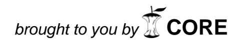
Centrum voor Wiskunde en Informatica

View metadata, citation and similar papers at core.ac.uk

brought to you by CORE



provided by CWI's Institutio

REPORT *RAPPORT*

SEN

Software Engineering

Software ENgineering

A distributed computational model for Reo

C.T.H. Everaars, D.F. de Oliveira Costa, N.K. Diakov,
F. Arbab

REPORT SEN-E0601 FEBRUARY 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2006, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A distributed computational model for Reo

ABSTRACT
The work described in this document aims at producing a formal computational model for the Reo coordination language, that can facilitate the implementation of Reo circuits in a distributed computing environment. The model introduced here partially covers what Reo requires - it implements a less strict form of the merge behavior of mixed nodes. While this already allows computing of a large class of useful circuits, it does not properly deal with some synchronous circuits that contain LossySync channels. This work has lead to a new and more powerful approach to computing the behavior of Reo circuits, called Connector Coloring.

1998 ACM Computing Classification System: C.2.4, D.1.3, D.1.m, D.3.2, D.3.3, F1.2
Keywords and Phrases: Coordination Language; Reo; Computational Model

A Distributed Computational Model for Reo

Functional Specification of a partial model

Kees Everaars, David Costa, Nikolay Diakov, Farhad Arbab
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

February 6, 2006

Abstract: The work described in this document aims at producing a formal computational model for the Reo coordination language, that can facilitate the implementation of Reo circuits in a distributed computing environment. The model introduced here partially covers what Reo requires - it implements a less strict form of the merge behavior of mixed nodes. While this already allows computing of a large class of useful circuits, it does not properly deal with some synchronous circuits that contain LossySync channels. This work has lead to a new and more powerful approach to computing the behavior of Reo circuits, called Connector Coloring.

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Rationale . . . 4
  - 1.2.1 Global View . . . 4
  - 1.2.2 Communication Infrastructure . . . 4
  - 1.2.3 Look-up and Commit vs Roll-back . . . 5
  - 1.2.4 Concurrency . . . 5
  - 1.2.5 Partial Failure . . . 5
- 1.3 Reo concepts . . . 5
- 1.4 Basics of the Operational Semantics . . . 6

2 Functional Notation **7**
- 2.1 Normal Function Declarations . . . 7
- 2.2 Function Declarations Using Pattern Matching . . . 8
- 2.3 List Comprehension a.k.a. ZF-notation . . . 8

3 Data Types **8**
- 3.1 Data Type Definition . . . 8
- 3.2 Auxiliary Functions per Data Type . . . 9
  - 3.2.1 *Node* . . . 9
  - 3.2.2 *CE* . . . 9
  - 3.2.3 *Channel* . . . 9
  - 3.2.4 *PendingOp* . . . 9

4 Simplified Computational Model **9**
- 4.1 Auxiliary Node Functions . . . 11
- 4.2 Simplified Node Behavior . . . 11
  - 4.2.1 Simplified *Accepts* . . . 11
  - 4.2.2 Simplified *Offers* . . . 12
  - 4.2.3 Simplified τ . . . 12
- 4.3 Some Simplified Synchronous Channels . . . 12
  - 4.3.1 Simplified `Sync` . . . 13
  - 4.3.2 Simplified `SyncDrain` . . . 13
  - 4.3.3 Simplified `SyncSpout` . . . 13
- 4.4 Some Examples . . . 14
  - 4.4.1 Circuit 1 . . . 14
  - 4.4.2 Circuit 2 . . . 18
  - 4.4.3 Circuit 3 . . . 21

5 Node Behavior **22**
- 5.1 Node Functions . . . 22
  - 5.1.1 *Accepts* . . . 23
  - 5.1.2 *Offers* . . . 24
  - 5.1.3 τ . . . 25

6 Channel Behavior **26**
- 6.1 Asynchronous Channels . . . 27
  - 6.1.1 `FIFO` . . . 27
  - 6.1.2 $\texttt{FIFO}_n$ . . . 27
- 6.2 Synchronous Channels . . . 28
  - 6.2.1 `Sync` . . . 28
  - 6.2.2 `SyncDrain` . . . 28

6.2.3 `SyncSpout` . . . 29
6.2.4 `LossySync` . . . 29
6.2.5 `Filter` . . . 30

7 Examples **31**
7.1 Loop categorization . . . 31
7.2 Loops . . . 32
7.2.1 Source-source-source loops (loop 1) . . . 32
7.2.2 Source-source-sink loops (loop 2) . . . 32
7.2.3 Source-sink-source loops (loop 3.1 and loop 3.2) . . . 34
7.2.4 Source-sink-sink loops (loop 4) . . . 35
7.2.5 Sink-source-source loops (loop 5) . . . 36
7.2.6 Sink-source-sink loops (loop 6) . . . 36
7.2.7 Sink-sink-source loops (loop 7) . . . 37
7.2.8 Sink-sink-sink loops (loop 8) . . . 37

8 Conclusions **37**

A Examples of traces of loops **39**
A.1 Trace of a source-source-source loop (loop 1) . . . 39
A.2 Trace of a source-sink-source loop (loop 3.1) . . . 42
A.3 Trace of a source-sink-source loop (loop 3.2) . . . 44
A.4 Trace of a source-sink-sink loop (loop 4) . . . 47
A.5 Trace of a sink-source-source loop (loop 5) . . . 48
A.6 Trace of a sink-source-sink loop (loop 6) . . . 50

1 Introduction

This document presents a computational model for Reo specified using a functional notation. This computational model can facilitate a fully distributed realization of the Reo coordination language.

1.1 Background

The Reo coordination language presents a paradigm for coordinated composition of software components. We refer to a software component composition specified in Reo as a *circuit*. Reo can serve not only as a specification language for component compositions, but also as their implementation language. For this we require an operational semantics (a formal computational model) that defines how one can compute a Reo circuit.

Modern computing environments involve powerful computers connected in a network by high-speed communication infrastructure. We observe, among others, the following interesting characteristics of the modern computing environment: high physical and geographical distribution of the individual computing devices; high availability of the network services; and an increasing growth in computing capabilities with the advances of technology. The work described in this document aims at producing an operational semantics for Reo, that facilitates implementation of Reo circuits in a distributed computing environment.

1.2 Rationale

Arbab discussed the wide variety of architectural models [1] that Reo can support as a specification language. Below we provide several important issues that an operational semantics for Reo should address in order to preserve the expressiveness of Reo for Reo as a specification language, during the computation of Reo circuits in a distributed environment.

1.2.1 Global View

In a geographically distributed environment, different parts of an instance of a Reo circuit may have to operate on remote hosts. An operational semantics that relies on a global view of the state of a Reo circuit imposes a centralized computational model. Centralized computational models promote implementations with contention and single-point-of-failure vulnerability. These often inhibit the parallelism inherent in physically distributed systems, by injecting additional delays necessary to maintain a consistent global view.

We acknowledge the high architectural and performance cost of centralized computational models. We concentrate on a computational model in which an authority computing an instance of a Reo circuit can obtain only a limited knowledge about the circuit. When such authority needs to perform some activity for which it does not have sufficient information, it delegates this activity to some other authority that has the necessary information. This principle of partial information underlies the Reo operational semantics that we present in this paper.

1.2.2 Communication Infrastructure

In Reo, channels encapsulate all communication-related activities. To compute an instance of a Reo circuit according to the principle outlined above, requires significant exchange of intermediate control information over the communication infrastructure. Since Reo offers only channels as communication medium, its operational semantics should not “cheat” by using an additional communication infrastructure

for signaling than the channels themselves. Using an additional communication infrastructure may violate the relations among the elements in a Reo circuit. Such violation may produce a difference between what the specification claims to do and what the operational semantics actually does in a distributed environment.

1.2.3 Look-up and Commit vs Roll-back

Reo has two types of channels: synchronous and asynchronous. Computing a Reo circuit consisting of synchronous channels requires an *atomic* evaluation of all conditions under which the circuit transports data through its synchronous channels. One can achieve atomicity following two approaches: (1) by performing the necessary transportation until either it succeeds or fails, in which case all intermediate transportation must roll back; or (2) by looking ahead at the result of every transportation, commit to these results, and only in the case of complete success, actually perform all transportation.

In order to compute an instance of a Reo circuit in the first approach, we need to restrict all channels used in the circuit to ones that support reversible transports. Such restriction would disallow the use of channels whose implementation has side effects that forbid reversing of transportation (or make it very difficult and costly). For example, transportation of volatile materials may prove too costly to perform in reverse if required in a roll back. Therefore, employing roll backs in the Reo operational semantics limits the expressive power of Reo as an implementation language, as compared to its power as a specification language for real-life coordination.

To overcome this limitation we rely on the second approach to computing Reo circuit instances. For this we require from a channel to provide a look ahead of the result of a transportation, before the actual transportation occurs. In this document we successfully construct an operational semantics that uses the look ahead capabilities of basic channels to provide look ahead capabilities for Reo circuit instances.

1.2.4 Concurrency

If we consider the principle of partial information together with the inherent concurrency in a distributed environment, it becomes clear that many authorities may start competing over the computation of an instance of the same Reo circuit in a distributed environment. This may lead to various race conditions, livelocks, deadlocks, etc. The Reo operational semantics should properly avoid, or detect and resolve race conditions during the computation of a Reo circuit.

1.2.5 Partial Failure

Having partial information in a large distributed environment prevents us from predicting failure in the system. Therefore, a part of the platform on which a Reo circuit instance runs may fail, while the information about this failure may not become available until someone actually tries to communicate with that part. The Reo operational semantics must properly deal with partial failure of the run-time platform of an instance of a Reo circuit.

1.3 Reo concepts

In this section we briefly introduce the basic Reo concepts. Arbab gives an extensive introduction to all Reo concepts [1]. Figure 1 shows the basic elements of a Reo circuit. A *channel* has precisely two *channel ends*. Reo introduces two types of channel ends: *sink* and *source*. A sink dispenses data out of its channel. A source accepts data into its channel. Channel ends coincide on *nodes*. Visually, one can

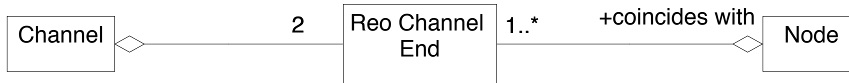


Figure 1: Reo elements and their basic relations

view a Reo circuit as a connected graph of nodes and edges, where one or more channel ends may coincide on every node, every channel end coincides on exactly one node, and an edge exists between two nodes if and only if there exists a channel whose channel ends coincide on those nodes.

Reo has three types of nodes: *source*, *sink*, and *mixed*. A source node contains only sources. A sink node contains only sinks. A mixed node contains both sources and sinks. Nodes typically serve as pumping stations for their channel ends, where from sinks (if any) the node takes an available data item, and replicates it to all of its sources (if any). Arbab gives a detailed description of node behavior [1].

1.4 Basics of the Operational Semantics

Reo defines a number of operations that application components can use to manipulate and interact with an instance of a Reo circuit. In the Reo operational semantics we consider only I/O operations: *take* and *write*. We disregard the rest of the operations, since they do not concern the traffic of data through an instance of a Reo circuit. A component can invoke an I/O operation on a node by using a channel end that coincides on that node. After an invocation, the operation becomes *pending* on the node. At this point, nodes, channels and their composition in a circuit, determine the working of a Reo circuit. The functional specification that we present in this paper defines the behavior of nodes and the behavior of channels.

The Reo operational semantics utilizes the principle of partial information in a distributed system in the following way. Nodes and Channels may act as active entities that execute on more than one host. Nodes constantly try to pump data from and to the channels whose ends coincide on them. Nodes also try to satisfy requests for Reo I/O operations from application components. Channels constantly try to transport data (channel behavior varies from one type of channel to another). We design the operational semantics in the form of queries posed (functions) to a Reo circuit element (node or channel) to determine what it can do. The Reo operational semantics presented in this paper answers all necessary queries to provide the look ahead capabilities necessary to ensure that a Reo circuit can successfully transport data items maintaining a result consistent with the algebraic semantics of Reo [2, 5].

One may ask a node what kind of *traffic* of data items it can perform at some moment in time. In the operational semantics, we represent this question using the τ function. In reply, the τ function returns the candidate data items. A mixed node, for example, pumps a data item out of one of its coincident sinks and into all its coincident sources. The traffic query may appear too complex, and therefore we decompose it further into two separate queries: (1) what data items can a node offer as available; and (2) what data items can a node accept? We represent the first query as the *Offers* function, and the second query as the *Accepts* function.

Since in Reo nodes may interact only through their coincident channel ends with their respective channels, the node queries naturally propagate through a Reo circuit in the form of some channel queries and a relation among the answers to those queries. We define the precise relations in detail later in the text. We again

consider the mixed node as an example. To evaluate its *Offers* function (denoted with a capital first letter), a mixed node needs to ask the channels of all of its coincident sinks whether they have anything to offer. We represent this channel query using an *offers* function (denoted with a small first letter). Similarly, to evaluate its *Accepts*, a mixed node needs to ask the channels of all of its coincident sources whether they can accept a certain data item. We represent this channel question using an *accepts* function. How *offers* and *accepts* propagate through a Reo circuit becomes a prerogative of the channels only. Since Reo does not depend on fixed definitions of channels, the Reo computational model remains open for consistent use with new channel behavior.

2 Functional Notation

In this section we present the bare minimum of the functional notation that we need to define the specification of the Reo operational semantics. For readers unfamiliar with a functional notation we refer to [4]. Furthermore, we introduce $d \ni p$ to designate that d matches with the pattern p.

2.1 Normal Function Declarations

The specification consists of a set of function declarations. The first line of each function declaration contains the function signature. A function signature consists of the name of the function, the type of its parameters and the type of its returned value (in this order).

$FunctionName :: Type_1 \rightarrow Type_2 \rightarrow ... \rightarrow Type_L \rightarrow ReturnType$
$FunctionName(a,\ b)\ =$
  body clause
 where
  local function definition$_1$
   ...
  local function definition$_n$

The very last type name always represents the function return type, and the number of parameters equals to one less than the number of types in the signature line (or just the number of $\rightarrow$'s). The specification of the function body, always follows after the function signature followed by an optional *where* clause. The function body consists of a head that defines the function argument variables, followed by the = sign, and a body clause that defines how the function utilizes the argument variables. The body clause may contain an **if then else** block to provide conditional results.

All the lines in a where clause represent definitions of local functions. A local function definition does not have a function signature. We define local functions with =, on the left of which we indicate the function name and argument names, and on the right of which we define a single line body. We use local function definitions in order to keep the function body more understandable. After a function declaration, we provide a detailed informal explanation of the function.

A type name between square brackets [ ] denotes a list of elements with type identified by the type name. We denote an empty list with []. The expression [h:t] states that a list has a head h (the first element of the list), and tail (the rest of the list) t. One can concatenate lists together using the operator ++. The *head* applied to a list returns the first element of the list. The *tail* applied to a list returns a list without the first element of the original list. The *length* applied to a list returns the current number of elements in a list.

A list of type names between brackets, e.g., (A, B, C), indicates a tuple with elements of the corresponding type names. The fst applied on a pair returns the first element of the pair. The *snd* applied on a pair returns the second element of the pair.

2.2 Function Declarations Using Pattern Matching

Alternatively, one can define functions using *pattern matching.* In pattern matching, the programmer supplies several alternative function bodies, with different head parts (tail parts may also differ). To select a proper body for processing a function invocation, we start with matching the head of the function body with the current function and its parameters. The matching starts in the order in which the programmer defines the alternative bodies. The first matching head determines the body selected for evaluation.

2.3 List Comprehension a.k.a. ZF-notation

A list comprehension has the form $[\ e \ \mid \ q_1,\ \cdots,\ q_n\ ]$, $n >= 1$, where a q_i qualifier constitutes one of the following:

- *generators* of the form $p \leftarrow g$, where p represents an element of type t and g represents an expression producing a list of type t, where p ranges over all the elements of the list. A list comprehension makes sense if it has at least one generator;
- *guards*, which represent arbitrary expressions of boolean type that filter the values that come from generators;
- *local bindings* that provide new definitions for use in the generated expression e or subsequent guards and generators.

3 Data Types

In this section we describe the Reo data types used in the specification. We explain to which Reo element a data type relates. Where applicable, we also provide a list of auxiliary functions per data type.

3.1 Data Type Definition

Table 1 introduces the data types we use in this document.

Data Type	Description
Node	A type that represents Reo nodes.
Channel	A type that represents Reo channels.
CE	A type that represents Reo channel ends.
Data	A type that represents the data items flowing through Reo channels.
Pattern	A type that represents the patterns passed to the *take* and *read* Reo I/O operations, to filter out required data items.
PendingOp	A type that represents Reo I/O operations pending on a node.
Bool	A type representing boolean values.

Table 1: Table of all basic data types

3.2 Auxiliary Functions per Data Type

We define two types of auxiliary functions: *selectors* and *predicates*. A selector takes as an argument an item of a basic data types, and "selects" some result based on the relation between the Reo element represented by the argument, with some other Reo element. We defined the possible relations in Section 1.3. A predicate checks for some condition on its argument. Note that $Data$ and $Pattern$ do not have selectors or predicates. We allow combining two patterns using the $\wedge$ operation.

3.2.1 $Node$

Table 2 introduces node functions.

Function Name	Type	Description
$pendingOp_{sel}(n)$	selector	Returns a list of all I/O operations pending on n.
$srcCE_{sel}(n)$	selector	Returns a list of all source channel ends coincident with n.
$sinkCE_{sel}(n)$	selector	Returns a list of all sink channel ends coincident with n.
$is_{sinknode}(n)$	predicate	Checks whether n represents a sink node.
$is_{sourcenode}(n)$	predicate	Checks whether n represents a source node.

Table 2: Node functions

3.2.2 CE

Table 3 introduces channel end functions.

Function Name	Type	Description
$channel_{sel}(ce)$	selector	Returns the channel to which the channel end ce belongs.
$node_{sel}(ce)$	selector	Returns the node on which the channel end ce coincides.

Table 3: Channel end functions

3.2.3 $Channel$

Table 4 introduces channel functions.

3.2.4 $PendingOp$

Table 5 introduces functions on pending I/O operations.

4 Simplified Computational Model

In this section we give the basic formulas underlaying Reo's operational semantics. We give them first in a simplified form and work out some examples with them. Thereby some shortcomings in the formulas come to light. The final form of the formulas are given later in §5 and §6. First we start with some auxiliary functions we need for the simplified computational model as well as for the more complex final version.

Function Name	Type	Description
$accepts_{sel}(ch)$	selector	Returns the *accepts* function of the channel ch.
$offers_{sel}(ch)$	selector	Returns the $offers$ function of the channel ch.
$channelend_{sel}(ch)$	selector	Returns a list of the channel ends that belong to channel ch.
$buffer_{sel}(ch)$	selector	Returns a list representing the buffer of a channel ch.
$capacity_{sel}(ch)$	selector	Returns the capacity of the buffer of a channel ch. Returns 0 for a channel with no buffer, and -1 for channels with unlimited capacity.
$pattern_{sel}(ch)$	selector	Returns the pattern of a channel (e.g., `Filter`).

