reset first. Another thing we have to do is empty the event list, because after every run you will have to start with an empty eventlist again. A flow chart of the initialization process is given in figure 3.9.

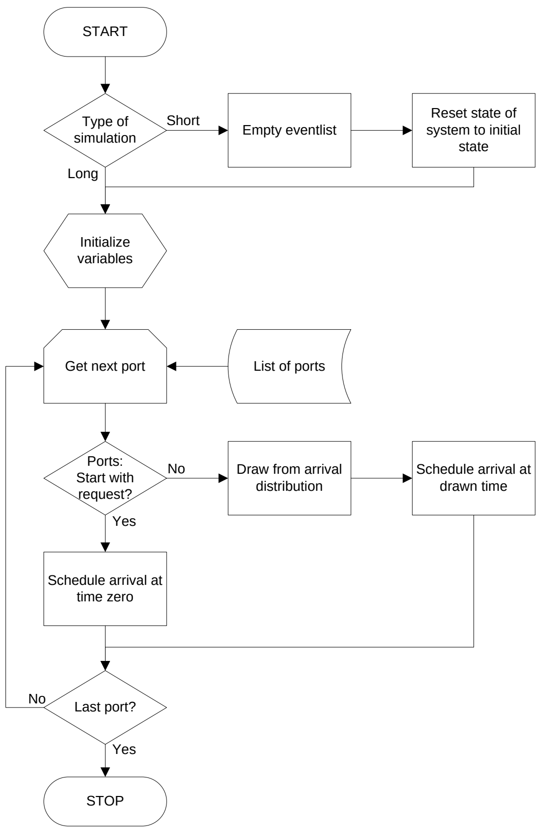

START
Type of simulation
Short
Empty eventlist
Reset state of system to initial state
Long
Initialize variables
Get next port
List of ports
Ports: Start with request?
No
Draw from arrival distribution
Schedule arrival at drawn time
Yes
Schedule arrival at time zero
No
Last port?
Yes
STOP

Figure 3.9: Flow chart for the initialization step

### 3.2.4 Events

As we can see in the flow chart of figures 3.7 and 3.8, there are two possible events in the event list. We can have the arrival of a request or we can have a finished colouring. Because we are using discrete event simulation, the state of the system might only change when one of these kind of events happens. We should keep track of a chronological list of events to find the next event to simulate. After every event we have to check if we want to end the simulation.

When we have multiple events at the same time, these events have to be processed in a specific way. First of all, the finishing of a colouring should be the first event to handle. This can be explained by taking a simple connector with a sync channel from A to B. If the flow from A to B finishes at the same time as an arrival at A or B, the behaviour of the system will be different depending of which event will be processed first. If the request will be processed first, this request will be blocked because the colouring has not ended yet. If the end of the colouring has priority over the request arrival, the channel will be idle first and the arriving request will be accepted and the corresponding port will go to waiting.

Before choosing a new colouring, all events of the same time should be processed. This will be a maximum of M + 1 events, where M is the number of ports and 1 from an end of colouring event. When a colouring would be selected before **all** ports are updated by the various events, we could get the wrong results. For example, if we have the connector of figure 3.10 and the requests at all ports will arrive at the same time. If we handle this requests one at a time and try to choose a colouring after every request, we will never choose the colouring we want (with flow from A to C and from B to the buffer).

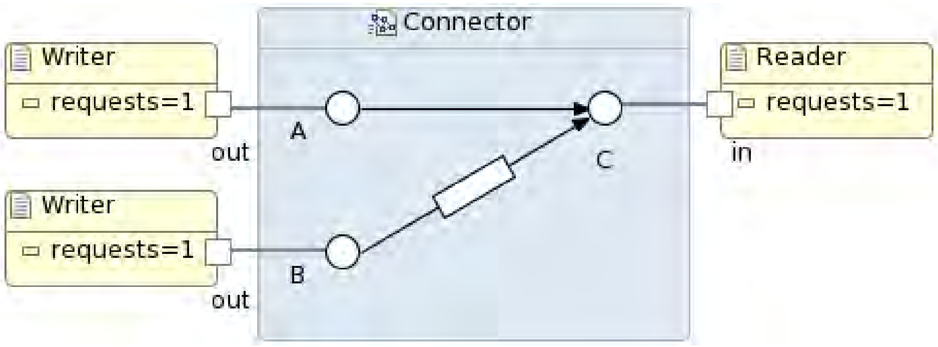


Figure 3.10: Connector with multiple flows in one colouring

The two types of request will be explained in more detail below.

### Request arrival

The first type of event is the arrival of a request at one of the ports. Recall that this request can observe the port in one of three states: empty meaning that there is no waiting request, waiting meaning that there is a request waiting at the port or the port can be busy meaning it is busy sending data. If the port is empty we can update the port availability to waiting. If the port is in the state 'waiting' or 'busy', the request will collide and will be lost.

After we have updated the port availability, we will also have to schedule when the next request will arrive at this port. This will be done by sampling from the arrival distribution at the port.

The last thing we have to do before we want to choose a new colouring is checking if there is already an active colouring with flow. Because we only use one colouring at a time and we will wait till this colouring is finished, we can only choose a new colouring if there is no active colouring at the moment.

**Finished colouring**

Another kind of event is the finishing of a colouring, when we have this event we have to update the availability of all the busy ports to empty. When the colouring includes a flow from or to a FIFO buffer, the buffer should also be updated. When we have flow into the FIFO buffer we will assume that the buffer is empty until the flow is finished. The conversely around, we will assume that the buffer will stay full until the flow is finished.

### 3.2.5 Next colouring

The most important part of the simulation is choosing the colourings. We will choose a new colouring when the last colouring is finished or when the state of the system changes because of an arriving request. But once a colouring is started, we will **never** choose another colouring before the current colouring is finished completely.

For example in the previous connector in figure 3.10 we have requests at A and C, which causes that we activate a colouring with flow from A to C. During this colouring a request at B arrives such that a flow from B to the buffer could also start together with the flow from A to C. But because we use only one colouring at a time until it is finished, we will not activate the flow from B to the buffer yet.

We have chosen to stick with this restriction because this is how Reo is designed and also how the colouring tables and colourings should be used. In every state of the Reo model, all nodes will come to a consensus about what happens next. Then this transition will happen and you are in a new state. Then we will decide what the next step should be and then this transition will be taken.

The colouring tables and colourings are also based on this principle. Every state of the system corresponds to a certain colouring table. This colouring table contains one or more possible next colourings which corresponds to a transition to a next state of the Reo system. The colouring contains a reference to the next colouring table which corresponds to the state of the system after the colouring is finished.

Based on the current colouring table we can choose the next colouring based on the availability of requests at the boundary nodes. All colourings in the current colouring table will be evaluated and can be rejected for two reasons:

1. The colouring has flow at a boundary node, and the boundary node has no waiting request
2. The colouring indicates that a boundary node gives a reason for delay, and there is a waiting request

If one or more of the colourings are accepted, one of these colourings will be chosen at random. After the colouring has been finished we will be at the next colouring table, which is specified with the chosen colouring.

As we can see in section 3.3, the restriction to stick to one colouring at a time until it is finished has some big drawbacks. But we could not drop this restriction, because then the colourings and colouring tables could not be used any more. When dropping the restriction, a whole new automata model should be invented with much more states which should be used as the driver for the simulation. More about this will be discussed in section 7.1. With some workarounds we are still able to model most of the systems we wanted to model, also with the current restriction.

### 3.2.6 Batches

When moving from one batch to the next batch we have to think about what belongs to which statistic. Counters based on instantaneous events are easy to handle, because the event is always in one of the batches. However, other statistics can begin in batch i and finish in batch i+1, so we have to think about what we should do with these statistics. The different types of batch behaviour for the statistics will be discussed here.

#### Instantaneous statistics

As mentioned before, for the instantaneous counters it is easy to place the event into the right event. This group includes the conditional probabilities (request arrivals and loss ratio). This group will also include the end-to-end delay and the inter-arrival times at a port, because for this statistic we can use the arrival time of the token.

Another statistic we have in this category is the merger directions, for this statistic we will determine the right batch based on the starting time of the flow (while the real arrival time at the merger might be in the next batch).

The actual channel utilization will also be in this category. If we look back at the connector in series of figure 3.1, we might have that the flow from A to D start in batch 1 and arrives in batch 2. So, if the batch change happens when the flow is going from B to C; we might want the channel utilization of A to B into batch 1, the flow from B to C separated over batch 1 and 2, and the flow from C to D into batch 2.

The problem with this is that we do not have separate events for the completion of flow in a channel. For this reason we will put all observations into the batch of the starting time of the flow. Because the goal of the simulation is to give an expected behaviour in the long-term, it should not make a big difference if we do it all in one batch or separated over multiple batches.

#### Split intervals

Other statistics might be based on intervals, so an observation might begin in batch i and finish in batch i+1. An example of such a statistic is the probability that an arbitrary incoming request will be in one of the three groups. The port we look at might begin in the empty state before the end of a batch until after the beginning of the next batch. If this is the case you have to think about what to do with this observation. One solution could be to leave out the entire observation, so once an observation start and ends in different batches you will leave out the observation.

Another possibility is to just put the observation in either batch i or in batch i+1, then you will not lose any data, but this can lead to a utilization over 100%. The last possibility is to split the observation, so the part which belongs to batch i will count in that batch and the part which belongs to batch i+1 will count in that batch.

We have chosen the last option because this does not lose data and it will also never exceeds 100%. If we would have chosen one of the other options, the results should not differ much because the results will only differ by at most one observation. If the results differ, the simulation period was not long enough. This can be seen by looking at the number of observations. If this number is not high enough, the method for choosing a batch for the intervals can make a difference.

**End of run and last batch**

When a run in short-term simulation or the last batch in long-term simulation is finished, the system might not be empty. So there could be active flows and waiting request, meaning we also have to decide what to do with these figures. For example, for the waiting time of a request we had specified that the time will be in the run or batch in which the request arrived. But because we want to stop the simulation after the last batch this request will never be processed.

So if you want to be strict with the runs and batches, you should simulate further until all waiting requests are gone and the active flows are finished. But maybe this can take a lot of extra simulation time. For this reason we will just throw away this partial waiting time, because it should not influence the final result. If it does influence the results the simulation period was not long enough.

## 3.3 Limitations and workarounds

In section 3.2.5 we have chosen that only one colouring will be active at a time until it is finished completely. Although this is the way Reo is designed, this approach gives problems when handling models with other assumptions. So in this section we will answer the research question: What are the limitations of the simulator or the approach used for the simulator?

For example, if we want to handle some basic queueing models, the Reo connector can not be build as expected. Most of the times some workarounds have to be used to make sure that the connector behaves as we would like it to behave. When there are alternative ways to choose colourings, these workarounds might not be needed any more, but until this time, the connectors can not be build in a different way. In this section we will explain what kind of limitations there are with the current approach and how to work around these limitations.

### 3.3.1 Exponential service duration

Figure 3.11 shows three FIFO buffers in series, this connector can be used to model a queueing model with four waiting places[2] and one server.

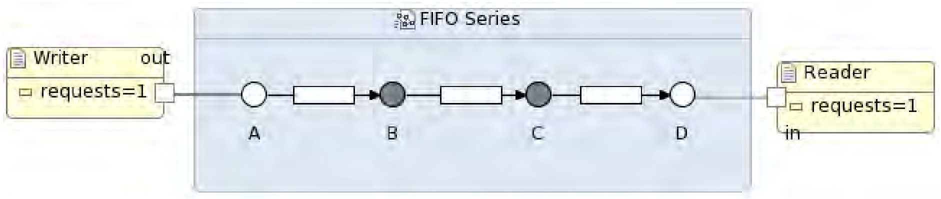


Figure 3.11: FIFO channels in series

An obvious idea would be to define the arrival rate on boundary node A and the service duration on the second part of the FIFO buffer from C to D. The rest of the channel delays should be set to zero and the arrival rate on the boundary node D should be set to 'IfNeeded' or 'Always'. Please refer to section 4.1 for a description of 'IfNeeded' and 'Always'.

But setting the service delay on this channel would not produce the right results. The reason for this is that once a colouring is accepted, it will not choose the next colouring before the current colouring is finished completely. So if the connector has started with processing its data from the last buffer to D, it is not able to process data from A to a buffer until the colouring is finished. So even if the buffers are empty a request at A will not move into a buffer until the 'service' has been finished. If, in the meantime another request arrives at A, it will be blocked while this was not needed because there was place in the queue.

To work around the restriction of the simulator, all channel delays should be set to zero and the service rate can be defined on the boundary nodes D. Now the availability of these boundary nodes indicate if the server is busy. When the state of the node is 'waiting' the server is idle and waiting for a request to process. When the state of the node is 'empty' it means that the server is actually busy processing a request. This might be counter intuitive, but at this moment there is no other alternative.

An important thing to note with this approach is that the end-to-end delay does not include the service duration. But the expected service duration can be added to the outputted end-to-end delay to get the actual end-to-end delay including service.

### 3.3.2 Non-exponential service duration

The approach suggested in the previous paragraph will work fine when dealing with an exponential service rate because of the memoryless property which is explained in section 2.4.

Because of this memoryless property, we do not care about the time the boundary

[2]the boundary node also has a place for the queue

node went from 'waiting' to 'empty'. The time until another request arrives is independent of this. When using another distribution, the memoryless property does not hold any more which causes that the previous approach fails to produce the right results.

Normally the arrivals of requests at a boundary node is a continuous process, so when a request arrives, the next request will be scheduled. For example, we want to model a single server without a queue[3] with a service duration of 4 using the workaround of the previous section. Then we have an arrival of a request every 4 seconds. We start with a request, which is reasonable because a waiting request indicates that the server is idle. So at time zero we will have the first arrival and we schedule the next arrival which will be at time 4. At time 4 we will schedule the next arrival which will be at time 8, and so on.

Now we also add an arrival rate of 1 request every 3 seconds. So a request will arrive at time 3, 6 and so on. The first arrival at time 3 will find the server in a waiting state so this request will be processed. Now the server is empty again until time 4 when the server will have its next request. At time 6 the next request arrives which also finds the server in a waiting state so this request will be processed immediately also.

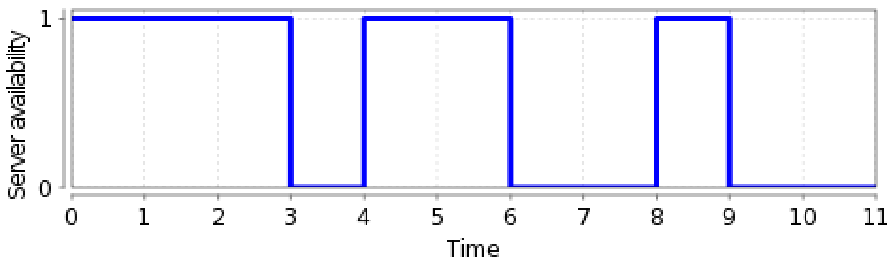
This behaviour is not what we would like to have, the service should start once it starts processing an item. So when a request will be processed at time 3, we expect that the server is available for another request at time 7 because we had a service duration of 4. So the server should not be available at time 4 already as explained above. When the server was available at time 7 instead of time 4, the request at time 6 will be lost in the LossySync channel. Figure 3.12 illustrates the different types of behaviours, where figure 3.12(a) indicates the behaviour when we use a regular deterministic distribution with inter arrival times of 4. Figure 3.12(b) indicates the behaviour we would like to have.

To solve this problem, we created an extra option to add a boolean value to the arrival rate to indicate that a new request should be sampled when a boundary node changes to empty. This indicates that a sample is made when the previous request is completely processed instead of when the previous request arrived. The request arrival behaviour can be changed by adding a 'true' value to the arrival rate distribution. For example by setting the arrival rate to Constant(4, true), if the boolean is omitted the regular Constant(4) distribution will be used.

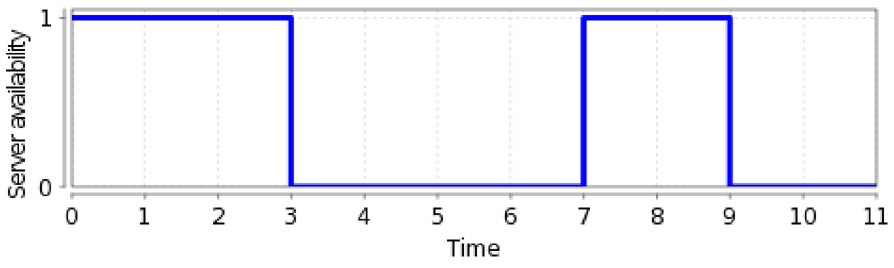
### 3.3.3 Waiting time after service

When we want to model systems where the server is **not** the end station of the request, so the request will not leave the system immediately after the processing has been finished, we can not use the previous approach any more. This is the case when a job will be handled by multiple servers or when a job will be processed by one server, and that it has to wait in a FIFO buffer until another request arrives at one of the boundary nodes after the processing in the server. In the previous approach, we modelled the system such that a job left the system when starting service, while the server itself will be blocked until the service is finished. This time was not in the end-to-end delay of the job itself any more, so this time should

[3]A server without a queue should be modelled using a LossySync channel



(a) Behaviour when service duration has been set to 4



(b) Behaviour when service duration starts when server is activated

Figure 3.12: Timeline for server with requests every 3 seconds

be added.

When a job has to wait after the processing at the server, this approach does not work any more because the waiting time and the service duration overlaps. For example, we can model a server with a constant service duration of 3 followed by a buffer. If we assume that a request arrives at time 0 to be processed in the server, and another arriving request at time 2 to read the request from the buffer. If we would model this system with an extra reader as specified in the previous section, the request will be in the buffer following the server after time 0, and the buffer will be blocked till time 3. After time 2 a request arrives to read from the buffer, so the request would get an end-to-end delay of 2. If we would add the service duration of 3, we would get an end-to-end delay of 5, which is obviously not the right result.

Another example of a system which would not give the right results when we would model it with just an extra reader is described in section 6.1 where a job will be processed by multiple servers and merged when all servers are finished. In this case a job will also have to wait after it has been processed by the servers.

To fix this problem we have to make a construction which will include the service duration(s) in the end-to-end delay. This can be achieved by using the construction as shown in figure 3.13. This example shows how you can model a service time duration between A and B, this is needed when the distribution of B is different than 'IfNeeded' or 'Always' because in that case, the request will have to wait after service. When the request is in the buffer it means that it is in service and once it left the buffer the service duration is finished.

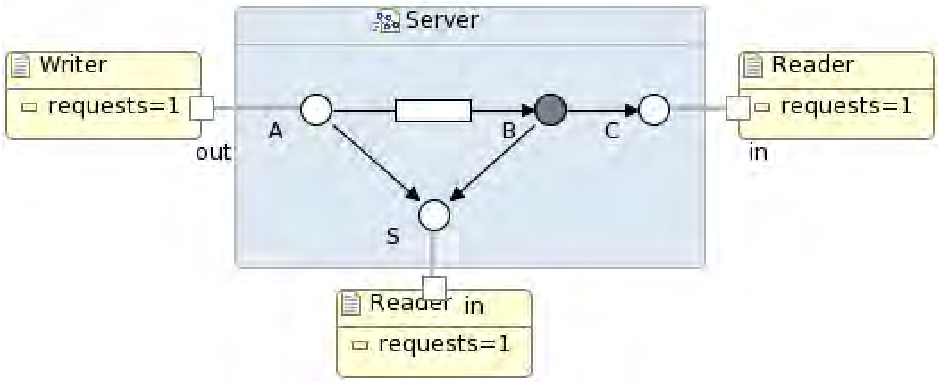


Figure 3.13: Construction for server

To make sure that this construction does what it should do, we needed another option for specifying the arrival rate of the boundary node S. We want that a request will flow into the server instantly once the server is free, and once the request is in the buffer we want to specify the service time distribution. To ensure this, we added another boolean value to the distribution to indicate that a distribution should iterate between zero and a sample of the distribution.

