Coordination models Orc and Reo compared

J.M.P. Proença, D.G. Clarke

REPORT SEN-R0802 MAY 2008
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)
coordination models Orc and Reo compared

ABSTRACT
Orc and Reo are two complementary approaches to the problem of coordinating components or services. On one hand, Orc is highly asynchronous, dynamic, and based on ephemeral connections to services. On the other hand, Reo is based on the interplay between synchronization and mutual exclusion, is more static, and establishes more continuous connections between components or services. The question of how Orc and Reo relate to each other naturally arises. In this paper, we present a detailed comparison of the two models. We demonstrate that embedding non-recursive Orc expressions into Reo connectors is straightforward, whereas recursive Orc expressions require an extension to the Reo model. For the other direction, we argue that embedding Reo into Orc would require, based on expressiveness results of Palamidessi, significantly more effort. We conclude with some general observations and comparisons between the two approaches.

1998 ACM Computing Classification System: D.3.1;D.3.2;F.1.1;F.1.2;F.3.2
Keywords and Phrases: Reo; Orc; Coordination languages; Software connectors
Coordination Models \textit{Orc} and \textit{Reo} Compared

José Proença†, Dave Clarke†

April 29, 2008

Abstract

\textit{Orc} and \textit{Reo} are two complementary approaches to the problem of coordinating components or services. On one hand, \textit{Orc} is highly asynchronous, dynamic, and based on ephemeral connections to services. On the other hand, \textit{Reo} is based on the interplay between synchronization and mutual exclusion, is more static, and establishes more continuous connections between components or services. The question of how \textit{Orc} and \textit{Reo} relate to each other naturally arises. In this paper, we present a detailed comparison of the two models. We demonstrate that embedding non-recursive \textit{Orc} expressions into \textit{Reo} connectors is straightforward, whereas recursive \textit{Orc} expressions require an extension to the \textit{Reo} model. For the other direction, we argue that embedding \textit{Reo} into \textit{Orc} would require, based on expressiveness results of Palamidessi, significantly more effort. We conclude with some general observations and comparisons between the two approaches.

1 Introduction

Although the field of coordination languages and models has been around for some time, the recent interest in Service-oriented Computing (SoC) and Web-service choreography and orchestration has precipitated greater interest in the field, resulting in both new models and new application domains for existing models. Service-oriented Computing is based on the idea that software is composed of services which reside on third-party machines [SH05]. Web services are a common realization of this idea [Cer02]. Since the conception of SoC, research has focused on developing languages to compose or coordinate services into either composite services or complete applications.

Coordination languages and models are based on the philosophy that an application or system should be divided into the parts that perform computation and the parts that coordinate the results and resources required to perform the computations. The original coordination language, Linda [Gel85], played only a passive rôle in coordination, by providing a blackboard (tuple space) which data can be written to and read from. Since then many coordination models have been proposed [PA98, AHM96], and the trend is towards developing models that play a more active rôle in the coordination process. Two recent and interesting active coordination models, \textit{Orc} [CM07] and \textit{Reo} [Arb04], sit diametrically opposite of each other in their approaches to coordinating services or components. This paper sets out to explore these models in detail.

\textit{Orc} is a simple orchestration language designed by Misra and Cook [CM07], based on three connectives and the simple notion of a site call to model computations—the external actions to be orchestrated. Central to \textit{Orc}’s design is the idea that accessing (web) sites is an asynchronous activity which can fail, and so the connectives are designed to be asynchronous and not susceptible to failure.

\textit{Reo} is a channel-based coordination language designed by Arbab [Arb04] that is based on a simple notion of channel composition. It differs from existing models in that composition of connectors from channels propagates synchronization and exclusion constraints through connectors. In combination with stateful channels, an expressive coordination language emerges.

\footnote{Supported by FCT grant 22485 – 2005, Portugal.}
\footnote{Email: \texttt{Jose.Proenca@cwi.nl D.G.Clarke@cwi.nl}}
\footnote{We take the words choreography and orchestration to fall under the more general notion of coordination.
This paper presents a comparison of \( \text{Orc} \) and \( \text{Reo} \). By choosing two coordination languages at different ends of the spectrum for our comparison, we hope to gain insight into the design choices and the advantages and disadvantages of various approaches. In the long run, we should hope for a synthesis of their key ideas, to get the best of both worlds. We present a number of examples, compare features and the underlying philosophies and design choices, and formally embed \( \text{Orc} \) into \( \text{Reo} \). We also discuss the difficulties of the embedding in the other direction, referring to known results about encoding the asynchronous \( \pi \)-calculus into the synchronous \( \pi \)-calculus. The main conclusions that we make from this work is that neither full \( \text{Orc} \) can be encoded into \( \text{Reo} \), nor can \( \text{Reo} \) be encoded into \( \text{Orc} \). We start by explaining informally why infinite \( \text{Orc} \) expressions, defined using recursion, cannot be encoded into \( \text{Reo} \). We then define \( \text{Orc}^- \) as the subset of \( \text{Orc} \) without recursion, and provide an encoding from \( \text{Orc}^- \) to \( \text{Reo} \), that we prove to be correct. This encoding is the main contribution of this paper. To show that the synchronous behaviour of \( \text{Reo} \) cannot be encoded into \( \text{Orc} \), we use previous results about the \( \pi \)-calculus due to Palamidessi [Pal94], by encoding \( \text{Orc} \) into the asynchronous \( \pi \)-calculus, and referring to the symmetric leader election problem. Here we do not formally prove the correctness of this last encoding.

Section 2 describes our encoding of \( \text{Orc} \) into \( \text{Reo} \). Section 3 presents our argument that the other direction is not possible, in general. Section 4 compares the two models on a variety of points. Section 5 discusses some related work, and Section 6 concludes and discusses future work. But first, we introduce \( \text{Orc} \) and \( \text{Reo} \), and give small examples.

### 1.1 \( \text{Orc} \)

In this section we present \( \text{Orc} \)'s syntax and reduction semantics, and give simple examples of \( \text{Orc} \) expressions. Work by Misra and others describes \( \text{Orc} \)'s semantics in more detail [CM07, KCM06].

\( \text{Orc} \) expressions have the following syntax, where \( E \) is an expression name, \( M \) is a site name, \( x \) is a variable, \( v \) is a constant value, and \( \overline{p} \) is a tuple of \( p \)'s:

\[
\begin{align*}
\text{Expr} &::= 0 \mid M(\overline{p}) \mid E(\overline{p}) \mid f \triangleright x \triangleright g \mid f \triangleright g \mid f \text{ where } x : \in g \\
\text{Actual} &::= x \mid v \\
\text{Definition} &::= E(\overline{p}) \overset{\text{def}}{=} f
\end{align*}
\]

An \( \text{Orc} \) program consists of an \( \text{Orc} \) expression together with a set of definitions. Basic services, such as data manipulation, are assumed to be provided by primitive sites. An \( \text{Orc} \) expression can be a primitive site call, a reference to another \( \text{Orc} \) expression, or a composition of \( \text{Orc} \) expressions. The computational model underlying \( \text{Orc} \) consists on a number of expressions running in parallel, which ultimately call sites. Each of these sites may publish a result, that can be discarded or used in other expressions.

A site call is written \( M(\overline{p}) \), where \( \overline{p} \) is a tuple of arguments, which can be constants or variables. During execution all variables have to be instantiated, that is, evaluation is strict, and the site returns at most one value. Example primitive sites include \( 0 \), a special primitive included in \( \text{Orc} \)'s syntax grammar which never responds, and \( \text{let}(v) \), which responds value \( v \). We use \( E \) to range over possibly recursive definitions of \( \text{Orc} \) expressions.

Three combinators exist for composing expressions \( f \) and \( g \): symmetric composition, written \( f \mid g \); sequential composition, written \( f \triangleright x \triangleright g \); and asymmetric composition, written \( f \text{ where } x : \in g \). The combinator \( f \mid g \) calls \( f \) and \( g \) simultaneously and executes them in parallel. The values that it can publish are exactly all the values that \( f \) and \( g \) can publish. The sequential composition \( f \triangleright x \triangleright g \) starts by calling \( f \), and for each published value by \( f \), a new thread of \( g \) is executed. The variable \( x \) is bound to each value published by \( f \) in the corresponding thread of \( g \). The values published by \( f \triangleright x \triangleright g \) consist of the values published by all threads of \( g \). The last operator \( f \text{ where } x : \in g \) calls \( f \) and \( g \) in parallel, replacing \( x \) in \( f \) by the first published value of \( g \). All the subsequent values published by \( g \) are discarded. This operator publishes only the values published by \( f \).

In the rest of this section we show some examples, borrowed from Kitchin et al. [KCM06], to provide a better understanding about the semantics of \( \text{Orc} \). In the end we formally present \( \text{Orc} \)'s semantics.

---

2These operations appear to be closely related to Friedman and Wise’s frons construct [FW80].
Sequential vs asymmetric parallel composition Consider the following Orc expressions:

\[
\begin{align*}
\text{EmailNews}(d) & \overset{\text{def}}{=} (\text{CNN}(uk, d) \mid \text{BBC}(uk)) >x> \text{email}(me, x) \\
\text{EmailNewsOnce}(d) & \overset{\text{def}}{=} \text{email}(me, x) \text{ where } x \in (\text{CNN}(uk, d) \mid \text{BBC}(uk))
\end{align*}
\]

Here, \(uk\) and \(me\) are constant values, \(x\) and \(d\) are variables, and \(\text{CNN}(uk, d)\), \(\text{BBC}(uk)\) and \(\text{email}(me, x)\) are site calls that retrieve the news for the UK on the day \(d\) from CNN, retrieve the news for the UK from BBC for today, and send an email to \(me\) with value \(x\). Thus \(\text{EmailNews}(d)\) and \(\text{EmailNewsOnce}(d)\) invoke the news service from CNN and BBC and send the resulting content by e-mail to \(me\). The difference between these two expressions is that \(\text{EmailNews}\) sends the news from both CNN and BBC (when the services reply), while \(\text{EmailNewsOnce}\) mails only the value of the first reply, ignoring the second reply.

Time-out Let \(R\text{timer}(t)\) be a site that, when called, waits \(t\) time units before publishing a signal. Using this site, we can express a call to a site \(M\) that can only wait \(t\) time units for its result using the following Orc expression.

\[
\text{let}(z) \text{ where } z :\in (M >x> \text{let}(x, \text{true}) \mid \text{R\text{timer}}(t) >x> \text{let}(x, \text{false}))
\]

In this example \(\text{true}\) and \(\text{false}\) are constants that indicate whether \(M\) succeeded in publishing a value or not. When \(M\) is faster to publish a value than \(\text{R\text{timer}}(t)\), then the full expression publishes the tuple \((x, \text{true})\), where \(x\) is the value published by \(M\). Otherwise, it publishes \((y, \text{false})\), where \(y\) is the signal published by the timer site. In the semantics of Orc that we present in the end of this Section we consider that, when both sides of the parallel composition are equally fast, then one is chosen non-deterministically.

Barrier Synchronization Consider the Orc expressions \(M >x> f\) and \(N >y> g\). We can execute them in parallel, imposing that \(f\) and \(g\) are called at the same time, after both sites \(M\) and \(N\) have completed.

\[
((\text{let}(u, v) \text{ where } u :\in M) \text{ where } v :\in N) > (x, y) > (f | g)
\]

The two asymmetric parallel combinators join the results of the calls of \(M\) and \(N\), and the result is forward to \(f\) and \(g\) via a single sequential composition combinator.

Recursion Infinite behaviour can be described using recursive definitions, as the following example shows.

\[
\text{Metronome} \overset{\text{def}}{=} \text{Signal} \mid (\text{R\text{timer}}(1) >x> \text{Metronome}) \\
\text{EmailNewsFrequently}(d) \overset{\text{def}}{=} \text{Metronome} >x> \text{EmailNewsOnce}(d)
\]

\(\text{Metronome}\) is an Orc expression that sends a signal published by \(\text{Signal}\) every time unit. The site \(\text{R\text{timer}}(t)\), as in the time-out example, waits \(t\) time units before publishing a signal. Therefore, \(\text{EmailNewsFrequently}\) calls \(\text{EmailNewsOnce}\) every time unit, which in turn sends \(me\) an email from either CNN or BBC.