Table 4: Channel functions

Function Name	Type	Description
$pattern_{sel}(tk_op)$	selector	Returns the pattern of a pending take operation tk_op.
$datavalue_{sel}(wr_op)$	selector	Returns the data item of a pending write operation wr_op.
$\pi_{write}(op)$	predicate	Checks whether op represents a pending write.
$\pi_{take}(op)$	predicate	Checks whether op represents a pending take.

Table 5: Pending I/O operation functions

4.1 Auxiliary Node Functions

We define two functions that allow us to work with the I/O operations pending on a node: P and W.

$$P \;::\; Node \rightarrow [Pattern]$$
$$P(n) \;=\; [\, pattern_{sel}(op) \;\mid\; op \leftarrow pendingOp_{sel}(n), \pi_{take}(op) \,]$$

The P function takes a node n as argument and returns the list of patterns of the *takes* pending on n.

$$W \;::\; (Node,\; Pattern) \rightarrow [Data]$$
$$W(n,p) \;=\; [\; datavalue_{sel}(op) \;\mid\; op \leftarrow pendingOp_{sel}(n),\; \pi_{write}(op) \wedge datavalue_{sel}(op) \ni p]$$

The W function takes an ordered pair consisting of a node n and a pattern p, and returns a list of those data items of the *writes* pending on n, that match the pattern p.

4.2 Simplified Node Behavior

In this section we define the simplified versions of the three node functions that define the behavior of nodes in Reo circuits: *Accepts*, *Offers*, and τ.

4.2.1 Simplified *Accepts*

The *Accepts* function takes an ordered 2-tuple (a pair) consisting of a node n and a data item d, and returns a boolean value indicating whether or not the node n accepts the data item d.

$$\begin{aligned}
&Accepts \;::\; (Node, Data) \rightarrow\; Bool\\
&Accepts\;(n,d) \quad = \textbf{if } is_{sinknode}(n) \textbf{ then}\\
&\qquad\qquad P(n) \;\neq\; [] \;\wedge\; \bigwedge [d \ni p \mid\; p \leftarrow P(n)]\\
&\qquad\quad \textbf{else}\\
&\qquad\qquad \bigwedge [accepts(x,d) \;\mid\; x \leftarrow scrCE_{sel}(n)]\\
&\quad \textbf{where}\\
&\qquad accepts = accepts_{sel}(ch)\\
&\qquad ch \qquad = channel_{sel}(x)
\end{aligned}$$

The *Accepts* function states that if n is a sink node the result is a boolean "AND" (the $\wedge$) of two boolean expressions. The first boolean expression ($P(n) \neq []$) checks whether there are pending take operations on the node. We examine this in a indirect way by checking whether the list of patterns of pending takes on n is empty or not (when there is no pending take there is also no pattern). In the second boolean expression ($\bigwedge[d \ni p \mid p \leftarrow P(n)]$) we apply the boolean "AND" (the $\bigwedge$) on a list of booleans (note the []) formed by the return values of checking whether or not d matched with the pattern of each of the pending takes ($p \leftarrow P(n)$). If n is not a sink node we apply the boolean "AND" on the list (note the []) of booleans formed by the return values of the *accepts* of each of the source channel ends. Note that the *accepts* function used while visiting each source channel end is channel specific. In the where-clause we pick out with the selector $accepts_{sel}(ch)$ the correct *accepts* function. The *ch* used thereby is determined by applying the selector $channel_{sel}$ on the channel end we currently deal with.

Briefly summed up: *Accepts* states that if n is a sink node, it accepts a data item d only if d matches with the read-patterns p of all take operations currently pending on n. Otherwise, n accepts d only if all the source channel ends in n accept d.

4.2.2 Simplified *Offers*

The *Offers* function takes an ordered 2-tuple consisting of a node n and a pattern p and returns a list of data items.

$$
\begin{array}{ll}
\mathit{Offers} :: (\mathit{Node},\ \mathit{Pattern}) \rightarrow\ [\mathit{Data}] & \\
\mathit{Offers}\ (n,p) & = \textbf{if}\ \mathit{is}_{sourcenode}(n)\ \textbf{then} \\
& \quad [\ d\ \mid\ d \leftarrow W(n,p)\ ] \\
& \ \textbf{else} \\
& \quad \bigcup\ [\ \mathit{offers}(x,p)\ \mid\ x \leftarrow \mathit{snkCE}_{sel}(n)\ ] \\
\quad \textbf{where} & \\
\qquad \mathit{offers} = \mathit{offers}_{sel}(ch) & \\
\qquad ch \quad\ = \mathit{channel}_{sel}(x) &
\end{array}
$$

The *Offers* function states that if n is a source node it offers the data values proposed by the write operations pending on that node that match the specified pattern p. If n is not a source node, it is either a mixed node or a sink node. A mixed node cannot be involved in any write operations in Reo. Therefore *Offers* returns in that case the union ($\bigcup$) of all values offered by all the sink nodes of n. In the where-clause the correct channel specific *offers* function is selected.

4.2.3 Simplified τ

The τ function takes an ordered 2-tuple consisting of a node n and a pattern p and returns a list of data items.

$$
\begin{array}{ll}
\tau\ ::\ (\mathit{Node},\ \mathit{Pattern}) \rightarrow\ [\mathit{Data}] & \\
\tau\ (n,p) & = [\ d\ \mid\ d \leftarrow \mathit{Offers}(n,p),\ \mathit{Accepts}(n,d)\ ==\ \mathit{True}\ ]
\end{array}
$$

The τ function defines the data items that are qualified to be chosen for transfer. These are the data items offered by the node n that match with the pattern p and which are also acceptable by the node.

4.3 Some Simplified Synchronous Channels

For every channel, the Reo operational semantics defines an *accepts* for its *source* ends, and an *offers* for its *sink* ends. The *accepts* function takes an ordered 2-tuple consisting of a channel end x and a data item d and returns a boolean that indicates the acceptability of the data item d into the channel. The *offers* function takes an ordered 2-tuple consisting of a channel end y and a pattern p and returns a list of data items offered by the channel that matches p. For completeness, below we define *accepts* for *sink* channel ends, and *offers* for *source* channel ends. Note that the Reo operational semantics properly distinguishes between sources and sinks coincident on a node, and performs the *accepts* on sources and *offers* on sinks only.

$$
\begin{array}{l}
\mathit{offers}\ ::\ (\mathit{CE},\ \mathit{Pattern}) \rightarrow [\mathit{Data}] \\
\mathit{offers}\ (x,d,S,E)\ =\ []
\end{array}
$$

A *source* never offers any data items, hence the *offers* returns [].

$$
\begin{array}{l}
\mathit{accepts}\ ::\ (\mathit{CE},\ \mathit{Data}) \rightarrow\ \mathit{Bool} \\
\mathit{accepts}\ (y,d)\ =\ \mathit{False}
\end{array}
$$

A *sink* never accepts data items, hence the *accepts* returns $False$.

Below we specify the behavior of only the three synchronous channels we need for the example in §4.4. Synchronous channels all share the characteristic that they propagate the queries made at one of their ends to the node at their other end, effectively synchronizing the behavior of the nodes at both their ends.

4.3.1 Simplified `Sync`

A `Sync` channel has a source and a sink and thus it can both accept and offer data. A `Sync` channel accepts a data item d through its source x only if the node n of its sink y accepts the data item d from this channel.

$$
\begin{aligned}
& \mathit{accepts} :: (\mathit{CE},\ \mathit{Data}) \rightarrow \mathit{Bool} \\
& \mathit{accepts}\ (x, d) = \mathit{Accepts}(n, d) \\
& \quad \textbf{where} \\
& \qquad n \ = \mathit{node}_{sel}(y) \\
& \qquad [y] = \mathit{channelend}_{sel}(c) \backslash [x] \\
& \qquad c \ = \mathit{channel}_{sel}(x)
\end{aligned}
$$

Note how node n is determined in the where-clause by $\mathit{node}_{sel}(y)$. The used parameter y comes from the singleton list $[y]$. List $[y]$ is determined as the subtraction of two lists. The first list comes from $\mathit{channelend}_{sel}(c)$ and yields the two channel ends of channel c. The c used thereby comes from $\mathit{channel}_{sel}(x)$ where x is the source channel end received as parameter on the left side. The second list contains only the source channel end x. Subtraction of these lists yields a singleton list $[y]$

A `Sync` channel offers a data item matching a pattern p through its sink y only if the node n of its source x can traffic a data item matching the pattern p into this channel.

$$
\begin{aligned}
& \mathit{offers} :: (\mathit{CE},\ \mathit{Pattern}) \rightarrow [\mathit{Data}] \\
& \mathit{offers}\ (y, p) = \tau\ (n, p) \\
& \quad \textbf{where} \\
& \qquad n \ = \mathit{node}_{sel}(x) \\
& \qquad [x] = \mathit{channelend}_{sel}(c) \backslash [y] \\
& \qquad c \ = \mathit{channel}_{sel}(y)
\end{aligned}
$$

A `Sync` is graphically represented as $\longrightarrow$. The arrow head identifies the sink channel end.

4.3.2 Simplified `SyncDrain`

A `SyncDrain` channel has two sources and thus it can only accept data. A `SyncDrain` channel accepts a data item d through its source x_1 only if the node n of its other source x_2 can traffic an arbitrary data item (pattern "*") into the channel as well.

$$
\begin{aligned}
& \mathit{accepts} :: (\mathit{CE},\ \mathit{Data}) \rightarrow \mathit{Bool} \\
& \mathit{accepts}\ (x_1, d) = \tau\ (n, *) \neq [] \\
& \quad \textbf{where} \\
& \qquad n \ \ = \mathit{node}_{sel}(x_2) \\
& \qquad [x_2] = \mathit{channelend}_{sel}(c) \backslash [x_1] \\
& \qquad c \ \ = \mathit{channel}_{sel}(x_1)
\end{aligned}
$$

A `SyncDrain` is graphically represented as $\succ\!\!\!\longrightarrow\!\!\!\prec$. The arrow heads identify the two sources.

4.3.3 Simplified `SyncSpout`

A `SyncSpout` channel has two sinks and thus it can only offer data. A `SyncSpout` channel offers a non-deterministically-generated data item d from its sink y_1 only if d matches the pattern p and the node n of its other sink y_2 accepts another non-deterministically-generated data item d' from this channel.

$$
\mathit{offers} :: (\mathit{CE},\ \mathit{Pattern},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow [(\mathit{Data}, [\mathit{CE}])]
$$

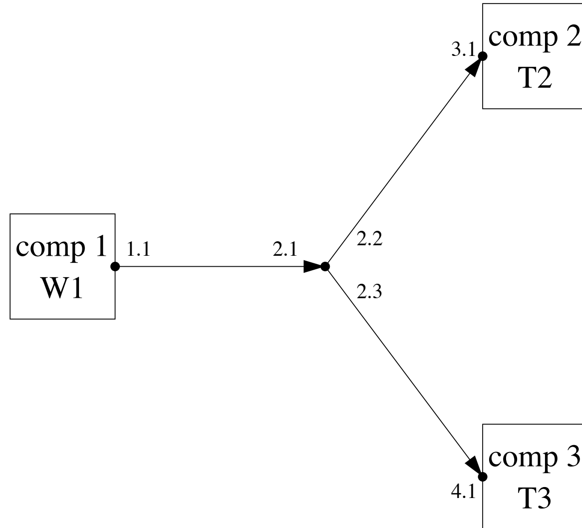


Figure 2: A simple circuit

$$offers\,(y_1, p) \;=\; [\; d \mid\; d \;\leftarrow\; random() \;\wedge\; d \ni p,\; d' \;\leftarrow\; random(), \; Accepts(n,\; d') \;==\; True]$$

where

$$\begin{aligned} n \quad &= node_{sel}(y_2) \\ [y_2] &= channelend_{sel}(c) \backslash [y_1] \\ c \quad &= channel_{sel}(y_1) \end{aligned}$$

Note that the *random*() function serves as a non-deterministic generator of data for the `SyncSpout`.

A `SyncSpout` is graphically represented as $\prec\!\!\longleftarrow\!\!\longrightarrow\!\!\succ$. The arrow heads both represent sink channel ends.

4.4 Some Examples

4.4.1 Circuit 1

In Figure 2 we show our first Reo circuit. This circuit consists of three component instances, three `Sync` channels and four nodes. We represent component instances as rectangles and nodes as small circles. Channel ends are represented identified using two integers separated by a dot. The first integer identifies the node where the channel end has its domicile and the second integer indentifies the channel end locally within that node. Together they form a unique identifier for a channel end. In Figure 2 we see that the three channels have the source sink pairs 1.1 and 2.1, 2.2 and 3.1, and 2.3 and 4.1, respectively. Node 1 is a source node and houses only the source channel end 1.1. Node 2 is a mixed node and houses sink channel end 2.1 and the source channel ends 2.2 and 2.3. Node 3 is a sink node and houses only the sink channel end 3.1. Node 4 is also a sink node and houses only the sink channel end 4.1. Further, we see that nodes 1, 3, and 4 are connected to, respectively, component 1, 2 and 3.

A component instance can invoke an I/O operation on a node by using a channel end that coincides on that node. After an invocation, the operation becomes *pending* on the node. Triggered by this invocation the node, which is an independent process, becomes active and tries to find out whether there is something to transport to channels connected to this node. Therefore the node computes the τ function (see

§4.2.3). When the result of the τ is empty there is nothing to transport; otherwise we can transport data. This main question concerning τ, results in a chain of sub-questions that propagate through the Reo circuit. Note that our only concern is whether we *can* transport data or not. When we know this, and the answer is yes, we can *do* it, but how we do it is not discussed in this report.

We consider the following command scenario for the components (see Figure 2).

- First, component 1 writes an integer (let us say 9) to channel end 1.1. (denoted by the $W1$ in the component's rectangle)
- After this, component 2 performs a take operation via channel end 3.1 (denoted by the $T2$ in the component's rectangle). The 2 in $T2$ refers to the later time of the command invocation. The pattern used in the take operation is the "*", so the take operation is not selective about for what it takes.
- After this, component 3 performs a take operation via channel end 4.1 (denoted by the $T3$ in the component's rectangle). Again, this take operation uses the "*" pattern.

We give for the $W1$, $T2$ and $T3$ commands the τ computations and show how its subsequent sub-questions propagate through the circuit. Intuitively, we know that the $W1$ invocation leads to an empty τ because there is nobody who can take the data. Also, the $T2$ invocation leads to an empty τ because component 3 does not take data which makes it also impossible for component 2 to get it. Only after component 3 performs the $T3$ command, τ will be not empty and the data can flow through the network, replicating the written data into channel end 2.2 and 2.3.

We now follow the τ computations for the different commands more precisely. After the $W1$ operation, we have a pending operation on node 1 and as a result of the operation, node 1 computes the traffic $\tau(1, *)$. According to §4.2.3 we first evaluate the data items the node offers and after this we test them for their acceptability. Because node 1 is a source node it offers the data item from writes pending on that node (*Offers(1)* evaluates to $[d \mid d \leftarrow W(n,p)]$; see §4.2.2). This yields the singleton list [9]. We now go back to continue evaluating the traffic of node 1 and check whether node 1 can accept the integer 9. Node 1 can accept this integer if the `Sync` channel of 1.1 can accept it. Such a channel can accept data through its source only if the node of its sink can accept the same data (see §4.3.1). So we now must evaluate *Accepts(2, 9)*. Because node 2 is not a sink node, we evaluate *accepts(2.2, 9)* and/or *accepts(2.3, 9)* (see $\bigwedge[accepts(x,d) \mid x \leftarrow scrCE_{sel}(n)]$ in §4.2.1). Because the `Sync` channel of 2.2 can accept data when node 3 can accept data (see §4.3.1) we evaluate *Accepts(3,9)*. Node 3, however, is a sink node so it can accept data only when there are pending take operations on that node (see §4.2.1). Because that is not the case, *Accepts(4,9)* returns false and with that *Accepts(2,9)* too, and thus also *Accepts(1,9)* returns false. This latter implies that $\tau(1, *)$ is empty because the offered data in the node (integer 9) is not acceptable in that node. Because $\tau(1, *)$ is empty no data can be transported.

Note that there is no reason to evaluate *accepts(2.3, 9)* because the *accepts(2.2, 9)* already failed. This lazy evaluation (i.e., we only evaluate what is needed) is always our basic strategy and fits well in the functional style of our query functions.

Below, we briefly sum up the computational path through the circuits for the $W1$ operation. We use thereby the following symbols:

- $\rightarrow$ is used to denote that the evaluation of the query function to the right of this arrow is the next step in the traffic computation.
- $\rightsquigarrow$ means the same as $\rightarrow$ except that there is a choice of which function we evaluate on the right. In the $W1$ computation, we have this situation

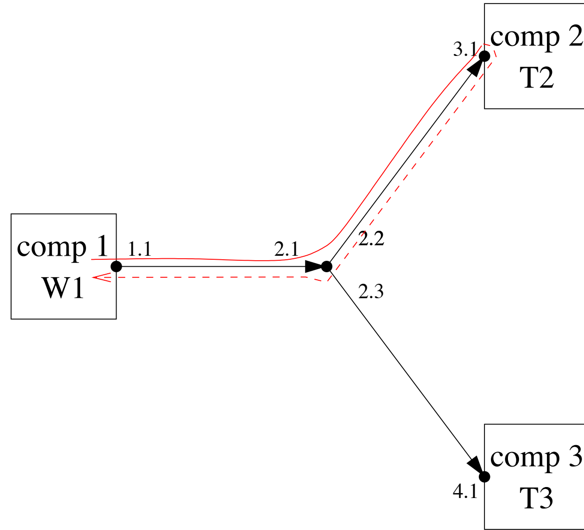


Figure 3: Direction of the computation for the write $W1$ of component 1.

when we evaluate *Accepts(2,9)*. There, we have the choice to proceed with the evaluation of *accepts(2.3,9)* or *accepts(2.2,9)*. This choice influences the computational path taken through the circuit, but of course, not the final result of the traffic computation.

- $\circlearrowleft$ behind a function means that this query function returns back to its caller. The caller can then proceed with its job.

With this notation, we can give the computational path for the $W1$ operation briefly and precisely.[1] It is:

$\tau(1,*)$ $\rightarrow$ *Offers(1,*)* $=$ [9] $\circlearrowleft$ (because it is a source node and there is a pending write) *Accepts(1,9)* $\rightarrow$ *accepts(1.1,9)* $\rightarrow$ *Accepts(2,9)* $\leadsto$ *accepts(2.2,9)* $\rightarrow$ *Accepts(3,9)* $=$ *False* $\circlearrowleft$ (because it is a sink node and there are no pending takes) *accepts(2.2,9)* $=$ *False* $\circlearrowleft$ *Accepts(2,9)* $=$ *False* $\circlearrowleft$ *accepts(1.1,9)* $=$ *False* $\circlearrowleft$ *Accepts(1,9)* $=$ *False* $\circlearrowleft$ $\tau(1,*)$ $=$ [] $\circlearrowleft$. Because τ is empty no data transport is possible.

Note the $\leadsto$ after *Accepts(2,9)*. According to the definition of the *Accepts* function (see §4.2.1) we have a choice to continue with *accepts(2.2,9)* (as we did) or *accepts(2.3,9)*.

In Figure 3 we show the computational path of the write command of component instance 1.