When the buffer is empty, the service is idle and should be waiting for an arriving request. So when the buffer is empty there should be a waiting request at the boundary node S. When a request arrives to flow into the buffer it will take the request from the boundary node S and will flow into the buffer. Then the request has to wait inside the buffer until another request is available at the boundary node S, the time till this request arrives will be sampled from the distribution of the boundary node. When this request is available at node S, the request in the buffer will leave and the server is idle again. Because the server is idle again we have to make sure that a request is available at node S. Because we have specified an alternating distribution at the boundary node, the next sample will be zero which indicates that the server is waiting again right after the last item left.

Although we restricted us to alternate between zero and a distribution, you might want to alternate between two different distributions in other examples. This can be achieved by having two different alternating distributions. For example, imagine that we want to alternate the distribution at boundary node A between an exponential(1) distribution and a uniform(0, 2) distribution. We can add sync channels from node A to nodes B and C and specify an alternating exponential(1) distribution[4] on node B and an alternating uniform(0, 2) distribution on node C.

Both distributions should not alternate in the same way, when one of the distributions gives a zero, the other distribution should sample. This can be achieved by specifying that one of the node starts with a request at its boundary node.

The construction of figure 3.13 can be used to model all delays in a connector, which makes the connector completely asynchronous. But this kind of construction also comes with a big disadvantage. Because you make the connector com-

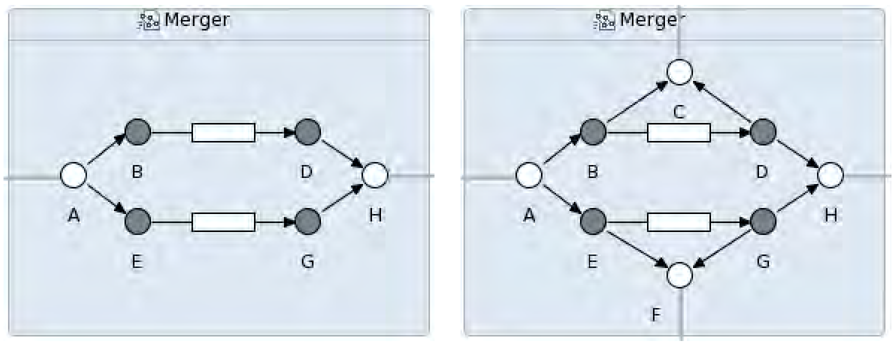
[4]An alternating exponential(1) distribution alternates between 0 and a sample from the exponential distribution and should be specified as exp(1, true, true)

pletely asynchronous, the number of colourings and colouring tables will increase rapidly, which slows down the simulator. When the number of buffers increases too much, there will also be problems with the memory usage which will be explained in section 6.1.

### 3.3.4 Mergers

Although we suggested that the method in the previous section could be used for all delays, the results can differ with the results with the delays on the channels. When looking at the merger given in figure 3.14(a) with a constant delay of zero on all channels except the channels D-H and G-H. In this case we assume that these two channels both have a delay of 1. Then the average end-to-end delay will be 1.5, because the request will flow instantly into both buffers. Then only one buffer is able to send to the reader which takes 1 time unit. Then the other buffer will fire, which added another time unit so this delay will be 2.

When we replace the delays with its counterparts as explained in the previous section, we got a system as in figure 3.14(b). Here all delays are zero and the arrival rate on the nodes C and F will be constant(1, true, true). In this case the average end-to-end delay will not be 1.5, but 1. The request will flow into both buffers which causes the boundary nodes to be empty. After 1 time unit both boundary nodes will have a request again and the flow to H can start again. Because the delays are set to zero, the end-to-end delay will also be one.



(a) Normal merger (b) Merger with separate delays

Figure 3.14: Different ways to model mergers

### 3.3.5 SyncDrains

Another connector which does not work as expected is given in figure 3.15. This time the error has nothing to do with the restriction that only one colouring is accepted at a time, because this connector has only one colouring. This time the problem is the way the connector will be traversed to get the end-to-end delays. We can evaluate this connector with arrival rates set to 'Always' and deterministic delays, where $A \rightarrow B = 1, A \rightarrow C = 2, B \rightarrow C = 0$ and $B \rightarrow D = 0$.

The end-to-end delay to node D will be 1 while you might expect it to be 2, because

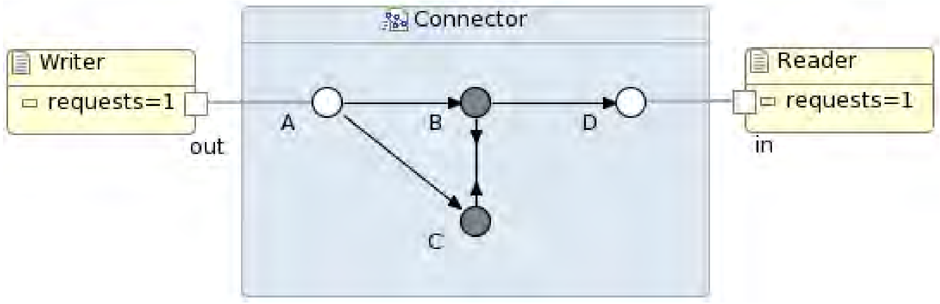


Figure 3.15: Connector with SyncDrain

an arriving request will be duplicated to two different Sync channels. One of them takes 1 time unit while the other one takes 2. Then it will be drained after 2 time units, so you might expect that this will also be the end-to-end delay to node D.

The reason why we get an end-to-end delay of 1 is that the flow from B to D starts before the flow from B to the drain starts. So whenever a request reaches B, it will be duplicated and one of the tokens will flow directly to D. The other token will wait until C is also available, because that is how the SyncDrain channel works. The request from B to D could not wait until C is available also, because this would not work in some cases. For example, if we change the Sync of A-B into a SyncDrain and the SyncDrain of B-C into a Sync from C to B. Now a request will never arrive at the other end of the SyncDrain if the request will not be sent whenever the other end is not available.

If you do want to have an end-to-end delay of 2, this can be solved in two ways. The first way is by changing the Sync channel from B to D into a FIFO channel. Then a request will wait in the buffer until the colouring has been finished. Because the colouring is finished after the drain is finished, the delay from A to D will also be this time. The second way to solve this is by changing the two Sync channels from A to B and A to C into a FIFO channel with two Sync channels to a reader as suggested in section 3.3.3.

# Chapter 4

# Implementation

In this chapter we discuss how we implemented the Reo simulator within the Eclipse Coordination Tools. Figure 4.1 gives a high level overview of the simulator in relation to the ECT and colouring semantics. The simulator is integrated in the Reo perspective in Eclipse and it uses the colouring semantics as the driver for the simulation. The simulator itself consists of three important parts, the specified distributions, the statistics gathering and the delay calculation algorithm. Furthermore, a simplified version of the data model can be found in appendix A and the source code of the simulator can be found in [6].

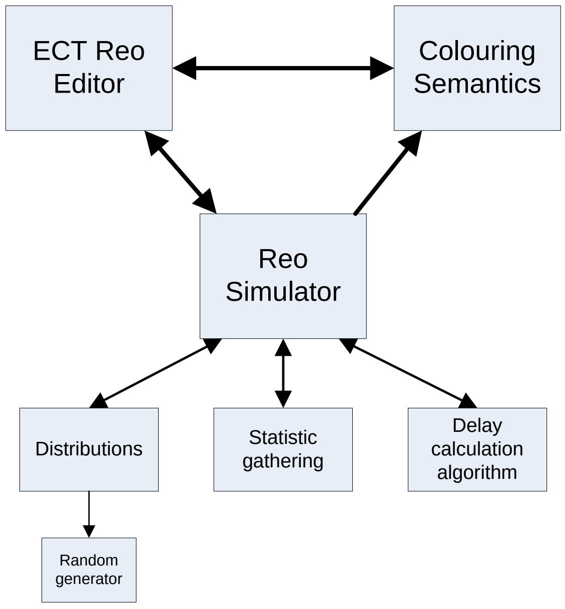


Figure 4.1: High level overview of simulator

In section 4.1 we introduce the distributions which can be used for the simulation, it will also explain some special cases created to model the systems as we wanted. Section 4.2 explains how we will choose a colouring given a certain state of the system. Section 4.3 explains how we will sample from the specified distributions, because for every colouring we want to sample every channel end only once. In section 4.4 we explain how we calculate the ending time of a colouring. This is one of the most important parts of the simulation model and it is possible to extend this in the future. Section 4.5 explains all the possible ways a simulation run can end. Finally, section 4.6 discusses how the different statistics will be saved. Some of the statistics are updated every time during the simulation, while others will be calculated at the end of the simulation.

## 4.1 Distributions

Within the simulator, the user is able to specify various distributions, some of them are general stochastic distributions, while some others are special cases created to create certain situations in the simulation. To sample from the regular distributions the open source JSci package [4] has been used. All types of distributions and their parameters are listed in table 4.1. The value after the parameters between the brackets indicates the type of the parameter, where b = boolean, d = double, i = integer and s = String.

| *Distribution* | *Par 1* | *Par 2* | *Par 3* | *Short* | *Remark* |
|---|---|---|---|---|---|
| Beta | $\alpha$ (d) | $\beta$ (d) | | | |
| Binomial | n (i) | p (d) | | Bino | |
| Chi$^2$ | k (i) | | | | |
| Constant | value (d) | | | Con | |
| Exponential | $\lambda$ (d) | | | Exp | |
| F | $d_1$ (d) | $d_2$ (d) | | | |
| Gamma | k (d) | | | Gam | Uses $\theta = 1$ |
| Lognormal | $\mu$ (d) | $\theta$ (d) | | Logn | |
| Poisson | $\lambda$ (d) | | | Poiss | |
| Triangular | low (d) | high (d) | avg (d) | Tri | |
| Uniform | low (d) | high (d) | | Unif | |
| Weibull | k (d) | | | Wbl | Uses $\lambda = 1$ |
| IfNeeded | | | | | |
| Always | | | | | |
| File | path (s) | loop (b) | | | loop is optional |

Table 4.1: Distributions

All distributions can also be provided with one or two extra boolean parameters. The first one indicates that a new sample is made whenever a boundary node is empty again, so when the previous request has been processed. This is different from the normal behaviour, when a new request is scheduled when the the current request arrives. The last boolean parameter is a parameter to let a distribution alternate between a sample from the distribution and zero. So the first sample will be zero, and the next sample will be a sample from the specified probability distribution. When these booleans are omitted, both values will be false. The reason for these extra parameters is explained in sections 3.3.2 and 3.3.3.

The special cases needs some extra explanation. The 'IfNeeded' and 'Always' are made for the arrival distributions on the boundary nodes, they could also be used for the delay distributions, but then they will just return zero all the time. The 'IfNeeded' indicates that a boundary node will always be empty or busy and never waiting. The node will never have a waiting request, and whenever a request is needed, it provides a request and the node will go to the state busy. The 'Always' indicates that the node is never empty, whenever the node is finished with the colouring, it will go to the waiting state immediately. This process is illustrated by table 4.2. This table illustrates the events for a Sync Channel with a deterministic delay of 2 and 'IfNeeded' arrivals on A and 'Always' arrivals on B. The table illustrates that port A is never in the state waiting because of the specified 'IfNeeded'. The 'Always' on B causes that after the colouring is finished on time 2, the port will go to waiting immediately.

| *Time* | *Event* | *Result* | *State A* | *State B* |
|---|---|---|---|---|
| 0 | Initial state | | empty | waiting |
| 0 | | Start flow | busy | busy |
| 2 | End colouring | ports empty | empty | empty |
| 2 | Always distribution | B waiting | empty | waiting |
| 2 | | Start flow | busy | busy |
| 4 | End colouring | ports empty | empty | empty |

Table 4.2: Events for Sync channel with IfNeeded on A and Always on B

The file distribution is introduced to be able to define any sequence of values the user wants. A trace file with data can be used for example. The values in the file should be inter-arrival times and not the arrival times of incoming request. The file distribution has a loop option, which indicates if the sequence of values will be repeated after the file has ended. This value will always be true for delay distributions (so setting it to false has no use), because the simulation should always be possible to sample from the delays. For the arrival distribution the default value is false (if the parameter is omitted), meaning that after the trace has ended, there will be no more arrivals.

Because the distributions are used for the inter-arrival times of requests at a boundary node and the delay of a channel, some distributions like the normal distribution have been left out because they can give values below zero. Negative values in this simulation would lead to very strange behaviour of the simulation. Some of the used distributions, like constant, uniform and triangular, can still be set to produce values smaller than zero. But by providing parameters larger than zero, they will always provide positive numbers.

## 4.2 Choosing a colouring

Because we have decided to use only one colouring at a time until it is completely finished we can use the colouring semantics. The reason for this choice is explained in section 3.2.5. Every possible colouring in a colouring table has a reference to the next colouring table which can be used after the colouring is finished. The algorithm to generate the colouring tables was already implemented and will be one of the inputs for the simulation. Every state of the system corresponds to a colouring table with all possible colourings which can be used next. In this case, the state corresponds to the availability of all FIFO buffers, so a connector contains up to $2^F$ tables where $F$ is the number of FIFO buffers. The number of tables might be less than this value, because of synchronization, which makes some of the states impossible to reach. Note that this state is not the same as the 'System state' used in the simulation, because that state also contains the availability of the boundary nodes. Every system state corresponds to exactly one colouring table, while every colouring table represents zero or more system states.

During the simulation we know for all boundary nodes if they have a waiting request. Using this information and all possible colourings, we can build a list of all possible colourings which can be chosen. If this list contains multiple options, one of the colourings has to be chosen at random.

To build up the list of all compatible colourings, we have to loop over all possible colourings. For every colouring, all boundary primitive ends have to be checked. The boundary primitive ends are the ends connected directly to a boundary node, so a source end for a source node and a sink end for a sink node. The colouring will be rejected if for any boundary primitive end one of these conditions holds:

- The end has flow and the request has no waiting request
- The end gives a reason for delay and there is a waiting request

If none of the boundary primitive ends is rejected, it will be added to the compatible colourings list. After choosing one of the compatible colourings at random, the colouring will be activated, which will be explained in section 4.4.

## 4.3 Sampling from distributions

For every primitive it is possible to define one or two delay distributions. Internally these distributions are stored inside the primitive ends of the primitive. Every supported primitive has exactly two primitive ends, however not every primitive uses the same types of primitive ends. An overview of the supported primitives, how many distributions the user can specify on the primitive and how they are stored in the primitive ends can be found in Table 4.3.

| *Primitive* | *#Dist* | *Source1* | *Source2* | *Sink1* | *Sink2* |
|---|---|---|---|---|---|
| Sync | 1 | Dist 1 | - | Constant(0) | - |
| LossySync | 2 | Dist 2 | - | Dist 1 | - |
| FIFO | 2 | Dist 1 | - | Dist 2 | - |
| SyncDrain | 1 | Dist 1 | Dist 1 | - | - |
| SyncSpout | 1 | - | - | Dist 1 | Dist 1 |
| AsyncDrain | 2 | Dist 1 | Dist 2 | - | - |
| AsyncSpout | 2 | - | - | Dist 1 | Dist 2 |

Table 4.3: Delay distributions

For the Sync channel, one of the ends has been set to Constant(0), because the user specifies a distribution for the complete channel. So by setting the delay for one end of the channel to zero, we get the total delay for the channel as specified. For the LossySync channel the first distribution is for the lossy when it behaves as a Sync while the second distribution is for the lossy when it loses its data. When the lossy loses its data, it will only use its source end. For this reason, the second delay is set to the source end of the primitive. For the SyncDrain and SyncSpout the distribution is set to both ends, when sampling from these and the LossySync channel we have to be careful which will be explained next.

For most primitive ends we can just sample from the specified delay distribution on that end. However, as mentioned before, we have to be careful when sampling from the SyncDrain, SyncSpout and LossySync channel. When sampling from the SyncDrain and SyncSpout, we have to make sure that both ends use the same delay. So when we are at the first end of the channel we can sample from the distribution and remember the sampled value to be used for the second end of the channel. For the LossySync we have to know if the colouring uses the Lossy as a Sync or not. We know that the lossy loses its data when the source end has flow while the sink end has no flow. So, if we want a sample for the source end, check if the sink end has flow also. If not, use a sample from distribution 2, else use a delay of zero for the source end and use a sample from delay 1 for the sink end of the channel.

When activating a colouring, it is possible that the same primitive (end) will be traversed multiple times. For this reason, before traversing the connector, a sample is made from every delay distribution to make sure that every step in the traversal takes the same sample on every end. So for every primitive end with flow in the concerned colouring a sample is made from the distribution specified on that end.

## 4.4 Activating a colouring

After the colouring has been chosen, the colouring has to be activated meaning that the involved boundary nodes will be set to busy and an event is scheduled to mark the end of the colouring. Because there may be different ways to calculate the colourings ending, the colouring event is created in a separate class. At this moment, only one class to calculate the ending time of the colouring has been implemented. Because this calculating is implemented in a separate class it should be easy to implement alternatives, some possible alternatives will be explained in section 7.1.

#### Depth first traversal

The implemented way to calculate the duration of the colouring is by using a depth first traversal through the channels and nodes with flow in the connector. Based on the colouring, the longest path from any starting point to an ending point will be determined which will mark the ending time of the colouring.

The first thing to do when doing the depth first traversal is generating the samples for all involved channels. This process is explained in section 4.3. Next the starting points of the traversal have to be determined, such a starting point can either be a boundary node or a channel, where a channel can only have initial flow if the channel is a FIFO, SyncSpout or AsyncSpout channel.

After the starting points have been determined, it is time to do the actual traversal. This will be done by looping over the starting points and taking the longest path that has been taken by any of the starting points.

**Definition 4.1.** ***Path***
*A path is a part of the connector with flow, where every colouring with flow will have one or more paths. A path indicates the starting point s and ending point e, excluding the intermediate points.*

A starting point itself can also have multiple paths, so for all these paths, it will save the end-to-end delay, but only the longest path counts for the ending of the colouring. For all starting points, the recursive function getDuration will be called, which will return the maximum time of any of the paths from the given starting point. This function will also update the end-to-end delay statistics for all paths taken. An overview of this function is given in figure 4.2. We will explain the diagram first, followed by an example of a traversal through a simple connector. In appendix A a data diagram is given which indicates the relationship between the different objects.

Before explaining the function we first introduce the notion of a token.

**Definition 4.2.** ***Token***
*A token will be used to traverse the connector, it contains a starting time, starting location, ending time, ending location and current location. Once a token receives its ending location the end-to-end delay corresponding to the path taken by the token will be updated.*

The function getDuration has three relevant parameters, a connectable, time and token. The connectable indicates the current location of the traversal, a *connectable* can either be a Node or a Primitive (a channel). The time indicates the point in time, when this connectable has been reached. The token is the actual item which will be traversed over the connector.