Orc’s semantics Instead of the standard, asynchronous semantics for Orc, we present a synchronous semantics which allows multiple events to occur at the same time. This approach enables a simpler formal comparison with Reo, without really changing the essence of Orc. The reduction rules for Orc expressions have the form \(f \overset{a}{\Rightarrow} g\) and are presented below. Here \(a\) is a set of observations of base events, defined as follows:

\[
\text{BaseEvent} ::= \tau \mid M_a(\tau) \mid k?v \mid !v
\]

We use the silent observation \(\tau\) mainly to represent the binding of a variable to a value. \(M_a(\tau)\) represents the call to site \(M\), indexed by a fresh \(k\) and where \(\tau\) is a tuple of values used as argument. \(k?v\) represents the return of value \(v\) by the site call indexed with \(k\). \(!v\) represents that a value \(v\) was published. Finally, we use \(a\) and \(b\) to range over sets of observations, following the convention that \(\overset{a}{\Rightarrow}\) denotes \(\Rightarrow\). The reduction rules are presented in Fig. [1].
primitive with arity $i.e.$

\[
\begin{array}{ll}
\text{(SiteCall)} & f \xrightarrow{g \ x} g' \\
\text{(SiteRet)} & let(v) \\
\text{(LET)} & let(v) \rightarrow 0 \\
\end{array}
\]

\[
\begin{array}{ll}
\text{(SYM1)} & f \xrightarrow{g \ x} f' \\
\text{(SYM2)} & g \xrightarrow{f} g' \\
\text{(SYM3)} & f \xrightarrow{g} f' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{(ASYM1N)} & g \xrightarrow{f} g' \\
\text{(ASYM3N)} & f \xrightarrow{g} f' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{(ASYM2)} & g \xrightarrow{x \xrightarrow{b} f} g' \\
\text{(ASYM2V)} & f \xrightarrow{x \xrightarrow{v} g} f' \\
\end{array}
\]

\[
\begin{array}{ll}
\text{(SEQ)} & f \xrightarrow{x \xrightarrow{v} g} f' \\
\end{array}
\]

Figure 1: Operational semantics of Orc

To describe the behaviour of a site call we introduce an extension to represent an intermediate state, $7k$, following [KCM06]. This denotes an instance of a site call that has not yet returned, and is used in rules (SiteCall) and (SiteRet). $k$ is a fresh value used to uniquely identify the specific call. We also use the primitive site $let(v)$ as an intermediate state, to capture a site that has just returned value $v$, and will publish that value. For the case of the asymmetric composition $g$ where $x \in f$, five different rules were defined to distinguish the cases when only $g$ is reduced, when only $f$ is reduced (publishing or not a value), and the combination of both.

The reductions rules in Fig. [1] yield the following reduction of the expression $EmailNewsOnce$ presented above. Here we assume that $a$ and $b$ are fresh, $v$ is the value published by the BBC site, and $v'$ is the value published by the email site.

$$
\frac{\text{BBC}_a(uk) \quad \text{email}(me, x) \quad x \in (CNN(uk, d) \mid BBC(uk))}{\text{email}(me, x) \quad x \in (CNN(uk, d) \mid ?a)}
$$

1.2 Reo

Reo is a powerful coordination model based on channel composition. Channels impose synchronisation and other constraints on their ends. Behaviour arises from the propagation of these constraints through connectors formed by plugging channels together to form nodes, which themselves impose mutual exclusive data merging and synchronous data replication constraints. A key characteristic of Reo is that synchrony and mutual exclusion constraints are propagated through composition. We present the semantics of Reo connectors in an adaptation of the constraint automata model [BSAR06].

Firstly, we assume that connectors are defined over a denumerable set of node names, $Node$. Each connector $C$ will have a set of input nodes $I \subseteq Node$, and a (disjoint) set of output nodes, $O \subseteq Node$. The input and output nodes of a connector define its $arity$, denoted $C : I \to O$.

We define $Names(C)$ to be $I \cup O$, which we call the boundary nodes of $C$.

The semantics of a connector $C$ is given as a reduction relation of the form $C \xrightarrow{N} C'$, where $N$ is a (partial) map from the set of boundary nodes to the values that flow through those nodes. For example, we write $I(v), O(v)$ to denote the map from the boundary nodes $I$ and $O$ to the value $v$, and we write $nodes(N)$ to denote the domain of $N$. We say that $C$

\[\text{For the purpose of this paper, we assume that primitive connectors are not plugged into themselves, i.e., for a primitive with arity } I \to O, \text{ we have that if } I \to O \text{ then } I \cap O = \emptyset.
\]

\[\text{We introduce a slightly different definition from the literature, where a boundary node is a node that is only an input or output node.} \]
evolves to $C'$ and fires nodes $\text{nodes}(N)$. $C'$ is the connector resulting from the particular step. Typically, $C$ and $C'$ will have the same primitives, just in different states. Table 1 presents some Reo primitives, their arity, and axioms describing their behaviour. Each axiom gives a valid reduction of the corresponding primitive.

<table>
<thead>
<tr>
<th>Visualisation</th>
<th>Representation</th>
<th>Arity</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{Sync}_{A,B}$</td>
<td>$A \rightarrow B$</td>
<td>$\text{Sync}<em>{A,B} \xrightarrow{A(v),B(v)} \text{Sync}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{SDrain}_{A,B}$</td>
<td>${A, B} \rightarrow \emptyset$</td>
<td>$\text{SDrain}<em>{A,B} \xrightarrow{A(v),B(w)} \text{SDrain}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{SSpout}_{A,B}$</td>
<td>$\emptyset \rightarrow {A, B}$</td>
<td>$\text{SSpout}<em>{A,B} \xrightarrow{A(v),B(w)} \text{SSpout}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Lossy}_{A,B}$</td>
<td>$A \rightarrow B$</td>
<td>$\text{Lossy}<em>{A,B} \xrightarrow{A(v)} \text{Lossy}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{ADrain}_{A,B}$</td>
<td>${A, B} \rightarrow \emptyset$</td>
<td>$\text{ADrain}<em>{A,B} \xrightarrow{A(v)} \text{ADrain}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{ASpout}_{A,B}$</td>
<td>$\emptyset \rightarrow {A, B}$</td>
<td>$\text{ASpout}<em>{A,B} \xrightarrow{A(v)} \text{ASpout}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{FIFO1}_{A,B}$</td>
<td>$A \rightarrow B$</td>
<td>$\text{FIFO1}<em>{A,B} \xrightarrow{A(v)} \text{FIFO1}</em>{A,B(v)}$</td>
</tr>
<tr>
<td></td>
<td>$\text{FIFO1}_{A,B}(v)$</td>
<td>$A \rightarrow B$</td>
<td>$\text{FIFO1}<em>{A,B}(v) \xrightarrow{B(v)} \text{FIFO1}</em>{A,B}$</td>
</tr>
<tr>
<td></td>
<td>$\text{Merger}_{A,B,C}$</td>
<td>${A, B} \rightarrow C$</td>
<td>$\text{Merger}<em>{A,B,C} \xrightarrow{A(v),C(v)} \text{Merger}</em>{A,B,C}$</td>
</tr>
</tbody>
</table>

Table 1: Arity and behaviour of some Reo primitives

The composition of connectors $C$ and $C'$ is denoted by $C * C'$. Well-formedness of the composition and the calculation of its arity is given by the following rule:

$$
C : I \rightarrow O \quad C' : I' \rightarrow O' \\
I'' \overset{\text{def}}{=} I \cup I' \quad O'' \overset{\text{def}}{=} O \cup O' \quad O \cap O' = \emptyset \\
C * C' : (I'' \setminus O'') \rightarrow O''
$$

This rule expresses that output and input nodes are plugged 1: n, i.e., each output node can be plugged into multiple input nodes. This results from the fact that the well-formedness conditions in the rule only impose that $O \cup O' = \emptyset$, and not that $I \cup I' = \emptyset$, and also from the fact that we only remove the repeated input nodes in the resulting arity. Regarding the behaviour, output nodes act as n-replicators, where data must flow to every connected input channel end. If $n = 0$, we assume that the data is consumed. The formal description we present differs slightly from the original description of Reo, where nodes could result from...
an $n : m$ plugging, without fundamentally changing anything, in order to simplify our formal results.

**Notation 1.1.** Given a map $N$ and a set $P$. With a slight abuse of notation, define $N \cap P \stackrel{\text{def}}{=} \{(n, d) \in N \mid n \in P\}$ and $N \setminus P \stackrel{\text{def}}{=} \{(n, d) \in N \mid n \notin P\}$.

The following two rules give the semantics for the composition of connectors $C_1 : I_1 \rightarrow O_1$ and $C_2 : I_2 \rightarrow O_2$. Note that a node set can only fire if it fires in both $C_1$ and $C_2$, with the same data value flowing in both cases.

$$
\begin{align*}
C_1 \xrightarrow{N_1} C_1' & \quad C_2 \xrightarrow{N_2} C_2' \\
N_1 \cap \text{Names}(C_2) = N_2 \cap \text{Names}(C_1) & \quad C_1 \circ C_2 \xrightarrow{N_1 \cup N_2} C_1' \circ C_2' \\
C_1 \xrightarrow{N_1} C_1' & \quad N_1 \cap \text{Names}(C_2) = \emptyset \\
C_1 \circ C_2 \xrightarrow{N_1} C_1' \circ C_2
\end{align*}
$$

Note that we do not address causality issues here, because the connectors we will build deliberately avoid causal loops. These can be trivially dealt with. We also introduce a restriction operator that hides some output nodes of a connector. Given a connector $C : I \rightarrow O$ and a set of nodes $\Omega \subseteq O$, define $C \mid \Omega = C : I \rightarrow \Omega$.

**Ordering example** Consider the services Politics, Sport and Email, that return news about politics or news about sport, or send an email of a given message, respectively. In Fig. 2 we present a connector that coordinates these three services. Initially, the connector receives data from the Politics and the Sport services, and forwards data from Politics to the Email in a single step. After that, the data previously sent by the Sport service is sent to the Email. This way we guarantee that the two services alternate, and that we can only have politics news if sports news is also available—presumably as a sanity-preserving measure.

![Figure 2: Example of a Reo connector](image)

Formally, we consider two different states of the connector:

$$
\begin{align*}
\text{Ord} & = (\text{FIFO}_{\text{Sport}, X} \circ \text{SDrain}_{\text{Politics}, \text{Sport}} \circ \text{Merger}_{\text{Politics}, X, \text{Email}}) \mid \text{Email} \\
\text{Ord}(x) & = (\text{FIFO}_{\text{Sport}, X}(x) \circ \text{SDrain}_{\text{Politics}, \text{Sport}} \circ \text{Merger}_{\text{Politics}, X, \text{Email}}) \mid \text{Email}
\end{align*}
$$

The former corresponds to the connector depicted in Fig. 2, while the latter corresponds to the same connector when the FIFO channel is full with data $x$. Using the rules above, we can calculate the connector’s arity, $\text{Ord} : \{\text{Politics}, \text{Sport}\} \rightarrow \text{Email}$, and behaviour:

$$
\begin{align*}
\text{Ord} & \xrightarrow{\text{Politics}(w), \text{Sport}(w), \text{Email}(v)} \text{Ord}(w) \\
\text{Ord}(w) & \xrightarrow{X(w), \text{Email}(w)} \text{Ord}.
\end{align*}
$$

These transitions represent the only possible behaviour of the connector, given the axioms of each primitive and the reduction rules. The first transition goes to a state where the buffer is full, and indicates that data is flowing on the nodes Politics, Sport, and Email. The second transition goes back to the original state, and indicates that data is flowing on nodes $X$ and Email.