We continue with the computational path for the $T2$ operation. It is as follows:

$\tau(3,*)$ $\rightarrow$ *Offers(3,*)* $\rightarrow$ *offers(3.1,*)* $\rightarrow$ $\tau(2,*)$ $\rightarrow$ *Offers(2,*)* $\rightarrow$ *offers(2.1,*)* $\rightarrow$ $\tau(1,*)$ $\rightarrow$ *Offers(1,*)* $=$ [9] $\circlearrowleft$ (because it is a sink node and there is a pending write) *Accepts(1,9)* $\rightarrow$ *accepts(1.1,9)*....

When we stop at this point in the computation, we may wonder why we have to evaluate *accepts(1.1,9)*. This evaluation is not necessary at all because the acceptance of the integer 9 for the `Sync` channel of 1.1 is a given, because this is the channel that initiated the offers request (see the *offers(2.1,*)* in the computational path). *Sync* channel of 1.1 issued an *offers* query to see if its node has anything to

[1] In order not to become dizzy during a traffic computation (especialy when there are loops in circuits), it is convient to write down with pen and paper the tree structure that evolves out of the evaluation of a τ (thus on one side the *Offers* part and on the other side the *Accepts* part). The resulting computational path is then easier to follow as a path through this tree.

offer it; the node, then, need not ask this channel if it will accept what it has to offer.

Note that this also happens when we try to evaluate $\tau(2,*)$. According to the τ formule (see §4.2.3) the evaluation of this function involves the computation of *Offers(2,*)*. In our case this yields the integer 9, so the next step in the computation is *Accepts(2,9)*. Therefore we must (according to §4.2.3) evaluate *accepts(2.2,9)* and/or *Accepts(2.3,9)*. But evaluating *accepts(2.2,9)* is not necessary because it is already clear that the `Sync` channel of 2.2 will accept what it asked the node had to offer. So in general we can say that for *accepts* functions of channels that requires calling τ (e.g., see §4.3.1) we should prevent the *Accepts* in τ computation from entering again the channel (with an *accepts* call) that has already called τ. In principle we can sharpen the previous statement because it is also valid for *offers* functions of channels that require calling τ (e.g., see §4.3.2). This leads to the next requirement with respect to the traffic computation.

> I) *For offers and accepts functions of channels that require calling τ we should prevent the Accepts in the τ computation from entering again the channel that has already called.*

We can easily solve this problem by given τ and *Accepts* functions an additional parameter (a third one) that represents the channel end (it is always a source) to be excluded in the *Accepts* in τ. With this added third parameter we give the computational path for the $T2$ command once again. When there is no channel end to exclude for the *Accepts* in τ we use "$-$" as the third parameter. Note that the τ function itself does not do anything with the third parameter. It just passes it to its *Accepts*.

$\tau(3,*,-) \rightarrow$ *Offers(3,*)* $\rightarrow$ *offers(3.1,*)* $\rightarrow \tau(2,*,2.2) \rightarrow$ *Offers(2,*)* $\rightarrow$ *offers(2.1,*)* $\rightarrow \tau(1,*,1.1) \rightarrow$ *Offers(1,*)=[9]* (because it is a sink node and there is a pending write) ⭯ *Accepts(1,9,1.1) = True* ⭯ (because there are no channels with source channel ends in node 1 for which we must evaluate their *accepts* function. The only channel with a source in node 1 is the `Sync` channel of 1.1 but this one is excluded in the *Accepts* in $\tau(1,*,1.1)$ because τ has been called from this channel end and therefore its acceptance is obviously True; see requirement I) $\tau(1,*,1.1)$ = [9] ⭯ *Offers(2,*)* = [9] ⭯ *Accepts(2,9,2.2)* (take over the excluded channel end as specified in the caller of this function, i.e., $\tau(2,*,2.2)$) $\rightarrow$ *accepts(2.3,9)* $\rightarrow$ *Accepts(4,9,-) = False* ⭯ (because it is a sink node and there are no pending takes) *accepts(2.3,9) = False* ⭯ *Accepts(2,9,2.2) = False* ⭯ $\tau(2,*,2.2)$ = [] ⭯ *offers(3.1,*)* = [] ⭯ *Offers(3,*)* = [] ⭯ $\tau(3,*,-)$ = []. Because τ is empty no data transport is possible. In Figure 4 we show this computational path.

The computational path for the $T3$ command is given below.

$\tau(4,*,-) \rightarrow$ *Offers(4,*)* $\rightarrow$ *offers(4.1,*)* $\rightarrow \tau(2,*,2.3) \rightarrow$ *Offers(2,*)* $\rightarrow$ *offers(2.1,*)* $\rightarrow \tau(1,*,1.1) \rightarrow$ *Offers(1,*)* = 9 ⭯ (because it is a source node and there is a pending write) *Accepts(1,9,1.1) = True* ⭯ (because according to requirement I, accepts(1.1,9) is skipped $\tau(1,*,1.1)$ = [9] ⭯ *offers(2.1,*)* = [9] ⭯ *Offers(2,*)* = [9] ⭯ *Accepts(2,9,2.3) = True* $\rightarrow$ *accepts(2.2,9)* $\rightarrow$ *Accepts(3,9,-) = True* ⭯ (because it is a sink node and there is a pending take) *accepts(2.2,9) = True* ⭯ *Accepts(2,9,2.3) = True* ⭯ $\tau(2,*,2.3)$ = [9] ⭯ *offers(4.1,*)* = [9] ⭯ *Offers(4,*)* = [9] ⭯ *Accepts(4,9,-) = True* ⭯. (because it is a sink node and there is a pending take) $\tau(4,*,-)$ = [9] ⭯. Because τ is not empty data transport is possible. In Figure 5 we show this computational path.

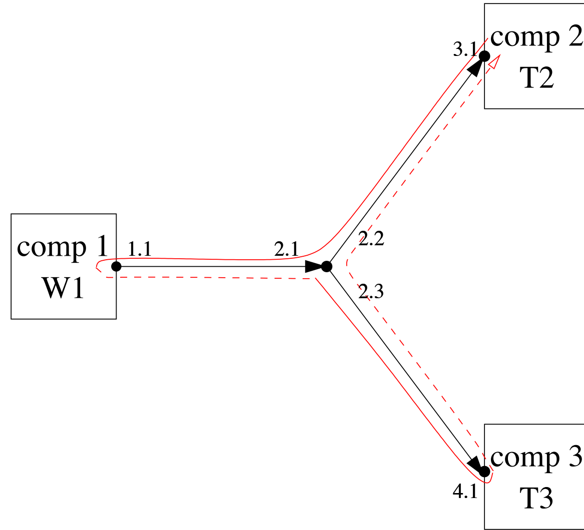


Figure 4: Direction of the computation for the take command of component 2 (T2)

4.4.2 Circuit 2

With the next Reo circuit (see Figure 6) we show that the proposed adaption for the τ and the *Accepts* functions, by introducing a third parameter representing the channel end to be excluded in the *Accepts* of the *tau* computation, is not sufficient.

This circuit is almost the same as the previous one, only now the components that perform the take operations disappear and are replaced by a `SyncDrain`. In this channel data can disappear when something is written on both sides, i.e., 3.2 and 4.2 (see §4.3.2).

Intuitively, we know that the write operation of the component instance can be performed. The data is replicated at node 2 into the `Sync` channels of 2.2 and 2.3. Via node 3 and node 4 the two copies disappear in the `SyncDrain`.

The computational path for the write command of component 1 is given below. $\tau(1, *, -)$ → *Offers(1,*)* = [9] (because it is a source node and there is pending write) ↻ *Accepts(1,9,-)* → *accepts(1.1,9)* → *Accepts(2,9,-)* ⇝ *accepts(2.2,9)* → *Accepts(3,9,-)* → *accepts(3.2,9)* → $\tau(4, *, 4.2)$ → *Offers(4,*)* → *offers(4.1,*)* → $\tau(2, *, 2.3)$ (we are back in node 2!) → .....

The first time we visit node 2 was with the *Accepts(2,9,-)*. We could naively continue with the $\tau(2, *, 2.3)$ and evaluate its *Offers*, but that is not necessary because we know already that the node offers something, namely the integer 9 for which we were testing acceptance to node 2. There is no need to evaluate this again by proceeding with the computation in the direction of node 1. The only problem is that the offered data is not available in the node. This leads to the following two requirements with respect to the traffic computation.

> II) *We must be able to recognize during the traffic computation whether or not we have already visited a node.*

> III) *After we have visited a node through an Accepts call, if by propagation of traffic computation we reenter the same node asking it to compute* τ*, we must be able to remember the data item that was the parameter of the Accepts call.*

We can fulfill requirement II in two ways. Either we store the information about

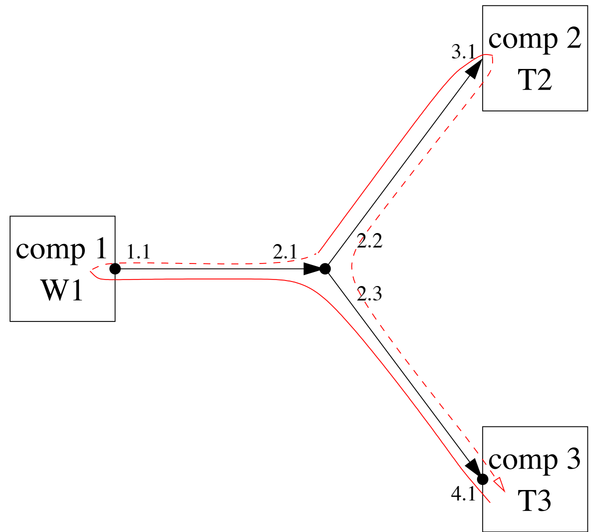


Figure 5: Direction of the computation for the take command of component 3 (T3)

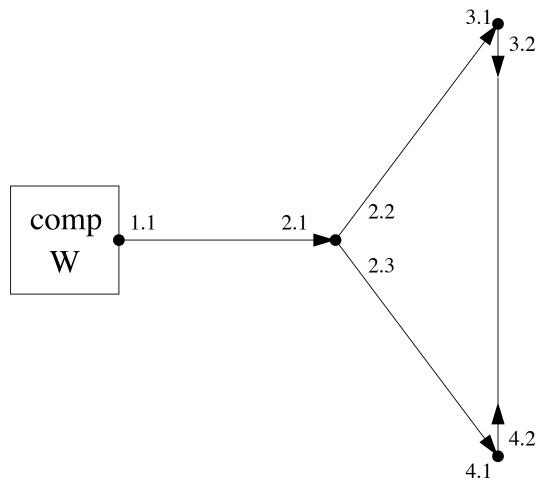


Figure 6: Another simple circuit

whether a node is visited or not in the node, or we take it with us as an additional parameter in the query functions. The same holds for the not yet (fully) checked offered data of a node with respect to requirement III.

A more serious problem is that once we visit a node for the second time with $\tau(2,*,2.3)$ (recall that we first visited it with *Accepts(2,9,2.3)*), we also have to evaluate (after τ's *Offers*) its *Accepts(2,9,2.3)* function. Because this leads to the evaluation of *accepts(2.2,9)* and this function is already used earlier in the computational part (as part of *Accepts(2,9,-)*) we have the situation that *accepts(2.2,9)* is based upon itself. This is the same as saying that *Accepts(2,9,-)* is based upon *Accepts(2,9,2.3)*. We have a loop. The only thing we can do to prevent this situation, is to adapt the τ computation such that once it detects that it has visited a node already with an *Accepts*, it returns the data item, whose acceptance we were testing. This leads to a fourth requirement that our traffic computation must fulfill.

> IV) *When we enter a node asking whether it can accept some data item this always results in the node asking whether some channel (through its source) accepts this data item. If this leads to a propagation that comes back to this node with a τ query, then we do not evaluate τ's* Offers *and* Accepts *functions as specified in §4.2.2 and §4.2.1. Instead we let τ return immediately to it caller with the data whose acceptance we were testing.*

When we resume our stranded computation from node 2 and follow the three additional new requirements, we get the following traffic path.

$\tau(2,*,2.3)$ = [] ↺ (according to IV) *offers(4.1,*)* = [9] ↺ *Offers(4,*)* = [9] ↺ *Accepts(4,9,4.2)* = *True* ↺ $\tau(4,*,4.2)$ = [9] ↺ *accepts(3.2,9)* = *True* ↺ *Accepts(3,9,-)* = *True* ↺ *accepts(2.2,9)*=*True* ↺ *accepts(2.3,9)* → .....

Now we are about to continue with the evaluation of the second *accepts* call in the not yet fully evaluated *Accepts(2,9,-)* function. Intuitively, we already know that evaluation of the *accepts(2.3,9)* is not necessary. If we still do it we get

accepts(2.3,9) → *Accepts(4,9,-)* → $\tau(3,*,3.2)$ → *Offers(3,*)* → *offers(3.1,*)* → $\tau(2,*,2.2)$ = [9] (because node is already visited and according to requirement IV) *offers(3.1,*)* = [9] ↺ *Offers(3,*)* = [9] ↺ *Accepts(3,9, 3.2)* = *True* ↺ $\tau(3,*,3.2)$ = [9] ↺ *accepts(4.2,9)* = *True* ↺ *Accepts(4,9,-)* = *True* ↺ *accepts(2.3,9)* = *True*.

This is exactly what we expected. When we test whether a node n accepts data this always results in the node asking whether some channel (through its source) accepts this data. Suppose we exit the node in order to answer this question via source a and by propagation of queries this end asks whether some synchronous channel, whose source b is also in node n, offers something (which leads to $\tau(n,*,b)$). The result of this propagation is the same as when we exit via source b (with a channel specific *accepts*) and reenter the node via source a (with a channel specific *offers* that leads to $\tau(n,*,a)$). In other words exiting a node via source a and reentering it along a synchronous part of the network via source b gives the same result as exiting via source b and reentering via source a.

This leads to a fifth requirement that our traffic computation must fulfill.

> V) *Organize the traffic computation for an Accept call on a node that is not a sink node, in such a way that the set of source channel ends that we visit through their channel specific accepts, can shrink during each accepts evaluation.*

Regarding this requirement V we can say that the upperbound for the number of accepts evaluations in an *Accepts* function is equal to the number of source channel ends in the node. However during the computation this number can shrink when for the evaluation of *Accepts* we reenter a node (when we have a cyclic part in the

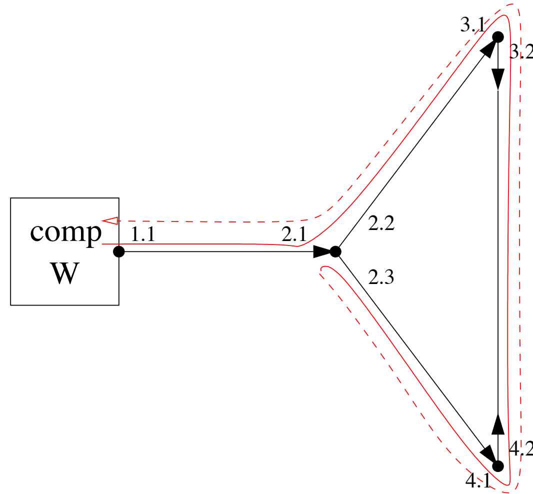


Figure 7: Direction of the computation for the write command of the component (W)

circuit). Such reentering always takes place via a synchronous part of the network. Note that a traffic computation always stops when it encounters an asynchronous channel.

To fulfill requirement V, we can store the source channel end via which we reenter a node in a list and carry this information with us as an additional parameter in the query functions. This means that the third parameter in τ and *Accepts* that represents, up to now, the excluded channel end, must change into a *list* of channel ends to be excluded. Moreover, the channel specific *offers* and *accepts* must also have this list parameter.

Below we give the full computation path for the write operation of the component instance taking into account all additional requirements with respect to the traffic computation.

$\tau(1, *, []$ $\rightarrow$ *Offers(1,*)* $= [9]$ (because it is a source node and there is pending write) $\circlearrowleft$ *Accepts(1,9,[]* $\rightarrow$ *accepts(1.1,9,[])* $\rightarrow$ *Accepts(2,9,[])* $\rightsquigarrow$ *accepts(2.2,9,[])* $\rightarrow$ *Accepts(3,9,[]* $\rightarrow$ *accepts(3.2,9,[]* $\rightarrow \tau(4, *, [4.2]) \rightarrow$ *Offers(4,*,[4.2])* $\rightarrow$ *offers(4.1,*,[4.2])* $\rightarrow \tau(2, *, [4.2, 2.3]) = [9] \circlearrowleft$ (according to IV we go back) *Offers(4,*,[4.2,2.3])* $= [] \circlearrowleft$ *Accepts(4,9,[4.2,2.3]* $=$ *True* $\circlearrowleft \tau(4, *, [4.2, 2.3]) = [9] \circlearrowleft$ *accepts(3.2,9,[4.2,2.3])* $=$ *True* $\circlearrowleft$ *Accepts(3,9,[4.2,2.3])* $=$ *True* $\circlearrowleft$ *accepts(2.2,9,[4.2,2.3]))* $=$ *True* $\circlearrowleft$ (so now *accepts(2.3,9)* is skipped, because according to requirement V the set of source channels has shrunk) *Accepts(2,9,[])* $=$ *True* $\circlearrowleft$ *accepts(1.1,9,[])* $=$ *True* $\circlearrowleft$ *Accepts(1,9,[])* $=$ *True* $\circlearrowleft \tau(1, *, []) = [9] \circlearrowleft$. Because τ is not empty data transport is possible. We show this computation path in Figure 7.

4.4.3 Circuit 3

In previous examples it was always the *write* operation of a component that offered the data for a transport. In our next Reo circuit (see Figure 8) this is different: it is a `SyncSpout` that generates the data. In this circuit, three `Sync` channels, one `SyncSpout`, and one `SyncDrain` are connected to each other as shown in Figure 8. The component instance performs a *take* operation. Intuitively, we see that the component instance can perform the take operation because the data comming from the `SyncSpout` (see §4.3.3) can flow via node 3 to node 2. There it is replicated into the source channel ends of 2.1 and 2.2, and can flow through their respective `Sync`

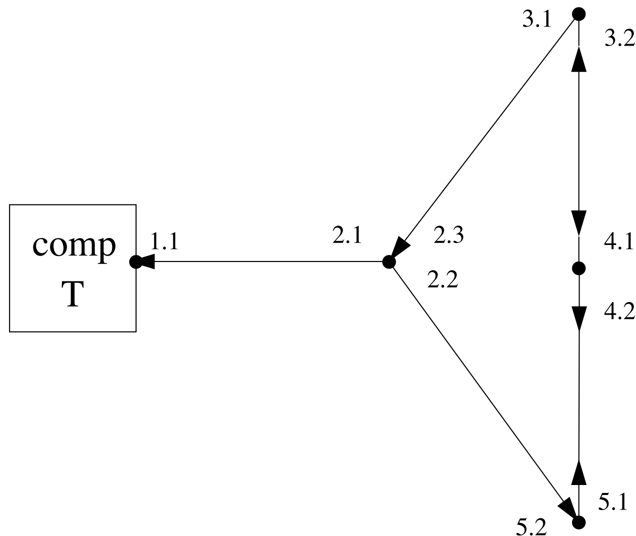


Figure 8: A simple third circuit

channels to the component instance, and via node 5 into the `SyncDrain`, whose other end at node 4 synchronously receives the data item produced by the other end of the `SyncSpout`.