When traversing the connector we will always be at a Node or in a Primitive. If we are at a sink node, we will also end the recursion because a sink node will always be the end of the path. When the node is a source or mixed node, the next step will be to traverse over all source ends with flow connected to the Node. If there are multiple source ends with flow, we will transfer the original token to the primitive of the first source end (which causes an update of the location of the token). In the other ends, we know that we have to make a copy of the token because the original token has another location than the current. This copy will only contain the starting point and starting time from the copied token, so not the current location and ending location and time.

When the connectable is a primitive, this indicates that the current location of the traversal is after the source end(s) and before the sink end(s) of the primitive. When we are in a primitive, we have to check if this primitive has sink ends, which indicates that we are dealing with a drain channel. If this drain channel is

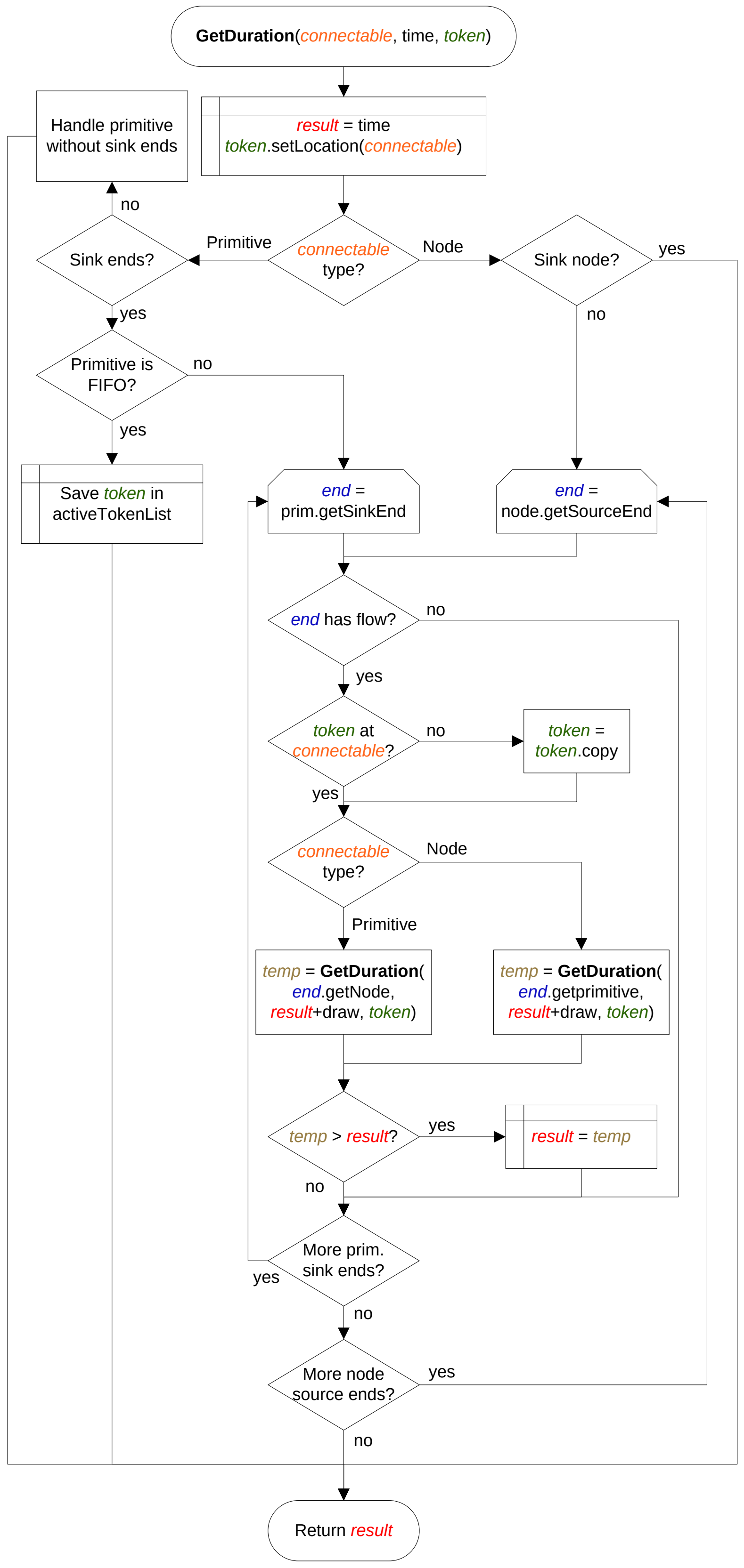


Figure 4.2: Flow chart for getDuration function used in Depth first traversal

a AsyncDrain we can just end the recursion because this will be the end of the path. If the primitive is a SyncDrain we have to make sure it will wait for the token to arrive at the other end of the drain until it will flow to get the right end-to-end delay. So the first token to arrive at the drain will be added to a list, and when the next token arrives, both tokens will be finished using the same end time. If the tokens into the SyncDrain have the same starting point (for example in a XOR), only one of them will be used for the end-to-end delay.

Another option to end the recursion in a primitive is if we end up in a FIFO buffer. If a token has flown into the buffer, it will not flow out of this buffer in the same colouring. For this reason, we end the recursion and add the current token to an activeTokenList. When the next colouring starts, it can use this token again to get the right end-to-end delay.

If the primitive is not a drain or a FIFO channel, we can loop over all sink ends with flow and traverse to the node(s) connected to these end(s). A SyncSpout channel is the only channel with multiple sink ends.

To summarize this part: we are always in either a node or in a primitive. When we are in a node, we will traverse over all source ends with flow. And when we are in a primitive we will traverse to the node the primitive is connected to. But there are three exceptions to this which will all end the recursion:

1. We end up in a sink node
2. There are no sink ends in the primitive (for example in a SyncDrain)
3. The sink end has no flow (for example in a FIFO or LossySync)

To make sure this part is clear we will also discuss an example with a simple connector shown in figure 4.3. In this example we use deterministic delays which are displayed on the channels.

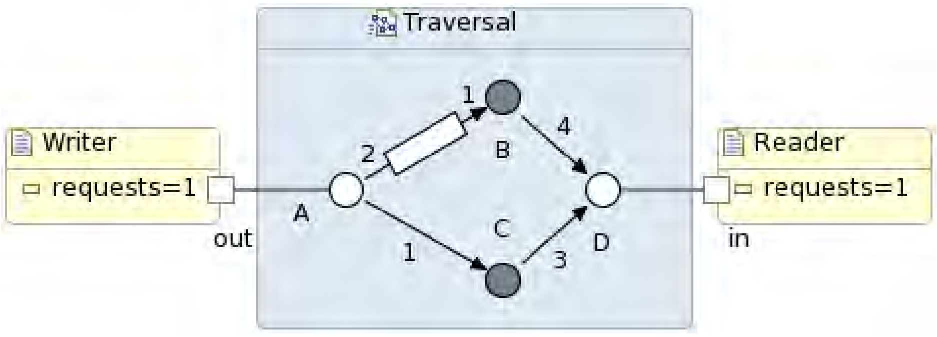


Figure 4.3: Example connector for depth first traversal

**Call 1: getDuration(A, 0, $T_1$)**
We assume that both A and D have a waiting request at time zero, so the colouring can start with flow from A to D and from A to the buffer. The first call to getDuration will be from the only starting point of the traversal at node A at time 0. Because the flow does not start at a FIFO buffer we have created a Token $T_1$ with starting time 0 and starting location A before calling getDuration for the

first time. Now we are at a node with two source ends with flow. We will first traverse to the primitive FIFO(A-B), this step will cost 2 time units as specified on the source end of the channel. The token $T_1$ is still at location A so this token can be used in calling getDuration.

**Call 2: getDuration(FIFO(A-B), 2, $T_1$)**
Now we are in the primitive FIFO(A-B). The first thing to do is updating the location of token token $T_1$ to FIFO(A-B). Because we have ended up in a FIFO buffer, we will save token $T_1$ to the activeTokenList and we return 2.

**Call 1: getDuration(A, 0, $T_1$)**
Because call 2 to getDuration returned a result of 2 we will be back at call 1 again. This temporary result of 2 is longer than the current result of 0 so this duration will be saved as the current longest path. Because node A has more than one source end, we can go on with the next primitive which is Sync(A-C). The duration of this channel has been set to 1, so we also end up in the sync channel after time 1. Now we can not use token $T_1$ any more, because the location of this token (FIFO(A-B)) is not the same as the current location (A) any more. For this reason we have to make a copy $T_2$ with the same starting time (0) and location (A) as $T_1$. With this new token we can call getDuration again.

**Call 3: getDuration(Sync(A-C), 1, $T_2$)**
Now we are in Sync(A-C), from now on the location of $T_2$ will be updated with every call to getDuration, but we do not mention this because the token will not be copied any more. The Sync channel is a channel with one sink end with flow (which will always be zero for sync channels), so we can call getDuration again.

**Call 4: getDuration(C, 1, $T_2$)**
Now we are in node C which has one source end. We add the duration of 3 of the source end of Sync(C-D) to call getDuration again with a total time of 4.

**Call 5: getDuration(Sync(C-D), 4, $T_2$)**
We arrive at a sync channel again, which will have a sink end with a delay of zero again. So we can traverse further to node D.

**Call 6: getDuration(D, 4, $T_2$)**
Finally we arrive at the sink node D, so we can stop the recursion. Because the path of token $T_2$ has actually ended, we can update the statistic of the delay from A to D with a delay of 4. Now the token $T_2$ can be removed, because the path has finished. The result of 4 will be returned by this call to getDuration.

**Call 1: getDuration(A, 0, $T_1$)**
The result from call 6 will be propagated to call 1 again. This result of 4 is longer than the previously saved result of 2, so the final result of the first call to getDuration will also be 4.

**Next colouring**
When a new request arrives at boundary node D, for example after time 6, the token $T_1$ will be used again, which will finish after time 11. Now this token is finished also, so this token can be removed from the activeTokenList.

## 4.5 Stopping criteria

The simulation has multiple ways to stop a simulation run, the various possibilities will be described below. After the simulation has been finished for every run (in case of short-term simulation) the simulated time, the number of events and the reason for stopping will be given. Next to that, also an overview is given with for every observed stop reason the count, percentage, mean, standard deviation and a confidence interval. Note that the confidence interval might be very bad if the count is very low.

#### Max simulation time or events

The first obvious reason for stopping a run is when the maximum simulation time or number of events has been reached. The maximum simulation time or events can be set in the settings of the simulation and is required. The run will stop whenever an event is chosen with a time larger than or equal to the maximum simulation time.

#### Cancelled

During the simulation, a progress bar is shown which gives an indication how far the simulation is. It will also display an estimation of the time left for the simulation. This indication will only be updated at the start of a new batch or run, but this indication is not very accurate at the beginning of the simulation. The progress bar will have a cancel button to cancel the simulation which will cancel all runs which are not completed yet.

#### Deadlock

The simulation can end up in a deadlock, which will freeze the system. A deadlock is easy to detect, because when we are in a deadlock the colouring table is empty (because the only colouring in the table would be a no flow colouring, but these are removed because they are not relevant for the simulation). In the options there is a button to enable or disable the deadlock detection. If enabled the simulation run will stop after ending up in the deadlock. The deadlock detection is optional because you might want to see what happens to the statistics after the system has ended up in a deadlock.

#### Livelock

Another option to end the simulation is by a livelock. Remember that we have a livelock when there are still actions inside the connector, but not at the boundary nodes. The simulation will detect the livelock by counting the number of consecutive chosen colourings without any involved boundary nodes, while all boundary nodes have a waiting request. If this count is above a maximum value which have to be specified in the options, the connector is assumed to be in a livelock.

Whenever a colouring is chosen with involved boundary nodes the count will be reset.

#### Empty eventlist

In normal circumstances an empty eventlist will not happen, but there is a possibility to get this type of ending. If deadlock detection is disabled and you specified a special distribution like 'IfNeeded' or 'Always' on every boundary node the eventlist will be empty when we end up in a deadlock.

#### Observed state

The last possibility to end the simulation is by specifying a state to stop the simulation. The state has to be specified in the same way as the system state, which consists of the state of the boundary nodes (sorted in an alphabetical order) and the state of the FIFO buffers. It is also possible to add the wild-card '?' to the specified state to indicate that certain parts of the state can be any value. For example, if you specify the value 'ww?' the simulation will stop if the simulation will reach the state 'wwe', 'www' or 'wwb'. Multiple states can be defined by separating them with a comma (,) sign. Then the simulation will stop if it reaches any of the specified states.

## 4.6 Statistics

This section covers the statistics which will be outputted after the simulation. All these statistics can be turned on or off before running the simulation to be able to reduce the number of outputs. These statistics can be separated into two groups, the primitive statistics which will be updated during the simulation, and the calculated statistics which will be derived from the primitive statistics. What every statistic represent can be found in section 3.1.

### 4.6.1 Primitive statistics

The primitive statistics are updated constantly during the simulation. How these statistics are updated during the simulation will be described in this section. The primitive statistics includes the following categories:

- Buffer utilization
- Channel utilization
- Colourings
- End-to-end delay
- Node state
- System state
- Special state

### Statistic updating

All the primitive statistics except the end-to-end delay are handled in the same way. Every category has one or more ’StateStatistics’ objects, for example the node state have such an object for every node, while the system state has only one ’StateStatistic’ object. This object remembers the current state and the moment it has reached this state. Please refer to the data diagram in appendix A for a representation of the different objects and their relations.

When the state of the ’StateStatistic’ changes, we know the duration the object has been in the previous state. If this is the first observation of this state, a new ’Statistic’ object will be created which will be updated from now on. This ’Statistic’ object contains the count and the total duration in all batches. It will also handle the chart for this statistic, which will be explained in the next section.

The end-to-end delay is a little different from all other primitive statistics because it does not use a ’StateStatistic’ object. In contrast with the other primitive statistics, the end-to-end delay is not only at one state at a time. When we have a colouring with multiple paths, we also have multiple end-to-end delays with the same starting time. For this reason, we only have a ’Statistic’ object for every observed end-to-end delay.

### Charts

For all primitive statistics it is also possible to show a chart after the simulation has been finished. This chart will show for all batches how the average statistic value evolves over time. To plot these results the open source package JFreeChart [3] has been used. By looking at this chart, you can see if the statistic value converges to a certain value. If it does not converge, the batch size can be too small. The charts can be zoomed in and out, so when checking the convergence it is important to look at the y-axis to see the deviation in the results.

Every state change of a statistic will produce two chart points, one to indicate the start of a state change and one to indicate the end. During the simulation there will be a lot of state changes which produces a lot of chart points. This will slow down the simulation and the output a lot, especially producing and viewing the chart is very slow with a lot of chart points. If the number of chart points would become very large, it can also produce out of memory errors.

For this reason, there is an option for the simulation called ’Max chart points’ to reduce the number of points for every chart. The way this works is as follows, when we assume the user specifies a max chart points value of 5000 and 50 batches. Then every batch will have a maximum of 100 points which will be connected by a line. At first all points will be saved to a list to produce the line later. Whenever we want to insert the $101^{st}$ point, we will first remove every second point from the list to reduce the list to half the points.

From this moment we will not save every point any more, but only every second point, because the first part also contains every second point. After we have reached the $101^{st}$ point again (which would be the $201^{st}$ point if we would have kept all chart points) we will remove every second point from the list again and save every fourth point from this moment on. By doing it this way, every batch line will always contain between 51 (if this number has been reached) and 100 chart points.

An example of such a chart is given in figure 4.4. In this chart the maximum number of chart points is set to 20.000 and the number of batches is 25. In the chart you can see that all batches have converged to approximately the same value in the end. However the difference between the smallest and largest value is around 0.03, which might be too large in some cases. If this is the case, a larger simulation time can be used.

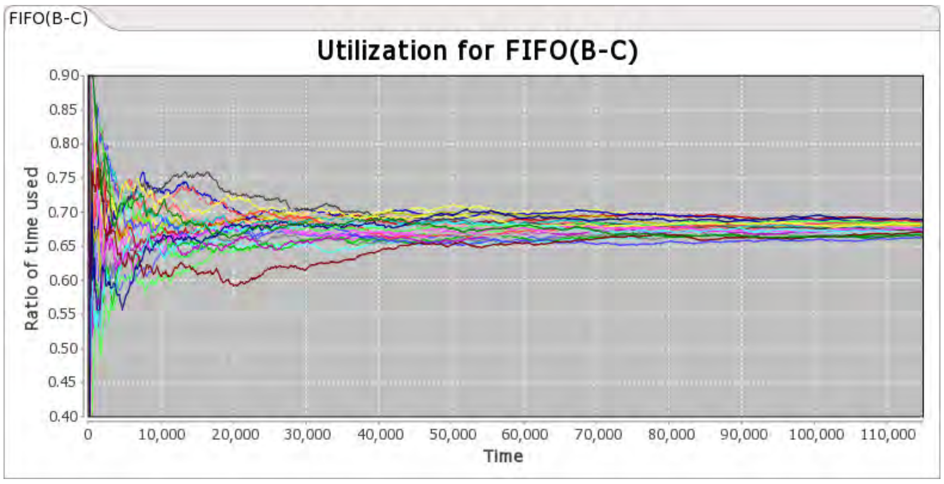


Figure 4.4: Chart example

### 4.6.2 Derived statistics

The derived statistics are derived from the primitive statistics and will be calculated after the simulation is finished. For this reason, these statistics will also not have a chart. The derived statistics includes the following statistics, how they are calculated from the primitive statistics will be explained next.

- Actual loss ratio
- Average conditional waiting time
- Average waiting time
- Channel locked
- Inter-arrival times
- Merger directions
- Request observations

#### Actual loss ratio

To calculate the actual loss ratio we need the colouring statistics. This statistic contains the number of times every colouring has been used, and the colouring itself contains information about the usage of a LossySync channel. To calculate the actual loss ratio of a LossySync channel we have to loop over all colourings and count the number of times the source and sink ends of the channel have flow. If the source end has flow, the colouring uses the LossySync and if the sink end has flow it means the channel has been used as a Sync channel. The loss ratio will be $1 - \frac{\text{Sink end count}}{\text{Source end count}}$.

### Waiting times and request observations

During the simulation we have an object which counts the observations of the request at the boundary nodes. So every node have three counters, one for every possible state. With this information we can easily calculate the percentage of times a request observes a certain state by dividing the count of the particular state with the total number of requests at this boundary node.

For the waiting times we need the request observations and the node state statistic which indicates that the node is in the state 'waiting'. This statistic contains the number of requests which have to wait, and the total time the node is in the state 'waiting'. When we divide the total time a boundary node is in the state 'waiting' by the total number of request at the boundary node we get the normal average waiting time. When we divide the total waiting time by the number of non-blocked requests, we get the average conditional waiting time.

### Channel locked

For the ratio of time a channel is locked because a colouring is active we have to use the colouring statistics. This statistic is a bit different based on the type of channel. For the Sync, SyncDrain and SyncSpout channel the value is given for the whole channel, because, when one end of the channel is used, the other end will also be used. For the LossySync, FIFO, AsyncDrain and AsyncSpout the value is given for an end of the channel, because the end can operate separately for these channels.

The values can be calculated by summing up all usage percentages of all colourings with flow for the primitive end. For the first group of channels the first end will be used, while for the other group both ends will be evaluated.

### Inter-arrival times

The inter-arrival times are easy to calculate, all we need is the end-to-end delay statistics. This statistic contains the number of flows from a certain starting to ending point. By dividing the length of a batch or run by the count, we obtain the inter-arrival times.