**Synchronizing merge example** We now present an example to illustrate how to define a more complex coordination pattern, without going into too much details. In Fig. 3 we represent a synchronizing merge connector, whose main idea is to control the execution of components or connectors A and B according to a specific pattern. This pattern is one of the workflow patterns defined by Van der Aalst [AHK03].
In the connector presented in Fig. 3, we introduce some special notation. The nodes with a cross (\(\times\)) (to be introduced in Table 2) denote exclusive routers, which output the received data to precisely one of its outputs. We use nodes with more than one input to represent mergers connected to the node. The components denoted by A and B have an input node on the left side and an output node on the right side, and represent two external components or connectors that receive a signal to start executing, and return another signal after executing.

The composition of the behaviour of the primitives in Fig. 3, after hiding the details of every output node except O, yields the following behaviour. Initially, only the input node I can be fired, causing data to flow into the \(FIFO1\) channel on the left and through the exclusive router on the left. Data will flow also either to A, to B, or to both, depending on who is ready to receive data, and to one of the three \(FIFO1\) channels in the middle. The connector evolves to a new state, where the only possible step is to output data through node O after the components that were initiated return a signal, emptying the existing \(FIFO1\) channels. Therefore, if only A was executed, then B cannot execute until A finishes, and if both A and B were executed, then they must finish before any of them can be executed again.

2 A Static Encoding of \(\Orc\) in \(\Reo\)

We present two translations of \(\Orc\) into \(\Reo\). The first translation, the merged-output encoding, attempts to directly model \(\Orc\) expressions, in particular, by merging the multiple results of a sequential composition. The second encoding, the multiple-output encoding, takes an alternative approach, duplicating the circuitry for each output of the \(\Orc\) expression. Note that we can only encode non-recursive \(\Orc\) expressions into a finite \(\Reo\) connector. For the remainder of the paper, we restrict ourselves to non-recursive \(\Orc\) expressions, denoted \(\Orc^-\). Basically, we assume that every invocation of a definition has been expanded. We also assume the existence of a \(\Reo\) component, with one input and one output node, for each primitive site. Initially, the component is ready to receive some data over the input node; after an unspecified amount of time, it may return a result over the output node.

Before presenting the encodings, we will introduce some useful \(\Reo\) connectors. We then give some formal properties concerning the second encoding on Section 2.4, and use weak bisimulation to prove its soundness with respect to \(\Orc\)'s semantics.

2.1 Warming up

We now introduce the \(\Reo\) connectors used in the translations. Each connector is defined by presenting its arity and axioms, although they could equally have been defined as the composition of primitives.\(^5\) The connectors are defined in Table 2.

Table 2 is divided into two parts. In the upper part we present a \(\Reo\) node and three connectors that play a rôle similar to nodes in that they connect a single output node to multiple input nodes A node (\(\bullet\)) with arity \(I \rightarrow \{O_1, \ldots, O_n\}\) receives data in \(I\) and replicates the same data synchronously to \(O_1, \ldots, O_n\). This can be derived from the rules introduced in Section 1.2. The behaviour of the remaining three connectors in the upper part is as follows. Firstly, an Exclusive Router (\(\times\)) receives data in the input node and sends data

\(^5\)The tupling connector \(T_n\) is an exception, as none of our primitives are capable of data manipulation.
synchronously to exactly one of its output nodes. If more than one output node can receive the data, a non-deterministic choice is made. Secondly, an Inclusive Router (⊗) is a variation of the Exclusive Router that can send data to multiple output nodes instead of performing a non-deterministic choice. Third is a connector that acts like a node for one step, and then prevents flow for eternity, by becoming the connector in the fourth row.

Now consider the lower part of Table 2. Connector \( T_n \) tuples \( n \) values. It is a synchronous connector, i.e., inputs and outputs succeed at the same time. Connector \( C_p \) always return a constant value \( p \). Connectors \( \text{Var} \) and \( \text{Var}(x) \) represent a possibly-undefined variable. It is a buffer that replaces its content when new data arrives to the connector, and can output its content as many times as required. The last connector, \( P_n \), coordinates \( n \) inputs into a single output. Data flows only if data can flow synchronously at one or more input nodes and the output node.

Before continuing with the encoding, an issue regarding the use of variables in Reo needs to be resolved. A variable can be read by multiple connectors, either all at the same time or just by some at each step. To coordinate access to a variable, we propose two different approaches in Fig. 4: (a) replicate the output of the variable when necessary, or (b) replicate the input and create a variable connector for each possible access. The second approach has the advantage that the access to a variable does not require any synchronisation between the connectors that may also access the variable. Although more storage locations are required, it reduces the cost of coordination. This is the approach we use.
2.2 Merged-Output Encoding

This section presents an encoding of an \(\text{Orc}^-\) expression into a connector which merges the multiple outputs of a parallel composition via a single output node. This is the most natural approach, but it is, as we shall see, problematic. We therefore only give an informal presentation, reserving a completely formal description for our second encoding.

An expression \(h \in \text{Orc}^-\) is encoded as a connector with arity \(\{I, X_1, \ldots, X_n\} \rightarrow O\), depicted in Fig. 5(a), where the \(X_i\) correspond to the free variables of \(h\). For example, the encoding of the expression \((\text{CNN}(uk, d) \mid \text{BBC}(uk)) > x > \text{email}(me, x)\) is presented in Fig. 5(b), recalling that \(d\) is a variable, whereas \(uk\) and \(me\) are constants. The connector starts by receiving data on input node \(I\) and buffering it. Site \(\text{BBC}\) can then be called, while site \(\text{CNN}\) needs to wait until data is available on node \(D\). The results from the site calls are stored in the \(\text{RVar}\) component one at a time, which subsequently provides the value to site \(\text{email}\), once for each value returned by \(\text{BBC}\) and \(\text{CNN}\).

Figure 5: Encodings into \(\text{Reo}\) connectors with a single output: (a) a general \(\text{Orc}\) expression; (b) a specific \(\text{Orc}\) expression \((\text{CNN}(uk, d) \mid \text{BBC}(uk)) > x > \text{email}(me, x)\). \(\text{RVar}\) is a resettable variable. It acts like a variable (e.g., \(\text{Var}\) from Section 2.1), but it cannot be updated until the reset (top) node is fired, removing the value of the variable.

The example encoding reveals the main problem of this approach. The outputs of \(\text{CNN}(uk, d) \mid \text{BBC}(uk)\) are forwarded to a single instance of \(\text{email}\), serializing the execution of \(\text{email}\). As a consequence, it is possible that \(\text{CNN}\) finishes before \(\text{BBC}\), but that site \(\text{email}\) hangs on the result of \(\text{CNN}\), preventing \(\text{email}\) from even getting the result from \(\text{BBC}\). The semantics of \(\text{Orc}^-\) [KCM06], however, dictate that \((\text{CNN}(uk, d) \mid \text{BBC}(uk)) > x > \text{email}(me, x)\) is strongly bisimilar to \((\text{CNN}(uk, d) > x > \text{email}(me, x)) \mid (\text{BBC}(uk) > x > \text{email}(me, x))\), which means that \(\text{email}\) is not serialized and could respond to either results from \(\text{CNN}\) or \(\text{BBC}\) irrespective of their ordering or failure. This, however, is not true for the connectors resulting from the encoding. In the next section, we overcome this problem by duplicating parts of the connector.

Another solution for this termination problem is possible by introducing some observational behaviour corresponding to when a service does not publish any value, as done by Bruni et al. in their encoding of \(\text{Orc}\) into Petri Nets [BMT06]. This could be achieved, for example, by adding timeouts to each primitive site call. An extension for \(\text{Reo}\) that includes connectors capable of dealing explicitly with time was proposed by Arbab et al. [ABBR07]. The authors introduce the Timed Constraint Automata, which can be used to formally model the timeouts in \(\text{Reo}\), allowing a precise definition of a component that fails to return any value. In our case we could attach a timeout connector to the input node of each site call, such as an expiring \(\text{FIFO1}\) channel, which loses the contents of its buffer after a certain time.
Using these ideas we expect that we could also encode recursive Orc expressions, but we chose not to use this approach because we consider it to be less faithful to Orc’s semantics, where the failure to return a value cannot be observed.

2.3 Multiple-Output Encoding

A more faithful encoding of Orc expressions is presented in this section. The encoding of an expression such as \( f >x> g \) duplicates \( g \) for each output of \( f \). The encoding is possible because we can obtain an upper bound on the number of outputs of an Orc expression—this is not possible with full Orc due to recursion. The following lemma captures this property.

**Lemma 2.1.** Define function \( \#(\cdot) \) on Orc expressions (and internal representations: \( ?k \) and \( \text{let}(\cdot) \)), and on sets of output actions as follows:

\[
\begin{align*}
\#(f | g) &= \#(f) + \#(g) \\
\#(f >x> g) &= \#(f) \times \#(g) \\
\#(g \text{ where } x \in f) &= \#(g) \\
\#(M(v_1, \ldots, v_n)) &= 1 \\
\#0 &= 0 \\
\#(\text{let}(v)) &= 1 \\
\#(\text{let}(v_1, \ldots, v_n, a)) &= n \\
\end{align*}
\]

This function gives an upper bound on the number of outputs produced by an Orc expression, i.e., for any Orc expression \( h \), \( h \xrightarrow{a} h' \) implies \( \#h \geq \#a + \#h' \).

**Proof.** Note that substitution does not effect the number of outputs, i.e., \( \#([\sigma]_h) = \#h \), because \( \#(M(x)) \) and \( \#(M(v)) \) are always 1 independently of the value of \( v \), where \( x \) is a variable and \( v \) is a constant value.

The proof follows by induction on the structure of \( h \). The base cases, \( 0 \), \( \text{let}(v) \), \( ?k \), and \( M(v) \), are trivial, since there is only one possible action for each, and only \( \text{let}(v) \) produces one output. We also omit the case when \( h = g \mid h \), since it is simpler than the other combinators, and the reasoning is analogous.

- **Case 1:** \( h = f >x> g \): In this case our induction hypothesis is: if \( f \xrightarrow{a} f' \), then \( \#f \geq \#a + \#f' \). Assume \( f \xrightarrow{a} f' \), where \( a = v_1, \ldots, v_n, a' \) and \( !w \notin a' \) (hence, \( \#a = n \)). The only possible reduction is given by rule Sequ: \( h \xrightarrow{a} h' \), where \( h' = f' >x> g \mid [v_1 / x].g \mid \ldots \mid [v_n / x].g \). Since \( a' \) has no outputs, we know that \( \#a' = 0 \). Then we can conclude that:

\[
\begin{align*}
\#a' + \#h' &= \#h' \\
&= \#(f' >x> g | [v_1 / x].g | \ldots | [v_n / x].g) \\
\{\text{Def. } (\#)\} &= \#(f' >x> g) + \#([v_1 / x].g) + \ldots + \#([v_n / x].g) \\
&= \#(f' >x> g) + \#a \times \#g \\
\{\text{IH on } f\} &\leq \#f \times \#g \\
&= \#h
\end{align*}
\]

- **Case 2:** \( h = g \text{ where } x \in f \): Again, our induction hypothesis is: if \( g \xrightarrow{a} g' \) and \( f \xrightarrow{a} f' \), then \( \#g \geq \#a + \#g' \) and \( \#f \geq \#b + \#f' \). We consider 2 cases. The first case assumes that \( g \xrightarrow{a} g' \) and \( f \xrightarrow{b} f' \), where \( !w \notin b \) (and therefore \( \#b = 0 \)). In this case we can apply rule Asym3N: \( h \xrightarrow{a,b} g' \text{ where } x \in f' \), and conclude that \( \#(a, b) + \#(g') \text{ where } x \in f' = \#a + \#b + \#g' \leq \#g = \#h \). For the second case we assume \( g \xrightarrow{a} g' \) and \( f \xrightarrow{b} f' \), where \( b \) can be any set of actions. In this case we can apply rule Asym2V: \( h \xrightarrow{a} [v / x].g' \), and conclude that \( \#a + \#([v / x].g') = \#a + \#g' \leq \#g = \#h \). We arrive at a similar conclusion in the remaining possibilities for the reduction of \( h \), after applying rule Asym1N, Asym2 or Asym1V.

\[\square\]

**Corollary 2.2.** Let \( f \in \text{Orc}^- \), and \( f \xrightarrow{a_1} f' \xrightarrow{a_2} \ldots \xrightarrow{a_n} f^{(n)} \) be a possible trace. Then \( \#f \geq \#a_1 + \ldots + \#a_n + \#f^{(n)} \).
We now define a function \([\cdot]\) that converts an expression \(f \in \text{Orc}^-\) into a \(\text{Reo}\) connector. The arity of the resulting connector will be \(\{I\} \cup V \rightarrow O\), where \(I \notin V\), \(I\) denotes the main input node, \(V\) denotes a set of nodes corresponding to the free variables of \(f\), and \(O\) is the set of output nodes. Node \(I\) is used to initiate the connector, though nodes in \(V\) can be fired beforehand, which corresponds to the setting of these variables. The function \([\cdot]\), presented in Fig. 6, is defined inductively on the shape of Orc expressions, in such a way that the number of output nodes is given by the function \(#\) defined above. In the definition of the encoding we use special primitives, also depicted in Fig. 6, where we show an \(F\) and \(G\) shaped hole to provide some intuition about where \([\cdot]\) is set of output nodes. Node \(X\) denotes a set of nodes corresponding to the free variables of \(f\), which will be connected to \(G\) where the value of \(x\) is used.