It is clear that when we start the traffic computation in node 1, the data that it offers is not yet determined. It becomes available later when we encounter the `SyncSpout` during traffic computation. This "later availability" is not a problem when we organize the computation as follows: We introduce a list in which we store pairs consisting of a visited node and the data it offers. We call this list *the list of visited nodes*. Every time we call a channel specific *offers* function during the traffic computation, we store such a pair in the visited nodes list. When it is not yet known what a node offers (as in our case with node 1) we use the symbol `nyd` (not yet determined) for the data item in the pair. If we carry this additional parameter with us during the traffic computation the `nyd` data can be filled in when we encounter a `SyncSpout`.

Note that by introducing this list of visited nodes and having it available during the traffic computation, we also solve the problems formulated in the requirements II and III. When we visit a node through some function, we can always look up in the visited nodes list whether we have already visited that node (i.e., solving requirement II) and we can also look up the data a node offers (i.e., solving requirement III).

The reader can work out the full computation path similarly to how we derived it for Figure 7. Doing this, it turns out that τ is a non-empty list. This means that the component *comp* can successfully perform the *take* operation. The computation path is shown in Figure 9.

5 Node Behavior

In this section we give the final version of the node functions.

5.1 Node Functions

In this section we define the three node functions that define the behavior of nodes in Reo circuits: *Accept*, *Offers*, and τ. All node functions take three common

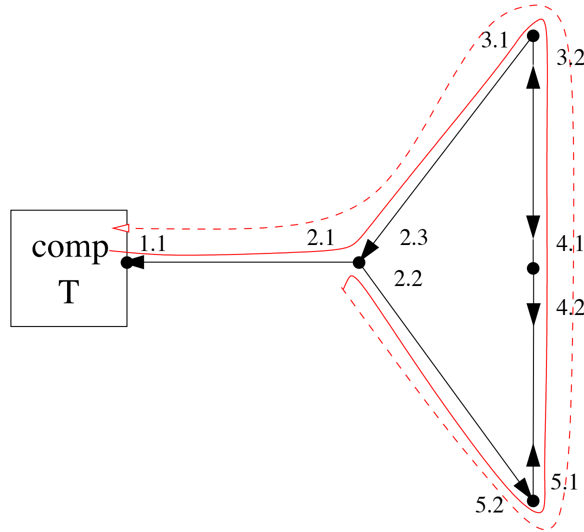


Figure 9: Direction of the computation for the take command of the component (T)

arguments: n - the node that executes the function (performs the query); S - the list of already visited nodes; and E - the list of excluded channel ends. We organize all arguments together using a tuple.

5.1.1 *Accepts*

The *Accepts* function takes an ordered 4-tuple consisting of a node n, a data item d, a list of visited nodes S, and a list of excluded channel ends E, and returns an ordered pair consisting of a boolean value and an updated list of excluded channels ends. The boolean value indicates whether or not n accepts d.

$$
\begin{array}{l}
\mathit{Accepts} :: (\mathit{Node}, \mathit{Data}, [(\mathit{Node}, \mathit{Data})], [\mathit{CE}]) \rightarrow (\mathit{Bool},\ [\mathit{CE}]) \\
\mathit{Accepts}\ (n, d, S, E) \quad = \textbf{if}\ (n, _) \in S\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad (\mathit{False}, E) \\
\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\quad \textbf{if}\ \mathit{is}_{sinknode}(n)\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad (P(n) \neq [] \wedge \bigwedge [d \ni p \mid p \leftarrow P(n)],\ E) \\
\qquad\qquad\qquad\qquad\quad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad \textbf{if}\ \mathit{cel} \neq []\ \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{bool}, E') \bowtie \mathit{Accepts}(n, d, S, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E' \cup [\mathit{head\ cel}]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathit{True}, E) \\
\quad \textbf{where} \\
\qquad \mathit{cel} \qquad\quad = \mathit{srcCE}_{sel}(n) \setminus E \\
\qquad (\mathit{bool}, E') = \mathit{accepts}\ (\mathit{head\ cel}, d, S \cup [(n, d)], E) \\
\qquad \mathit{accepts} \quad = \mathit{accepts}_{sel}(\mathit{ch}) \\
\qquad \mathit{ch} \qquad\quad\ = \mathit{channel}_{sel}(\mathit{head\ cel})
\end{array}
$$

In the *Accepts* function, we check whether we have visited n, by testing whether S already contains an ordered pair $(n, _)$ that contains n. Note that we use "_" as a shorthand for any data item. We exit a node that we visit in only two cases: (1) we exit it through a source (by asking *accepts* within its *Accepts*), or (2) we exit it through a sink (by asking *offers* in its *Offers* - see below). Suppose we visit an

already visited node n to ask (with *Accepts*) whether it can accept d. To accept d in the first case would mean that whatever n accepts depends on whatever n accepts (i.e., a self reference). To accept d in the second case would mean that n tries to synchronize two sinks (the first one we get from the *offers* called when exiting the visited n, and the second one we get from the call to *Accepts*). This violates the rule that a node non-deterministically selects precisely one of its sinks to take data from. In both cases *Accepts* returns $(False, E)$ indicating that n cannot accept d.

A sink node that we visit for the first time accepts a data item only if the patterns of all *takes* (there must exist at least one) pending on the node, match the data item. Any mixed or source node that we visit for the first time accepts d only if the channels of all of its sources that do not belong to E also accept d. We perform this check in a recursive manner.

We define *cel* in the where-clause as the source channel end list in a node minus the list of excluded channel ends E. We first evaluate whether one source (*head cel*) from the *cel* list accepts d, and then we recursively evaluate *Accepts* over the rest of the sources in n. Evaluating $(bool, E')$ requires calling *accepts* of the channel that corresponds to *head cel*. Furthermore, we pass to *accepts* of the channel an S added with (n, d) to indicate that we have visited this node with some data element d. *Accepts* returns the result of $(bool, E')$ combined with the result of the recursive call to *Accepts*, by using the $\bowtie$ operation. Note that the recursive call to *Accepts* uses E', which contains E updated with any channel ends from possible loops that we may have encountered during the call to the *accepts* on *head cel*.

The recursion ends when the *cel* becomes empty, in which case Accepts returns $(True, E)$ indicating that it has accepted d. We make sure that *cel* becomes empty, because every time we make the recursive call to *Accepts* we effectively reduce *cel* by one element.

Below we define the function $\bowtie$:

$$\bowtie :: (Bool,\ [CE]) \rightarrow (Bool,\ [CE]) \rightarrow (Bool,\ [CE])$$
$$\bowtie\ (b1,\ ce1)\,(b2,\ ce2)\ =\ (b1\ \wedge\ b2,\ ce1\ +\!\!+\ ce2)$$

We use $\bowtie$ to combine the result of a testing for acceptance (*accepts*) of a data item from a single channel and the result of testing for acceptance (*Accepts*) from the rest of the channels whose sources coincide on n. The $\bowtie$ function returns a pair, the first element of which represents a boolean obtained by applying a boolean $\wedge$ on the first element of the ordered pair of both its arguments. The second element of the result pair consists of the concatenation of the list of excluded channel ends of the second elements of the same ordered pairs. Note that the $\bowtie$ function may produce repetitions in the second element of the resulting pair, because we use concatenation (not union) in the last rule. This does not affect the function result with respect to the node behavior.

5.1.2 *Offers*

The *Offers* function takes an ordered 4-tuple consisting of a node n, a pattern p, a list of visited nodes S, and a list of excluded channel ends E, and returns a list of ordered pairs, each consisting of a data item and its corresponding list of excluded channel ends.

$$Offers\ ::\ (Node,\ Pattern, [(Node, Data)], [CE]) \rightarrow\ [(Data, [CE])]$$

$$
\begin{array}{lll}
\textit{Offers}\ (n, p, S, E) & = & \textbf{if}\ (n, _)\ \in\ S\ \textbf{then} \\
 & & \quad [(d, E) \mid (n1, d) \leftarrow S, n1 == n \wedge d \ni p\ ] \\
 & & \textbf{else} \\
 & & \quad \textbf{if}\ is_{sourcenode}(n)\ \textbf{then} \\
 & & \quad\quad [(d, E) \mid d \leftarrow W(n, p)] \\
 & & \quad \textbf{else} \\
 & & \quad\quad \bigcup\ [\textit{offers}(x, p, S \cup [(n, \mathtt{nyd})], E) \\
 & & \quad\quad\quad\quad\quad\quad \mid\ x \leftarrow \textit{snkCE}_{sel}(n)] \\
\end{array}
$$

where

$$
\begin{array}{ll}
\textit{offers} & = \textit{offers}_{sel}(ch) \\
ch & = \textit{channel}_{sel}(x)
\end{array}
$$

In the *Offers* function, we first check whether we have already visited n. Having visited n can mean that this node already has something to offer. Therefore, *Offers* returns an ordered pair (d, E) containing the data item d for which (n, d) belongs to S and d matches p. Note that `nyd` does not match any pattern including "*" - it matches "_"only. Therefore, *offers* does not return any pairs containing `nyd`.

A source node that we visit for the first time offers ordered pairs (d, E), where d's come from the *writes* pending on the node that match p (produced by $W(n, p)$). Any mixed or sink node that we visit for the first time offers a union of what the channels of its sinks have to offer. We call *offers* with an S updated with $(n, \mathtt{nyd})$ to indicate that we have visited n, but because we do not know what this node has to offer yet, we pair n with the special data item `nyd` (not yet determined). Effectively, we delay determining what n has to offer, until we come to a node that has something to offer, which would also imply that n has something to offer.

5.1.3 τ

The τ function takes an ordered 4-tuple consisting of a node n, a pattern p, a list of visited nodes S, and a list of excluded channel ends E, and returns a list of ordered pairs, each consisting of a data item and its corresponding list of excluded channel ends.

$$
\begin{array}{lll}
\tau\ ::\ (\textit{Node},\ \textit{Pattern},\ [(\textit{Node}, \textit{Data})],\ [\textit{CE}]) \rightarrow\ [(\textit{Data}, [\textit{CE}])] & & \\
\tau\ (n, p, S, E) & = & \textbf{if}\ (n, _)\ \in\ S\ \textbf{then} \\
 & & \quad \textit{Offers}(n, p, S, E) \\
 & & \textbf{else} \\
 & & \quad [\ (d, E'')\ \mid\ (d, E') \leftarrow \textit{Offers}(n, p, S, E), \\
 & & \quad\quad\quad\quad\quad (b, E'') \leftarrow \textit{Accepts}(n, d, S', E'), \\
 & & \quad\quad\quad\quad\quad b == \textit{True}] \\
\end{array}
$$

where

$$
S' = \textit{update}\ S\ d
$$

In the τ function, we first check whether we have already visited n. Having visited n can mean that we have already determined what this node has to offer. Therefore, we delegate to *Offers* to determine the result. Note that we do not need to test for acceptance as we have done it before when we visited the node for the first time.

For a node that we have not visited yet, we return the ordered pairs (d, E''), such that d represents a data item that the node offers and accepts. We determine the offered data from the ordered pairs (d, E') returned by *Offers*. We consider only data items d for which *Accepts* returns as a result $(True, E'')$. Passing E' to *Accepts* makes sure that we do not enter a loop encountered by *Offers*. By returning E'' with each offered and accepted data item we make sure that we do not enter again loops encountered either by *Offers* or *Accepts*. Furthermore, we pass an updated visited nodes list S' to *Accepts*.

Previous calls to *Offers* up to and including the current call to τ, may have accumulated in S' pairs with `nyd` in them, indicating that we have visited some nodes, but have not yet determined the data they offer. In τ, when we obtain the result of *Offers* for n, we proceed to test for acceptance of the pairs in the result. The moment we select a data item d from the result of *Offers*, we effectively assume that n has d to offer, and we continue to test this hypothesis by calling *Accepts*. At this point we need to replace all `nyd` data items in pairs in S, because the d has become available in them too. Note that previous calls to *Offers* and τ, chain up only for synchronous channels (see definitions in the next section). We require the replacement of `nyd` as early as this point, because the subsequent call to *Accepts* may result in a loop ending on one of the nodes that previously had `nyd` in S. Effectively, here we resolve the delay in determining what some previously visited nodes have to offer. Note that the `SyncSpout` channel (see the next section) has a similar behavior with respect to generating and offering data. We make S' from S using the *update* function defined below:

$$\begin{array}{l}
update :: [(Node, Data)] \rightarrow Data \rightarrow [(Node, Data)] \\
update\ []\ d = [] \\
update\ ((n, \texttt{nyd}) : t)\ d = [(n, d\ )] +\!\!+ (update\ t\ d) \\
update\ ((n, d1\ ) : t)\ d = [(n, d1)] +\!\!+ (update\ t\ d)
\end{array}$$

The *update* function takes as arguments a visited nodes list and a data item, and returns an updated visited nodes list. We update all pairs in the visited nodes list argument, so that if a pair contains a `nyd` data item, we replace it with the data item argument. The lines from the function declaration read as follows. Updating an empty list results in an empty list. Updating a list with a head, a pair that contains a `nyd` as data item, results in a list with a head, the same pair with the `nyd` replaced by d, followed by the updated tail of the argument list. A list with a head pair that does not contain `nyd` as data item, results in a list with the same head followed by the updated tail of the argument list.

6 Channel Behavior

For every channel, the Reo operational semantics defines an *accepts* for its *source* ends, and an *offers* for its *sink* ends.

The *accepts* function takes an ordered 4-tuple consisting of a channel end x, a data item d, a list of visited nodes S, and a list of excluded channel ends E. The *accepts* function returns an ordered pair consisting of a boolean and an updated excluded channel ends list. The boolean indicates whether or not the channel accepts data item d.

The *offers* function takes an ordered 4-tuple consisting of a channel end y, a pattern p, a list of visited nodes S, and a list of excluded channel ends E, and returns a list of ordered pairs consisting of a data item offered by the channel that matches p, and its corresponding list of excluded channel ends.

For completeness, below we define *accepts* for *sink* channel ends, and *offers* for *source* channel ends. Note that the Reo operational semantics properly distinguishes between sources and sinks coincident on a node, and performs the *accepts* on sources and *offers* on sinks only.

$$\begin{array}{l}
offers :: (CE,\ Pattern,\ [(Node, Data)],\ [CE]) \rightarrow [(Data, [CE])] \\
offers\ (x, d, S, E) = ([], E)
\end{array}$$

A *source* never offers any data items, hence its *offers* returns [].

$$accepts :: (CE,\ Data,\ [(Node, Data)],\ [CE]) \rightarrow (Bool, [CE])$$

$$accepts\ (y, d, S, E) \ = \ (False,\ E)$$

A *sink* never accepts data items, hence its *accepts* returns $False$.

6.1 Asynchronous Channels

In this section we specify the behavior of several useful asynchronous channels. Asynchronous channels typically share the characteristic that they do not propagate the queries made at one of their ends to the node at their other end.

6.1.1 `FIFO`

A `FIFO` channel has a source and a sink and thus it can both accept and offer data. A `FIFO` channel also has an unbounded buffer for storing data items. A `FIFO` channel always accepts a data item d through its source x.

$$\begin{array}{l} accepts\ ::\ (CE,\ Data,\ [(Node, Data)],\ [CE]) \rightarrow\ (Bool, [CE]) \\ accepts\ (x, d, S, E) \ = \ (True,\ E) \end{array}$$

A `FIFO` channel offers nothing through its sink y if its buffer contains no data item. Otherwise, it offers the data item d that is the oldest in its buffer, if d matches the pattern p.

$$\begin{array}{l} offers\ ::\ (CE,\ Pattern,\ [(Node, Data)],\ [CE]) \rightarrow [(Data, [CE])] \\ offers\ (y, p, S, E) \quad = \ \mathbf{if}\ B\ \neq\ []\ \wedge\ d\ \ni\ p\ \mathbf{then} \\ \qquad\qquad\qquad\qquad\quad [(d,\ E)] \\ \qquad\qquad\qquad\qquad \mathbf{else} \\ \qquad\qquad\qquad\qquad\quad [] \\ \quad \mathbf{where} \\ \qquad B = buffer_{sel}(c) \\ \qquad c\ = channel_{sel}(y) \\ \qquad d\ = head\ B \end{array}$$

6.1.2 $\texttt{FIFO}_n$

A $\texttt{FIFO}_n$ channel has a source and a sink and thus it can both accept and offer data. A $\texttt{FIFO}_n$ channel also has a buffer for storing up to n data items. A $\texttt{FIFO}_n$ channel accepts a data item d through its source x, if its buffer B can hold at least one more data item.

$$\begin{array}{l} accepts\ ::\ (CE,\ Data,\ [(Node, Data)],\ [CE]) \rightarrow\ (Bool, [CE]) \\ accepts\ (x, d, S, E) \ = \ (length\ B\ >\ n,\ E) \\ \quad \mathbf{where} \\ \qquad B = buffer_{sel}(c) \\ \qquad c\ = channel_{sel}(x) \\ \qquad n\ = capacity_{sel}(c) \end{array}$$

A $\texttt{FIFO}_n$ channel offers through its sink y nothing if its buffer is empty. Otherwise, it offers the data item d in its buffer, if this item matches the pattern p.

$$\begin{array}{l} offers\ ::\ (CE,\ Pattern,\ [(Node, Data)],\ [CE]) \rightarrow [(Data, [CE])] \\ offers\ (y, p, S, E) \quad = \ \mathbf{if}\ B\ \neq\ []\ \wedge\ d\ \ni\ p\ \mathbf{then} \\ \qquad\qquad\qquad\qquad\quad [(d,\ E)] \\ \qquad\qquad\qquad\qquad \mathbf{else} \\ \qquad\qquad\qquad\qquad\quad [] \end{array}$$

where
$B = \mathit{buffer}_{sel}(c)$
$c = \mathit{channel}_{sel}(y)$
$d = \mathit{head}\ B$

6.2 Synchronous Channels

In this section we specify the behavior of several useful synchronous channels. Synchronous channels all share the characteristic that they propagate the queries made at one of their ends to the nodes at their other ends, effectively synchronizing the behavior of the nodes at both their ends.

6.2.1 `Sync`

A `Sync` channel has a source and a sink and thus it can both accept and offer data. A `Sync` channel accepts a data item d through its source x only if the node n of its sink y accepts the data item d from this channel.

$\mathit{accepts} :: (\mathit{CE},\ \mathit{Data},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow (\mathit{Bool}, [\mathit{CE}])$
$\mathit{accepts}\ (x, d, S, E) = \mathit{Accepts}(n, d, S, E)$
where
$n = \mathit{node}_{sel}(y)$
$[y] = \mathit{channelend}_{sel}(c) \backslash [x]$
$c = \mathit{channel}_{sel}(x)$

A `Sync` channel offers a data item matching a pattern p from its sink y only if the node n of its source x can traffic a data item matching the pattern p into this channel.

$\mathit{offers} :: (\mathit{CE},\ \mathit{Pattern},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow [(\mathit{Data}, [\mathit{CE}])]$
$\mathit{offers}\ (y, p, S, E) = \tau\ (n, p, S, E \cup [x])$
where
$n = \mathit{node}_{sel}(x)$
$[x] = \mathit{channelend}_{sel}(c) \backslash [y]$
$c = \mathit{channel}_{sel}(y)$

We call the τ function with an excluded channel end list added with the source of the channel to prevent τ from entering the same channel, when testing for acceptance in its call to *Accepts*, after having selected some data item from the result of its call to *Offers*. Furthermore, adding the source to the excluded channel end list also makes sure that it will appear as an excluded channel end in the result of the *offers* function (node behavior takes care of this). This makes sure that should we later (in τ) discover that we have already visited n (and hence we have a loop), the channel end through which we looped belongs to the excluded channel ends list in the result. This contributes to preventing entering a loop from the other side (its other channel end on the node on which the loop begins).