### Merger directions

This statistic is given for every non-source node with more than one sink end and it can also be calculated from the colouring statistics. For all sink ends of such a node we can sum up the counts of all colourings which has flow on the sink end. Then the direction can be calculated by dividing the total count of one end by the total count of all ends.

# Chapter 5

# Validation

In this chapter we will validate if the Reo simulator produces the right output and we will answer the research question: Does the Reo simulator produce the same results as other simulators? At the same time it will answer another research question: What kind of systems can we model with Reo?.

This validation will be done by comparing the results of the Reo simulator with results from QIA, queueing theory and other simulators. In section 5.1 we will use an ordering connector to compare the results of the Reo simulator with the results produced by the CTMC of QIA. In this section we will also check if all different simulation methods will produce the same results. In section 5.2 we validate another model using the CTMC of the system, this time we validate if the steady state results are the same. In this section we will also check if the Reo simulation results converges to the same value as the CTMC when the simulation length increases. In section 5.3 we discuss a connector using a lossy sync because this channel will not produce the same results as the CTMC. In section 5.4 we compare the results of the Reo simulator by building a queueing model and comparing the results with the results known in queueing theory. Finally, section 5.5 compares the results of the Reo simulator with the results of a simulator build by the University of Malta. In section 6.1 of chapter 6 we also we also validated the end-to-end delay with the results of a model build in ExtendSim, but because this was only part of that analysis, this validation is not in this chapter.

## 5.1 Ordering connector

The ordering connector is one of the standard examples of Reo which can be seen in figure 5.1. This connector ensures that the starting location of every token arriving at C, is alternated between A and B. This connector has also been covered in the paper about QIA [11], however this automata model has the restriction that every distribution should be exponential. But if we keep this restriction within the simulation model, the result from that paper should be reproducible using the simulator for Reo.

When converting the ordering connector of figure 5.1 into a QIA model first, and

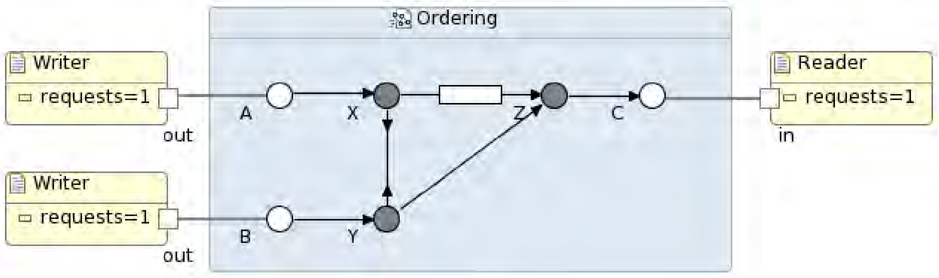


Figure 5.1: Ordering connector

then into a continuous time Markov chain (CTMC). This CTMC can be analysed using PRISM to get the steady state probabilities. This CTMC has five delays: dA, dB, dC, dY0 and dZ0. In the simulator dA, dB and dC should be defined on the arrival distributions of the boundary nodes. The delays dY0 and dZ0 should be defined in the delay distribution of the FIFO channel. In the simulator, we also have to define the delay distributions of the Sync channels and the SyncDrain. Because these are not specified in the CTMC, these will be set to be zero.

Due to some bugs which are fixed in the latest version of the QIA converter, the results of the experiments by PRISM for this connectors are a little different from the results of figure 12 in the QIA paper. The results of the new version can be seen in figure 5.2.

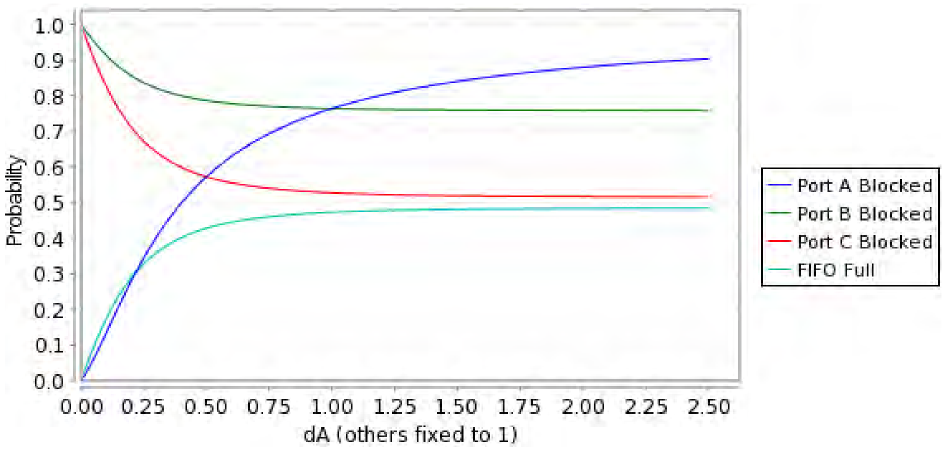


Figure 5.2: Blocking probabilities found by PRISM

Currently, it is not possible to define a range of parameters to do multiple simulations in a row in the Reo simulator, so we cannot easily create a chart as generated by PRISM. For this reason we have compared the values by setting dA to 0.1, 1.0 and 2.5. We used different kind of simulation types here, to see if there are any differences, for every type of simulation approximately 20,000 events in every batch have been used, this number of events corresponds to about 5,000 time units in every batch. The results of the simulation can be found in table 5.1.

When looking at the results, we can see that the results are reasonably close

(a) $dA = 0.1$

| *Type* | *A blocked* | *B blocked* | *C blocked* | *FIFO Full* |
|---|---|---|---|---|
| Short - Events | 13.52% | 91.32% | 82.40% | 17.59% |
| Short - Time | 13.64% | 91.19% | 82.92% | 17.19% |
| Long - Events | 13.31% | 91.31% | 82.69% | 17.22% |
| Long - Time | 13.33% | 91.31% | 82.41% | 17.47% |
| PRISM | 13.34% | 91.33% | 82.69% | 17.33% |

(b) $dA = 1.0$

| *Type* | *A blocked* | *B blocked* | *C blocked* | *FIFO Full* |
|---|---|---|---|---|
| Short - Events | 76.40% | 76.38% | 52.73% | 47.31% |
| Short - Time | 76.35% | 76.44% | 52.82% | 47.16% |
| Long - Events | 76.01% | 76.40% | 52.70% | 47.19% |
| Long - Time | 76.02% | 76.38% | 52.63% | 47.19% |
| PRISM | 76.37% | 76.37% | 52.74% | 47.26% |

(c) $dA = 2.5$

| *Type* | *A blocked* | *B blocked* | *C blocked* | *FIFO Full* |
|---|---|---|---|---|
| Short - Events | 90.37% | 75.92% | 51.68% | 48.29% |
| Short - Time | 90.30% | 75.50% | 51.66% | 48.15% |
| Long - Events | 90.42% | 75.49% | 51.69% | 48.50% |
| Long - Time | 90.42% | 75.50% | 51.64% | 48.45% |
| PRISM | 90.32% | 75.79% | 51.58% | 48.41% |

Table 5.1: Results for ordering connector with different simulation methods

to the results found by PRISM. The differences in the results of the simulation and the results of PRISM can most probably be declared by the length of the simulation. To test this hypothesis, we have evaluated some of the results again to see if the results are better. The results we have checked again, are some of the results with a large difference with the results of PRISM. For the short term simulation with time and $dA = 2.5$ the difference in blocking probability of port B went from 0.39 to 0.01 when using a simulation period ten times longer. The buffer utilization decreased from 0.26 to 0.02. For the long term simulation with events and $dA = 1.0$ the difference in blocking probability of port A went from 0.36 to 0.01 when using a simulation period ten times longer.

## 5.2 Barrier Synchronizer

Another standard Reo example is the barrier synchronizer as shown in figure 5.3. This connector ensures that there can only be flow from A to E if there is also flow from B to F. In this example, we will not look to the blocking percentages of the boundary nodes, but to all steady states. Every steady state in the CTMC corresponds to one or more 'system states' given by the simulation. Each state in the CTMC corresponds to the availability of the boundary nodes, so for example {A,F} indicates that boundary nodes A and F are available. This state in the CTMC corresponds to the system state 'weew' where boundary nodes A and F are waiting, because available in the CTMC state corresponds to a waiting request in the simulation. An exact description about the system state can be found in section 3.1.1.

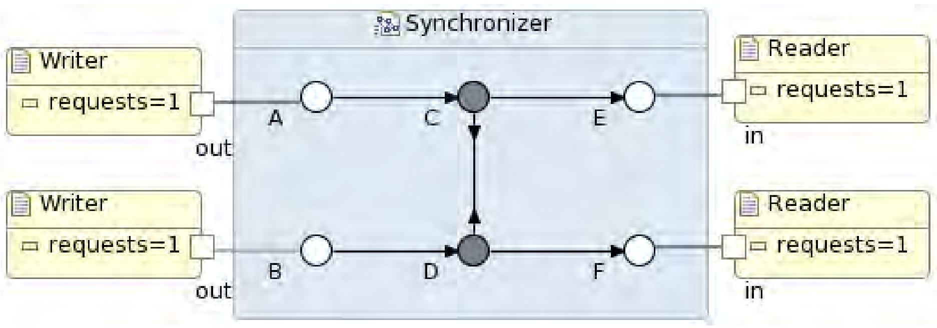


Figure 5.3: Barrier Synchronizer

In this case we will not only check if the steady states of the CTMC equals the values of the system states. We will also check if the deviation in results will become smaller if the simulation period is longer. The results of this validation can be found in table 5.2.

In this table the state is the state of the CTMC as explained above, the system state is the corresponding state from the simulation. The results shown are the results of the steady states of the CTMC and the results of the simulation using different number of events. The number of events listed in the table are the number of events in every batch excluding 1000 events warm-up period. In this example the following exponential rates have been used: $dA = 6$, $dB = 3$, $dE = 5$,

$dF = 4$ and $dCD = 100000$.

| *State* | *Sys St* | *PRISM* | *500* | *5000* | *50000* | *500000* |
|---|---|---|---|---|---|---|
| {} | eeee | 0.1085 | 0.1197 | 0.1086 | 0.1091 | 0.1087 |
| {B} | ewee | 0.0217 | 0.0185 | 0.0227 | 0.0219 | 0.0216 |
| {F} | eeew | 0.0310 | 0.0324 | 0.0311 | 0.0310 | 0.0310 |
| {E} | eewe | 0.0417 | 0.0411 | 0.0418 | 0.0415 | 0.0417 |
| {A} | weee | 0.0543 | 0.0547 | 0.0525 | 0.0541 | 0.0543 |
| {A,B} | wwee | 0.0326 | 0.0336 | 0.0327 | 0.0330 | 0.0324 |
| {A,F} | weew | 0.0504 | 0.0630 | 0.0525 | 0.0504 | 0.0507 |
| {A,E} | wewe | 0.0746 | 0.0722 | 0.0752 | 0.0746 | 0.0749 |
| {A,B,E} | wwwe | 0.1317 | 0.1262 | 0.1265 | 0.1317 | 0.1314 |
| {A,E,F} | weww | 0.2550 | 0.2319 | 0.2579 | 0.2540 | 0.2550 |
| {A,B,E,F} | wwww | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| {A,B,F} | wwew | 0.0759 | 0.0916 | 0.0748 | 0.0756 | 0.0761 |
| {B,E} | ewwe | 0.0234 | 0.0227 | 0.0231 | 0.0236 | 0.0234 |
| {E,F} | eeww | 0.0358 | 0.0320 | 0.0374 | 0.0355 | 0.0356 |
| {B,E,F} | ewww | 0.0471 | 0.0451 | 0.0458 | 0.0476 | 0.0469 |
| {B,F} | ewew | 0.0164 | 0.0153 | 0.0174 | 0.0164 | 0.0163 |
| Avg deviation | | | 0.0053 | 0.0012 | 0.0003 | 0.0001 |
| Max deviation | | | 0.0231 | 0.0052 | 0.0010 | 0.0003 |

Table 5.2: Results Synchronizer with different number of events

We would like to know if the values converges to the steady states from the CTMC. For this, we should not look at the individual values because by coincident a result with a smaller number of events could be better than the same result with more events. The important values we should look at are the average and maximum deviation displayed at the bottom of the table, which indeed converges to 0 when the simulation length increases.

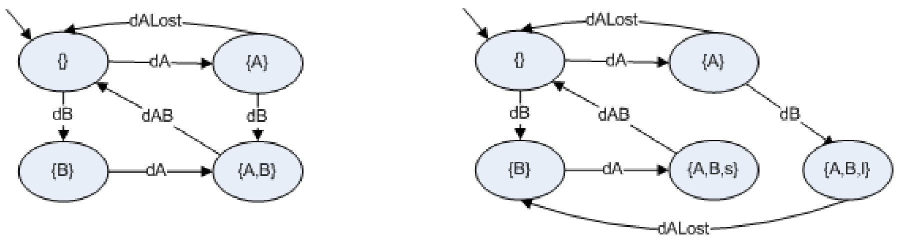
## 5.3 Lossy Sync channel

When evaluating a lossy sync channel with the Reo simulator, this will not work in the same way as the CTMC provided by QIA. In the simulator the lossy sync will always lose its data when the source end is available before the sink end. In the CTMC however, the lossy sync will not lose its data when the sink end is available before the data is lost.

The CTMC provided by QIA can be found in figure 5.4(a). This is a simplified version of the automata given in figure 8 of the QIA paper. One important thing to mention when looking at this CTMC is the presence of an arrow from {A} to {A,B}. So whenever A is available and B is not available, there is still a chance that the request at A will not be lost when a request at B arrives before the request at A is lost.

In the simulation this is different, whenever A is available before B is available, it will lose its data, also if B is available before the data is lost. If B is available before the data is lost, B will have a waiting request when the losing of the data is finished. The Markov chain which represents the working of the simulator is given in figure 5.4(b). In this figure another state {A,B,l} has been added where A and B are available, but the request of A will still be lost. The state {A,B} in the middle has been renamed to {A,B,s} indicating that both A and B are available and the channel will behave as a sync channel.

The blocking probabilities given by both Markov chains and the Reo simulator are given in table 5.3. In this example dA and dB are set to 2, dAB is set to 3 and dALost is 4. As expected, the results of the two Markov chains produces different results, while simulating the connector in Reo indeed gave the same results as the CTMC of figure 5.4(b). Using this example it is also easy to validate if the channel utilization and the colouring results are correct. In this case these can be derived from the system state statistics as shown in table 5.4. The values given in the table derived from the state statistics are exactly the same as the values given for the colourings and the channel utilization which proves that these results are correct.



(a) QIA (b) Simulation

Figure 5.4: Markov chains for the Lossy Sync channel

| *Method* | *A blocked* | *B blocked* |
|---|---|---|
| CTMC of figure 5.4(a) | 36.84% | 52.63% |
| CTMC of figure 5.4(b) | 37.31% | 64.17% |
| Reo Simulator | 37.31% | 64.17% |

Table 5.3: Blocking probabilities for Lossy Sync

| *System state* | *Percentage* | *Sync* | *Loss* | *No flow* |
|---|---|---|---|---|
| bb | 23.95% | X | | |
| be | 8.91% | | X | |
| bw | 4.43% | | X | |
| ee | 26.84% | | | X |
| ew | 35.85% | | | X |
| Total | | 13.43% | 23.95% | 62.69% |

Table 5.4: Colouring and channel utilization validation

## 5.4 M/G/s/c queueing model

In this section we will validate the simulator by comparing the results from the simulator by the theoretic values known in queueing theory. In principal, Reo is only suitable for queueing models with a finite queue. But the number of buffers can be chosen such that the probability that a request is blocked is very small. If this probability is small enough we can approximate the result of the model with an infinite buffer.

A queueing model which is very well suited to validate with the simulator in Reo is the M/G/s/c model, also known as the Erlang X model. This model can be used in call center, where a limited number of servers are available. Whenever all servers are busy, the next customers will be queued. When the queue is full, the call will be blocked. A call center may use a limited queue to make sure that a caller will not wait for a very long time, instead he may call back later when it is not busy. This model has the following properties:

- Exponential arrival distribution
- General service time distribution
- s servers
- c capacity of the system (including the servers)

Every arriving customers observing that the full capacity is used will be blocked and the customer will also not attempt again. The Erlang X model also has the ability to specify an average time to abandonment, but this is not used because the simulator is not able to model this. As an example we will model a M/G/2/5 queue and check if the results of the simulator are similar to the results of the Erlang X model.

In Reo, this queueing model has been modelled as in figure 5.5. Two important notions have to mentioned when looking at the connector. First of all, the connector contains only two FIFO buffers while the model have three queueing places. The reason for this, is that a request can wait at port A also. The second important part to notice is that the merger node Y has an x on it, meaning it behaves as a router node and it will send its data to exactly one sink end. The same behaviour could be achieved by using an XOR connector.

Because there is always a possibility for a request to wait at port A, an extra channel has to be added when using a model without a queue (for example the Erlang B model). Such a model can be made by adding an extra LossySync channel after the boundary node.

After modelling the connector, we have to define the arrival rates and the delay on the channels. The arrival rate can be defined on port A and should be exponential. An obvious thought would be to define the service time distribution on the last two Sync channels (Y-B and Y-C). But this choice would not work as expected in the current simulator as explained in section 3.3.1. Instead, all delays on every channel is set to zero while the service rate is defined on the boundary nodes B and C.

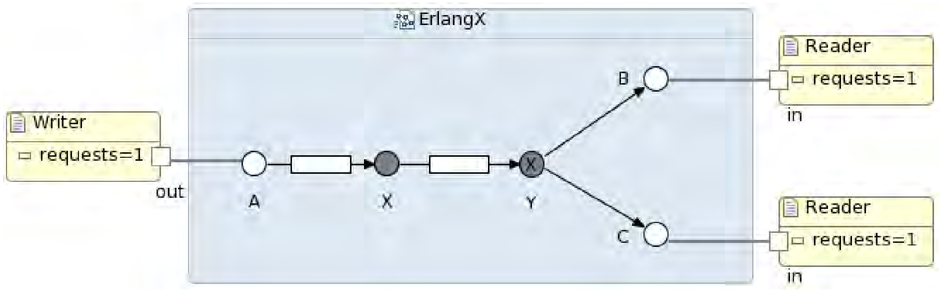


Figure 5.5: M/G/2/5 model

### Results

To compare the results of the Erlang X model and the simulator, an Erlang X calculator [1] has been used to get the blocking probabilities and average waiting time. In the simulator, the blocking probability is outputted directly while the waiting time should be derived from other results.

When simulating the model with $\lambda = 5$ and $\mu = 1$, we will find a blocking probability of 60.48% with an interval of [60.27%, 60.68%]. This probability can be found by looking to the state of node A, whenever this state is 'waiting' or 'busy', all incoming requests will be blocked. Because the delays of all channels are zero, node A will never be in the state 'busy' for more than zero seconds.

The blocking probability given by the Erlang X calculator is 60.43%, which is close to the found value and also within the confidence interval.

The average waiting time of 1.21 minutes given by the Erlang X calculator can be derived from the results of the simulation in two ways. The easiest way is by adding the average conditional waiting at port A to the end-to-end delay of A-B or A-C. This conditional waiting time indicates the time a request have to wait at A before it will go into the queue. The end-to-end delay in this case represents the total time from A to B, without the waiting time at A and also without service time. Adding the conditional waiting time of 0.31 to the end-to-end delay of 0.90 produces the waiting time of 1.21.