**Symmetric Parallel Composition:**

\[
\|\theta,\alpha,\beta : I \rightarrow \{I_f, I_g\}
\]

Initially \(\theta = \alpha = \beta = 0\). The intuition behind the connector \(\|\theta,\alpha,\beta\), illustrated in Fig. 6(a), is that it is initialized by flow in node \(I\), after which sends an initialization signal on nodes \(I_f\) and \(I_g\). The data is buffered in buffers that can fire nodes \(I_f\) and \(I_g\) as soon as they are ready to be fired. As \([f \parallel g]\) = \([0,0,0] * F * G\), firing \(I_f\) and \(I_g\) will trigger the connectors \(F\) and \(G\). The behaviour of \(\|\theta,\alpha,\beta\) is depicted in the diagram below.

![Diagram](image)

**Sequential Composition:**

\[
\|x\theta,\alpha_1,..,\alpha_n : \{I, O_f1, \ldots, O_f n\} \rightarrow \{I_f, I_g1, \ldots, I_g n, X_1, \ldots, X_n\}
\]

The connector is illustrated in Fig. 6(b). The main idea is to execute \(F\) when data flows through the input node \(I\), and to buffer each of its outputs in a different FIFO channel. Each of these FIFO channels is connected to a different instance of the encoded \(G\), which can be executed in parallel after the corresponding FIFO channel is filled.

To make the behaviour easier to describe, we factor \(\|x\theta,\alpha_1,..,\alpha_n\) into \(n + 1\) different connectors corresponding to unconnected parts of the main connector:

\[
\|x\theta,\alpha_1,..,\alpha_n = \|x\theta, F * \|x\alpha_1, G_1 * \cdots * \|x\alpha_n, G_n,
\]

where \(\|x\theta, F : I \rightarrow I_f\), \(\|x\alpha_j, G_j : O_f j \rightarrow \{I_{gj}, X_{j}\}\), and \(1 \leq j \leq n\). Initially \(\theta = \alpha_1 = \ldots = \alpha_n = 0\). The possible behaviour of each of the subparts is the following:

\[
\begin{align*}
\|x\theta, F &\rightarrow I_f(v), I_f(v) \\
\|x\alpha_j, G_j &\rightarrow O_f j(v), X_{j}(v), I_{gj}(v), I_{gj}(v), \|x\alpha_j, G_j \\
\end{align*}
\]

where \(1 \leq j \leq n\). This means that \(\|x\theta, 0, \ldots, 0\), when triggered by node \(I\), synchronously triggers the input node of \(F\). For each output of \(F\) (in node \(O_f j\)), the connector \(\|x\alpha_j, G_j\) also synchronously fires node \(X_j\) (making the contents of variable \(x\) available in \(G\)), and evolves to a configuration where the input node of \(G_j\) can be fired whenever possible.

**Asymmetric Parallel Composition:**

\[
\mathbb{W}^\theta,\alpha,\beta,\delta : \{I, O_f1, \ldots, O_f n\} \rightarrow \{I_f, I_g, X\}
\]

The connector is illustrated in Fig. 6(c). The intuition is that \(F\) and \(G\) are executed in parallel. The output nodes of \(F\) are merged in such a way that only the first output value will flow through node \(X\), which will be connected to \(G\) where the value of \(x\) is used. The output nodes of the connector \(\mathbb{W}^\theta,\alpha,\beta,\delta\) are precisely the output nodes of \(G\).
\[
[f | g] = (F * \{0,0,0 \} * G) \upharpoonright_{O_f \cup O_g}
\]
where \[
\|_{\theta,\alpha,\beta} : I \rightarrow \{I_f, I_g\} \quad \text{where}
\]
\[
\begin{array}{c}
I_f \quad \alpha \quad \beta \\
\downarrow \\
I_g \\
\end{array}
\Rightarrow
\begin{array}{c}
F \\
G \\
\end{array}
\]

\[
F := \{f\} \cup V_f \rightarrow O_f \\
G := \{g\} \cup V_g \rightarrow O_g
\]

(a)

\[
[f > x > g] = (F * \{x\}_{0,0,0} * G_1 \cdots * G_n) \upharpoonright_{\bigcup_{i=1}^{n} O_{gi}}
\]
where \[
\|_{\theta,\alpha,\beta,\gamma} : I \rightarrow \{I_f, I_g, \ldots, I_{fn}, X_1, \ldots, X_n\} \\
\]
\[
\begin{array}{c}
I_f \quad \alpha_1 \quad \beta_1 \\
\downarrow \\
I_g \quad \alpha_n \quad \beta_n \\
\end{array}
\Rightarrow
\begin{array}{c}
F_1 \\
G_1 \\
\ldots \\
G_n \\
X_1 \\
\ldots \\
X_n \\
\end{array}
\]

\[
F := \{f\} \cup V_f \rightarrow \{O_{f1}, \ldots, O_{fn}\} \\
\]
for \(j \in \{1, \ldots, n\}:
\]
\[
G_j := \{x_j/x\}_{g_j} : \{I_{gj}\} \cup V_{gj} \rightarrow O_{gj}
\]
\(x_j\) is a fresh variable name

(b)

\[
[g \text{ where } x \in f] = (W_{\theta,\alpha,\beta,\gamma} * F * G) \upharpoonright_{O_g}
\]
where \[
\|_{\theta,\alpha,\beta,\gamma} : I \rightarrow \{I_f, I_g, X\} \\
\]
\[
\begin{array}{c}
I_f \quad \alpha \\
\downarrow \\
I_g \\
\end{array}
\Rightarrow
\begin{array}{c}
F \\
G \\
X \\
\end{array}
\]

\[
F := \{f\} \cup V_f \rightarrow \{O_{f1}, \ldots, O_{fn}\} \\
G := \{g\} : \{I_g\} \cup V_g \rightarrow O_g
\]

(c)

\[
[M(x_1, \ldots, x_n, v_1, \ldots, v_m)] = (M_{\theta,\alpha,\beta,\gamma} \cdot V_{0,0,0} * M_k) \upharpoonright_{k}
\]
where \[
\|_{\theta,\alpha,\beta,\gamma} : I \rightarrow \{X_1, \ldots, X_n, ?k\} \\
\]
\[
\begin{array}{c}
I \\
\downarrow \\
X_1 \\
\ldots \\
X_n \\
0 \quad \downarrow \\
0 \quad \downarrow \\
0 \\
\end{array}
\Rightarrow
\begin{array}{c}
M_k \\
\beta \\
\end{array}
\]
\[
V = \{v_1, \ldots, v_m\}
\]
\(x_1, \ldots, x_n\) are variables \(v_1, \ldots, v_m\) are values
\[
\begin{array}{c}
\text{for } j \in \{1, \ldots, n\}:
\end{array}
\]
\[
\begin{array}{c}
t_j := \begin{cases} 0 & \text{if } \theta = \alpha_j = 0 \\ 1 & \text{otherwise} \end{cases}
\end{array}
\]
\[
M_k : M_k \rightarrow ?k := \text{Reo component of site } M \\
k \text{ is fresh}
\]

(d)

Figure 6: Definition of the encoding function \([\cdot]\) from Orc into Reo, where \(\alpha, \beta, \theta,\) and \(\delta\) stand for the value of buffers (FIFO's or One Time nodes), whose value can be 0 (no value), 1 (some value), or a constant value. Nodes in the environment are associated with the variable with the same name in lower case. For example, node \(X_1\) in Reo corresponds to variable \(x_1\) in Orc.
To make the behaviour easier to describe, we factor $W_{\theta,\alpha,\beta,\delta}$ into two different connectors corresponding to unconnected parts of the main connector:

$$W_{\theta,\alpha,\beta,\delta} = W_{\theta,\alpha,\beta} \ast W_{\delta}^\ast,$$

where $W_{\theta,\alpha,\beta} : I \to \{I_1, I_2\}$ and $W_{\delta}^\ast : \{O_{f1}, \ldots, O_{fn}\} \to \{X\}$. Initially $\theta = \alpha = \beta = \delta = 0$. The connector $W_{\theta,\alpha,\beta}$ is exactly the same as connector $P$ and, therefore $W_{0,0,0}^\ast$ behaves as $\|_{0,0,0}$. The possible behaviour of $W_0^\ast$ is the following:

$$W_0^\ast \xrightarrow{Ox(v_k)} W_1^\ast,$$

where $O \subseteq \{O_{f1}(v_1), \ldots, O_{fn}(v_n)\}$, and $v_k \in \{v_1, \ldots, v_n\}$ such that $O_{fk}(v_k) \in O$. The choice of which node in $\{O_{f1}(v_1), \ldots, O_{fn}(v_n)\}$ will write into node $X$ is made by connector $P$ (see Table 2). This means that $W_{0,0,0,0}^\ast \ast \ast F \ast G$ behaves similarly to $\|_{0,0,0} \ast \ast \ast F \ast G$, except that the output nodes of $F$ trigger the connector $W_{\delta}^\ast$. The output nodes of $W_{\theta,\alpha,\beta,\delta}$ are restricted to the output nodes of $F$. This connector allows data to flow to node $X$, which is part of the environment of $G$ and is made available to this instance. Note that the difference between $\parallel_{\theta,\alpha,\beta}$ and $W_{\theta,\alpha,\beta,\delta}$ is captured, in part, by the combinator $W_{\delta}^\ast$.

**Site call:**

$$M_{\theta,\Sigma,\nu,\beta,\delta} : \{I, X_1, \ldots, X_n, ?k\} \to \{\mu_k, !k\}$$

The connector is illustrated in Fig. 3(d). The main idea is to tuple all the arguments required by site $M$ before the site is executed. As in previous cases, we factor this connector into two different connectors corresponding to unconnected parts of $M$ to make the behaviour easier to describe:

$$M_{\theta,\Sigma,\nu,\beta,\delta} = M_{\theta,\Sigma,\nu} \ast M_{\beta,\delta}.$$
Lemma 2.4. Let \( h \in \text{Orc}^- \). Each node in Names([h]) can be fired at most once.

\( \text{Proof.} \) Recall that the One Time node, depicted as \( \circ \), only allows data to flow once. We can verify that, in every rule of the translation, the input nodes are connected to a One Time \( \circ \) fired once. The proof for the output nodes follows by induction on the structure of a site call \( M \).

**Lemma 2.3.** Let \( h \in \text{Orc}^- \). Each node in Names([h]) can be fired at most once.

**Definition 2.5.** Let \( f \in \text{Orc}^- \) and \( F = [f] : \{I_f\} \cup V_f \rightarrow O_f \). We define two partitions of reachable configurations of \( F \):

- \( F^-=\{F' \mid F \xrightarrow{a_1} \ldots \xrightarrow{a_n} F' \land \{I_f\} \neq \text{nodes}(a_1 \cup \ldots \cup a_n) \land n \geq 0\} \)
- \( F^+=\{F' \mid F \xrightarrow{a_1} \ldots \xrightarrow{a_n} F' \land \{I_f\} \in \text{nodes}(a_1 \cup \ldots \cup a_n) \land n \geq 1\} \)

Figure 7: Multiple-output encoding of \( (CNN(uk,d) \mid BBC(uk)) \rightarrow email(me,x) \)
The first set consists of the configurations of $F$ after zero or more steps up to when the input node is fired, and the second set consists of the possible configurations after the input node has fired. We say that a connector $F'$ is reachable from $F$ if $F' \in F^{-I} \cup F^{+I}$. Combining Definition 2.5 with Lemmas 2.3 and 2.4 we arrive at the following corollary.