6.2.2 `SyncDrain`

A `SyncDrain` channel has two sources and thus it can only accept data. A `SyncDrain` channel accepts a data item d through its source x_1 only if the node n of its other source x_2 can traffic an arbitrary data item (pattern "*") into the channel as well.

$\mathit{accepts} :: (\mathit{CE},\ \mathit{Data},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow (\mathit{Bool}, [\mathit{CE}])$
$\mathit{accepts}\ (x_1, d, S, E) = (\mathit{tau} \neq [],\ \oplus\ \mathit{tau})$

where

$$\begin{aligned}
tau &= \tau\,(n, *, S, E \cup [x_2]) \\
n &= node_{sel}(x_2) \\
[x_2] &= channelend_{sel}(c)\backslash[x_1] \\
c &= channel_{sel}(x_1)
\end{aligned}$$

The *accepts* returns an ordered pair consisting of a boolean expression indicating whether *tau* (see its definition in the where-clause) contains at least one data item, and an expression $\oplus tau$ (the defintion of $\oplus$ is given below), which produces a list of excluded channel ends that contains the excluded channel end lists of all data items returned by τ. We call the τ function with an excluded channel ends list added with the other source of the channel to prevent τ from entering the same channel, when testing for acceptance in its call to *Accepts*, after having selected some data item from the result of its call to *Offers*.

The $\oplus$ function takes as its argument the result of τ, and returns as its result the concatenation of all the channel ends lists of the ordered pairs from its argument list. Below, we provide the function declaration using pattern matching:

$$\begin{aligned}
&\oplus :: \ [(Data, [CE])] \rightarrow \ [CE] \\
&\oplus\, [\,] \ = \ [\,] \\
&\oplus\, (e : t) \ = \ (snd\ e) \ +\!\!+ (\oplus \ \ t)
\end{aligned}$$

The declaration lines read as follows. Performing the function on the empty list results in an empty list. Performing the function on the list with the head pair e results in the concatenation of the second element of e and the result of performing the function recursively to the tail of the argument list.

6.2.3 `SyncSpout`

A `SyncSpout` channel has two sinks and thus it can only offer data. A `SyncSpout` channel offers a non-deterministically-generated data item d from its sink y_1 only if d matches the pattern p and the node n of its other sink y_2 accept another non-deterministically-generated data item d' from this channel.

$$\begin{aligned}
&offers\ ::\ (CE,\ Pattern,\ [(Node, Data)],\ [CE]) \rightarrow [(Data, [CE])] \\
&offers\,(y_1, p, S, E)\ =\ [\,(d, E')\mid\ d\ \leftarrow\ random()\ \wedge\ d \ni p, \\
&\qquad\qquad d'\ \leftarrow\ random(), \\
&\qquad\qquad (b,\ E')\ \leftarrow\ Accepts(n,\ d',\ S',\ E), \\
&\qquad\qquad b == True]
\end{aligned}$$

where

$$\begin{aligned}
n &= node_{sel}(y_2) \\
[y_2] &= channelend_{sel}(c)\backslash[y_1] \\
c &= channel_{sel}(y_1) \\
S' &= update\ S\ d
\end{aligned}$$

Note that the $random()$ function serves as a non-deterministic generator of data for the `SyncSpout` channel. Furthermore, the `SyncSpout` channel serves as a data generator and therefore we pass S', derived from S by replacing all `nyd` in its pairs with d', to *Accepts*. Effectively, with *update* here we resolve a previous delay in determining what some already visited nodes have to offer.

6.2.4 `LossySync`

A `LossySync` channel has a source and a sink and thus it can both accept and offer data. A `LossySync` channel always accepts any data item d through its source x.

However a `LossySync` channel always tests whether the node n of its sink y accepts the data item d from this channel.

$$
\begin{aligned}
&\mathit{accepts} :: (\mathit{CE},\ \mathit{Data},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow (\mathit{Bool}, [\mathit{CE}]) \\
&\mathit{accepts}\ (x, d, S, E)\ =\ (\mathit{True},\ \mathit{snd}\ \mathit{Accepts}(n, d, S, E)) \\
&\quad \textbf{where} \\
&\qquad p\ \ = \mathit{pattern}_{sel}(c) \\
&\qquad n\ \ = \mathit{node}_{sel}(y) \\
&\qquad [y] = \mathit{channelend}_{sel}(c)\backslash[x] \\
&\qquad c\ \ = \mathit{channel}_{sel}(x)
\end{aligned}
$$

We require the `LossySync` to test for acceptance on n, because if the n accepts, we need to return the resulting excluded channel ends list, to prevent re-entering possible loops from the other side.

A `LossySync` channel offers a data item matching a pattern p from its sink y only if the node n of its source x can traffic a data item matching the pattern p into this channel.

$$
\begin{aligned}
&\mathit{offers} :: (\mathit{CE},\ \mathit{Pattern},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow [(\mathit{Data}, [\mathit{CE}])] \\
&\mathit{offers}\ (y, p, S, E)\ =\ \tau\ (n, p, S, E \cup [x]) \\
&\quad \textbf{where} \\
&\qquad n\ \ = \mathit{node}_{sel}(x) \\
&\qquad [x] = \mathit{channelend}_{sel}(c)\backslash[y] \\
&\qquad c\ \ = \mathit{channel}_{sel}(y)
\end{aligned}
$$

6.2.5 `Filter`

A `Filter` channel has a source and a sink and thus it can both accept and offer data. A `Filter` channel always accepts through its source x any data item d that does not match its pattern p. A `Filter` channel accepts a data item d that matches its pattern p only if the node n of its sink y accepts the data item d from this channel.

$$
\begin{aligned}
&\mathit{accepts} :: (\mathit{CE},\ \mathit{Data},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow (\mathit{Bool}, [\mathit{CE}]) \\
&\mathit{accepts}\ (x, d, S, E)\ =\quad = \textbf{if}\ d\ \not\ni\ p\ \textbf{then} \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{True},\ E) \\
&\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Accepts}(n, d, S, E) \\
&\quad \textbf{where} \\
&\qquad p\ \ = \mathit{pattern}_{sel}(c) \\
&\qquad n\ \ = \mathit{node}_{sel}(y) \\
&\qquad [y] = \mathit{channelend}_{sel}(c)\backslash[x] \\
&\qquad c\ \ = \mathit{channel}_{sel}(x)
\end{aligned}
$$

A `Filter` channel offers a data item matching a pattern p from its sink y only if the node n of its source x can traffic a data item matching the pattern p and the `Filter`'s pattern fp into this channel.

$$
\begin{aligned}
&\mathit{offers} :: (\mathit{CE},\ \mathit{Pattern},\ [(\mathit{Node}, \mathit{Data})],\ [\mathit{CE}]) \rightarrow [(\mathit{Data}, [\mathit{CE}])] \\
&\mathit{offers}\ (y, p, S, E)\ =\ \tau\ (n, p\ \wedge\ \mathit{fp}, S, E \cup [x]) \\
&\quad \textbf{where} \\
&\qquad n\ \ = \mathit{node}_{sel}(x) \\
&\qquad \mathit{fp}\ = \mathit{pattern}_{sel}(c) \\
&\qquad [x] = \mathit{channelend}_{sel}(c)\backslash[y] \\
&\qquad c\ \ = \mathit{channel}_{sel}(y)
\end{aligned}
$$

7 Examples

In this section we demonstrate how one can apply our operational of Reo semantics to compute the data traffic in a set of example circuits. Computing the traffic in a Reo circuit with an acyclic graph does not pose any significant challenge. Furthermore, we focus only on circuits that contain synchronous channels, because only they propagate queries to their opposite ends. Therefore, we select examples that demonstrate how we can compute the traffic in synchronous Reo circuits containing various loops.

As an example selection criteria we use an exhaustive classification of loop types in Reo circuits. By selecting a set of examples containing every possible loop we increase our confidence in the correctness of the proposed specification. In a separate ongoing work, we also plan to provide an actual proof that the proposed operational semantics produces results equivalent to the algebraic semantics of Reo.

7.1 Loop categorization

In Reo, data flow through a sink of a channel into a node, on which the sink coincides. Data flow through a source into its channel from a node on which the source coincides. When computing the traffic in a Reo circuit, individual queries propagate through channels and nodes via channel ends. Therefore, we can enter and exit a node through either a sink or a source. Using *sink* and *source* we can describe a path in the computation using a list. For example, assuming we have a path that starts and finishes at a node,

[*sink, source, sink, sink, source, source*]

indicates that the starting node asks what some channel had to offer, then the channel asks the node on its other side what it has to traffic, the node then asks another channel what it has to offer, the channel now changes the flow direction asking whether the node at its other side can accept something, then the node asks a channel whether it can accept something, then the channel changes the flow direction asking what the node on its other side can traffic. Note that alternations of *sink* and *source* mean that data flow in one direction only, while two subsequent *sinks* or two subsequent *sources* mean that we have a switch in the direction of the flow of data. `SyncSpout` and `SyncDrain` do such switching. Furthermore, a node also may perform switching: when asked to evaluate its τ (always through a source), after evaluating *Offers* and determining the data it offers, the node evaluates *Accepts* (always through a source), resulting in a [*source, source*] direction switch at this node (exiting through *Offers* does not count because it returns some result successfully).

In our operational semantics, we keep track of which nodes we have already visited. When we encounter an already visited node, we detect a loop. We use the notation we introduced above to categorize loops based on the characteristics of the propagations that occur at the node that we visit for a second time.

We can enter, exit, and re-enter a node (re-entering means a loop) either through a sink or through a source. This produces $2^3 = 8$ combinations of propagations that can form a loop. Since we want to visit every node precisely once, during a second visit the operational semantics of Reo should provide a return value without further propagation. In Table 6 we list all loop types together with the return values at node re-entry.

Figure 10 depicts the different loop cases. In each case we zoom in only on the node that we visit for the second time. For simplicity, wherever necessary we depict other nodes as black dots. We also abstract away from whole channels, depicting only the channel ends that belong to the visited node and also depict a line and

Nr.	Enter	Exit	Re-enter	Switch count	Result to return at re-entry
1	source	source	source	N/A	[d]
2	source	source	sink	N/A	False
3.1	source	sink	source	0	[ ]
3.2	source	sink	source	$2 * k, k \geq 1$	[d]
4	source	sink	sink	N/A	False
5	sink	source	source	N/A	[d]
6	sink	source	sink	N/A	False
7	sink	sink	source	N/A	N/A
8	sink	sink	sink	N/A	N/A

Table 6: Table of all possible loops

an arrow to indicate the type of the channel end. An arrow pointing to the node indicates a sink. An arrow pointing away from the node indicates a source.

7.2 Loops

7.2.1 Source-source-source loops (loop 1)

In this type of loops, shown in Figure 10, we enter the node asking about its traffic (i.e., τ). This results in the node asking the channel of its coincident sinks about data. Suppose that the node offers some data item d (shown by a "d" on Figure 10) but as part of the traffic query we need to test for acceptance. This results in the node asking a channel through one of its coincident sources whether it accepts d. We abstract away from what happens next, but at some point the propagation of queries returns to this node with asking what the node has to traffic. Since during the first visit we have already decided that the node offers d, we answer that the node traffics the data item d.

In Figure 11 we show a circuit with a loop of this type. This circuit contains two `Sync` channels, a `SyncDrain`, three nodes, and two application component instances (*comp1* and *comp2*). First *comp1* performs a *write* operation on node 1 (*W1*) and later *comp2* performs a *take* operation on node 2 with a pattern "*" (*T2*).

In this example, we expect that the take operation on node 2 will succeed to retrieve the data item of the write operation on node 1, because the computation of the traffic on node 2 results in source-source-source loop (loop 1) in node 3. In Figure 11 we also show the computation path for the *take* operation.

Note that, when we evaluate the traffic in node 1 for the *W1* operation along the path [1.1, 3.1, 3.2, 3.3] or along [1.1, 3.1, 3.3, 3.2] we have a sink-source-source loop in node 3 (loop 5; see §7.2.5). Note also that the other possibility to propagate along the path [1.1, 3.1, 3.4, 2.1] will not yield a loop.

7.2.2 Source-source-sink loops (loop 2)

In this type of loops, shown in Figure 10, we enter the node asking about its traffic. Because a traffic computation (i.e., τ) always starts with asking the channels of the node's coincident sinks what data they offer (i.e., the *offers* part of τ) we always exit via a sink with an *offers* query. Suppose we return from this *offers* query with some data item d (shown by the a "d" in Figure 10) in the node, and again we exit the node via a source channel end (so with an *accepts* query). We abstract away from what happens next, but at some point the propagation of queries returns to this node via a sink (i.e., reenters the node) asking whether the node can accept some data item (this can be d but not necessarily so). We answer this last question with *False*, because otherwise we allow this circuit to synchronize two sinks - the

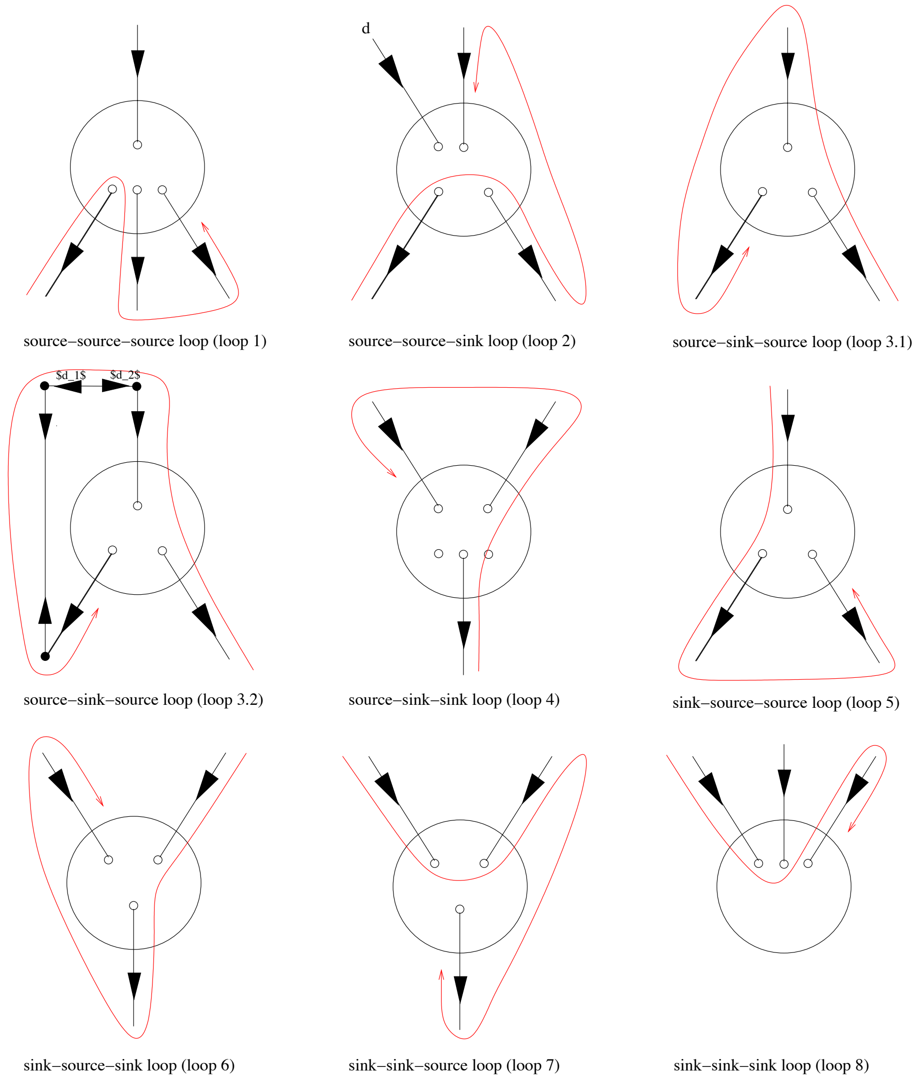


Figure 10: Loop types illustrated

one through which the data item is intended to come and the one through which we re-enter the node. For this reason, the channels of these sinks can never offer anything to this node in a circuit containing this kind of loop.

In Figure 12 we show a circuit with a loop of this type. This circuit contains three `Sync` channels, three nodes, and two application component instances (*comp1* and *comp2*). First *comp1* performs a *write* operation on node 3 (*W1*) and later *comp2* performs a *take* operation on node 1 with a pattern "*" (*T2*). In this example, we expect that the *take* operation on node 2 will fail to retrieve the data item of the *write* operation on node 3, because the computation of the traffic on node 2 results in a source-source-sink loop (loop 2) in node 3. We also show the computation path for the *take* operation in Figure 12.

Note that if we evaluate the traffic in node 1 for the *W1* operation along the path [1.1, 3.2, 3.3, 3.4], we encounter a sink-source-sink loop in node 3 (loop 6; see §7.2.6).

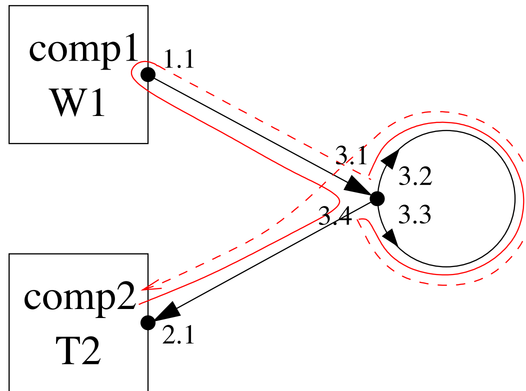


Figure 11: A source-source-source loop (loop 1)

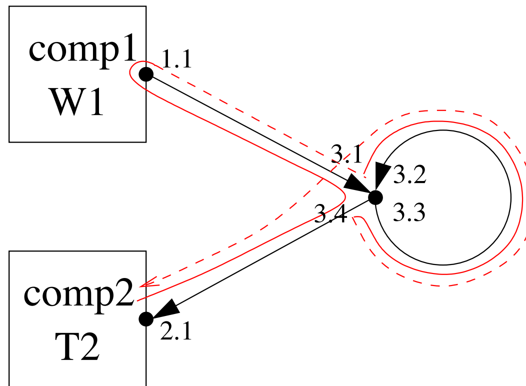


Figure 12: A source-source-sink loop (loop 2)

7.2.3 Source-sink-source loops (loop 3.1 and loop 3.2)

In this type of loops, shown in Figure 10, we enter the node asking about its traffic. This results in the node asking what some channel (through its sink) offers. We abstract away from what happens next, but at some point the propagation of the queries returns to this node asking what the node has to offer.

We distinguish two subtypes of these loops based on for the path between the *exit* and the *re-entry* of the node: (1) data generation has occurred in the path (see Figure 10, loop 3.1), and (2) no data generation had occurred in the path (see Figure 10, loop 3.2). Note that data generation occurs when at least one [*sink, sink*] switch occurs in the path. To have this type of a loop, the path we speak about must start with a sink and end with a source. Therefore, after a [*sink, sink*] switch we must have a subsequent [*source, source*] switch. A path that generates data has $2 * k, k \geq 1$ switches in total, while a loop path that does not, has 0 switches.