For an alternative way to calculate the waiting time we need the blocking probability and the buffer utilizations. When we have this information, we can calculate the average queue length, and apply Little's Law [17] to get the average waiting time.

Little's law gives a very important relation between the long-term average number of customers in a system, the time a customer spends in the system and the arrival rate. Little's law is only valid when the system is stable, meaning the number of customers in the system does not grow to infinity. Because we are dealing with a bounded buffer, this will not happen and we can apply Little's law: $L = \lambda W$, where $L$ is the average number of customers in the queue, $\lambda$ the effective arrival rate and $W$ the average waiting time.

So, all we need to calculate the average waiting time is the average number of customers in the queue. The values to calculate the average queue length are:

- Blocking probability: 60.48%
- Buffer utilization (A-X): 84.74%
- Buffer utilization (X-Y): 94.39%

How to calculate the probability that there are x customers waiting for service can be found in table 5.5.

| *Prob* | *Calculation* | *Value* |
|---|---|---|
| $P(X = 3)$ | Blocking probability | 60.48% |
| $P(X = 2)$ | Buffer util. (A-X) - Blocking probability | 24.27% |
| $P(X = 1)$ | Buffer util. (X-Y) - Buffer util. (A-X) | 9.65% |
| $P(X = 0)$ | 1 - Buffer util. (X-Y) | 5.61% |

Table 5.5: Queue length probabilities

Now we can calculate the average queue length L with equation 5.1.

$$L = \sum_{i=0}^{3} iP(X = i) = 2.40 \tag{5.1}$$

When the average queue length has been calculated, the average waiting time can be calculated by applying Little's law. The arrival rate to use in this formula is the non-blocked arrival rate, so in this case the arrival will be $(1 - 0.6048) \times 5 = 1.98$. The average waiting time given by the simulation is $2.40/1.98 = 1.21$ which is exactly the same value as given by the Erlang X calculator.

## 5.5 G/G/s/c queue

In the previous examples, we have only used exponential distributions, which have the advantage that they are memoryless. This property[1] is very important for all calculations, and these models can not be used any more if we will use different distributions.

In the CTMC, every state might have multiple transitions to a next state, and when we are in that next state we do not have to know what has happened before this state. So we can just sample again to know which state to go to next. If we want to use different distributions, this CTMC will not work any more and we have to use simulations. To validate the results of the Reo simulator with non-exponential distributions we have used a simulator made by the University of Malta [7].

In this simulator we have made a model with four servers and four waiting places in the queue. For the arrival rate we have used the uniform(0,1) distribution and for the service rate of all servers we have used the uniform(0,3) distribution.

In Reo this can be modelled in the same way as in the previous example with the M/G/s/c model. For the arrival rate we can just specify the uniform distribution

[1]The memoryless property has been explained in section 2.4

on boundary node A. For the service rate however, we can not just specify the uniform(0,3) distribution. Instead, we have to specify uniform(0,3,true). The value 'true' indicates that a sample is made whenever a boundary node changes from the state 'busy' to 'empty', so in this case none of the arriving read request will be blocked. The reason to model it like this is explained in section 3.3.2.

**Results**

One of the results generated by the simulator of the University of Malta (UM) is the average waiting time and average queue length, which are related to each other by Little's law as explained in the previous section. In the Reo simulator, we can find the average waiting time by adding the waiting time at port A to the end-to-end delay. This produces a result of 0.174 against the 0.176 of the UM simulator. The average queue length can be found by applying Little's law with an effective arrival rate of 1.99, which gives us a result of 0.346 against 0.350.

Another figure produced by the UM simulator is the blocked customers rate of 0.601%. The Reo simulator generates this rate directly also. In the previous examples it did not matter if we took the 'Node state' or the 'Req observations'. This was due to the PASTA (Poisson Arrivals See Time Averages) property which only holds if we have Poisson arrivals. In this case we do not have Poisson arrivals so there is an actual difference between these values. The 'Node state' indicates that port A is in a blocking state in 0.910% of the time, while the 'Req observations' gives us the actual blocking rate of 0.585%.

The last values we can compare are the average number of busy servers and the server utilization. The average number of busy servers can be found by summing up the occupancy rate of the separate servers. In the Reo simulator, this occupancy rate can be found by taking the percentage of time a sink node is in the state 'empty'. This produces a average number of busy servers of 2.976 by Reo compared to 2.983. The server utilization is 0.744 by Reo compared to 0.746.

The results are summarized in table 5.6. All values are reasonably close to each other, but by increasing the simulation time for both simulators will decrease the difference between the simulators even more.

| *Result* | *Reo* | *UM* |
|---|---|---|
| Average waiting time | 0.174 | 0.176 |
| Average queue length | 0.348 | 0.350 |
| Blocked customer rate | 0.585 | 0.601 |
| Average number of busy servers | 2.976 | 2.983 |
| Server utilization | 0.744 | 0.746 |

Table 5.6: Comparison of simulator of Reo and University of Malta (UM)

This kind of system could also be modelled in a different way. In this alternative way, all servers also have a space for an item in service as explained in section 3.3.3. This type of system is shown in figure 5.6. Because all requests are only handled by one server, this alternative type of modelling the system does not matter for the results in table 5.6. Only the end to end delay given by the simulator will be different, because in this alternative method this delay also includes the service time duration. But by adding the expected service time duration to the end-to-end delay given by the first model, the results are the same.

The expected service time could be known based on the service time distribution, but we can also calculate the average service time if the expected service time is unknown, for example if we are using a file to generate the service times.

This average service duration can be calculated with equation 5.2, where $s =$ average service duration, $i_{avg}$ is the average inter-arrival times, $i_c$ the count of the inter-arrival times, $e_t$ the ratio of time the boundary node of the server is in the state 'empty' and $e_c$ the number of observations for the server being in the state 'empty'. We use the time and count the boundary node of the server is in the state 'empty' because this actually indicates that the server is busy with a request.

$$s = \frac{i_{avg} \times i_c \times e_t}{e_c} \tag{5.2}$$

But this alternative way to model the G/G/s/c queue comes with a serious disadvantage. Because we have added four extra FIFO buffers and four extra readers the number of colouring tables and colourings have increased a lot. While the first model only had 8 tables with a total of 372 colourings, the alternative model has a total of 127 tables with 70.491 colourings. First of all, it is much slower to calculate these colouring tables. Secondly, the simulation itself is also much slower because choosing the next state will also be more time consuming, instead of an average of 42 colourings in every table, now there are 555 colourings in every table. Also the amount of virtual memory needed to store the tables is very large, but we will return to this point in section 6.1.

So however both types of ways to model the G/G/s/c queue produces the same results, you should not use the alternative approach because of the big disadvantages.

## 5.6 Conclusions

In this chapter we showed that the results of the Reo simulation are the same as the results with different methods. We validated the different output measures of the simulation by the results using CTMC of QIA, Erlang-X calculator and a simulator by the University of Malta for simulating queuing models with general distributions. Only the lossy sync channel produces different results than the CTMC, but we explained what causes this difference.

We also shown that the results of the different simulation methods (short or long with events or time) produced the same results in section 5.1. In section 5.2 we showed that the results also converges to the same values as the CTMC. When simulating different systems in Reo, it is important to use a decent simulation period. If the simulation period is not long enough, the confidence interval can be large indicating that the mean value is not very accurate. We can see if

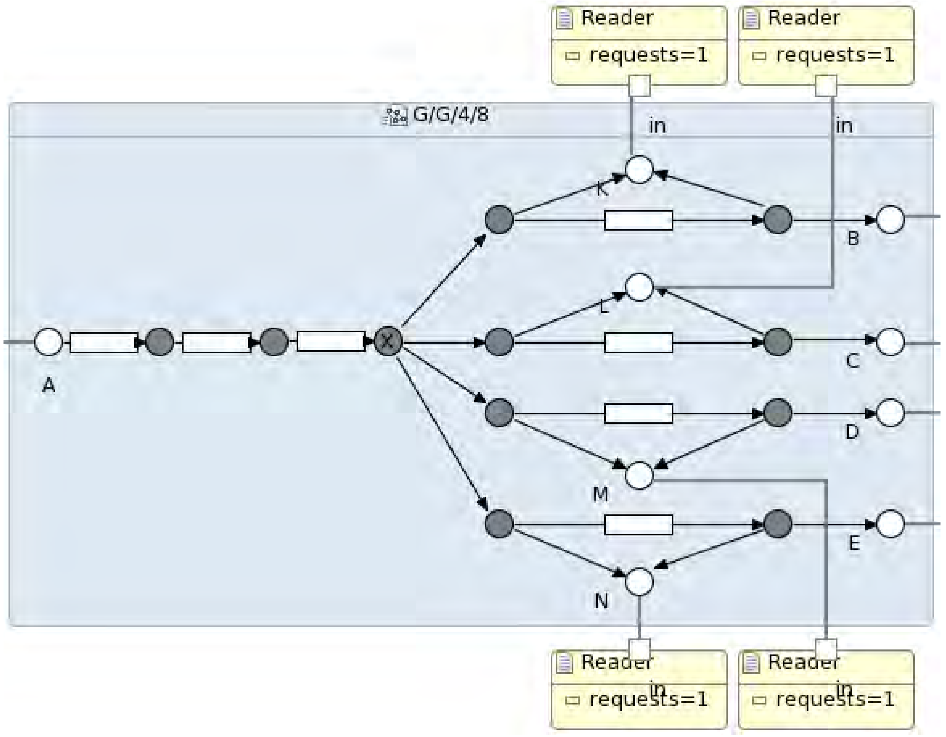


Figure 5.6: Alternative for G/G/4/8 model

the simulation period is long enough by looking at the charts generated for the statistics and check if the results of the different batches converged to the same value.

# Chapter 6

# Results

In this chapter we will discuss some models with non-exponential delays which can not be evaluated without the Reo simulator. So this chapter will give an answer to the research question: What kind of systems can we model with Reo? In section 6.1 we will discuss a system where a job will be processed by multiple servers and merged after that. In this section we will also experience a problem when handling systems with too many buffers. Section 6.2 shows how to model more complex systems, by modelling a system with a variable number of servers. Finally, section 6.3 models and analyses a manufacturing process.

## 6.1 Synchronizing servers

In this section we will look at a system with multiple parallel queues and synchronization. The idea is that a job will be duplicated to multiple servers and when all servers have processed the job, the job is finished. In this example, synchronization is very important, when a server has finished a request, the request will wait until all servers are finished. A practical example of such a system is a holiday reservation system, where both a flight and an accommodation should be available. So whenever a customer wants to book the holiday, the job will be send to both systems, and when both of them have approved, the job is finished. In this system we will investigate what happens to the blocking probability, end-to-end delay and the average queue length when the number of servers increases.

Every server has its own queue of items to be processed and a queue with processed items. Because of the queue with processed items, the server does not have to wait till all servers have processed a certain job. So whenever the server has processed an item, it can continue with the next job. Whenever a job has been processed by all the servers, it will be removed from all processed items queues. A diagram of this kind of process is given in figure 6.1.

When looking at this kind of system, you might want to use infinite buffers for all queues, but as explained before this is not possible in Reo. To keep the problem relatively small, we have restricted the buffers to four per server. One before the server, one for an item in service and two for finished requests. In this way, every

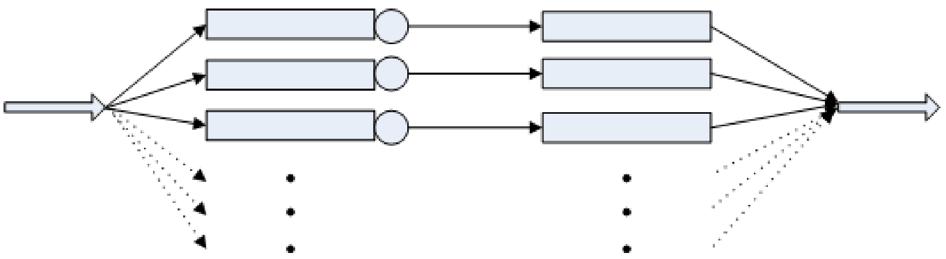


Figure 6.1: Diagram of synchronizing servers

server is able to be two jobs ahead of the slowest server. When one of the servers have processed two requests before one of the other servers have processed the first request, the faster server will wait. There can be a waiting request at the boundary node, but this will not flow into the server yet because it is not possible to flow into all queues before the servers.

The Reo connector which models this kind of system with 3 servers is given in figure 6.2. In this figure we have indicated what the current state of the connector could be. So server 1 has processed request A and B already, server 2 is still processing request A and server 3 has processed A and is busy with request B. Request C is waiting at the boundary node, because the request can not flow into all queues yet.

The service duration of all servers has been modelled by a queue, and two sync channels to a boundary node K. This boundary node has an alternating arrival rate between its distribution and zero. The reason to model it like this is explained in section 3.3.3.

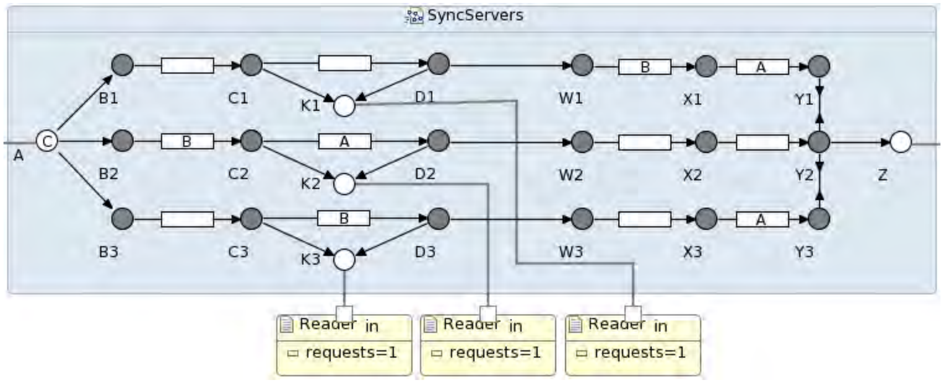


Figure 6.2: Reo connector of synchronizing servers

To model the system in Reo, every server has a limited number of four buffers, but this still gives problems when evaluating systems with multiple servers. The problem is that every FIFO buffer adds another synchronized region which might multiply the number of colouring tables by two. The readers itself increases the number of colourings in every table, because every reader can give or require a reason for delay. So this approach does increases the number of colourings and colouring tables, but at this moment there is no other way to model this service duration.

Table 6.1 gives an indication of the number of colourings and colouring tables compared to the number of synchronized servers. Every time a server is added the number of buffers increase by 4, while the number of tables multiply by around 4. The number of colourings multiply by about 9 after adding another server.

| *Servers* | *Tables* | *Colourings* |
|---|---|---|
| 1 | 16 | 97 |
| 2 | 58 | 783 |
| 3 | 238 | 6925 |
| 4 | 1030 | 62607 |

Table 6.1: Servers vs number of colourings

We would have liked to continue with adding more and more servers, but computing and storing the colouring tables with four servers was already very hard and caused problems with the virtual memory. Storing all colourings and colouring tables costs about 4 GB of the virtual memory, which caused out of memory errors in Eclipse. By increasing the maximum memory space of Eclipse from the default of 256M to 4096M solved the problem for the case with 4 servers, but adding another server would increase the memory so much that it was not possible any more.

The calculation and storing of the colouring tables is done before the actual simulation starts, and was also already implemented before building the simulator. It is also outside the scope of this research to change or improve the algorithm to calculate the colouring tables. But this observation shows that Reo is not very well suited for queueing theory. Queueing models always needs at least a few FIFO channels, but these channels creates asynchronous regions, which rapidly increases the number of colouring tables and colourings. Reo was designed to model systems with synchronization and without a time constraint. A lot of FIFO channels will make the whole system asynchronous, and this makes Reo not the right tool to model this at this moment.

In section 7.3 we suggested to add a channel to model a FIFO$_k$ channel. With this kind of channel, only the number of requests in the buffer is important instead of the number and the position in the queue. With this approach we only need one buffer for every queue, which can drastically decrease the number of colourings.

## Results

We will investigate what happens to the blocking probability, end-to-end delay and the average queue length when the number of servers increases. We will keep the arrival rate and the service rate of every individual server the same for all experiments. For the arrival rate we will use a uniform distribution between 0 and 1, while we use a uniform distribution between 0 and 0.9 for all servers. The arrival rate for the boundary node Z has been set to 'IfNeeded', because that node only has to wait till a request has been processed by all servers. It does not have a service rate on its own.

As explained above we would have liked to add more servers, but this was not possible. So, in this section we will look at the results of one to four synchronous servers, and we try to give an indication of what would happen if we add more servers.

We have also validated the end-to-end delay results by building the model in Extend [2]. The model we have build in Extend can be found in appendix B. The results of the Reo model and the Extend model were the same.

### Blocking probability

The first statistic we will look at is the blocking probability, a request will be blocked if one of the buffers between $B_i$ and $C_i$ is occupied and a request is waiting at boundary node A also. Because the average service duration is 0.45 and the average inter-arrival time is 0.50, the blocking rate is not very high. A blocked request will also not return, which reduces the load on the servers.

When looking at the results in figure 6.3, we will see that the blocking probability will increase with the number of servers. This is reasonable, because when the number of servers increase, the probability that one of the servers takes a long time will increase also. Because of this, every request takes a longer time to leave the system, so more request will be blocked. We can also see that the blocking probability increases with a decreasing rate, so the blocking probability will converge to a certain point. Because we could not evaluate more than four servers, this point is unknown for us.

Note that the graph is based on just 4 points, and is not very accurate for this reason. But because you can still see a trend in the results, the results are still shown in a graph. Also note that it is not possible to have a fractional amount of servers, in the graph the dots are connected to each other, but in practice this is not possible.

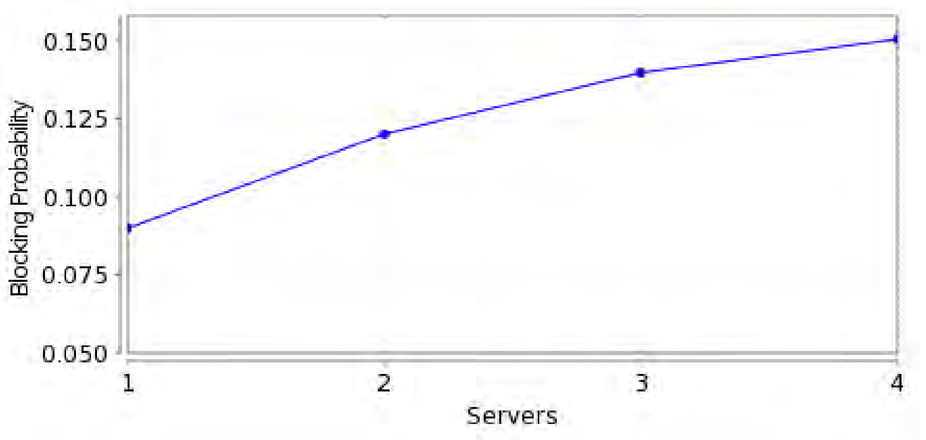


Figure 6.3: Blocking probabilities for Sync Servers

### End-to-end delay

The end-to-end delay for a request is an important measure for the system. When the number of servers increase, the end-to-end delay will also increase. The end-to-end delay is the total time from the arrival of a non blocked request at node A until it leaves the system at node Z. The end to end delay given by the Reo simulator does not include the waiting time at the boundary node so this has to be added to get the real end-to-end delay. The waiting time at the boundary node is given as the conditional waiting time at the boundary node.