**Corollary 2.6.** Let $f \in \text{Orc}^-$ and $F \equiv [f]: \{I_f\} \cup V_f \rightarrow O_f$. Then:

- Assume $H \in F^{-I}$. If $H \not\sim H'$, then $\text{nodes}(a) \cap O_f = \emptyset$. Furthermore, for all $H''$ reachable from $F$, if $H'' \not\sim H$, then $H'' \in F^{-I}$, $I_f \not\in \text{nodes}(a)$, and either $\bar{a} = \emptyset$ or $\bar{a} = \tau$.

- If $H \in F^{+I}$ and $H \not\sim H'$, then $I \not\in \text{nodes}(a)$ and $H' \in F^{+I}$.

The main result of this section is the existence of a weak bisimulation between an Orc expression and its translation into Reo. We define the notion of weak transition and weak bisimulation inspired by Milner's definition of weak bisimilarity [Mil99].

**Definition 2.7.** Let $Q$ and $Q'$ be Orc expressions (or Reo connectors), and $a$ be a set of actions. We write $Q \sim Q'$ to denote $Q(\bigwedge) \xrightarrow{a'} (\bigwedge)'^* Q'$, whenever $a\{\tau\} = a'\{\tau\}$, i.e., $Q$ evolves to $Q'$ after performing a transition $a'$ and any number of $\tau$ transitions before or after $a'$. When $a = \{\tau\}$, then $\Delta = (\bigwedge)'^*$.

**Definition 2.8.** We say $f \sim C$, where $f$ is an Orc expression, $C$ is a connector configuration, and $a \subseteq \text{BaseEvents}$, we have:

(i) if $f \xrightarrow{a} f'$, then $\exists b, C'$ such that $\hat{b} = a$, $C \xrightarrow{b} C'$ and $f' \approx C'$; and

(ii) if $C \not\approx C''$, then there is an expression $f'$ such that $f \xrightarrow{a} f'$ and $f' \approx C'$.

We say $f$ is weakly bisimilar to $C$, written $f \sim C$, if there is a weak bisimulation $\approx$ such that $f \approx C$.

Lemma 2.9 captures that substituting a variable in an Orc expression is the same as triggering the input node associated with the corresponding variable.

**Lemma 2.9.** Let $h \in \text{Orc}^-$ and $h_v \triangleq [v/x]h$, where $x$ is a free variable in $h$, and $v$ is a data value. Substitution does not change the behaviour of the translation, i.e.,

$$\text{If } h \sim [h] \text{ and } [h] \xrightarrow{X(v)} H_v \text{ then } h_v \sim H_v, $$

where $H_v$ is obtained by sending value $v$ in node $X$.

**Proof Outline.** We start by verifying that the only relevant case is when $h = M(\bar{v})$, and $x \in \bar{v}$, because that is the only place where $v$ can be used. We can prove that, in this case, the possible behaviour of $[h_v]$ is the same as $H_v$, concluding that $h \sim [h]$ implies $h_v \sim H_v$.

Theorem 2.10 is the main result of this section, which relates Orc expressions with their Reo encodings. The proof uses the lemmas introduced above, in particular, Corollary 2.6 deals with inductive applications of the construction, and Lemma 2.9 handles the substitution of variables.

**Theorem 2.10.** Let $h \in \text{Orc}^-$. We claim that $h \sim [h] : I \cup V \rightarrow O$, where $V$ contains only nodes associated to free variables of $h$.

**Proof Outline.** This theorem follows by induction on the structure of $h$. We define the relation $\approx$ inductively for each constructor of Orc as follows. We omit the prove that $\approx$ is a weak bisimulation, which can be done by analysing every possible element of $\approx$.

- $h = M(x_1, \ldots, x_n, v_1, \ldots, v_m)$

Where $x_1, \ldots, x_n$ are variables and $v_1, \ldots, v_m$ are values. We assume that the last variables are always the first to be instantiated.
• \( h = f \mid g \)

Let \( F = \{f\} \) and \( G = \{g\} \). By the induction hypothesis there are two bisimulations, \( \approx_f \)
and \( \approx_g \), such that \( f \approx_f F \) and \( g \approx_g G \).

\[
\approx = \{(f' \mid g', F' \parallel v_0,0,0 \parallel G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^I \land G' \in G^I\}
\]

\[
\cup \{(f' \mid g', F' \parallel v_0,0,0 \parallel G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^{I+1} \land G' \in G^{I+1}\}
\]

\[
\cup \{(f' \mid g', F' \parallel v_0,0,0 \parallel G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^I \land G' \in G^{I+1}\}
\]

\[
\cup \{(f' \mid g', F' \parallel v_0,0,0 \parallel G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^{I+1} \land G' \in G^I\}
\]

• \( h = f \gg g \)

Let \( n = \#f \), and \( 1 \leq j \leq n \). Also let \( F = \{f\} \) and \( G_j = \{g\} \). By the induction hypothesis there are \( n + 1 \) bisimulations, \( \approx_f \) and \( \approx_{g_j} \), such that \( f \approx_f F \) and \( g \approx_{g_j} G_j \).

\[
\approx = \{(f' \gg g', F' \parallel v_0,0,0 \parallel G'_1 \parallel \ldots \parallel G'_n) \mid f' \approx_f F' \land g' \approx_{g_j} G'_j \land F' \in F^I \land G'_j \in G^I\}
\]

\[
\cup \{(f' \gg g', F' \parallel v_0,0,0 \parallel G'_1 \parallel \ldots \parallel G'_n) \mid f' \approx_f F' \land \land F' \in F^{I+1} \land G'_j \in G^{I+1}\}
\]

\[
\cup \{(f' \gg g', F' \parallel v_0,0,0 \parallel G'_1 \parallel \ldots \parallel G'_n) \mid f' \approx_f F' \land \land F' \in F^I \land G'_j \in G^{I+1}\}
\]

\[
\cup \{(f' \gg g', F' \parallel v_0,0,0 \parallel G'_1 \parallel \ldots \parallel G'_n) \mid f' \approx_f F' \land \land F' \in F^{I+1} \land G'_j \in G^I\}
\]

3 Encoded Reo into Orc

The encoding of Orc into Reo is local, in the sense that each Orc combinator and each site
call in an Orc expression can be independently translated, and their composition yields the
encoding of the main expression. On the other hand, we anticipate that the encoding of Reo
into Orc would be global, since each Reo connector needs to be considered as whole. Note
that such an encoding will not be compositional. For example, \( C(Sync_{A,B} \bowtie Merge_{B,C,D}) \)
would not correspond to \( C(Sync_{A,B}) \parallel C(Merge_{B,C,D}) \), since in the second case it is possible
for data to flow from \( A \) to \( B \), whereas in Reo this could not occur if there was also data
flowing from \( C \) to \( D \). The encoding would become roughly the implementation of one of the
known algorithms to combine the synchronous constraints imposed by Reo primitives, such as
Connector Colouring [CC07].

The expressiveness of Orc is closely related to the set of base primitive sites considered.
An example use of more complex primitive site calls can be found in the work by Cook et al.
[CPMR86], where the authors encode into Orc the set of workflow patterns proposed by
Van der Aalst [AHK98]. A similar approach could be attempted to encode Reo into Orc,
using complex primitive site calls that can synchronize with each other, but still the encoding
will not be compositional. We also analysed a synchronous semantics of Orc, presented by
Cook et al. [CM07], where all events other than external response are processed as soon as possible. This allows, for example, to impose an order on how two primitive sites are called, which was not possible with the asynchronous semantics. However, it is still not possible to describe atomic blocks that can either succeed or rollback if one of the actions is not possible. A stronger model, for example, a transactional model, is required to capture the synchrony imposed by \( R \).

Formal comparisons between synchronous and asynchronous communication have been explored in the context of the \( \pi \)-calculus. The asynchronous \( \pi \)-calculus, or \( \pi_a \)-calculus for short, is a subset of the \( \pi \)-calculus with no mixed choice operator, and whose syntax mandates that a process finishes after outputting a message in a channel. To have (polyadic) synchronisation in the \( \pi \)-calculus means that it is possible to constrain a fixed-sized tuple of more than one channel so that each element can be executed only if all the other elements of this tuple can also be executed. This notion of synchrony is closely related to synchrony in \( R \), since \( R \) allows for the definition of constraints on the firing of more than one port in the same step. Unlike \( R \), the \( \pi \)-calculus does not propagate synchrony through composition.

In the remaining of this section we explore expressiveness results in the context of the \( \pi \)-calculus, in particular, the work by Palamidessi where she proves that the (synchronous) \( \pi \)-calculus cannot be encoded in the \( \pi_a \)-calculus [Pa97].

**Sketch of a non-encodability result via \( \pi \)-calculus**

We sketch a proof of the non-encodability of the \( R \) into full \( \mathcal{O}c \), according to a reasonable notion of encodability, reusing the results of Palamidessi wherein she compares the expressiveness of the (synchronous) \( \pi \)-calculus and the \( \pi_a \)-calculus [Pa97]. Our argument relies on the assumption that different sites in \( \mathcal{O}c \) can only communicate with each other through \( \mathcal{O}c \)'s combinators. Without this assumption it would be possible to use arbitrarily complex sites to produce the desired coordination.

Palamidessi proved that there is no uniform encoding from the \( \pi \)-calculus to the \( \pi_a \)-calculus that preserves a reasonable semantics. She defines an encoding to be uniform if it preserves distribution and permutations, i.e., if the parallel operator on the \( \pi \)-calculus is encoded into the parallel operator on the \( \pi_a \)-calculus, and if for each renaming of variables before the encoding there is some permutation on the encoded process such that certain conditions hold. A reasonable semantics is characterised by distinguishing two processes \( P \) and \( Q \) whenever \( P \) can produce actions on certain intended channels that cannot be produced by \( Q \). In her proof Palamidessi uses the argument that the leader election on a symmetric system cannot be solved using the \( \pi_a \)-calculus because the symmetry cannot be broken, while it is possible in the \( \pi \)-calculus, mainly because of the existence of the guarded choice construct.

It is not immediately clear how these results apply in our setting, as it is difficult to know the meaning of preserving the parallel operator on an encoding of \( R \) into \( \mathcal{O}c \). We sketched our proof as follows. We present an encoding of \( \mathcal{O}c \) into the \( \pi_a \)-calculus, allowing us to conclude that the symmetric leader election problem cannot be solved in \( \mathcal{O}c \). Note that we do not prove the correctness of the encoding. We then conclude by giving a brief explanation on how the leader election problem can be trivially solved in \( R \).

**Encoding \( \mathcal{O}c \) into the \( \pi_a \)-calculus**

In this section we present briefly the syntax of the \( \pi_a \)-calculus, we define an encoding function \( \Pi \) from \( \mathcal{O}c \) to the \( \pi_a \)-calculus, and we translate our running example into the \( \pi_a \)-calculus.

The syntax of a \( \pi_a \)-calculus process is defined as follows, where \( x \) is a channel, and \( y \) is the message (or a tuple of messages) sent over a channel, which can be again a channel.

\[
\text{Processes} \quad P ::= \pi(y) \mid x(y).P \mid (\nu x)P \mid P|P \mid !P
\]

Informally \( \pi(y) \) represents the output of message \( y \) through channel \( x \), \( x(y).P \) represents the reception of message through channel \( x \), which becomes bounded to name \( y \), and evolves to process \( P \), \( (\nu x)P \) represents the creation of a new channel name \( x \), which can occur in \( P \), \( P|P \) stands for the parallel execution of two processes, that can communicate over common channels, and \( !P \) represents the replication of process \( P \), i.e., an unbounded parallel execution of copies of the same process \( P \). Note that replication in the \( \pi_a \)-calculus has been proven to
be equivalent to a set of recursive processes. We omit the formal semantics of the \(\pi\)-calculus, which can be easily found in the literature \[\text{Pa97, CM03} \].