If the case we have a path without switches the visited node has nothing to offer (none of its channels can generate anything within the loop). If we have a path that generates data, as we discussed above, the node offers a data item d_1 (see in Figure 10, loop 3.2) generated by the first switch in the path, because the node has delegated the answer to the *offers* query (asked upon entry) to one of its channels, and somewhere in the loop path some entity (a channel or another node) answered it. The re-entry askes the same question as the first entry, and hence should get the same answer.

In Figure 13 we show a circuit with a loop of this type where no data is generated

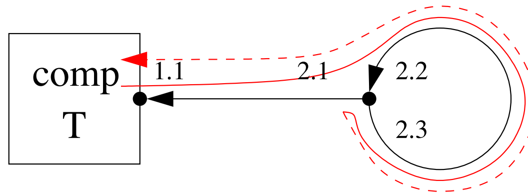


Figure 13: A source-sink-source loop (loop 3.1)

in the loop path. This example contains two `Sync` channels, two nodes and an application component instance that performs a *take* operation on node 1. In this example, we expect that the *take* operation on node 1 will never succeed to retrieve any data, because the computation of the traffic on node 1 results in a source-sink-source loop without data generation (loop 3.1), which tries to take data from a node that does not offer anything (yet). In Figure 13 we also show the computation path for *take* operation.

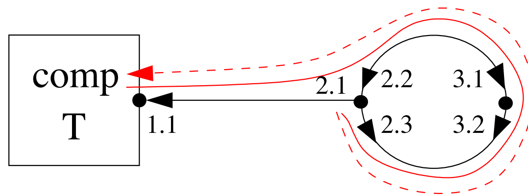


Figure 14: A source-sink-source loop (loop 3.2)

In Figure 14 we show a circuit with a loop of this type where a data item is generated in the loop path. This example contains a `Sync`, a `SyncSpout`, a `SyncDrain`, three nodes and an application component instance that performs a *take* operation on node 1. In this example, we expect that the *take* operation on node 1 will succeed to retrieve a data item, because the computation of the traffic on node 1 results in a source-sink-source loop with data generation (loop 3.2), which takes data from a node that takes its data from a `SyncSpout`. In Figure 14 we also show the computation path for the *take* operation.

7.2.4 Source-sink-sink loops (loop 4)

In this type of loops, shown in Figure 10, we enter the node asking about its traffic. This results in the node asking what some channel (through its sink) offers. We abstract away from what happens next, but at some point the propagation of the queries returns to this node with asking whether the node can accept some data item. We answer this last question with *False*, because otherwise we allow this circuit to synchronize two sinks on the same node: the one through which we exit the node and the one through which we re-enter the node. For this reason, the channels of the exit and re-entry sinks cannot offer anything to this node in a circuit containing this kind of a loop.

In Figure 15 we show a circuit with a loop of this type. This example contains a `Sync`, a `SyncSpout`, two nodes, and an application component instance that performs a *take* operation on node 1. In this example, we expect that the *take* operation on node 1 will never succeed to retrieve any data, because the computation of the traffic on node 1 results in a source-sink-sink loop (loop 4), which tries to synchronize two sinks on the same node. In Figure 15 we also show the computation path for the *take* operation.

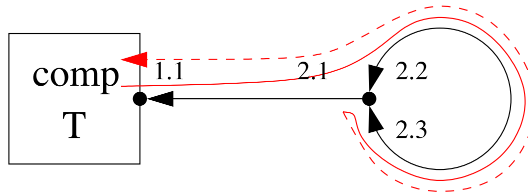


Figure 15: A source-sink-sink loop (loop 4)

7.2.5 Sink-source-source loops (loop 5)

In this type of loops, shown in Figure 10, we enter the node asking whether it can accept some data item. This results in the node asking whether some coincident channel (through its source) accepts the data item. We abstract away from what happens next, but at some point the propagation of queries returns to this node asking what the node has to offer. Since we have already visited the node testing its acceptance of d, we answer that the node offers d.

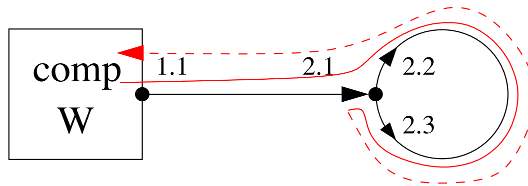


Figure 16: A sink-source-source loop (loop 5)

In Figure 16 we show a circuit with a loop of this type. This example contains a `Sync`, a `SyncDrain`, two nodes, and an application component instance that performs a *write* operation on node 1. In this example, we expect that the write operation on node 1 succeeds, because the computation of the traffic on node 1 results in a sink-source-source loop (loop 5). In Figure 16 we also show the computational path for the *take* operation.

7.2.6 Sink-source-sink loops (loop 6)

In this type loops, shown in Figure 10, we enter the node asking whether it can accept some data item. This results in the node asking whether some channel (through its source) accepts the data item. We abstract away from what happens next, but at some point the propagation of queries returns to this node asking whether the node can accept some data item. We answer this last question with *False*, because otherwise we allow this circuit to synchronize two sinks on the same node: the one through which we enter the node and the one through which we re-enter the node. For this reason, the channels of the entry and re-entry sinks can never offer anything to this node in a circuit containing this kind of a loop.

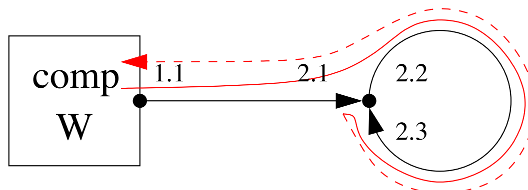


Figure 17: A sink-source-sink loop (loop 6)

In Figure 17 we show a circuit with a loop of this type. This example contains two `Sync` channels, two nodes and an application component instance that performs a *write* operation on node 1. In this example, we expect that the *write* operation on node 1 always fails, because the computation of the traffic on node 1 results in a sink-source-sink loop (loop 6), which tries to synchronize two sinks on the same node. In Figure 17 we also show the computation path for the *write* operation.

7.2.7 Sink-sink-source loops (loop 7)

In this type of loops, shown in Figure 10, we enter a node asking whether it can accept some data item. According to the *Accepts* definition (see §4.2.1) this results in the node asking some channel (through its source) whether it can accept the data item. This means that the only way we can exit the node is via a source; exiting via a sink does not constitute a valid node behavior. Therefore, a sink-sink-source loop cannot occur.

Because by definition a loop of this type violates node behavior we cannot give an example.

7.2.8 Sink-sink-sink loops (loop 8)

This loop, shown in Figure 10, cannot occur because it violates the rules for node behavior in the same way as a sink-sink-source loop (loop 7) does.

Because by definition a loop of this type violates node behavior, we cannot give an example.

8 Conclusions

In this report we presented a basic computational model for the Reo coordination language. We provide a functional specification for node behavior and channel behavior. The specification covers only the "look up" phase of computing a Reo circuit. We leave out the implementation of the computational model for "committing" the results in nodes and channels, and performing the actual transportation of data. Furthermore, we do not supply a locking scheme to manage multiple authorities that attempt to compute the same Reo circuit. We also do not supply a scheme to support partial fault tolerance. These two issues constitute future work that extends our proposed computational model.

The model presented in this paper has a major limitation - it can properly compute only a subset of all possible Reo circuits. The main problem comes from the way this model handles merging in *Offers* as described in Section 5.1.2. When selecting a data element the merger should "suppress" data from the other channels (i.e., exclude the behavior supplying that data), which the formulae does not do properly. Suppressing requires propagation of the exclusion constraint through channels and nodes. This problem manifests itself in circuits which contain, for example, a *LossySync* channel. In such circuits, improper propagation of an exclusion constraint through the sink channel end into a *LossySync* channel does not allow this channel to switch to its alternative behavior: to accept and lose the data from its source end.

Extending the model with propagation of exclusion constraints is possible, but considerably complicates our functional specification. In order to develop a more complete model, Costa et al. adopt a different approach to modeling circuits, called Connector Coloring [3]. This approach, among other things, abstracts away from the type of flow (inbound or outbound) in order to simplify the representation of different channel behavior.

From our experience with the presented approach, we also observe that in most practical examples, a circuit typically passes a lot of data between two dynamic reconfigurations, i.e., most of the time circuits remain static relative to the data flow through them. This calls for a different approach that instead of computing the complete circuit behavior every time when components try use the connector, allows to determine all possible behavior alternatives once and then only select the relevant alternative when data needs to flow. In such approach, circuit reconfiguration triggers a new behavior computation.

Acknowledgements

We thank David Clarke for his comments on the drafts of this report. Furthermore, we extend our gratitude to all other members of SEN3, who have contributed in one way or another to the development of the computational model of Reo.

A Examples of traces of loops

In this appendix we provide the full traces of the evaluation of the traffic for some of the loop examples given in §7.2. In the traffic computation (τ) we first evaluate *Offers* in order to determine the data items offered at the node, so that we can then check them for acceptance.

A.1 Trace of a source-source-source loop (loop 1)

This trace corresponds to the example in §7.2.1 and the circuit in Figure 11. In this example the application component instance performs a *write* operation on node 3 with data item 661. Below, we provide a full trace of the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned}\tau(1,\ *,\ [],\ []) = [(d_1, E'') \ \mid\ & (d_1, E') \leftarrow \mathit{Offers}(1, *, [], []),\\ & (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}].\end{aligned}$$

Node 1 offers what the channel of its sink has to offer:

$$\begin{aligned}\mathit{Offers}(1, *, [], []) &= \bigcup [\ \mathit{offers}(x, *, [] \cup [(1, \mathtt{nyd})], []) \mid x \leftarrow \mathit{snkCE}_{sel}(1)]\\ &= \bigcup [\ \mathit{offers}(x, *, [(1, \mathtt{nyd})], []) \mid x \leftarrow [1.1]]\\ &= \mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []).\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []) = \tau(2, *, [(1, \mathtt{nyd})], [2.1]).$$

We now evaluate the traffic at node 2:

$$\begin{aligned}\tau(2, *, [(1, \mathtt{nyd})], [2.1]) = [(d_2, E'') \ \mid\ & (d_2, E') \leftarrow \mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1]),\\ & (b_2, E'') \leftarrow \mathit{Accepts}(2, d_2, S', E'),\\ & b_2 == \mathit{True}].\end{aligned}$$

We evaluate what node 2 has to offer. It has only one sink 2.2. So:

$$\begin{aligned}&\mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1])\\ &\quad = \bigcup [\mathit{offers}(x, *, [(1, \mathtt{nyd})] \cup [(2, \mathtt{nyd})], [2.1]) \mid x \leftarrow \mathit{snkCE}_{sel}(2)]\\ &\quad = \bigcup [\mathit{offers}(x, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) \mid x \leftarrow [2.2]]\\ &\quad = \mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]).\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) = \tau(3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1, 3.1]).$$

We now evaluate the traffic at node 3. First we evaluate what the node offers and which of the offered data items the node accepts.

$$\begin{aligned}&\tau(3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1, 3.1])\\ &\quad = [(d_3, E'') \ \mid\ (d_3, E') \leftarrow \mathit{Offers}(3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1, 3.1]),\\ &\qquad\qquad (b_3, E'') \leftarrow \mathit{Accepts}(3, d_3, S', E'), b_3 == \mathit{True}]\end{aligned}$$

We evaluate what node 3 has to offer. Node 3 has only one source and therefore it offers the data item 661 of its pending write operation (661 matches *):

$$\begin{aligned}&\mathit{Offers}(3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1, 3.1])\\ &\quad = [(d, [2.1, 3.1]) \mid d \leftarrow W(3, *)] = [(661, [2.1, 3.1])].\end{aligned}$$

We now return to continue evaluating the traffic at node 3:

$$\begin{aligned}
&\tau(3, *, [(1, \texttt{nyd}), (2, \texttt{nyd})], [2.1, 3.1]) \\
&\quad = [(d_3, E'') \mid (d_3, E') \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_3, E'') \leftarrow \mathit{Accepts}(3, d_3, S', E'), b_3 == \mathit{True}] \\
&\quad = [(661, E'') \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_3, E'') \leftarrow \mathit{Accepts}(3, 661, [(1, 661), (2, 661)], [2.1, 3.1]), \\
&\qquad\qquad b_3 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 3 accepts 661. It does, because

$$\mathit{cel} = \mathit{srcCE}_{sel}(3)\backslash[2.1, 3.1] = [3.1]\backslash[2.1, 3.1] = []$$

and therefore

$$\mathit{Accepts}(3, 661, [(1, 661), (2, 661)], [2.1, 3.1]) = (\mathit{True}, [2.1, 3.1]).$$

We now return to finish evaluating the traffic at node 3:

$$\begin{aligned}
&\tau(3, *, [(1, \texttt{nyd}), (2, \texttt{nyd})], [2.1, 3.1]) \\
&\quad = [(661, E'') \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_3, E'') \leftarrow (\mathit{True}, [2.1, 3.1]), b_3 == \mathit{True}] \\
&\quad = [(661, [2.1, 3.1]) \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (\mathit{True}, [2.1, 3.1]) \leftarrow (\mathit{True}, [2.1, 3.1]), \\
&\qquad\qquad \mathit{True} == \mathit{True}] \\
&\quad = [(661, [2.1, 3.1]).]
\end{aligned}$$

We now return to evaluate what the channel of 2.2 offers:

$$\mathit{offers}(2.2, *, [(1, \texttt{nyd}), (2, \texttt{nyd})], [2.1]) = [(661, [2.1, 3.1])].$$

We now return to finish evaluating what node 2 has to offer:

$$\mathit{Offers}(2, *, [(1, \texttt{nyd})], [2.1]) = [(661, [2.1, 3.1])].$$

We now return to continue evaluating the traffic at node 2:

$$\begin{aligned}
&\tau(2, *, [(1, \texttt{nyd})], [2.1]) \\
&\quad = [(d_2, E'') \mid (d_2, E') \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_2, E'') \leftarrow \mathit{Accepts}(2, d_2, S', E'), b_2 == \mathit{True}] \\
&\quad = [(661, E'') \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_2, E'') \leftarrow \mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1]), \\
&\qquad\qquad b_2 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 2 accepts 661:

$$\begin{aligned}
&\mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1]) \\
&\qquad = (\mathit{bool}, E') \bowtie \mathit{Accepts}(2, 661, [(1, 661)], E' \cup [\mathit{head}\ \mathit{cel}])
\end{aligned}$$

where

$$\begin{aligned}
\mathit{cel} &= \mathit{srcCE}_{sel}(2)\backslash[2.1, 3.1] = [2.1, 2.3, 2.4]\backslash[2.1, 3.1] = [2.3, 2.4] \\
(\mathit{bool}, E') &= \mathit{accepts}(\mathit{head}\ [2.3, 2.4], 661, [(1, 661)] \cup [(2, 661)], []) \\
&= \mathit{accepts}(2.3, 661, [(1, 661), (2, 661)], [2.1, 3.1]).
\end{aligned}$$

We evaluate whether the channel of 2.3 accepts 661. We use the *accepts* definition of the `SyncDrain` channel given in §6.2.2:

$$
\begin{aligned}
&\mathit{accepts}(2.3, 661, [(1, 661), (2, 661)], [2.1, 3.1]) \\
&\qquad = (\tau(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]) \neq [], \\
&\qquad\quad \oplus\ \tau(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]).
\end{aligned}
$$

We now evaluate the traffic at the visited node 2. Visited nodes traffic whatever they offer:

$$
\begin{aligned}
&\tau(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]) \\
&\qquad = \mathit{Offers}(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]).
\end{aligned}
$$

We now evaluate what the visited node 2 offers. A visited node offers data items matching the required pattern, which come from the node's pair in the visited nodes list:

$$
\begin{aligned}
&\mathit{Offers}(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]) \\
&\qquad = [(d, [2.1, 3.1, 2.4]) \mid (n1, d) \leftarrow [(1, 661), (2, 661)], n1 == 2 \wedge d \ni *] \\
&\qquad = [(661, [2.1, 3.1, 2.4])].
\end{aligned}
$$

We now return to finish evaluating the traffic at the visited node 2:

$$
\tau(2, *, [(1, 661), (2, 661)], [2.1, 3.1, 2.4]) = [(661, [2.1, 3.1, 2.4])].
$$

We now return to finish evaluating whether the channel of 2.3 accepts 661:

$$
\begin{aligned}
&\mathit{accepts}(2.3, 661, [(1, 661), (2, 661)], [2.1, 3.1]) \\
&\qquad = ([(661, [2.1, 3.1, 2.4])] \neq [], \oplus [(661, [2.1, 3.1, 2.4])] \\
&\qquad = (\mathit{True},\ [2.1, 3.1, 2.4]).
\end{aligned}
$$

We can now continue to evaluate whether node 2 accepts 661:

$$
\begin{aligned}
&\mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1]) \\
&\qquad = (\mathit{True}, [2.1, 3.1, 2.4]) \bowtie \mathit{Accepts}(2, 661, [(1, 661)], \\
&\qquad\qquad [2.1, 3.1, 2.4] \cup [\mathit{head}\ [2.3, 2.4]]) \\
&\qquad = (\mathit{True}, [2.1, 3.1, 2.4]) \bowtie \mathit{Accepts}(2, 661, [(1, 661)], \\
&\qquad\qquad [2.1, 3.1, 2.4] \cup [2.3]) \\
&\qquad = (\mathit{True}, [2.1, 3.1, 2.4]) \bowtie \mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1, 2.4, 2.3]).
\end{aligned}
$$

We now evaluate the recursive call to Accepts for node 2. Since

$$
\mathit{cel} = \mathit{srcCE}_{sel}(2) \backslash [2.1, 3.1, 2.4, 2.3] = [2.1, 2.4, 2.3] \backslash [2.1, 3.1, 2.4, 2.3] = []
$$

therefore

$$
\mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1, 2.4, 2.3]) = (\mathit{True}, [2.1, 3.1, 2.4, 2.3]).
$$

We now return to finish evaluating whether node 2 accepts 661 (we ignore any duplicates that $\bowtie$ produces because of its implementation with $+\!\!+$ instead of $\cup$):

$$
\begin{aligned}
&\mathit{Accepts}(2, 661, [(1, 661)], [2.1, 3.1]) \\
&\qquad = (\mathit{True},\ [2.1, 3.1, 2.4]) \bowtie (\mathit{True}, [2.1, 3.1, 2.4, 2.3]) \\
&\qquad = (\mathit{True},\ [2.1, 3.1, 2.4, 2.3]).
\end{aligned}
$$

We now return to finish evaluating the traffic at node 2:

$$
\begin{aligned}
&\tau(2, *, [(1, \mathtt{nyd})], [2.1]) \\
&\qquad = [(661, E'') \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (b_2, E'') \leftarrow (\mathit{True},\ [2.1, 3.1, 2.4, 2.3]), b_2 == \mathit{True}] \\
&\qquad = [(661, [2.1, 3.1, 2.4, 2.3]) \mid (661, [2.1, 3.1]) \leftarrow [(661, [2.1, 3.1])], \\
&\qquad\qquad (\mathit{True}, [2.1, 3.1, 2.4, 2.3]) \\
&\qquad\qquad\quad \leftarrow (\mathit{True},\ [2.1, 3.1, 2.4, 2.3]), \\
&\qquad\qquad \mathit{True} == \mathit{True}] \\
&\qquad = [(661, [2.1, 3.1, 2.4, 2.3])].
\end{aligned}
$$