The results of this delay can be found in figure 6.4. As we can see this end-to-end delay has a similar behaviour as the blocking probability. It increases with a decreasing rate. Eventually, it will also converge to a certain value, but based on the 4 chart points it is not clear to see where it converges to.

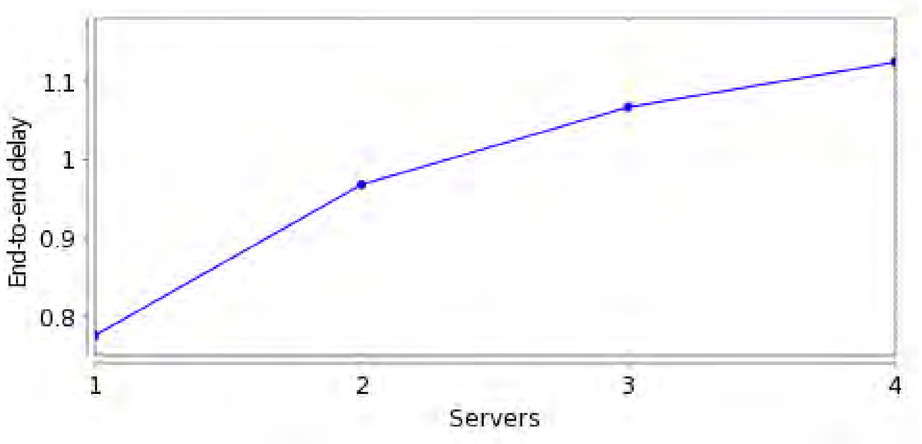


Figure 6.4: End-to-end delay for Sync Servers

### Queue length

The synchronizing servers have three different places where a request can wait: at the boundary node, before or in the server and after the server waiting for the other servers to finish a request. Because the service rate is the same for all servers, we will average the queue lengths over all servers. The calculation of the queue lengths can be done in a similar way as in section 5.4.

The results of this analysis can be found in figure 6.5. In this figure you can see that the queue before or in the server decreases with the number of servers, while the average queue length after the server increases. This is what we expected, because the number of jobs in the system is always limited to a maximum of two. Increasing the number of servers in the system also increases the probability that a request has to wait for other servers to finish a job, and therefore increasing the queue length after the servers. The queue length before the server decreases for the same reason.

The queue length at the boundary node increases from 0.1416 to 0.2381, but this is not visible very well in the figure. As with the other measures, all lengths will converge to a certain value, but more servers should be added to indicate what these values should be.

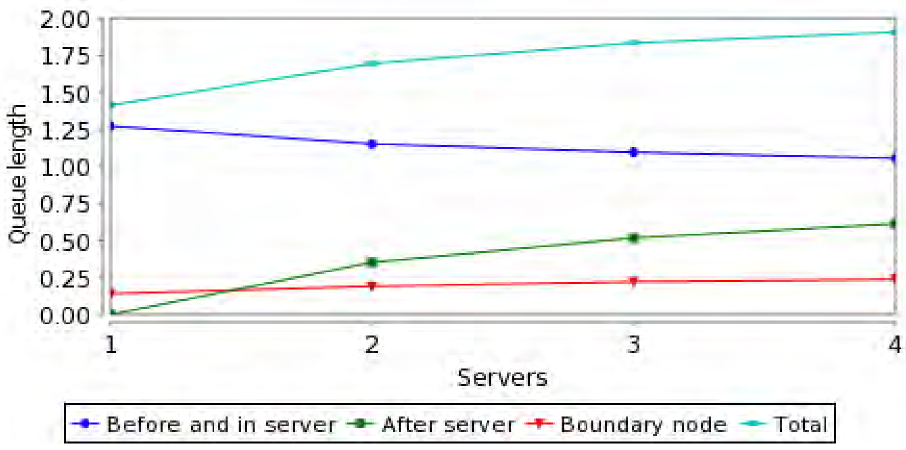


Figure 6.5: Queue lengths for Sync Servers

## 6.2 Variable number of servers

To illustrate the abilities of Reo, we will also describe a more difficult example. The example we will discuss is a queueing model with a variable number of servers. In this model we assume that the system has a limited amount of M parallel servers, where only r (r < M) are permanently on duty. The remaining M - r servers can be activated when there are too many waiting customers (requests).

Figure 6.6 shows such a model with 3 permanent servers and 2 extra available servers. In this system every incoming job will only be handled by one of the servers. An incoming job will go the first queue, and if there is space in the next queue or in one of the permanent servers, it will go there instantly. Once a job arrives when the queue for the permanent servers is full, it will be routed to one of the extra servers which do not have a queue of its own.

In practice, this kind of models can be used in call centers. Whenever it is too busy to handle all incoming calls with the current number of employees, another employee can help until it is quiet again.

We will model this system with Reo using 1 permanent server and 1 additional server whenever there are 3 customers in the queue. Whenever a request is assigned to the queue, it will wait until it has been serviced by server 1, so it will never go to server 2. The arrival rate and the service rate of the additional server will be kept the same over all experiments, while we vary the service rate of the base server to see what happens to the queue length, end-to-end delay, blocking probability and the ratio of requests handled by the second server.

The way to model this kind of system is shown in figure 6.7. Because this connector does not look very trivial, we will explain the ideas behind this connector.

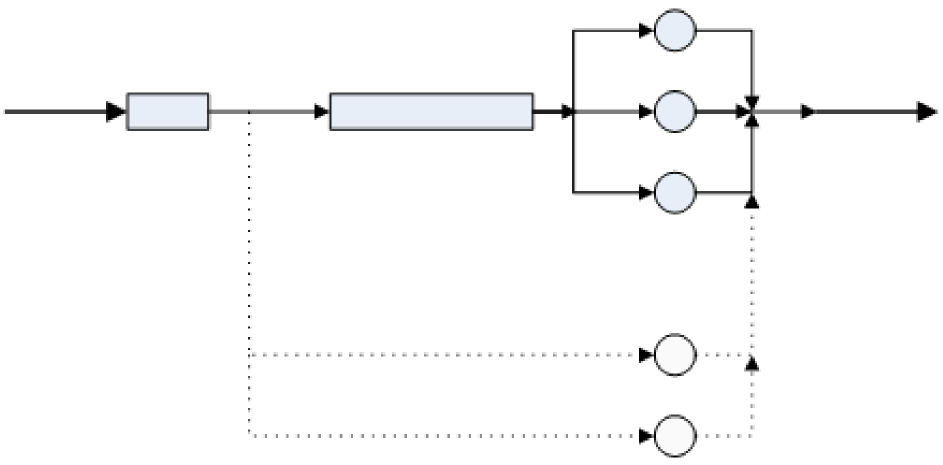


Figure 6.6: Diagram of variable number of servers

First, we have a normal queue with 3 waiting places between A and S1 traversing over B1 till B4, this is the normal waiting line for the permanent server. Because we want to use an alternative server when all 3 places are taken, we make an additional path from A to S2. Note that A is a router node, such that a job will be sent to one server only.

Only the paths from A to S1 and S2 would not be enough, because we have to make sure that S2 is only used if the buffer is full. If we would leave out the rest of the connector, the servers 1 and 2 will be chosen at random if server 2 is available and server one has space in the buffers. To be able to block requests flowing to S2 when the buffer is not full we introduced a shadow queue which is exactly the opposite of the normal queue. This shadow queue is build between C1 and C6, and the channels from the normal queue makes sure that every buffer in the shadow queue is empty when the normal buffer is full and vice versa. For this reason the shadow queue will start as full buffers because the normal queue is empty.

With this shadow queue we are able to activate S2 only when all spaces in the normal queue are occupied. Sync channels from E1 to all buffers in the shadow queue ensures that the channel from E1 to S2 is only used if all buffers in the shadow queue are empty. When these are empty, the normal queue is full, so in this case we need to use the second server. When one of the shadow buffers is occupied, the normal queue is not full and S2 can not be taken.

The last part of the connector we have not explained yet is the buffer from E1 to E2 and the drains following E2. The reason for this is the fact that taking the direction to E1 will fill up all buffers. But because the normal queue is still full, the shadow queue have to be emptied again. Having the extra buffer between E1 and E2 allows us to empty the shadow again after server 2 has been used.

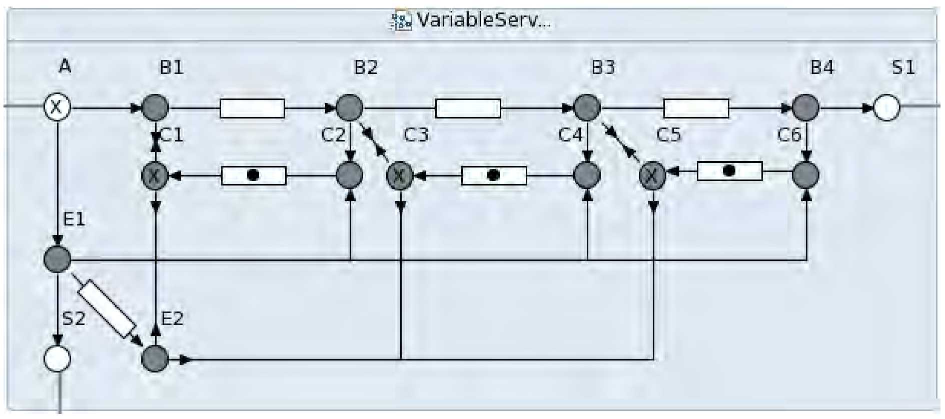


Figure 6.7: Reo connector for variable number of servers

## Results

Now we will evaluate the system and look what happens to the queue length, end-to-end delay, blocking probability and the ratio of requests handled by the second server. For the arrival rate we have chosen a Weibull distribution with $k = 1.5$. The second server has a log-normal distribution with $\mu = 0$ and $\sigma = 1$. The distribution of the first server will also be a log-normal distribution. We will keep the $\sigma = 1$ and vary the $\mu$ to see the results on the performance indicators.

### Average queue length

The average queue length of the queue before the permanent server is shown in figure 6.8. The average inter-arrival times at boundary node A is around 0.9, so when the average server duration of the base server exceeds this time, the server is not capable of handling all request. Because of this, the queue will fill up and the second server will be used to help the first server.

When the average service duration is around 0.9, the average queue length increases rapidly, until it converges to the maximum queue size. When the service time becomes large enough, almost all of the requests will be redirected to server 2 or blocked if server 2 is also not available.

### Ratio of requests handled by the second server

The ratio of requests handled by the second server is outputted directly by the simulator. This ratio can be found by the statistic 'Merger direction' at node C2, C4 and C6. The ratio displayed there is the number of requests handled by server 2 compared to the total number of non-blocked requests.

This ratio can be seen in figure 6.9, and you can see a similar behaviour as with the average queue length. After the average service duration of the first server has exceeded the arrival rate, more and more requests will be send to server 2, but because the base server is still able to process jobs by itself, it will not converge as

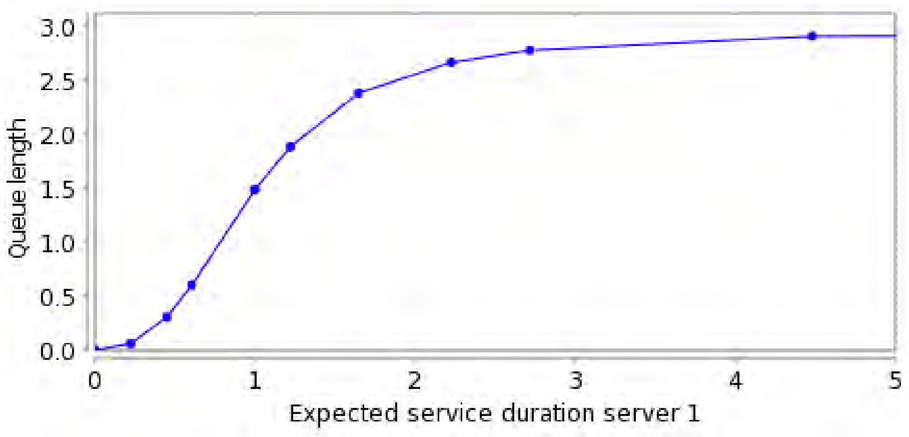


Figure 6.8: Average queue length for variable number of servers

fast as the queue length. Note that this chart has a larger x-axis than figure 6.8 because it did not converge as fast as the queue length. Eventually the ratio will be almost one when the service rate of the base server is so slow that almost all requests will be blocked or handled by server 2.

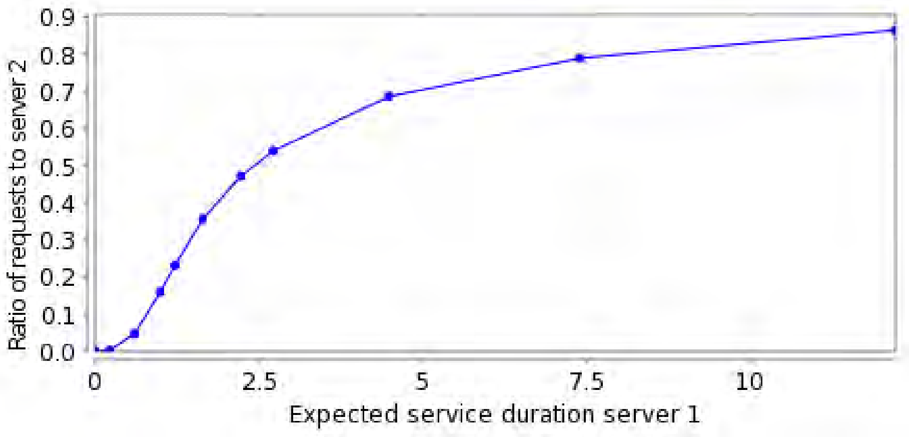


Figure 6.9: Ratio of requests to S2 for variable number of servers

### Blocking probability

When the queue of server 1 is full and server 2 is not ready yet, the request will wait at the boundary node. When another request arrives before one of the servers is available it will be blocked. The chart of this blocking probability is given in figure 6.10, this chart looks a lot like the figure in the ratio of requests by server 2.

Because the service rate of the second server is the same during all experiments, the blocking probability will converge to a certain value. Given the specified arrival rate and service distributions, the arrival process generates around 1.11 requests per time unit, while server 2 is able to handle about 0.61 requests per time unit, meaning server 2 is able to handle 55% of the arriving requests. As a result of this, if the service duration for server 1 is so long that it can hardly process any requests, the blocking probability would be at least 45%. Simulating this type of system with a very large service duration for server 1 gave a blocking probability of 51%, which is a bit larger than than the given 45% because of the randomness in the system.

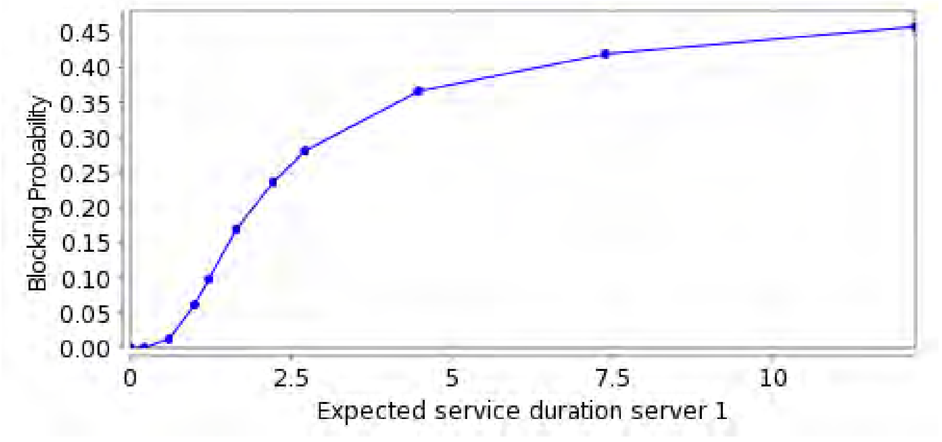


Figure 6.10: Blocking probability for variable number of servers

### End-to-end delay

The end-to-end delay of the second server is not relevant because this server has no queue, so the average end-to-end delay to this server is always the same. The end-to-end delay to the permanent server however is dependent on the waiting time in the queue. This end-to-end delay can be calculated in two ways, one is by adding some of the outputted statistics, the other is by applying Little's law. We already calculated the average queue length, so we can use that here.

In this case the easiest way is to get the end-to-end delay from the outputted statistics. This delay does not include the service time of the server and the waiting time at the boundary node A, so these values have to be added.

The other way to calculate the end-to-end delay is by applying Little's law. Because we already have the average queue length, the blocking probability and the ratio to server 2 we can calculate the delay. The end-to-end delay can be calculated with equation 6.1, where $W$ is the average queue length, $\lambda$ the arrival rate, $b$ the blocked requests ratio and $r$ the ratio of requests to server 2.

$$\frac{W}{\lambda(1-b)(1-r)} \tag{6.1}$$

Both methods produced the same results, which are shown in figure 6.11. In this figure we can see that the end-to-end delay is approximately linear over the average service duration. This is reasonable, because once the server is not able to handle all requests any more, the server will use its full capacity almost all the time, and when that is the case, the end-to-end delay will just be a function of the average service duration of the server. At the beginning the delay is not linear yet, because the server is not working at its full capacity, so it also has to wait sometimes before a new request arrives.

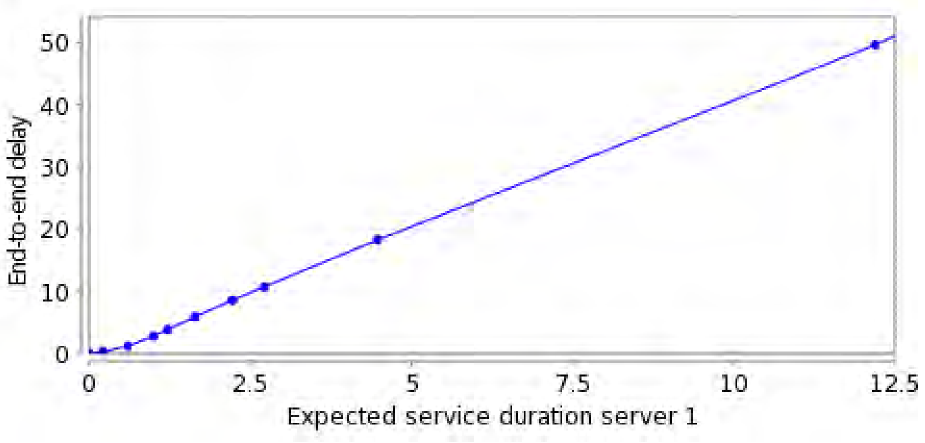


Figure 6.11: End-to-end delay for variable number of servers

## 6.3 Manufacturing Process

Another process which can be modelled with Reo is a manufacturing process. A job arrives in the system, then it will distributed over multiple machines, then some parts might need to be fitted together until the final product is finished.

The process we will model is given in figure 6.12. An arriving job will be duplicated twice to get 3 separate jobs to each of the servers 1, 2 and 3. When servers 1 and 2 are finished both, the products will be merged by server 4, but this merging fails with probability $p$. When a merge at server 4 succeeds and server 3 is also finished the products of server 3 and server 4 will be merged by server 5. After server 5 is finished, the request will leave the system. During the processing of a job, no other jobs will be allowed into the system. So there will always be only one job in the system, once this job leaves the system, a new job can flow into the system.