The general idea is, given an \(\text{Orc}\) expression \(f\) and a channel name \(s\), produce its translation \(P\) in the \(\pi\)-calculus such that \(P\) can be executed by sending a message \(\text{out}\) through channel \(s\). We denote it by \(\langle f, s \rangle = P\). The message \(\text{out}\) is the channel used by the resulting expression to output the possible results of the corresponding \(\text{Orc}\) expression. The names of variables in \(\text{Orc}\) are used as the names of the channels in the \(\pi\)-calculus, where the corresponding value is passed. We present the encoding in Fig\(8\), where we use the notation \((\nu x)(\nu y)(\nu z)P\) to denote \((\nu x y z)P\).

\[
\langle f \mid g, \text{start} \rangle = \text{start}(\text{out}).(\nu \text{startf} \text{startg})
\]
\[
(\langle f, \text{startf} \rangle | \text{startf}(\text{out})
| (\langle g, \text{startg} \rangle | \text{startg}(\text{out}))
\]

\[
\langle \text{g where } x \in f, \text{start} \rangle = \text{start}(\text{out}).(\nu \text{startf} \text{startg} \text{outf} x)
\]
\[
(\langle f, \text{startf} \rangle | \text{startf}(\text{out})
| (\langle g, \text{startg} \rangle | \text{startg}(\text{out})
| \text{outf}(x').!\pi(x'))
\]

\[
\langle f >x > g, \text{start} \rangle = \text{start}(\text{out}).(\nu \text{startf} \text{startg} \text{outf})
\]
\[
(\langle f, \text{startf} \rangle | \text{startf}(\text{out})
| (!\text{outf}(x').(\nu \text{startg} x)
| (\langle g, \text{startg} \rangle | \text{startg}(\text{out}) | !\pi(x'))))
\]

\[
\langle M(p), \text{start} \rangle = \text{start}(\text{out}).p(p').M(p', \text{out})
\]

\[
\langle E(p), \text{start} \rangle = \text{start}(\text{out}).E(p, \text{out})
\]

Figure 8: Translation of \(\text{Orc}\) into \(\pi\)-calculus

The general definition of the encoding of an \(\text{Orc}\) expression \(f\) into the \(\pi\)-calculus, after introducing a set \(\text{Defs} = \{D_1, \ldots, D_n\}\) of definitions of auxiliary \(\text{Orc}\) expressions, is as follows:

\[
\langle f, \text{Defs, start} \rangle = \langle f, \text{start} \rangle | \langle D_1 \rangle_{\text{def}} | \cdots | \langle D_n \rangle_{\text{def}} | \text{ Sites}
\]

where \(\langle E(p) \rangle_{\text{def}} = !\langle E(p, \text{out}).(\nu \text{startf})(\langle f, \text{startf} \rangle | \text{startf}(\text{out})) \rangle\). Furthermore, we assume \(\text{Sites}\) to consist on several processes in parallel, one for each site \(M\) used in \(f\) and in \(\text{Defs}\), such that it can always receive a message through channel \(M\) to start the computation corresponding to the site call to \(M\). Note that the parameters of \(M\) will be all the arguments of site \(M\), and also the channel that should be used to return the result of the site call.

Fig.\(9\) presents the encoding \(\langle \cdot \rangle\) applied to our running example, where \(\text{start}\) is the name of the channel that needs to be used to start the execution of the translated process.

Separation result

The encoding of \(\text{Orc}\) into the \(\pi\)-calculus, if proved to be correct, shows that \(\text{Orc}\) is not expressive enough to break the symmetry when solving the leader election problem. Carbone and Maffeis \[\text{CM03}\] extended Palamidessi’s result to show that the expressive power of the \(\pi\)-calculus with polyadic synchronisation that can synchronize at most \(n\) channels is less than of the one that can synchronize at most \(n+1\) channels. These results emphasise the idea that \(\text{Reo}\) cannot be encoded in \(\text{Orc}\), because \(\text{Orc}\) is asynchronous whereas \(\text{Reo}\) can synchronize an arbitrary number of ports.

The synchrony and exclusion inherent to \(\text{Reo}\), unlike in \(\text{Orc}\) and the \(\pi\)-calculus, allows the symmetry of a system to be easily broken. Combined with fact that the symmetric leader election problem cannot be solved in \(\text{Orc}\), this is enough to show the non-encodability of \(\text{Reo}\) into \(\text{Orc}\). Although we do not prove formally that the leader election problem can be solved in the context of \(\text{Reo}\), the connector in Fig.\(10\) provides the necessary intuition.
\[
((\text{CNN}(uk, d) \mid \text{BBC}(uk)) > x > \text{email}(me, x), \text{start}) = \\
\text{start}(out), (\nu\text{startf} \text{startg} \text{o}utf \ x)
\]

\[
\begin{align*}
\text{startf}(out). & (\nu\text{startf} \text{startg}) \\
& (\text{startf}(out).\text{uk}(uk').d(d').\text{CNN}(uk', d', \text{out}) \\
& \text{startf}(out) \\
& \text{startg}(out).\text{uk}(uk').\text{BBC}(uk', \text{out}) \\
& \text{startg}(out)
\end{align*}
\]

\[
\begin{align*}
& \text{startf}(\text{out}) \\
& !\text{(outf}(x').(\nu\text{startg} \ x) \\
& (\text{startg}(out).\text{me}(me').x(x').\text{email}(me', x', \text{out}) \\
& \text{startg}(out) | !\text{f}(x')))
\end{align*}
\]

Figure 9: Example of the encoding of \textsc{Orc} into \textit{\pi}_a\text{-calculus}

to understand how it could be solved. This example shows a connector built from three symmetric sub-connectors, each having an input port on the left and an output port on the right, connected with each other. The resulting connector guarantees that, after one step, all the output nodes will have received the same message from exactly one input port \textit{I}_n, chosen non-deterministically. This results from (1) the synchronous replication of each of the messages received, which guarantees that data can only flow in an input node if the same data can also flow in all the output nodes; and (2) from the merge of the messages, that guarantees that the data flowing in each of the outputs can only come from one of the inputs, excluding the possibility of dataflow on the remaining inputs.

Figure 10: Leader election in \textsc{Reo}

4 Discussion

We now compare \textsc{Orc} and \textsc{Reo} on some issues of philosophy and design.

**Focus of Control** In \textsc{Orc}, control lies with the orchestrator: an \textsc{Orc} expression initiates contact with external sites. On the other hand, \textsc{Reo} assumes that control is initiated externally to a connector by a component. The request to write data to or read data from a node is subsequently handled by the connector. This is how \textsc{Reo} coordinates, by controlling when such requests can succeed, though from the perspective of web services, control is inverted.

**Component/Service Instantiation** In \textsc{Reo}, components are attached externally to a connector, whereas \textsc{Orc} can dynamically initiate contact with services. \textsc{Orc} is thus more dynamic, although it is tightly bound to the actual sites being called. These limitations seem easy to lift.

**One-off Interaction vs. Streams** \textsc{Orc} expressions unfold over their life-time, so each piece of syntax is reduced once and each site call is performed once. On the other hand, \textsc{Reo} establishes rigid connections between parties, as it makes the assumption that parties will continuously communicate.
Dynamics As an $Orc$ expression reduces, its ‘configuration’ changes dynamically. For instance, $f >s> g$ creates a new instance of $g$ for each value produced by $f$. This was encoded in $Reo$ by calculating a bound on the number of values produced by $f$ and duplicating the circuitry for $g$. As $Reo$’s connectivity is more or less fixed, and $Orc$ expressions ‘fire’ only once, our encoding introduces a lot of connectors that are used only once. In very recent work by Koehler et al. [KLA07, KCPA08] the authors present how to model dynamic reconfiguration of $Reo$ connectors using a high level approach based on graph transformation techniques. The authors go further with this idea, and propose a framework where dataflow triggers the reconfiguration process. This framework can be the basis for self adapting and dynamically reconfigurable connectors.

Asynchrony vs. Synchrony $Orc$ offers highly asynchronous connectives that gracefully deal with failing sites. $Reo$ is highly synchronous and susceptible to failure. Recall that failure can also be handled with timed connectors, as mentioned in Section 2.2 although this solution is less transparent, as failure must explicitly be handled. In principle, synchrony (or in any case, atomicity) can form the basis of high-level abstractions.

5 Related Work

Bruni et al. [BMT06] present a static encoding of $Orc$ into Petri nets. However, their encoding is not faithful to the $Orc$ model, as it assumes that each primitive site returns either a valid value or some value to state that it will not return a value. $Orc$, on the other hand, gracefully deals with sites which do not return values. Our encoding into $Reo$ more accurately handles the absence of dataflow. Our encoding also considers the data values passed around, in contrast to Bruni et al.’s encoding, which passes only Petri net tokens. Bruni et al. also present an encoding of full $Orc$ into the $Join$ calculus—an expressive calculus for concurrent processes developed at INRIA. The $Join$ calculus provides a simple support for distributed programming, intentionally avoiding some communication constructs that are difficult to implement in a distributed setting. This calculus supports some synchrony, by introducing patterns that correspond to multiple events which must all be present for the pattern to be recognized. However, the $Join$-calculus is not highly synchronous like $Reo$, as it does not propagate synchrony through composition. The precise relationship between the $Join$ calculus and $Reo$ is left for future work.

Many other coordination languages exist, and these are compared in some earlier surveys [PA98, AHM96]. We can fairly safely say that few (coordination) languages offer the degree of synchrony that $Reo$ offers. Obvious exceptions are synchronous languages such as Esterel [Ber00]. These languages are useful for programming reactive systems, though they lack non-determinism, and in general seem not to be directly useful for coordinating distributed systems. To remedy this situation, the GALS (globally asynchronous, locally synchronous) model [Cha84, DMK+06] has been adopted, whereby local computation is synchronous and communication between different machines is asynchronous.

As with $Orc$, the GALS model adopts the arguably correct view that distributed systems must be programmed asynchronously. $Reo$ is also able to express such distinctions, and more, through the many choices of synchrony or asynchrony—the result depends upon how a connector is deployed to a distributed system. $Reo$ claims that instead of synchrony, it is really implementing atomicity, and hence a basic form of transaction [Arb04]. This has not yet been convincingly demonstrated.

A method for comparing expressiveness was proposed by de Boer and Palamidessi [BP94], where they introduce a notion of language embedding refined with some “reasonable” conditions. Brogi and Jaquet used this method to compare coordination models with Linda-like operations and a shared dataspace [BJ03]. Our attempt to prove that $Reo$ could not be encoded is based on a result of Palamidessi where she compares the expressiveness of the $\pi$-calculus and the $\pi_a$-calculus, which follows a similar approach to [BP94], but not for the same class of languages. However, it is not clear how this result could be used to prove the encodability of $Orc$ into $Reo$.

The idea of reusing the expressiveness results with the $\pi$-calculus was already successfully used to compare expressiveness in other contexts. A good example is the work from Philips and Vigliotti [PV04], where they compare the expressiveness of ambient calculi against different
dialects of the \( \pi \)-calculus, providing also a good overall perspective on existing expressiveness results with respect to the \( \pi \)-calculus.

6 Conclusion and Future Work

We have compared \( \text{Orc} \) and \( \text{Reo} \), by encoding the non-recursive fragment of \( \text{Orc} \) into \( \text{Reo} \), by discussing the failure of the encoding in the other direction, and by comparing a number of design decisions. \( \text{Orc} \) is highly asynchronous and deals well with failure. \( \text{Reo} \) supports a high degree of synchrony, and potentially high-level abstractions. An obvious omission is a comparison of the efficiency of the two models. Unfortunately, both implementations are too preliminary for this to have any real meaning. The extension of our encoding to full \( \text{Orc} \) requires either recursively-defined or dynamically reconfigurable \( \text{Reo} \) connectors. These extensions to \( \text{Reo} \) are interesting on their own, and are the subject of future work.