We now return back to finish evanuating what the channel of 1.1 has to offer:

$$\mathit{offers}(1.1, *, [(1, \texttt{nyd})], []) = [(661, [2.1, 3.1, 2.4, 2.3])].$$

We now return back to finish evaluating what node 1 offers:

$$\mathit{Offers}(1, *, [], []) = [(661, [2.1, 3.1, 2.4, 2.3])].$$

We now continue evaluating the traffic of node 1:

$$\begin{aligned}
&\tau(1, *, [],\ [])\\
&\quad = [(d_1, E'') \mid\ (d_1, E') \leftarrow [(661, [2.1, 3.1, 2.4, 2.3])],\\
&\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}]\\
&\quad = [(661, E'') \mid\ (661, [(661, [2.1, 3.1, 2.4, 2.3])]) \leftarrow [(661, [2.1, 3.1, 2.4, 2.3])],\\
&\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, 661, [], [(661, [2.1, 3.1, 2.4, 2.3])]), b_1 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 1 accepts 661. Node 1 contains only one sink and it has a *take* pending with pattern *:

$$\begin{aligned}
&\mathit{Accepts}(1, 661, [], [(661, [2.1, 3.1, 2.4, 2.3])])\\
&\qquad = (P(1)\ \neq\ []\ \wedge\ \bigwedge [991 \ni p \mid\ p \leftarrow P(1)],\ [2.1, 3.1, 2.4, 2.3])\\
&\qquad = (\mathit{True}\ \wedge\ 991 \ni *, [2.1, 3.1, 2.4, 2.3])\\
&\qquad = (\mathit{True}, [2.1, 3.1, 2.4, 2.3]).
\end{aligned}$$

We now return to finish evaluating the traffic of node 1:

$$\begin{aligned}
&\tau(1, *, [],\ [])\\
&\qquad = [(661, E'') \mid\ (661, [(661, [2.1, 3.1, 2.4, 2.3])]) \leftarrow\\
&\qquad\qquad\qquad\qquad [(661, [2.1, 3.1, 2.4, 2.3])],\\
&\qquad\quad (b_1, E'') \leftarrow (\mathit{True}, [2.1, 3.1, 2.4, 2.3]), b_1 == \mathit{True}]\\
&\qquad = [(661, [2.1, 3.1, 2.4, 2.3]) \mid\ (661, [(661, [2.1, 3.1, 2.4, 2.3])]) \leftarrow\\
&\qquad\qquad\qquad\qquad [(661, [2.1, 3.1, 2.4, 2.3])],\\
&\qquad\quad (\mathit{True}, [2.1, 3.1, 2.4, 2.3]) \leftarrow (\mathit{True}, [2.1, 3.1, 2.4, 2.3]),\\
&\qquad\quad \mathit{True} == \mathit{True}]\\
&\qquad = [(661, [2.1, 3.1, 2.4, 2.3])].
\end{aligned}$$

Because this traffic list τ is not empty, the components *comp1* and *comp2* in Figure 11 succesfully perform, respectively, a *write* and a *take* operation.

A.2 Trace of a source-sink-source loop (loop 3.1)

This trace corresponds to the example in §7.2.3 and the circuit in Figure 13. Below, we provide a full trace of the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned}
\tau(1,\ *,\ [],\ []) = [(d_1, E'')\ \mid\ &(d_1, E') \leftarrow \mathit{Offers}(1, *, [], []),\\
&(b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}].
\end{aligned}$$

Node 1 offers what the channel of its sink has to offer:

$$\begin{aligned}
\mathit{Offers}(1, *, [], []) &= \bigcup [\ \mathit{offers}(x, *, [] \cup [(1, \texttt{nyd})], []) \mid x \leftarrow \mathit{snkCE}_{sel}(1)]\\
&= \bigcup [\ \mathit{offers}(x, *, [(1, \texttt{nyd})], []) \mid x \leftarrow [1.1]]\\
&= \mathit{offers}(1.1, *, [(1, \texttt{nyd})], []).
\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(1.1, *, [(1, \texttt{nyd})], []) = \tau(2, *, [(1, \texttt{nyd})], [2.1]).$$

We now evaluate the traffic at node 2. In the same way as node 1, we first evaluate the data items the node offers and then test for their acceptance:

$$\begin{aligned}\tau(2,*,[(1,\texttt{nyd})],[2.1]) = [(d_2,E'') \mid\ & (d_2,E') \leftarrow \mathit{Offers}(2,*,[(1,\texttt{nyd})],[2.1]),\\ & (b_2,E'') \leftarrow \mathit{Accepts}(2,d_2,S',E'),\\ & b_2 == \mathit{True}].\end{aligned}$$

Node 2 offers what the channel of its sink has to offer:

$$\begin{aligned}&\mathit{Offers}(2,*,[(1,\texttt{nyd})],[2.1])\\ &\quad= \bigcup[\mathit{offers}(x,*,[(1,\texttt{nyd})] \cup [(2,\texttt{nyd})],[2.1]) \mid x \leftarrow \mathit{snkCE}_{sel}(2)]\\ &\quad= \bigcup[\mathit{offers}(x,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1]) \mid x \leftarrow [2.2]]\\ &\quad= \mathit{offers}(2.2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1]).\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(2.2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1]) = \tau(2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1,2.3]).$$

We have already visited node 2. A visited node traffics what it offers:

$$\tau(2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1,2.3]) = \mathit{Offers}(2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1,2.3]).$$

A visited node offers any data item that we visited it with. Node 2 appears in the visited nodes list with a `nyd` data item, which means that the node cannot offer anything (the `nyd` does not match any valid pattern including *). So:

$$\begin{aligned}&\mathit{Offers}(2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1,2.3])\\ &\quad= [(d,E) \mid (n1,d) \leftarrow [(1,\texttt{nyd}),(2,\texttt{nyd})], n1 == 2 \wedge d \ni *]\\ &\quad= [].\end{aligned}$$

We now return to finish evaluating what the visited node 2 traffics:

$$\tau(2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1,2.3]) = [].$$

We now return to finish evaluating what the `Sync` offers:

$$\mathit{offers}(2.2,*,[(1,\texttt{nyd}),(2,\texttt{nyd})],[2.1]) = [].$$

We now return to finish evaluating what node 2 offers:

$$\mathit{Offers}(2,*,[(1,\texttt{nyd})],[2.1]) = [].$$

We now return to finish evaluating the traffic at node 2:

$$\begin{aligned}&\tau(2,*,[(1,\texttt{nyd})],[2.1])\\ &\quad= [[] \mid [] \leftarrow [], (b_2,E'') \leftarrow \mathit{Accepts}(2,d_2,S',E'), b_2 == \mathit{True}]\\ &\quad= [].\end{aligned}$$

We now return to finish evaluating the *offers* of the channel of 1.1:

$$\mathit{offers}(1.1,*,[(1,\texttt{nyd})],[]) = [].$$

We now return to finish evaluating what node 1 offers:

$$\mathit{Offers}(1,*,[],[]) = [].$$

We now return to finish evaluating the traffic at node 1:

$$\begin{aligned}\tau(1,\ *,\ [],\ []) &= [[] \mid [] \leftarrow [], (b_1,E'') \leftarrow \mathit{Accepts}(1,d_1,S',E'), b_1 == \mathit{True}]\\ &= [].\end{aligned}$$

Because this traffic list τ is empty, the component *comp* in Figure 13 cannot perform the *take* operation (i.e., it remains pending).

A.3 Trace of a source-sink-source loop (loop 3.2)

This trace corresponds to the example in §7.2.3 and the circuit in Figure 14. Below, we provide a full trace of the the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned}\tau(1,\ *,\ [],\ []) = [(d_1, E'') \mid\ & (d_1, E') \leftarrow \mathit{Offers}(1, *, [], []),\\ & (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}].\end{aligned}$$

Node 1 offers what the channel of its sink has to offer:

$$\begin{aligned}\mathit{Offers}(1, *, [], []) &= \bigcup [\ \mathit{offers}(x, *, [] \cup [(1, \mathtt{nyd})], []) \mid x \leftarrow \mathit{snkCE}_{sel}(1)]\\ &= \bigcup [\ \mathit{offers}(x, *, [(1, \mathtt{nyd})], []) \mid x \leftarrow [1.1]]\\ &= \mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []).\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []) = \tau(2, *, [(1, \mathtt{nyd})], [2.1]).$$

We now evaluate the traffic at node 2:

$$\begin{aligned}\tau(2, *, [(1, \mathtt{nyd})], [2.1]) = [(d_2, E'') \mid\ & (d_2, E') \leftarrow \mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1]),\\ & (b_2, E'') \leftarrow \mathit{Accepts}(2, d_2, S', E'),\\ & b_2 == \mathit{True}].\end{aligned}$$

Node 2 offers what the channel of its sink has to offer:

$$\begin{aligned}&\mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1])\\ &\quad= \bigcup[\mathit{offers}(x, *, [(1, \mathtt{nyd})] \cup [(2, \mathtt{nyd})], [2.1]) \mid x \leftarrow \mathit{snkCE}_{sel}(2)]\\ &\quad= \bigcup[\mathit{offers}(x, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) \mid x \leftarrow [2.2]]\\ &\quad= \mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]).\end{aligned}$$

We use the *offers* definition of the `SyncSpout` channel given in §6.2.3. We assume that its random generator produces 991 and 992:

$$\begin{aligned}&\mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1])\\ &\quad= [\ (991, E')\ \mid 991 \leftarrow \mathit{random}() \wedge 991 \ni *, 992 \leftarrow \mathit{random}(),\\ &\qquad\qquad (b, E') \leftarrow \mathit{Accepts}(3, 992, [(1, 991), (2, 991)], [2.1]),\\ &\qquad\qquad b == \mathit{True}].\end{aligned}$$

We evaluate whether node 3 accepts 992:

$$\begin{aligned}&\mathit{Accepts}(3, 992, [(1, 991), (2, 991)], [2.1])\\ &\quad= (\mathit{bool}, E') \bowtie \mathit{Accepts}(3, 992, [(1, 991), (2, 991)], E' \cup [\mathit{head}\ \ \mathit{cel}])\end{aligned}$$

where

$$\begin{aligned}\mathit{cel} &= \mathit{srcCE}_{sel}(3)\backslash[2.1] = [3.2]\backslash[2.1] = [3.2]\\ (\mathit{bool}, E') &= \mathit{accepts}(\mathit{head}\ [3.2], 992, [(1, 991), (2, 991)] \cup [(3, 992)], [2.1])\\ &= \mathit{accepts}(3.2, 992, [(1, 991), (2, 991), (3, 992)], [2.1]).\end{aligned}$$

We use the *accepts* definition of `SyncDrain` given in §6.2.2:

$$\begin{aligned}&\mathit{accepts}(3.2, 992, [(1, 991), (2, 991), (3, 992)], [2.1])\\ &\quad= (\tau(2, *, [(1, 991), (2, 991), (3, 992)], [2.1, 2.3])\ \neq\ [],\\ &\quad\ \oplus\ \tau(2, *, [(1, 991), (2, 991), (3, 992)], [2.1, 2.3])).\end{aligned}$$

We evaluate the traffic at the visited node 2:

$$\tau(2, *, [(1,991),(2,991),(3,992)], [2.1, 2.3]) \\ \quad = \mathit{Offers}(2, *, [(1,991),(2,991),(3,992)], [2.1, 2.3]).$$

The visited node 2 offers the data item that it has in a pair in the visited nodes list. This data item should also match the pattern *:

$$\mathit{Offers}(2, *, [(1,991),(2,991),(3,992)], [2.1, 2.3]) \\ \quad = [(d, [2.1, 2.3]) \mid (n1, d) \leftarrow [(1,991),(2,991),(3,992)], n1 == 2 \wedge d \ni *] \\ \quad = [(991, [2.1, 2.3])].$$

We now return to finish evaluating what the visited node 2 can traffic:

$$\tau(2, *, [(1,991),(2,991),(3,992)], [2.1, 2.3]) = [(991, [2.1, 2.3])].$$

We now return to finish evaluating whether the `SyncDrain` of 3.2 accepts 992:

$$\mathit{accepts}(3.2, 992, [(1,991),(2,991),(3,992)], [2.1]) \\ \quad = ([(991, [2.1, 2.3])] \neq [], \oplus [(991, [2.1, 2.3])]) \\ \quad = (\mathit{True}, [2.1, 2.3]).$$

We now return to continue evaluating whether node 3 accepts 992:

$$\mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1]) \\ \quad = (\mathit{bool}, E') \bowtie \mathit{Accepts}(3, 992, [(1,991),(2,991)], E' \cup [\mathit{head}\ \mathit{cel}]) \\ \quad = (\mathit{True}, [2.1, 2.3]) \bowtie \\ \qquad \mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1, 2.3] \cup [3.2]) \\ \quad = (\mathit{True}, [2.1, 2.3]) \bowtie \mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1, 2.3, 3.2]).$$

We now evaluate the recursive call to *Accepts* for node 3. Since

$$\mathit{cel} = \mathit{srcCE}_{sel}(3) \backslash [2.1, 2.3, 3.2] = [3.2] \backslash [2.1, 2.3, 3.2] = []$$

therefore

$$\mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1, 2.3, 3.2]) = (\mathit{True}, [2.1, 2.3, 3.2]).$$

We now finish evaluating whether node 3 accepts 992:

$$\mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1]) \\ \quad = (\mathit{True}, [2.1, 2.3]) \bowtie \\ \qquad \mathit{Accepts}(3, 992, [(1,991),(2,991)], [2.1, 2.3, 3.2]) \\ \quad = (\mathit{True}, [2.1, 2.3]) \bowtie (\mathit{True}, [2.1, 2.3, 3.2]) \\ \quad = (\mathit{True}, [2.1, 2.3, 3.2]).$$

We now return to evaluate what the `SyncSpout` of 2.2 has to offer:

$$\mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) \\ \quad = [\ (991, E') \mid 991 \leftarrow \mathit{random}() \wedge 991 \ni *, 992 \leftarrow \mathit{random}(), \\ \qquad (b, E') \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), b == \mathit{True}] \\ \quad = [(991, [2.1, 2.3, 3.2]) \mid 991 \leftarrow \mathit{random}() \wedge \mathit{True}, 992 \leftarrow \mathit{random}(), \\ \qquad (\mathit{True}, [2.1, 2.3, 3.2]) \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), \\ \qquad \mathit{True} == \mathit{True}] \\ \quad = [(991, [2.1, 2.3, 3.2])].$$

We now return to finish evaluating what node 2 has to offer:

$$\mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1]) = [(991, [2.1, 2.3, 3.2])].$$

We now return to continue evaluating the traffic at node 2:

$$\begin{aligned}
&\tau(2, *, [(1, \texttt{nyd})], [2.1]) \\
&\qquad = [(d_2, E'') \mid (d_2, E') \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_2, E'') \leftarrow \mathit{Accepts}(2, d_2, S', E'), b_2 == \mathit{True}] \\
&\qquad = [(991, E'') \mid (991, [2.1, 2.3, 3.2]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_2, E'') \leftarrow \mathit{Accepts}(2, 991, [(1, 991)], [2.1, 2.3, 3.2]), \\
&\qquad\qquad b_2 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 2 accepts 991. It does, because

$$\mathit{cel} = \mathit{srcCE}_{sel}(2) \backslash [2.1, 2.3, 3.2] = [2.1, 2.3] \backslash [2.1, 2.3, 3.2] = []$$

and therefore

$$\mathit{Accepts}(2, 991, [(1, 991)], [2.1, 2.3]) = (\mathit{True}, [2.1, 2.3, 3.2]).$$

We now return to finish evaluating the traffic at node 2:

$$\begin{aligned}
&\tau(2, *, [(1, \texttt{nyd})], [2.1]) \\
&\qquad = [(991, E'') \mid (991, [2.1, 2.3]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_2, E'') \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), b_2 == \mathit{True}] \\
&\qquad = [(991, [2.1, 2.3, 3.2]) \mid (991, [2.1, 2.3, 3.2]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (\mathit{True}, [2.1, 2.3, 3.2]) \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), \\
&\qquad\qquad \mathit{True} == \mathit{True}] \\
&\qquad = [(991, [2.1, 2.3, 3.2])].
\end{aligned}$$

We now return to finish evaluating what the `Sync` of 1.1 has to offer:

$$\mathit{offers}(1.1, *, [(1, \texttt{nyd})], []) = [(991, [2.1, 2.3, 3.2])].$$

We now return to finish evaluating what node 1 has to offer:

$$\mathit{Offers}(1, *, [], []) = [(991, [2.1, 2.3, 3.2])].$$

We now return to continue evaluating the traffic at node 1:

$$\begin{aligned}
&\tau(1, *, [], []) \\
&\qquad = [(d_1, E'') \mid (d_1, E') \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}] \\
&\qquad = [(991, E'') \mid ((991, [2.1, 2.3, 3.2]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, 991, [], [2.1, 2.3, 3.2]), \\
&\qquad\qquad b_1 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 1 accepts 991. Node 1 contains only one sink and it has a take pending with pattern *:

$$\begin{aligned}
&\mathit{Accepts}(1, 991, [], [2.1, 2.3, 3.2]) \\
&\qquad = (P(1) \neq [] \wedge \bigwedge [991 \ni p \mid p \leftarrow P(1)],\ [2.1, 2.3, 3.2]) \\
&\qquad = (\mathit{True} \wedge 991 \ni *, [2.1, 2.3, 3.2]) \\
&\qquad = (\mathit{True}, [2.1, 2.3, 3.2]).
\end{aligned}$$

We now return to finish evaluating the traffic at node 1:

$$\begin{aligned}
&\tau(1, *, [], []) \\
&\qquad = [(991, E'') \mid ((991, [2.1, 2.3]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (b_1, E'') \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), b_1 == \mathit{True}] \\
&\qquad = [(991, [2.1, 2.3, 3.2]) \mid ((991, [2.1, 2.3, 3.2]) \leftarrow [(991, [2.1, 2.3, 3.2])], \\
&\qquad\qquad (\mathit{True}, [2.1, 2.3, 3.2]) \leftarrow (\mathit{True}, [2.1, 2.3, 3.2]), \\
&\qquad\qquad \mathit{True} == \mathit{True}] \\
&\qquad = [(991, [2.1, 2.3, 3.2])].
\end{aligned}$$

Because this traffic list τ is not empty, the component *comp* in Figure 14 successfully performs the *take* operation.