If we did not have the feedback loop in the system, the system can easily be modelled in Reo without the use of any FIFO buffer. However, because we have this loop we still need some extra constructions in the system. Figure 6.13 shows how we have modelled the process in Reo.

The first construction we used is making server 3 independent of server 1 and 2. If the merging of server 1 and 2 has failed, we do not want that it has to wait until server 3 is finished with its job before server 1 and 2 starts again. This can be achieved by adding a buffer and two sync channels to a reader with an alternating distribution for server 3. The reason for this kind of construction is explained in

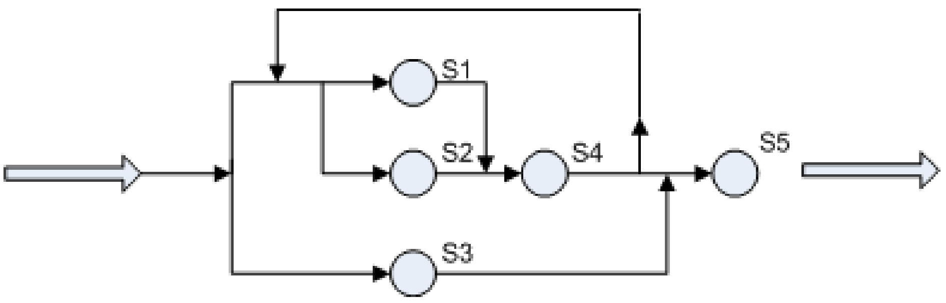


Figure 6.12: Manufacturing process

section 3.3.3. Once a request is in the buffer from K to M it is finished at server 3 and it has to wait until server 1 and 2 are finished.

Note that we only used the construction with an extra reader for server 3 and not for server 1, 2 and 4 also. The reason for this is that server 3 is only used once, while servers 1 and 2 might be used multiple times. If server 3 takes longer than server 1, 2 and 4, than the request from server 4 will just wait in the buffer from L to X. If server 1, 2 and 4 are slower, the request in server 3 might stay longer in the buffer from C to K than necessary. A request might be available at G already, but it has to wait until a colouring involving server 1, 2 and 4 has been finished until it can use that request, but this does not change the end-to-end delay.

The second construction we need is adding a FIFO buffer from A to R, the reason for this is to ensure that only one request at a time is present in the system. Once this request has left the system, the buffer from A to R is empty again and another request can enter the system.

The next part is adding two extra buffers, one from B to D, and one from I to B. In Reo it is not possible to have a direct loop without buffers, so extra buffers have to be added to make the loop possible. A direct loop is not possible because a node accepts only one of its sink ends, so you can not have multiple ends firing to a node in one colouring.

The last construction we need is the router node to model the failure rate. Normally a router node will choose one of its outputs at random, which gives a failure rate of 50%. But by adding more sync channels to one of the connected nodes will influence this failure rate. So in the given example there are 9 channels from J to I which gives a 90% probability to take one of the 9 channels to I. Because the failure rate should be independent of the state of server 3, another buffer is added from L to X.

## Results

When you look at the process in figure 6.12, you will see five servers. In the Reo model we have set these as shown in table 6.2. All other channel delays have been set to zero, while the delays on the boundary nodes A and Z have been set to 'Always' because we are only interested in what happens inside the connector. In this kind of system we are interested in the end-to-end delay and the buffer

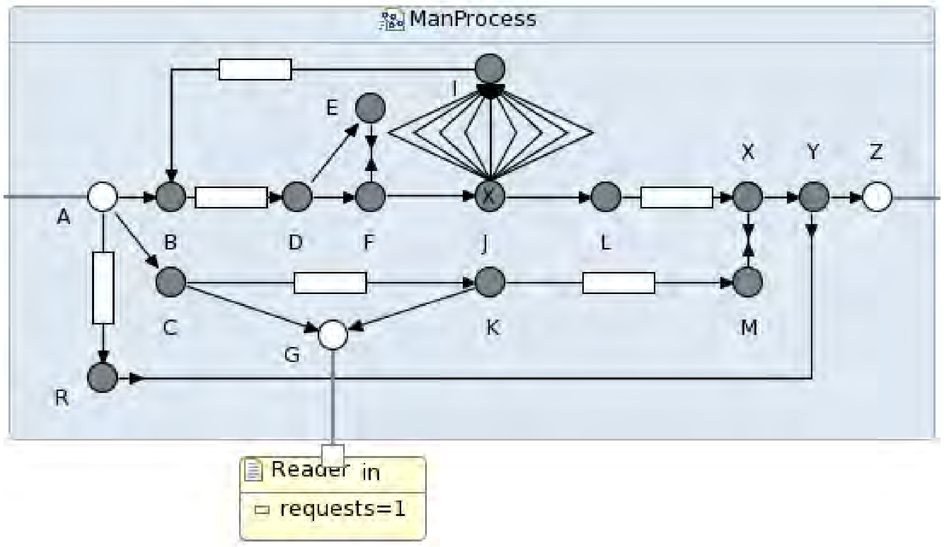


Figure 6.13: Manufacturing process in Reo

utilizations.

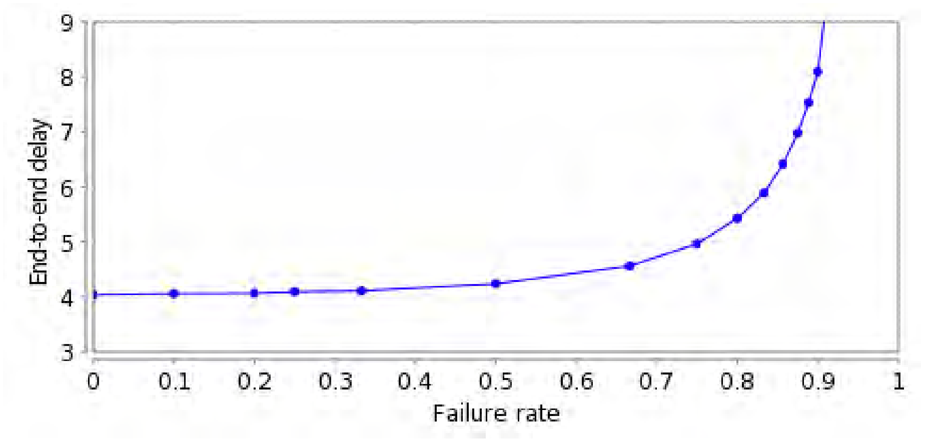
| *Server* | *Distribution* | *Reo object* |
|---|---|---|
| 1 | Beta(1, 1) | Sync(D-E) |
| 2 | Beta(1, 3) | Sync(D-F) |
| 3 | Chi2(3, true, true) | Node G |
| 4 | Tri(0, 0.5, 0.2) | Sync(F-J) |
| 5 | Exp(1) | Sync(X-Y) |

Table 6.2: Delays for manufacturing process

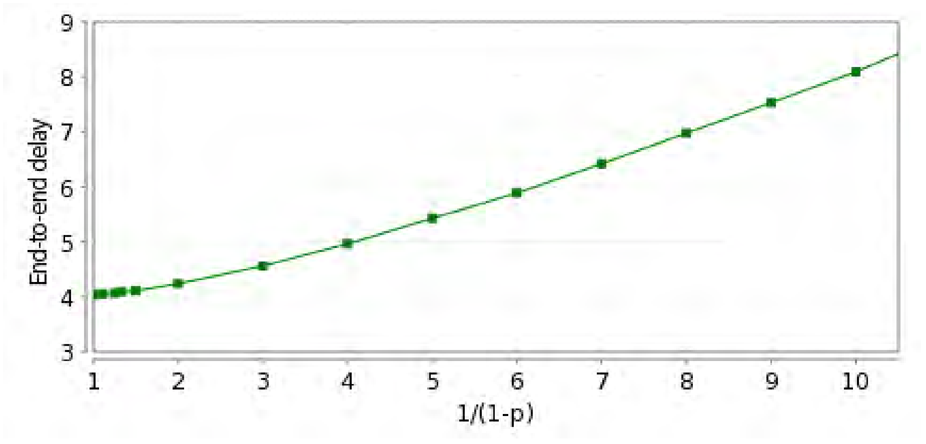
### End-to-end delay

First we will look at the end to delay compared to the failure rate. Figure 6.14 shows this comparison in two ways, figure 6.14(a) gives the failure rate against the end-to-end delay while figure 6.14(b) transformed the x-axis. The transformation to $1/(1-p)$ can be interpreted as the average number of attempts before the merging succeeds.

In figure 6.14(a) we can see that the end-to-end delay barely changes until the failure rate is over 0.5. After that, the failure probability is higher than the probability of success so the expected number of failures increases. The delay still does not increase very fast because the expected duration of the third server is greater than the expected duration of one iteration in the upper part of the connector (server 1, 2 and 4). However, when the expected number of iterations increases with the larger failure probability, the probability that the third server has to wait on the other part also increases. The delay will go to infinity when the failure probability goes to 1, which is what we have expected.



(a) Failure rate vs delay



(b) 1 / (1 - failure rate) vs delay

Figure 6.14: Failure probability vs end-to-end delay for manufacturing process

In figure 6.14(b) the axis is transformed, which indicates approximately a straight line after an expected number of around 4 attempts before a successful merge. The expected duration of the upper part is a function of the number of attempts, and after a certain expected number of attempts, this expectation will be much higher than the expectation of the last part.

When we have no failure at all, you might expect an end-to-end delay of 4 which is the sum of the expectation of server 3 and server 5, which are respectively 3 and 1. But the delay given by the simulator is 4.0373 (with an 95% confidence interval of [4.0248, 4.0513]). The reason for this is that server 3 is not always slower than server 1, 2 and 4, so sometimes an item has to wait in buffer K-M. We can confirm this by looking at the buffer utilization for buffer K-M and comparing this with the utilization for Sync channel X-Y. If none of the items have to wait in this buffer these two utilization should be the same. The difference of 0.00003 confirms that some of the requests have waited indeed.

### Buffer and channel utilization

In this type of process we can also look at the utilization of the buffers and the utilization of server 5. Server 5 will be used after server 1 till 4 are finished and successful, so the utilization of 5 can be seen as the influence of the fifth server on the total end-to-end delay. The results of these utilizations is shown in figure 6.15. The buffers A-R and I-B are left out of this figure because these utilizations are respectively 1 and 0 over all experiments.

When we look at this figure we can see that the utilization of buffer B-D and L-X are mirrored in the line $y = 0.5$. This is logical because an item is always in buffer B-D when it is in service by server 1, 2 and 4 or in buffer L-X when it is finished with servers 1, 2 and 4. When it is in buffer L-X it is either busy with server 5 or waiting for server 3 to finish. The difference between these two options can be seen by looking at the utilization of X-Y, which is the utilization of server 5. The difference between the utilization of L-X and the utilization of X-Y indicates that an item has to wait for server 3.

The utilizations of C-K and K-M are also mirrored, where C-K indicates that an item is in service by server 3 and K-M indicates that an item is finished by server 3 and it is in service by server 5 or that it has to wait on server 1, 2 and 4.

When there is no failure at all, we can see that the utilization of L-X and X-Y are almost equal, meaning that server 1, 2 and 4 are almost always finished later than server 3. This is also what we expected because the expected service duration of server 3 (3) is much greater than the combined service duration of server 1, 2 and 4 (about 0.6).

When the failure rate is close to one, server 3 is hardly used any more so the utilization of C-K will go to zero, while the utilization of K-M will go to one. Also the utilization of L-X will go to zero because the items will never leave the loop, which causes that the utilization of B-D will go to one. Because server 5 is also not used any more when the failure rate goes to one, this utilization will also go to zero.

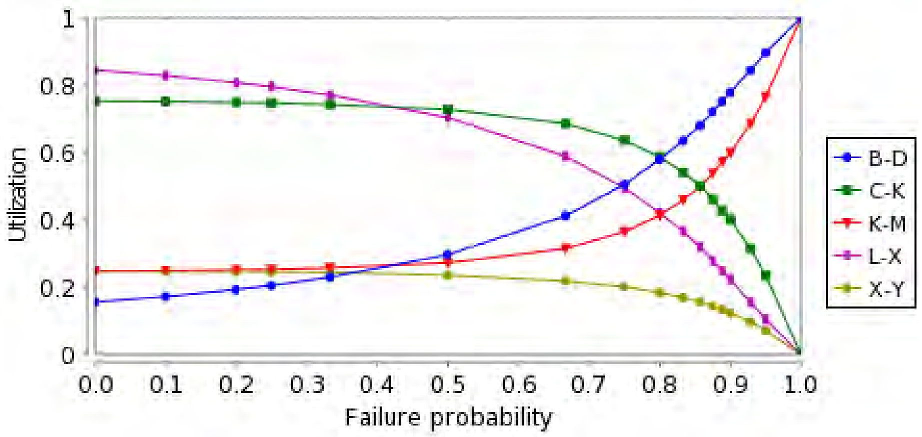


Figure 6.15: Buffer and channel utilization for manufacturing process

## 6.4 Conclusions

In this chapter we showed the abilities of the Reo simulator with different general stochastic distributions. Especially the system in section 6.1 is typical for Reo, because in this case synchronization is very important. Although it is very hard or even impossible to evaluate this system analytically, the Reo model also gave problems. When the number of servers increases, the number of colourings increased very fast too. This caused memory problems when we used too many servers. Four servers was already very hard and costs 4GB of the virtual memory. Adding another server was already not possible any more. For this reason we could not continue with our analysis of this type of system, causing we had only four observations for every statistic, which made it impossible to predict the behaviour with an infinite number of servers.

In section 6.2, we showed that it is also possible to model more difficult systems. Although we restricted us to one permanent server with a queue of three places, and one additional server when the queue is full we could still get some nice results. In principle it is possible to extend the number of servers, but this would cost a lot of extra channels.

Finally, in section 6.3, we modelled a manufacturing process where a job is processed by multiple servers. One of the servers could fail with a probability p, after which the job will be processed again by some servers. Normally, the failure rate will be 50% because a Reo node will choose one of its source ends at random. By adding the number of channels to one of the connected nodes, the failure could be adjusted. In this system we evaluated what happened to the end-to-end delay and buffer utilizations when the failure rate increases.

# Chapter 7

# Future work

In this chapter we will discuss the future work which can be performed on the Reo simulator. During the implementation, we have made certain choices which influenced the behaviour of the simulator. With some workarounds, we were still able to analyse the systems we want, but with some improvements the simulator will be even more powerful. In section 7.1 we discuss the possibility to have other drivers for the simulator, for example to make the system asynchronous. In section 7.2 we suggest to add a possibility to do multiple simulations in a row, to do parameter variation. Section 7.3 discusses the possibility to add a channel which changes the current $\text{FIFO}_1$ into a $\text{FIFO}_k$ buffer to save colourings. Section 7.4 discusses the possibility to have a variable instead of a fixed number of batches in the simulation. Section 7.5 suggests to produce a chart with the distribution of the individual results instead of using aggregated results instantly. In the simulation we only support the channels which are most frequently used, in section 7.6 we stated that the other less frequently used channels should also be supported in the future. Finally, section 7.7 indicates that the simulator should be able to detect and inform the user about (possible) wrong input.

## 7.1 Different drivers

The most important part of the simulator is determining the states and the duration of states. At this moment we use the colouring algorithm to determine the possible next steps, and based on the availability of the boundary nodes we choose one of the possible colourings at random. When we have chosen one of these colourings we will activate this colouring for a certain amount of time until it is completely finished before choosing the next colouring.

This approach can be changed in two ways in the future, first of all the calculation of the end time of the colouring can be changed. At this moment the calculation will be done using a depth first traversal through the connector, where the longest path will count as the ending time of the colouring. This calculation is implemented in a separate class to ensure that this method can be changed easily.

For example, the calculation of the ending time of the colouring can be changed by traversing through the animation steps, which also involves the used channels. In each step we can take the maximum of the samples from the involved channels in that step, then we can sum up these separate values. This approach would give ending times which are equal to or greater than the current approach based on the depth first traversal.

Another more important change to the current simulator would be to change states before a colouring has been finished. The restriction to activate only one colouring until it has been finished completely, is a reasonable restriction when we look at the way Reo is designed. Reo is designed as a synchronization language where all nodes will come to a consensus about what happens next. Then this transition will happen and you are in a new state. But at the same time, this restriction gave some limitations to the systems we wanted to model as explained in section 3.3. With some workarounds it was still possible to model almost everything we wanted, but it always created a lot of colourings and colouring tables.

The models we tried to model always needed asynchronization, where every asynchronous region should be independent of the other regions. So whenever a certain region is busy, we would like to be able to activate another region also. Multiple regions can be activated within one colouring, but once a colouring has been started, another region can not be added until the colouring is finished.

In the future it would be very handy to have a driver which handles every region independent of the others. For this we might need a new automata model which should be used as the driver for the simulator. But, inventing, proving and using this automata model will take a lot more time which could not be done within this internship. This would also require some more changing to the simulator than just the calculation of the end time of the colouring. But when this change has been implemented, the workarounds discussed in section 3.3 will not be needed any more.

## 7.2 Multiple simulations

A very handy addition to the simulator can be to add a possibility to do multiple simulations in a row. In section 5.1 we used PRISM to get steady state behaviour of continuous time Markov chains. In PRISM it is possible to vary one or more of the parameters to see what happens to the steady state behaviour of the system. In this section we varied the arrival rate of a boundary node from 0.1 to 2.5 to produce a graph with the blocking probabilities at every boundary node.

In the Reo simulator however, it is not possible to vary one parameter of the simulation. So for every choice of the arrival rate, another simulation have to be started. For this reason we have chosen to simulate only a limited number of arrival rates at the boundary node.

In the future it would be a good addition to make it possible to vary one of the parameters over multiple simulations in a row. With this extra option it would also be nice to change the result tabs generated by the simulator. At this moment it generates the averages over all batches in one simulation. When we are able to do multiple simulations in a row, the batch results of all simulations should be displayed. Another graph can be added also to show the average behaviour of the statistic over all simulation runs.

## 7.3 Buffers

In all queueing models we investigated with Reo we had the problem that the FIFO buffer of Reo has only one waiting place. When investigating some of the models, we would have liked to have infinite or a large enough finite number of buffers. Adding a lot of buffers to a Reo circuit will make the calculation of the colourings and the simulator itself slow. After enough buffers have been added we also got problems with the memory usage, which is explained in section 6.1.

A possible solution to this problem is to have another Reo channel where you can specify the number of places, then you do not need a lot of buffers in series, which will decrease the number of colourings drastically. When this channel has been added, the semantics for the colouring of this channel also have to be defined. We also have to adjust the buffer statistics outputted by the simulator, now it will only output the percentage of time the buffer is full. But when we have a buffer with multiple waiting spaces we also want to know the average queue length.

The delays for the buffer can stay the same as defined for the FIFO channel with one place, one delay for flowing into the buffer and one delay for flowing out of the buffer. When connecting multiple buffers in series we also set every delay to zero to indicate that a request will flow into the last empty buffer instantly. So the restriction that we only have two delays with a buffer with multiple places will not make a difference.

## 7.4 Variable number of batches

In the current simulator, it is only possible to specify a fixed number of batches. In the future this can be extended to allow a variable number of batches. Instead of specifying the number of batches, the user have to specify how accurate the confidence interval should be. For example, the user should be able to specify that the 95% confidence interval should have a maximum length of 0.01.

It is probably not enough to specify that a confidence interval should have a maximum width because the simulator have many statistics. Until all of these statistics have reached an interval smaller than the specified value could take a very long time. For this reason it might be needed to specify which of the statistic or statistic category should have a confidence interval smaller than the specified value.