Note that, despite the expressiveness power provided by \( \text{Reo} \), we can still have feasible implementations in asynchronous networks. This is mainly because problems such as the leader election can be solved in real networks by assuming that the system is not completely symmetric, \( i.e. \), we can assume unique identifiers exist for every entity in a network which can be used to break the symmetry.

References


## A Proofs

*Proof. (Lemma 2.4)* The proof follows by induction on the structure of $h$:
• \( h = f | g \)

We know that \([h] = (F \parallel \ll_0,0,0 \parallel G) \parallel [\alpha_f \cup \alpha_g] \), where \( F := [f] \) : \([I_f] \cup V_f \rightarrow \bar{O}_f \) and \( G := [g] : \([I_g] \cup V_g \rightarrow \bar{O}_g \). Therefore \([h] : \([I] \cup V_f \cup V_g \rightarrow \bar{O}_f \cup \bar{O}_g \) The only possible

• reduction step of \( \ll_0,0,0 \) is by \( I(v) \) to \( \ll_1, v, v \), where \( v \) is a data value, and only after this

• reduction step the input nodes of \( F \) and \( G \) can be fired. Since we know by induction hypothesis that the property is valid for \( F \) and \( G \), then we conclude that node \( I \) is always triggered before the outputs. By induction hypothesis we also know that nodes in \( V_f \) and \( V_g \) can still be fired by \( F \) or \( G \) before node \( I \) is fired. We conclude that only nodes in \( V_f \cup V_g \) can be fired by \([h] \) before node \( I \) is fired.

• \( h = f \triangleright x > g \)

We know that \([h] = (F \parallel \ll x_{[0,0,0]} \parallel G_1 \times \ldots \times G_n) \parallel [\alpha_1 \cup \ldots \cup \alpha_n] \), where \( \ll x_{[0,0,0]} \) = \( \{I_1, O_1, \ldots, O_f\} \rightarrow \{I_f, I_g, X\} \). \( F := [f] : \{O_1, \ldots, O_f\} \) and \( G_j := [g_j] : \{I_f, I_g\} \cup V_g \rightarrow O_g \), for any \( j \) between 1 and \( n \). Therefore \([h] : \{I\} \cup V_f \cup V_g \rightarrow O_g \), and by induction hypothesis we know that the output nodes of \( F \) and \( G \) cannot occur until their input nodes are fired, and consequently the nodes \( X_1, \ldots, X_n \) cannot be fired either. The only possible boundary nodes of \([h] \) that can be fired are \( I \) and nodes in \( V_f \cup V_g \). Note that firing of nodes in \( V_f \cup V_g \) can occur before their input node is fired. Node \( I_f \) is only fired when node \( I \) is fired (by definition of \( \ll x_{[0,0,0]} \)), and only after the output nodes of \( F \) are fired can the input nodes of \( G_j \) be fired.

• \( h = g \) where \( x \in f \)

This case is very similar to when \( h = f | g \). The same arguments presented for that case are also valid here: by induction hypothesis we can also claim that \( F \) and \( G \) can only fire their output nodes after their input nodes are fired, which only occurs after node \( I \) is fired. The difference with respect to \( f | g \) is that we only need to consider the output nodes of \( G \) and \( G \) may have an input node labelled by \( X \), which will be dependant on the flow of data on one of the output nodes of \( F \). The only possible observation of \([h] \) until node \( I \) is fired correspond to dataflow in \( V_f \) or in \( V_g \setminus X \).

• \( h = M(x_1, \ldots, x_n, v_1, \ldots, v_m) \)

We know that \([h] = (M_{0,0,0} \parallel V_0, 0, 0 \parallel M_k) \parallel [k] \), where \( k \) is fresh, \( M_k \) is a \( Rneo \) component corresponding to site \( M \), and \( V = \{v_1, \ldots, v_m\} \). We can derive that \([h] : \{I, X_1, \ldots, X_n, ?k\} \rightarrow \{k\} \). The component \( M_k \) can only be executed when node \( M_k \) is fired, which can only occur in the same synchronous step as the firing of node \( I \) and of the output ends of the FIFO1 channels associated to the arguments of \( M \) (recall that the component \( T_{n+1} \) is synchronous). Initially the only possible behaviour of \([h] \) is to fire input nodes other than \( I \), i.e., nodes in \( \{X_1, \ldots, X_n\} \), until every FIFO1 channel associated to each variable is full. When this occurs, the only possible behaviour is:

\[ M_{0,0,0} \parallel v_{(\alpha_1, \ldots, \alpha_n)}, \ldots, v_{(\alpha_1, \ldots, \alpha_n)} \parallel 0, 0 \rightarrow \frac{I(y), M_k(v_{(\alpha_1, \ldots, \alpha_n)}, v_{1, \ldots, v_m})}{M_{1,0,0}, 0, 0, 0, 0, 0} \]

triggering the execution of site \( M \), and only after the site returns a value in node \( ?k \) the connector evolves, flowing data in its only output node \( !k \).

\[ \square \]

Proof. (Theorem 2.10)

In Section 2.4 we express a weak bisimulation between a non-recursive \( Orc \) expression \( f \) and its encoding into \( Rneo [f] \). Here we present an exaustive proof that \( f \sim [f] \), by presenting a valid bisimulation.

For this proof we use Corollary 2.6, which guarantees that only nodes in the environment can succeed until the input node is fired, and that each input node is fired at most once. The proof follows by induction on the structure of \( h \).

• \( h = M(v_1, \ldots, v_m) \), where \( v_1, \ldots, v_m \) are values.

Let \( v = \langle v_{1, \ldots, v_m} \rangle \). The only possible reduction of \( h \) is:

\[ M(v) \xrightarrow{M_k(v)} ?k \xrightarrow{kHz'} \xrightarrow{let(v') \xrightarrow{kHz'}} 0, \]

23
where \( v' \) is the value returned by site \( M \). The only possible reduction of \( [h] \) is:

\[
\begin{align*}
M_0(\cdot, v, 0, 0) &\cdot M_k \xrightarrow{I(x), \eta(x)} M_1(\cdot, 0, 0, 0, 0) \cdot M_k \\
&\xrightarrow{\tau(v')} M_1(\cdot, 0, 0, 1, v') \cdot M_k
\end{align*}
\]

Since \( \{ I(x), M_k(v) \} = \{ M_k(v), \tau \} \), \( \{ ?k(\tau') \} = \{ k' \} \), and \( \{ !k(\tau') \} = \{ \tau' \} \), we can define a weak bisimulation \( R_M = \{ (M(v), M_0(\cdot, v, 0, 0) \cdot M_k), (\eta, M_1(\cdot, 0, 0, 0, 0) \cdot M_k), (let(v'), M_1(\cdot, 0, 0, 1, v') \cdot M_k), (\tau, M_1(\cdot, 0, 0, 0, 0), 1, v' \cdot M_k) \} \), which allows us to conclude that \( h \sim [h] \).

Note that the value \( k \) referred in both systems is the same value. This means that the translation of a primitive site does not choose any fresh \( k \) but the exact same value as the reduction semantics of \( \text{Orc} \). Since we need one different value for each instance of the site \( M \) in both reduction semantics, the value will still be fresh.

- \( h = M(x_1, \ldots, x_n, v_1, \ldots, v_m) \), where \( x_1, \ldots, x_n \) are variables.

To make the explanation easier, we will assume that the arguments of \( M \) are sorted: the first \( n \) arguments are variables, and the following \( m \) arguments are values. Furthermore, we assume that the last variables are always the first to be instantiated. Since the evaluation of \( M \) is strict, then the only possible behaviour is to instantiate variables, followed by data values. We consider the application of a substitution to be an internal action:

\[
M(x_1, \ldots, x_n, v_1, \ldots, v_m) \xrightarrow{\pi} [v'_1/x_1, \ldots, v'_n/x_n].M(x_1, \ldots, x_n, v_1, \ldots, v_m)
\]

We can label this action by \( \tau \) because, in \( \text{Orc} \) semantics, when a substitution occurs it is either labelled by \( \tau \), or ignored if some other action also occurs.

In this case \( [h] = M_0(\cdot, 0, 0, 0, 0) \cdot M_k : (I, X_1, \ldots, X_n) \rightarrow O \). Equivalently to the reduction of the \( \text{Orc} \) expression, we have:

\[
M_0(\cdot, 0, 0, 0, 0) \cdot M_k \xrightarrow{X_1(v'_1), \ldots, X_n(v'_n)} M_0(\cdot, 0, 0, v'_1, \ldots, v'_n) \cdot (v_1, \ldots, v_m) \cdot 0 \cdot M_k
\]

Note that \( \{ X_1(v'_1), \ldots, X_n(v'_n) \} = \{ \tau \} \). We can then define a relation

\[
R_M' = \{ (M(x_1, \ldots, x_{i-1}, v'_1, \ldots, v'_i, v_1, \ldots, v_m), M_0(\cdot, 0, 0, v'_1, \ldots, v'_n), (v_1, \ldots, v_m), 0 \cdot M_k) \mid 1 \leq i \leq n \}
\]

\[
\cup \{ (\eta, M_1(\cdot, 0, 0, 0, 0) \cdot M_k), (let(v'), M_1(\cdot, 0, 0, 0, 0), 1, v' \cdot M_k), (\tau, M_1(\cdot, 0, 0, 0, 0), 1, 0 \cdot M_k) \}
\]

The second part of \( R_M' \) is a bisimulation for the same reasons we presented to show that the relation \( R_M \) is a bisimulation, defined in the first case of the proof. The main difference with \( R_M \) is that the \( R_M' \) has all the possible combinations for when there are variables that are not instantiated. In this case \( (h, [h]) \in R_M' \), and the fact that the only possible behaviour of an expression with the same format as \( h \) and its translation is the behaviour described before yields that \( R_M' \) is in fact a bisimulation.

- \( h = f \mid g \)

Let \( F = [f] \) and \( G = [g] \). We know by the induction hypothesis that there exist two bisimulations, \( \approx_f \) and \( \approx_g \), such that \( f \approx_f F \) and \( g \approx_g G \). Let \( v \) be a data value. Based on these bisimulations, we define \( \approx \) such that \( h \approx [h] \):

\[
\approx = \{ (f' \mid g', F' \mid \leq_{0, 0, 0} G'), (f' \mid g', F' \mid \leq_{1, v, v} G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^{-1} \land G' \in G^{-1} \}
\]

\[
\cup \{ (f' \mid g', F' \mid \leq_{1, 0, v} G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^{-1} \land G' \in G^{-1} \}
\]

\[
\cup \{ (f' \mid g', F' \mid \leq_{1, 0, 0} G') \mid f' \approx_f F' \land g' \approx_g G' \land F' \in F^{-1} \land G' \in G^{-1} \}
\]

We now prove that \( \approx \) is a weak bisimulation as defined in Definition 2.8 by proving that the two implications, numbered by (i) and (ii), are verified for every element in \( \approx \).
- $f' \mid g' \approx F' * ||_{0,0,0} * G'$ where $f' \approx_f F'$, $g' \approx_g G'$, $F' \in F^{-1}$, and $G' \in G^{-1}$.

(i) The possible reduction steps for $f' \mid g'$ are:

$$f' \mid g' \overset{\alpha}{\rightarrow} f'' \mid g' \quad f' \mid g' \overset{\beta}{\rightarrow} f' \mid g'' \quad f' \mid g' \overset{a,b}{\rightarrow} f'' \mid g''$$

Since $f' \approx_f F'$ and $g' \approx_g G'$, we can conclude for each of these cases:

(i.1) Exists $f''$ and $b$ such that $\bar{a} = a, \bar{b} = b, F' \overset{\delta}{\rightarrow} F''$ and $f'' \approx_f F''$. With respect to $F''$ we can still consider two cases:

(ii.1.1) $F'' \in F^{-1}$: The input node is not fired so $F' * ||_{0,0,0} * G' \overset{\varphi}{\rightarrow} F'' * ||_{0,0,0} * G'$ and $f'' \approx F'' * ||_{0,0,0} * G'$ (because $f'' \approx_f F'' \wedge g'' \approx_g G' \wedge F'' \in F^{-1} \wedge G' \in G^{-1}$).