A.4 Trace of a source-sink-sink loop (loop 4)

This trace corresponds to the example in §7.2.4 and the circuit in Figure 15. Below, we provide a full trace of the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned}\tau(1,\ *,\ [],\ []) = [(d_1, E'') \mid\ & (d_1, E') \leftarrow \mathit{Offers}(1, *, [], []),\\ & (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}].\end{aligned}$$

Node 1 offers what the channel of its sink has to offer:

$$\begin{aligned}\mathit{Offers}(1, *, [], []) &= \bigcup [\ \mathit{offers}(x, *, [] \cup [(1, \mathtt{nyd})], []) \mid x \leftarrow \mathit{snkCE}_{sel}(1)]\\ &= \bigcup [\ \mathit{offers}(x, *, [(1, \mathtt{nyd})], []) \mid x \leftarrow [1.1]]\\ &= \mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []).\end{aligned}$$

We use the *offers* definition of the `Sync` channel given in §6.2.1:

$$\mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []) = \tau(2, *, [(1, \mathtt{nyd})], [2.1]).$$

We now evaluate the traffic at node 2:

$$\begin{aligned}\tau(2, *, [(1, \mathtt{nyd})], [2.1]) = [(d_2, E'') \mid\ & (d_2, E') \leftarrow \mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1]),\\ & (b_2, E'') \leftarrow \mathit{Accepts}(2, d_2, S', E'),\\ & b_2 == \mathit{True}].\end{aligned}$$

Node 2 offers what the channels of its two sinks have to offer:

$$\begin{aligned}&\mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1])\\ &\quad = \bigcup [\mathit{offers}(x, *, [(1, \mathtt{nyd})] \cup [(2, \mathtt{nyd})], [2.1]) \mid x \leftarrow \mathit{snkCE}_{sel}(2)]\\ &\quad = \bigcup [\mathit{offers}(x, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) \mid x \leftarrow [2.2, 2.3]]\\ &\quad = \mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1])\\ &\quad\quad \cup\ \mathit{offers}(2.3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]).\end{aligned}$$

We first evaluate what the channel of 2.2 has to offer. We use the *offers* definition of the `SyncSpout` channel given in §6.2.3. We assume that its random generator produces 881 and 822:

$$\begin{aligned}&\mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1])\\ &\quad = [\ (881, E') \mid 881 \leftarrow \mathit{random}() \wedge 881 \ni *, 882 \leftarrow \mathit{random}(),\\ &\quad\quad (b,\ E') \leftarrow \mathit{Accepts}(2,\ 882,\ [(1, 881), (2, 881)],\ [2.1]),\\ &\quad\quad b == \mathit{True}].\end{aligned}$$

We evaluate whether node 2 accepts 882. It does not, because we have already visited node 2:

$$\mathit{Accepts}(2,\ 882,\ [(1, 881), (2, 881)],\ [2.1]) = (\mathit{False}, [2.1]).$$

We now return to finish evaluating what the `SyncSpout` of 2.2 has to offer:

$$\begin{aligned}&\mathit{offers}(2.2, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1])\\ &\quad = [(881, E') \mid 881 \leftarrow \mathit{random}() \wedge 881 \ni *, 882 \leftarrow \mathit{random}(),\\ &\quad\quad (b,\ E') \leftarrow (\mathit{False}, [2.1]), b == \mathit{True}]\\ &\quad = [\ (881, [2.1]) \mid 881 \leftarrow \mathit{random}() \wedge 881 \ni *,\ 882 \leftarrow \mathit{random}(),\\ &\quad\quad (\mathit{False}, [2.1]),\ \mathit{False} == \mathit{True}]\\ &\quad = [].\end{aligned}$$

Similarly, the `SyncSpout` (in this example, the same channel) of 2.3 has nothing to offer, and thus

$$\mathit{offers}(2.3, *, [(1, \mathtt{nyd}), (2, \mathtt{nyd})], [2.1]) = [].$$

We now return to finish evaluating what node 2 has to offer:

$$\mathit{Offers}(2, *, [(1, \mathtt{nyd})], [2.1]) = [] \bigcup [] = [].$$

We now return to finish evaluating the traffic at node 2:

$$\tau(2, *, [(1, \mathtt{nyd})], [2.1]) = [].$$

We now return to finish evaluating the offers of the channel of 1.1:

$$\mathit{offers}(1.1, *, [(1, \mathtt{nyd})], []) = [].$$

We now return to finish evaluating what node 1 has to offer:

$$\mathit{Offers}(1, *, [], []) = [].$$

We now return to finish evaluating what node 1 can traffic:

$$\begin{aligned}\tau(1, *, [], []) &= [[] \mid [] \leftarrow [], (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}] \\ &= [].\end{aligned}$$

Because this traffic list τ is empty, the component *comp* in Figure 15 cannot perform the *take* operation (i.e., it remains pending).

A.5 Trace of a sink-source-source loop (loop 5)

This trace corresponds to the example in §7.2.5 and the circuit in Figure 16. In this example the application component instance performs a *write* operation on node 1 with data item 441. Below, we provide a full trace of the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned}\tau(1,\ *,\ [],\ []) = [(d_1, E'') \mid\ & (d_1, E') \leftarrow \mathit{Offers}(1, *, [], []), \\ & (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}].\end{aligned}$$

Node 1 has only one source, therefore it offers the data items of the *write* operation pending on the node:

$$\mathit{Offers}(1, *, [], []) = [(441, []) \mid 441 \leftarrow W(1, *)] = [(441, [])].$$

We now return to continue evaluating the traffic at node 1:

$$\begin{aligned}&\tau(1, *, [], []) \\ &\quad = [(d_1, E'') \mid (d_1, E') \leftarrow [(441, [])], \\ &\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}] \\ &\quad = [(441, E'') \mid (441, []) \leftarrow [(441, [])], \\ &\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, 441, [], []), b_1 == \mathit{True}].\end{aligned}$$

We now evaluate whether node 1 accepts 441:

$$\begin{aligned}&\mathit{Accepts}(1, 441, [], []) \\ &\qquad = (\mathit{bool}, E') \bowtie \mathit{Accepts}(1, 441, [], E' \cup [\mathit{head}\ \ \mathit{cel}])\end{aligned}$$

where

$$\begin{aligned}\mathit{cel} &= \mathit{srcCE}_{sel}(1)\backslash[] = [1.1]\backslash[] = [1.1] \\ (\mathit{bool}, E') &= \mathit{accepts}(\mathit{head}\ [1.1], 441, [] \cup [(1, 441)], [])\end{aligned}$$

$$= accepts(1.1, 441, [(1, 441)], []).$$

We use the *accepts* definition of the `Sync` channel given in §6.2.1:

$$accepts(1.1, 441, [(1, 441)], []) = Accepts(2, 441, [(1, 441)], []).$$

We now evaluate whether node 2 accepts 441:

$$\begin{aligned} & Accepts(2, 441, [(1, 441)], []) \\ & \quad = (bool, E') \bowtie Accepts(2, 441, [(1, 441)], E' \cup [head\ cel]) \end{aligned}$$

where

$$\begin{aligned} cel &= srcCE_{sel}(2) \backslash [] = [2.2, 2.3] \backslash [] = [2.2, 2.3] \\ (bool, E') &= accepts(head\ [2.2, 2.3], 441, [(1, 441)] \cup [(2, 441)], []) \\ &= accepts(2.2, 441, [(1, 441), (2, 441)], []). \end{aligned}$$

We use the *accepts* definition of the `SyncDrain` channel given in §6.2.2:

$$\begin{aligned} & accepts(2.2, 441, [(1, 441), (2, 441)], []) \\ & \quad = (\tau(2, *, [(1, 441), (2, 441)], [2.3]) \neq [], \\ & \quad \oplus\ \tau(2, *, [(1, 441), (2, 441)], [2.3]). \end{aligned}$$

We now evaluate the traffic at the visited node 2:

$$\tau(2, *, [(1, 441), (2, 441)], [2.3]) = Offers(2, *, [(1, 441), (2, 441)], [2.3]).$$

The visited node 2 offers the data item that in has in a pair in the visited nodes list. This data item also matches the pattern *:

$$\begin{aligned} & Offers(2, *, [(1, 441), (2, 441)], [2.3]) \\ & \quad = [(d, [2.3]) \mid (n1, d) \leftarrow [(1, 441), (2, 441)], n1 == 2 \wedge d \ni *] \\ & \quad = [(441, [2.3])]. \end{aligned}$$

We now return to finish evaluating the traffic at the visited node 2:

$$\tau(2, *, [(1, 441), (2, 441)], [2.3]) = [(441, [2.3])].$$

We now return to finish evaluating whether the channel of 2.2 accepts 441:

$$\begin{aligned} & accepts(2.2, 441, [(1, 441), (2, 441)], []) \\ & \quad = ([(441, [2.3])] \neq [], \oplus[(441, [2.3])] = (True, [2.3]). \end{aligned}$$

We now return to continue evaluating whether node 2 accepts 441:

$$\begin{aligned} & Accepts(2, 441, [(1, 441)], []) \\ & \quad = (True, [2.3]) \bowtie Accepts(2, 441, [(1, 441)], [2.3] \cup [head\ [2.2, 2.3]]) \\ & \quad = (True, [2.3]) \bowtie Accepts(2, 441, [(1, 441)], [2.3] \cup [2.2]) \\ & \quad = (True, [2.3]) \bowtie Accepts(2, 441, [(1, 441)], [2.3, 2.2]). \end{aligned}$$

We now evaluate the recursive call to *Accepts* for node 2. Since

$$cel = srcCE_{sel}(2) \backslash [] = [2.2, 2.3] \backslash [2.3, 2.2] = []$$

therefore

$$Accepts(2, 441, [(1, 441)], [2.3, 2.2]) = (True, [2.3, 2.2]).$$

We can now return to finish evaluating whether node 2 accepts 441:

$$\begin{aligned} &\mathit{Accepts}(2, 441, [(1,441)], [\,]) \\ &\qquad = (\mathit{True}, [2.3]) \bowtie (\mathit{True}, [2.3, 2.2]) \\ &\qquad = (\mathit{True}, [2.3, 2.2]). \end{aligned}$$

We now return to evaluate whether the channel of 1.1 accepts 441:

$$\mathit{accepts}(1.1, 441, [(1,441)], [\,]) = (\mathit{True}, [2.3, 2.2]).$$

We now return to continue evaluating whether node 1 accepts 441:

$$\begin{aligned} &\mathit{Accepts}(1, 441, [\,], [\,]) \\ &\qquad = (\mathit{True}, [2.3, 2.2]) \bowtie \mathit{Accepts}(1, 441, [\,], [2.3, 2.2] \cup [\mathit{head}\ [1.1]]) \\ &\qquad = (\mathit{True}, [2.3, 2.2]) \bowtie \mathit{Accepts}(1, 441, [\,], [2.3, 2.2] \cup [1.1]) \\ &\qquad = (\mathit{True}, [2.3, 2.2]) \bowtie \mathit{Accepts}(1, 441, [\,], [2.3, 2.2, 1.1]). \end{aligned}$$

We now evaluate the recursive call to *Accepts* for node 1. Since

$$\mathit{cel} = \mathit{srcCE}_{sel}(1) \backslash [2.3, 2.2, 1.1] = [1.1] \backslash [2.3, 2.2, 1.1] = [\,].$$

therefore

$$\mathit{Accepts}(1, 441, [\,], [2.3, 2.2, 1.1]) = (\mathit{True}, [2.3, 2.2, 1.1]).$$

We now return to finish evaluating whether node 1 accepts 441:

$$\begin{aligned} &\mathit{Accepts}(1, 441, [\,], [\,]) \\ &\qquad = (\mathit{True}, [2.3, 2.2]) \bowtie (\mathit{True}, [2.3, 2.2, 1.1]) \\ &\qquad = (\mathit{True}, [2.3, 2.2, 1.1]). \end{aligned}$$

We now return to finish evaluating the traffic at node 1:

$$\begin{aligned} &\tau(1, *, [\,], [\,]) \\ &\quad = [(441, E'') \mid (441, [\,]) \leftarrow [(441, [\,])], \\ &\qquad\qquad (b_1, E'') \leftarrow (\mathit{True}, [2.3, 2.2, 1.1]), b_1 == \mathit{True}] \\ &\quad = [(441, [2.3, 2.2, 1.1]) \mid (441, [\,]) \leftarrow [(441, [\,])], \\ &\qquad\qquad (\mathit{True}, [2.3, 2.2, 1.1]) \leftarrow (\mathit{True}, [2.3, 2.2, 1.1]), \\ &\qquad\qquad \mathit{True} == \mathit{True}] \\ &\quad = [(441, [2.3, 2.2, 1.1])]. \end{aligned}$$

Because this traffic list τ is not empty, the component *comp* in Figure 16 successfully performs the *write* operation.

A.6 Trace of a sink-source-sink loop (loop 6)

This trace corresponds to the example in §7.2.6 and the circuit in Figure 17. In this example the application component instance performs a *write* operation on node 1 with data item 331. Below, we provide a full trace of the evaluation of the traffic at node 1.

The traffic at node 1 is:

$$\begin{aligned} \tau(1,\ *,\ [\,],\ [\,]) = [(d_1, E'') \mid\ & (d_1, E') \leftarrow \mathit{Offers}(1, *, [\,], [\,]), \\ & (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}]. \end{aligned}$$

Node 1 only has sources, therefore it offers the data items from *write* operations pending on the node:

$$\mathit{Offers}(1, *, [\,], [\,]) = [(331, [\,]) \mid 331 \leftarrow W(1, *)] = [(331, [\,])].$$

We now return to continue evaluating the traffic at node 1:

$$\begin{aligned}
&\tau(1, *, [], []) \\
&\qquad = [(d_1, E'') \mid (d_1, E') \leftarrow [(331, [])], \\
&\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, d_1, S', E'), b_1 == \mathit{True}] \\
&\qquad = [(331, E'') \mid (331, []) \leftarrow [(331, [])], \\
&\qquad\qquad (b_1, E'') \leftarrow \mathit{Accepts}(1, 331, [], []), b_1 == \mathit{True}].
\end{aligned}$$

We now evaluate whether node 1 accepts 331:

$$\begin{aligned}
&\mathit{Accepts}(1, 331, [], []) \\
&\qquad = (\mathit{bool}, E') \bowtie \mathit{Accepts}(1, 331, [], E' \cup [\mathit{head}\ \mathit{cel}])
\end{aligned}$$

where

$$\begin{aligned}
\mathit{cel} &= \mathit{srcCE}_{sel}(1) \backslash [] = [1.1] \backslash [] = [1.1] \\
(\mathit{bool}, E') &= \mathit{accepts}(\mathit{head}\ [1.1], 331, [] \cup [(1, 331)], []) \\
&= \mathit{accepts}(1.1, 331, [(1, 331)], []).
\end{aligned}$$

We use the *accepts* definition of the `Sync` channel given in §6.2.1:

$$\mathit{accepts}(1.1, 331, [(1, 331)], []) = \mathit{Accepts}(2, 331, [(1, 331)], []).$$

We now evaluate whether node 2 accepts 331:

$$\begin{aligned}
&\mathit{Accepts}(2, 331, [(1, 331)], []) \\
&\qquad = (\mathit{bool}, E') \bowtie \mathit{Accepts}(2, 331, [(1, 331)], E' \cup [\mathit{head}\ \mathit{cel}])
\end{aligned}$$

where

$$\begin{aligned}
\mathit{cel} &= \mathit{srcCE}_{sel}(2) \backslash [] = [2.2] \backslash [] = [2.2] \\
(\mathit{bool}, E') &= \mathit{accepts}(\mathit{head}\ [2.2], 331, [(1, 331)] \cup [(2, 331)], []) \\
&= \mathit{accepts}(2.2, 331, [(1, 331), (2, 331)], []).
\end{aligned}$$

We use the *accepts* definition of the `Sync` channel given in §6.2.1:

$$\mathit{accepts}(2.2, 331, [(1, 331), (2, 331)], []) = \mathit{Accepts}(2, 331, [(1, 331), (2, 331)], []).$$

We now evaluate whether the visited node 2 accepts 331, however visited nodes do not accept anything (see §5.1.1):

$$\mathit{Accepts}(2, 331, [(1, 331), (2, 331)], []) = (\mathit{False}, []).$$

We now return to finish evaluating whether the channel of 2.2 accepts 331:

$$\mathit{accepts}(2.2, 331, [(1, 331), (2, 331)], []) = (\mathit{False}, []).$$

We now return to continue evaluating whether node 2 accepts 331:

$$\begin{aligned}
&\mathit{Accepts}(2, 331, [(1, 331)], []) \\
&\qquad = (\mathit{False}, []) \bowtie \mathit{Accepts}(2, 331, [(1, 331)], [] \cup [2.2]) \\
&\qquad = (\mathit{False}, []) \bowtie \mathit{Accepts}(2, 331, [(1, 331)], [2.2]).
\end{aligned}$$

We now evaluate the recursive call to *Accepts* for node 2. Since

$$\mathit{cel} = \mathit{srcCE}_{sel}(2) \backslash [2.2] = [2.2] \backslash [2.2] = []$$

therefore

$$\mathit{Accepts}(2, 331, [(1, 331)], [2.2]) = (\mathit{True}, [2.2]).$$

We now return to finish evaluating whether node 2 accepts 331:

$$Accepts(2, 331, [(1, 331)], []) = (False, []) \bowtie (True, [2.2]) = (False, [2.2]).$$

We now return to finish evaluating whether the channel of 1.1 accepts 331:

$$accepts(1.1, 331, [(1, 331)], []) = (False, [2.2]).$$

We now return to continue evaluating whether node 1 accepts 331:

$$\begin{aligned} &Accepts(1, 331, [], []) \\ &\quad = (False, [2.2]) \bowtie Accepts(1, 331, [], [2.2] \cup [1.1]) \\ &\quad = (False, [2.2]) \bowtie Accepts(1, 331, [], [2.2, 1.1]). \end{aligned}$$

We now evaluate the recursive call to *Accepts* for node 1. Since

$$cel = srcCE_{sel}(1) \backslash [2.2, 1.1] = [1.1] \backslash [2.2, 1.1] = []$$

therefore

$$Accepts(1, 331, [], [2.2, 1.1]) = (True, [2.2, 1.1]).$$

We now return to finish evaluating whether node 1 accepts 331:

$$\begin{aligned} &Accepts(1, 331, [], []) \\ &\quad = (False, [2.2]) \bowtie (True, [2.2, 1.1]) \\ &\quad = (False, [2.2, 1.1]). \end{aligned}$$

We now return to finish evaluating the traffic at node 1:

$$\begin{aligned} &\tau(1, *, [], []) \\ &\quad = [(331, E'') \mid (331, []) \leftarrow [(331, [])], \\ &\qquad\qquad (b_1, E'') \leftarrow (False, [2.2, 1.1]), b_1 == True] \\ &\quad = [(331, [2.2, 1.1]) \mid (331, []) \leftarrow [(331, [])], \\ &\qquad\qquad (False, [2.2, 1.1]) \leftarrow (False, [2.2, 1.1]), \\ &\qquad\qquad False == True] \\ &\quad = []. \end{aligned}$$

Because this traffic list τ is empty, the component *comp* in Figure 17 cannot perform the *write* operation (i.e., it remains pending).

References

[1] Farhad Arbab. Reo: A channel-based coordination model for components composition. volume 14 of *Math. Struct. in Comp. Science*, pages 329–366. Cambridge University Press, 2004.

[2] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005.

[3] David Costa, David Clarke, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency. In *4th International Workshop on the Foundations of Coordination Languages and software Architectures (FOCLASA)*, pages 85–102, August 2005. To appear in ENTCS in 2006.

[4] Jerud J. Mead. Haskell - a tutorial introduction. Available online at http://www.cs.drexel.edu/˜souter/cs360/sp04/handouts/hugs.pdf.

[5] J.J.M.M. Rutten, M. Kwiatkowska, N. Gethin, and D. Parker. *Mathematical Techniques for analysing concurrent and probabilistic systems*, volume 23 of *CRM Monograph series*, chapter 5: Component Connectors, page 215. American Mathematical Society, 2004.