Once this value has been specified we can simulate as usual, with the difference that we check the confidence interval after each batch. If the interval is not small enough, the simulation continues with the next batch until the interval is small enough. The simulation can stop whenever all specified statistics have an interval smaller than the specified value.

## 7.5 Distribution of results

The simulator produces only the averages of the statistics, but it might also be nice to add the possibility to add charts which displays the distribution of the individual observations. A disadvantage of this could be that every single observation has to be saved individually instead of aggregated. Plotting all these points could become a problem, as experienced with the current charts with the averages over time.

The package used to create the charts had the problem that it is was really slow with a lot of chart points. For this reason we created an option in the simulation to specify the maximum number of chart points in the graph. To create graphs with the distribution of the result, possibly another package should be used.

Some statistics should also be changed to enable the possibility to display the distribution of the results. For example, with the buffer utilization we display the ratio of time the buffer is full. So we could save the points where the buffer changes from empty to full, but this will not give any relevant results. If we change the statistic to the time the buffer has been empty or full, it is possible to give a distribution of the results.

## 7.6 More Reo channels

At this moment we have restricted the simulator to work with the basic Reo channels (Sync, LossySync, FIFO, SyncDrain, SyncSpout, AsyncDrain and AsyncSpout). But there are some more channels which are not supported by the simulator, these are the Filter, Transform and Timer channels. In the future these channels can also be supported, together with any possible future channels added to Reo.

Another possibility could be to include support for components. At this moment the simulation is done on a connector level, which do not contain components. By converting a connector into a component, all channels will be hidden which saves a lot of colourings. But it is not clear how the delays should be specified on such components. This possibility should be investigated further before it can be used.

## 7.7 Warnings and errors

Before the simulation starts, the simulator only checks if the simulation options are of the right format. So for example if the number of batches is an integer and if the confidence interval is a double. If any of these parameters is of the wrong type, an error will be shown that the options are not correct. An addition to this would be to highlight the values which are not correct.

Next to that it would be nice to check the other input values also. When a distribution field is left blank, the constant(0) distribution will be used. If the user fills in a distribution with an incorrect syntax, the constant(0) distribution will also be used. Instead, a warning or error should be produced to indicate to the user that the entered value is not correct. The parameters of the distributions will also not be checked, the parameters should be chosen such that it can not produce negative results. For example, if we declare a uniform(-1, 1) distribution, there should be an error to indicate that this kind of distribution can have negative values.

When the distribution of one of the boundary nodes have been left blank, we end up with an infinite simulation when we base the simulation end on time. Because every request arrives at this boundary node will arrive at time zero, we will never simulate further than time zero and the simulation will never stop. If events are used for the end of the simulation, we do not have problems with an infinite simulation, but we still do not get any relevant results. It is also possible to have an infinite simulation when the distribution of the boundary nodes have been set to 'IfNeeded' or 'Always' and all delays are zero.

Another possibility can be to give an indication of the number of events in every batch. When the simulation end is based on time, it is hardly possible for the user to know how many events will happen in the simulation. If an estimation will be given, the user knows if the simulation period is too long or too short. The estimation can be given based on the number of arrivals at every boundary node, which can be calculated easily. The number of colouring events is harder to estimate, but an indication based on the average duration of every colouring should be possible.

# Chapter 8

# Conclusion

The Reo coordination language was designed to model synchronization between different parts of complex systems. Originally, Reo was designed without a notion of time. Later, additional models made it possible to make Reo stochastic by defining delays on channels and arrival rates on the boundary nodes of the system. With this approach it is possible to model such systems as continuous time Markov chains to perform quality of service analysis. However, this conversion is only possible if the Markov property has been met, which is only the case when using exponential distributions to model the arrival rates and delays.

When using other general distributions, we can not generate Markov chains any more to get quality of service measures. Analytically, it is also very hard to solve complex systems with synchronization and various general distributions on the channels and boundary nodes. For this reason, the goal of this research was to develop a simulation model to evaluate complex systems modelled in Reo.

This simulation model has been implemented within the Eclipse Coordination Tools framework, which includes the graphical user interface and other plug-ins for Reo. The simulator uses discrete event simulation to simulate a Reo system. For this discrete event simulation, it will use the colouring semantics as a driver for the simulator. This colouring semantics indicates which parts of a connector should have data flow given a configuration of the system. With this semantics we are able to determine all possible next steps in the discrete event simulation.

An important part of the simulator is choosing and executing the colouring. In our approach, we limited us to choose a colouring and execute this colouring until it is finished completely because this is how Reo is designed, and how the colouring semantics should be used. When we have chosen a colouring, we should decide when this colouring should end. This is implemented in a separated module of the simulator to make it possible to add different ways to calculate the ending time of the colouring later. At this moment, only one option to calculate the ending time of a colouring has been implemented using a depth first traversal through the connector.

Because we have chosen to execute a colouring until it is finished completely, the system is handled completely synchronous as Reo is designed. When modelling systems where parts of the system should be asynchronous, this gave some complications. For example, when modelling queueing models in Reo, we need asynchronization which can be achieved by using some workarounds. With these workarounds we were able to model and simulate any system we wanted, but in the future these workaround might not be needed any more when a new engine for the simulator has been designed.

With the simulator, we aimed to detect all types of 'strange' behaviour of the system, for example when the system will not converge to a steady state. For this purpose, we have the option to detect deadlocks and livelocks explained in section 3.1.2. When a deadlock occurs, the system will got stuck, which will ruin the results of the simulation. Another option we created to detect if a statistic has converged to a certain value, is a chart for every statistic with the average statistic value plotted over time for every batch. In this chart, we can see if the statistic converges to a certain value, and also if all batches converges to the same value.

When using the simulator, we also discovered a problem with the colouring semantics when evaluating large systems. If we model systems with a lot of FIFO buffers, we created asynchonization in the Reo system causing a lot of extra colourings. When we have too many colourings, it will cost a lot of virtual memory causing memory problems. The calculation of the colourings is not the scope of this research, but it is something which needs attention.

We validated the simulator by comparing the results of the simulator with results of CTMC, Erlang-X calculator and other simulators. The results of the Reo simulator were the same as with the other methods when the simulation period was long enough. After this validation we used the simulator to analyse systems which could not be analysed before.

So with this simulator in Reo we are able to perform quality of service analysis on almost any system modelled in Reo with general stochastic distributions on the delays of the channels and the arrival rate on the boundary nodes.

# References

[1] Erlang-X Calculator. `http://www.math.vu.nl/~koole/ccmath/ErlangX/`.

[2] ExtendSim Simulation Software. `http://www.extendsim.com/`.

[3] JFreeChart. `http://www.jfree.org/jfreechart/`.

[4] JSci - A science API for Java. `http://www.jfree.org/jfreechart/`.

[5] Reo Coordination Language. `http://reo.project.cwi.nl/`.

[6] Simulation source code. `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/browser/reo/cwi.reo.stochastic`.

[7] Single Queue Simulator. `http://staff.um.edu.mt/jskl1/simweb/sq1/sq1.html`.

[8] F. Arbab. Reo: a channel-based coordination model for component composition. *MSCS*, 14(3):329–366, 2004.

[9] F. Arbab. Abstract Behavior Types: A Foundation Model For Components And Their Composition. *Science of Computer Programming*, 55(1-3):3 – 52, 2005.

[10] F. Arbab, T. Chothia, S. Meng, and Y.-J. Moon. Component Connectors with QoS Guarantees. In *COORDINATION*, pages 286–304, 2007.

[11] F. Arbab, T. Clothia, R. D. van der Mei, M. Sun, Y.-J. Moon, and C. Verhoef. From Coordination To Stochastic Models Of QoS. volume 5521 of *Lecture notes in computer science*, pages 268 – 287. Springer, 2009.

[12] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. In *WADT*, pages 34–55, 2002.

[13] C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[14] D. Clarke, D. Costa, and F. Arbab. Connector colouring i: Synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007.

[15] D. Costa. *Formal Models for Context Dependent Connectors for Distributed Software Components and Services.* Phd thesis, 2009.

[16] S. Meng and F. Arbab. QoS-Driven Service Selection and Composition. In *ACSD*, pages 160–169. IEEE Computer Society, 2008.

[17] H. Tijms. *Operationele Analyse.* Epsilon Uitgaven, 2002.

# Index

# Appendix A: Data diagram

Figure 8.1 gives a representation of the most important objects, attributes and functions. This representation does not give a full overview of all elements, but it does give an indication.

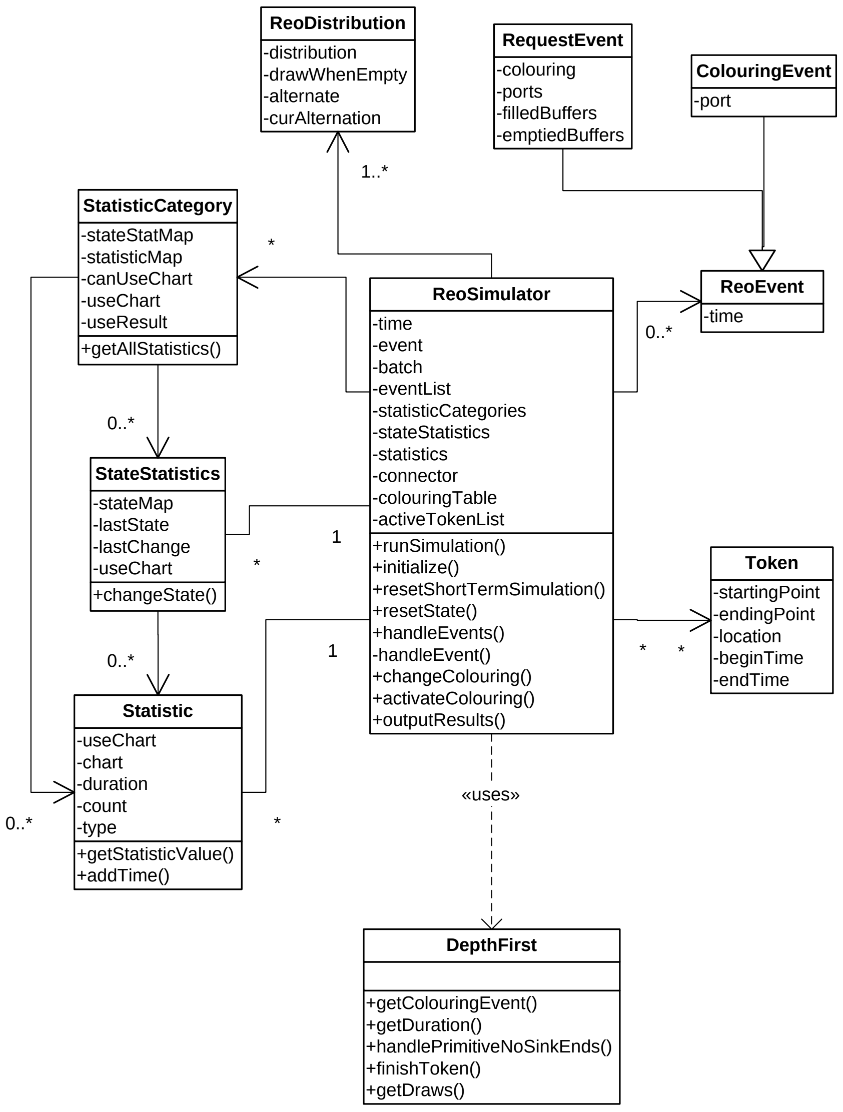


Figure 8.1: Data diagram of the simulator

# Appendix B: Extend model

To check the end-to-end delay in the synchronizing server model of section 6.1, we also modelled the system using ExtendSim [2]. Figure 8.2 shows how we have modelled that system.

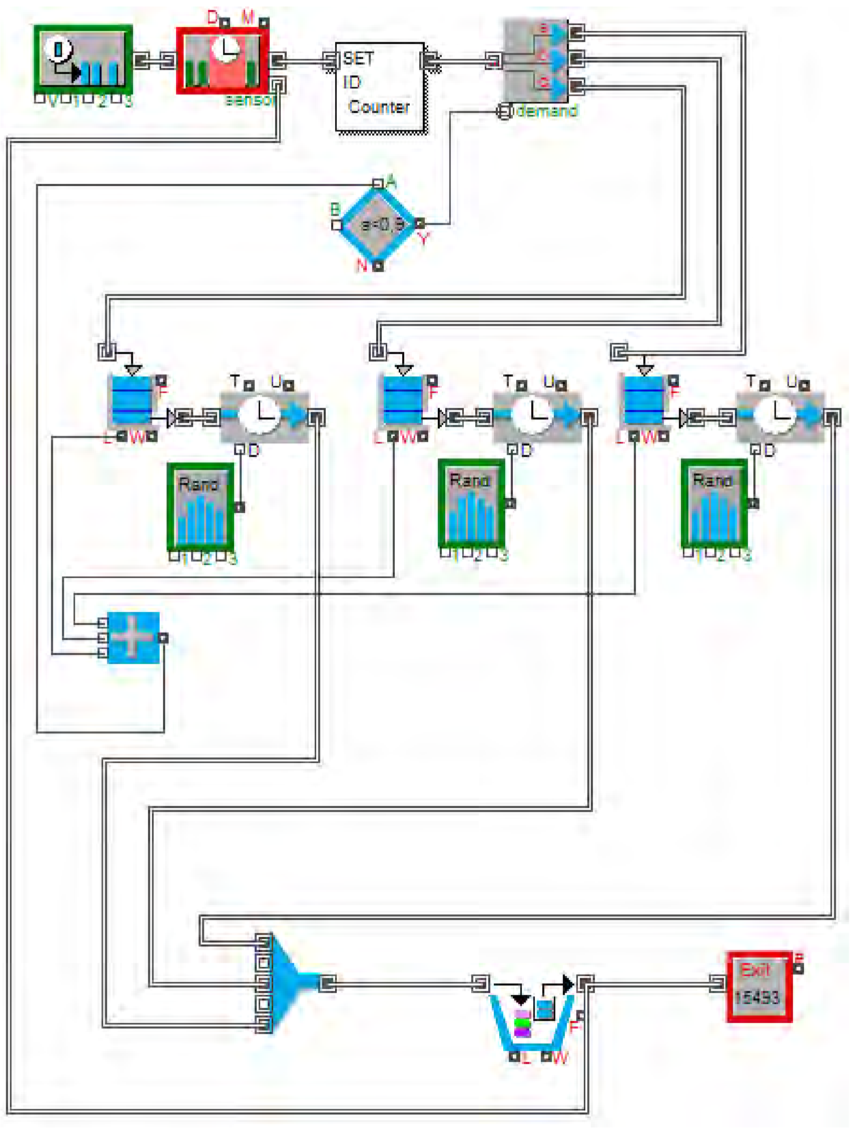


Figure 8.2: ExtendSim model for synchronizing servers

# Appendix C: Quick start guide

This appendix is meant as a quick start guide for using the Reo simulator. For a description how to use Reo, see [5]. After installing Reo, you might have to add the simulation view to the Reo perspective via 'window' - 'show view' - 'other'. In this appendix we will discuss which distributions can be used and where these have to be defined. This appendix will also give a short description of the simulation options and the output it will generate.

## Distributions

We can define the distributions on the channels by clicking on a channel and going to the 'Delay' tab in the properties view in Eclipse. Depending on the channel, you can specify one or two distributions. All possible distributions are displayed in table 8.1. The inter-arrival rate on the boundary node can be specified by clicking on a boundary node and going to the 'Arrivals' tab in the properties view. Here you can specify the same distributions as with the channels, and you specify if you start with a request by setting 'Start with request' to 'true'.

| *Distribution* | *Par 1* | *Par 2* | *Par 3* | *Short* | *Remark* |
|---|---|---|---|---|---|
| Beta | $\alpha$ (d) | $\beta$ (d) | | | |
| Binomial | n (i) | p (d) | | Bino | |
| Chi$^2$ | k (i) | | | | |
| Constant | value (d) | | | Con | |
| Exponential | $\lambda$ (d) | | | Exp | |
| F | $d_1$ (d) | $d_2$ (d) | | | |
| Gamma | k (d) | | | Gam | Uses $\theta = 1$ |
| Lognormal | $\mu$ (d) | $\theta$ (d) | | Logn | |
| Poisson | $\lambda$ (d) | | | Poiss | |
| Triangular | low (d) | high (d) | avg (d) | Tri | |
| Uniform | low (d) | high (d) | | Unif | |
| Weibull | k (d) | | | Wbl | Uses $\lambda = 1$ |
| IfNeeded | | | | | |
| Always | | | | | |
| File | path (s) | loop (b) | | | loop is optional |

Table 8.1: Distributions

For a description of the special cases 'IfNeeded' and 'Always', please refer to section 4.1. For the arrival rate at a boundary node it is also possible to add one or two extra booleans, the first is to indicate that a request should be sampled when the boundary node is empty again. The second parameter is to indicate that the boundary node should iterate between samples of the distribution and 0. When you will need these options is explained in sections 3.3.2 and 3.3.3.

### Options

The following list indicates the options to use the simulation. For some of the options the type of the option is displayed between brackets.

- Type of simulation: long- or short-term simulation
- Base simulation end on: events or time
- Warm-up period (double): time or number of events till the simulation starts gathering statistics
- Simulation length (double): time or events till the simulation stops
- Number of batches (integer): the simulation length will be split into multiple batches to be able to give a confidence interval for the statistics. The number of batches is normally chosen between 25 and 50 [17]
- Confidence interval (double between 0 and 1): how accurate the confidence interval should be
- Detect deadlock: if enabled, the simulation will stop whenever we are in a deadlock. If disabled, the simulation will go on so the user can see what happens with the statistics after the deadlock.
- Detect livelock: ability to specify if a livelock should be detected
- Internal colourings for livelock (integer): specify how many colourings in a row without any involved boundary nodes should be chosen to indicate a livelock. Also see section 3.1.2
- State to stop simulation (optional): possibility to define a certain system state in which the simulation should stop. Also see section 4.5
- Special state (optional): possibility to define a system state to get statistics from. Also see section 3.1.1
- Seed (integer): define a seed if you want to produce the same results in every consecutive simulation with the same parameters
- Max chart points (integer): maximum number of chart points for the charts. Also see section 4.6.1.

### Output

The following list indicates the possible outputs for the simulation. The tab 'results options' in the simulation view allows the user to disable certain output statistics. It is recommended to disable the 'System state' and 'Colourings' when using large connectors, because the number of colourings and states can become very large. For a more extensive description of the different outputs, please refer to section 3.1.

- Actual loss ratio: ratio of requests lost in LossySync channel, compare to the total number of requests into the channel
- Average conditional waiting time: average waiting time of all requests which will not be blocked
- Average waiting time: average waiting time over all requests
- Buffer utilization: ratio of time the FIFO buffer is full

- Channel locked: ratio of time a channel is locked because a colouring involving this channel is active
- Channel utilization: ratio of time a channel is actually in use
- Colourings: ratio of time a colouring is active
- End-to-end delay: average total delay between two points
- Inter-arrival times: average time between two arrivals at an ending point from a certain starting point
- Merger directions: ratio of requests from a certain sink end of a node compared to the total number of requests into the node
- Node state: the ratio of time a node is in a certain state
- Request observations: the ratio of requests which observes the node in a certain state compared to the total number of requests arriving at the node
- System state: ratio of time the system is in a certain state
- Special state: ratio of time the system is in the specified special state