(ii.1.2) $F'' \in F^{-1}$: The input node is fired so $F' * ||_{0,0,0} * G' \overset{\varphi}{\rightarrow} F'' * ||_{1,0,v} * G'$ and $f'' \approx F'' * ||_{1,0,v} * G'$ (because $f'' \approx_f F'' \wedge g'' \approx g', G' \wedge F'' \in F^{+1} \wedge G' \in G^{-1}$).

(i.2) Analogous to (i.1).

(i.3) Exists $f''$, $G''$, $a'$, and $b'$ such that $\bar{a} = a, \bar{b} = b, F' \overset{\delta}{\rightarrow} F''$, $G' \overset{\delta}{\rightarrow} G''$, $f'' \approx_f F''$, and $g'' \approx_g G''$. With respect to $F''$ and $G''$ we have four cases:

(i.3.1) $F'' \in F^{-1}$ and $G'' \in G^{-1}$. The input node of $F''$ is fired and the input node of $G''$ is not fired. This means that $F' * ||_{0,0,0} * G' \overset{\varphi}{\rightarrow} F'' * ||_{1,0,v} * G''$, and $f'' \approx F'' * ||_{1,0,v} * G''$, for some data value $v$ (because $f'' \approx_f F'' \wedge g'' \approx_g G'' \wedge F'' \in F^{+1} \wedge G'' \in G^{-1}$).

(i.3.2) $F'' \in F^{-1}$ and $G'' \in G^{+1}$. Analogous to (i.3.1).

(i.3.3) $F'' \in F^{+1}$ and $G'' \in G^{-1}$. Analogous to (i.3.1).

(i.3.4) $F'' \in F^{+1}$ and $G'' \in G^{+1}$. Analogous to (i.3.1).

(ii) The possible reduction steps for $F' * ||_{0,0,0} * G'$ are:

(ii.1) $||_{0,0,0} * F' * G' \overset{\varphi}{\rightarrow} F' * ||_{1,0,v} * G'$.

We know that $f' \mid g' \approx F' \mid g', f' \approx_f F'$, $g' \approx_g G'$, $F' \in F^{-1}$, and $F' \in F^{-1}$, therefore $f' \mid g' \approx F' * ||_{1,0,v} * G'$.

(ii.2) $F' * ||_{0,0,0} * G' \overset{\varphi}{\rightarrow} F'' * ||_{0,0,0} * G'$.

Since $F' \in F^{-1}$ and the input node of $F''$ is not fired (because of the combinator $||_{0,0}$), then, by Corollary [2.6] $F'' \in F^{-1}$. Since $F'' \overset{\varphi}{\rightarrow} f''$ and $f'' \approx_f F''$, therefore $f' \mid g' \approx F'' \mid g'$ and $f'' \approx F'' * ||_{0,0,0} * G'$.

(ii.3) $F' * ||_{0,0,0} * G' \overset{\varphi}{\rightarrow} F' * ||_{0,0,0} * G''$.

Analogous to (ii.2).

(ii.4) Combination of the previous cases, for which the proves are analogous:

(ii.4.1) and (ii.1) and (ii.2), (ii.1) and (ii.3), and (ii.1), (ii.2) and (ii.3).

- $f' \mid g' \approx F' * ||_{1,0,v} * G'$ where $f' \approx_f F'$, $g' \approx_g G'$, $F' \in F^{-1}$, and $G' \in G^{-1}$.

This case is very similar to the previous one.

- $f' \mid g' \approx F' * ||_{1,0,v} * G'$ where $f' \approx_f F'$, $g' \approx_g G'$, $F' \in F^{+1}$, and $G' \in G^{-1}$.

(i) The possible reduction steps for $f' \mid g'$ are:

$$f' \mid g' \overset{\alpha}{\rightarrow} f'' \mid g' \quad f' \mid g' \overset{\beta}{\rightarrow} f' \mid g'' \quad f' \mid g' \overset{a,b}{\rightarrow} f'' \mid g''$$

Since $f' \approx_f F'$ and $g' \approx_g G'$, we can conclude for each of these cases:

(i.1) Exists $f''$ and $a'$ such that $\bar{a} = a, F' \overset{\delta}{\rightarrow} F''$ and $f'' \approx_f F''$. Since $F' \in F^{-1}$, then by Corollary [2.6] $F'' \in F^{-1}$. Therefore $F' * ||_{1,0,v} * G' \overset{\delta}{\rightarrow} F' * ||_{1,0,v} * G'$ and $f'' \approx F' * ||_{1,0,v} * G'$ (because $f'' \approx_f F' \wedge g'' \approx_g G' \wedge F'' \in F^{+1} \wedge G' \in G^{-1}$).

(i.2) Exists $G''$ and $b'$ such that $\bar{b} = b, G' \overset{\delta}{\rightarrow} G''$ and $g'' \approx_g G''$. With respect to $G''$ we can still consider two cases:

(i.2.1) $G'' \in G^{-1}$: The input node is not fired so $F' * ||_{1,0,v} * G' \overset{\delta}{\rightarrow} F' * ||_{1,0,v} * G''$ and $f'' \approx F' * ||_{1,0,v} * G''$ (because $f'' \approx_f F' \wedge g'' \approx_g G'' \wedge F'' \in F^{+1} \wedge G'' \in G^{-1}$).
where \( X \) by rule (d) in Fig. 6. Connector \( H \)

Proof. We need to prove that \( \{1,0,0 \cdot G' \} \approx \{1,0,0 \cdot G'' \} \) (because \( f' \approx_f F' \) and \( g' \approx_g G'' \)).

(i.2) \( G'' \in G^{+1} \): The input node is fired so \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \).

(i.3) Exists \( F'', G'', a', \) and \( b' \) such that \( a = b = b', F' \approx F'', G' \approx G'' \), \( f'' \approx_f F'' \), and \( g'' \approx_g G'' \). We know \( F'' \in F^{+1} \), as explained in (i.1).

With respect to \( G'' \) we have two cases:

(i.3.1) \( G'' \in G^{-1} \): The input node is not fired so \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \) (because \( f'' \approx_f F'' \) and \( g'' \approx_g G'' \)).

(ii.3) \( G' \in G^{-1} \): The input node is fired so \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \) (because \( f'' \approx_f F'' \) and \( g'' \approx_g G'' \)).

(ii) The possible reduction steps for \( F' * \|_{1,0,0} * G' \) are:

(ii.1) \( \|_{1,0,0} * F' * G' \approx \|_{1,0,0} * G' \).

By Corollary 2.6 \( F'' \in F^{+1} \). Since \( F' \approx F'' \), then \( f' \approx_f f'' \) and \( f'' \approx_f F'' \). Therefore \( f' | g' \approx_f f'' | g'' \).

(ii.2) \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \).

Since \( G' \in G^{-1} \) and the input node of \( G' \) is not fired (because of the combinator \( |_{1,0,0} \)), then by Corollary 2.6 \( G'' \in G^{-1} \). Then \( g' \approx_g g'' \).

(ii.3) \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \).

Since \( G' \in G^{-1} \) and the combinator \( |_{1,0,0} \) evolves to \( |_{1,0,0} \), then the input node is fired, i.e., \( G'' \in G^{-1} \). Then \( g' \approx_g g'' \).

(ii.4) \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \).

Analogous to proofs in (ii.1) and (ii.2).

(ii.5) \( F' * \|_{1,0,0} * G' \approx \|_{1,0,0} * G'' \).

Analogous to proofs in (ii.1) and (ii.3).

The remaining cases are analogous.

Before proving the cases when \( h = f >x> g \) and when \( h = g \) where \( x \in f \), we introduce another necessary lemma.

**Lemma A.1.** (The same as Lemma 2.4) Let \( h \in \text{Orc}^- \) and \( h_v \equiv [v/x].h \). We claim that substitution does not change the behaviour of the translation, i.e.,

\[
If h \sim [h] \quad and \quad [h] \quad \xrightarrow{X(v)} \quad H_v \quad then \quad h_v \sim H_v,
\]

where \( x \) is a free variable in \( h \), \( v \) is a data value, and \( H_v \) is obtained by sending value \( v \) in node \( X \).

**Proof.** We need to prove that \( h_v \sim H_v \), where \( h_v \) is obtained by substituting variable \( x \) by value \( v \), and \( H_v \) is obtained by sending value \( v \) in node \( X \). Recall that we are assuming that each variable name is unique, and node \( X \) is associated with variable \( x \).

Using induction on the structure of \( h \), we can easily verify that, for \( h = f | g, h = f >y> g \), or \( h = g \) where \( y \in f \), where \( x \neq y \), the result follows directly. The node \( X \) can only exist in \( [f] \) and \( [h] \) (and can be fired), and if the property holds for \( f \) and \( g \), then it also holds for \( h \). Therefore, it is enough to consider the case when \( h = M(x_1, \ldots, x_n, v_1, \ldots, v_n) \).

Furthermore, the only relevant case is when \( x \in \{x_1, \ldots, x_n\} \).

We now consider, without lost of generality, that \( h = M(x_1, \ldots, x_n, v_1, \ldots, v_n) \). Then \( h_v = M(x_1, \ldots, x_n, v, v_1, \ldots, v_n) \), and \([h_v] = M_0(0, \ldots, 0).v, v_1, \ldots, v_n) \). Since \( h \) and \( h_v \) are calls to primitive sites, then we can conclude, by the beginning of the proof of Theorem 2.10 that \( h \sim [h] \) and \( h_v \sim [h_v] \). Consider now the connector \([h] \), obtained by rule (d) in Fig. 6. Connector \( H_v \) corresponds to the same connector after data \( v \) flows in input node \( X \), i.e., \( H_v = M_0(0, \ldots, 0).(v_1, \ldots, v_n) \). It is now enough to prove that
the possible behaviour of \([h_v]\) and \(H_v\) is the same, since the assumption that \(h_v \sim [h_v]\) will guarantee that also \(h_v \sim H_v\).

The only difference between \([h_v]\) and \(H_v\) is that in the former the value \(v\) is in FIFO1 channel whose input end is connected to a primitive that never returns data, while in the latter the value \(v\) is in a FIFO1 channel whose input end is connected to a One Time node labelled by one. The One Time node, although also connected to node \(X\), will always guarantee that no flow will occur on any of its ends, which corresponds to the behaviour of the primitive that never returns data and a node that can never flow data again. Therefore \([h_v]\) and \(H_v\) have the same behaviour.

- \(h = f \triangleright x \triangleright g\)

Let \(n = \#f\), and \(1 \leq j \leq n\). Also let \(F = \{f\}\) and \(G_j = \{g\}\). We know by the induction hypothesis that there exist \(n+1\) bisimulations, \(\approx_f\) and \(\approx_{g,j}\), such that \(f \approx_f F\) and \(g \approx_{g,j} G_j\). Based on these, we define \(\approx\) such that \(h \approx [h]\):

\[
\approx = \{ (f' > x > g', F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n) \}
\]

\[
\quad \quad \cup \{ (f' \triangleright x \triangleright g' | \triangleright x \triangleright \{0, \ldots, 0\} * F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n) \}
\]

\[
\quad \quad \quad \quad \quad \quad \quad | f' \approx_f F' \land g' \approx_{g,j} G_j \land F' \in F^{-1} \land G_j \in G^{-1}
\]

\[
\quad \quad \quad \quad \quad \quad \quad \land \forall m \in \{1, \ldots, r\}. (g_m \approx_{g,j} G_m \land G_m \in G^{-1} \land a_m = 0)
\]

\[
\quad \quad \quad \quad \quad \quad \quad \land \forall m \in \{1, \ldots, r\}. (g_m \approx_{g,j} G_m \land G_m \in G^{-1} \land a_m = 0)
\]

We now prove that \(\approx\) is a weak bisimulation as defined in Definition 2.8 by proving that the two implications, numbered by (i) and (ii), are verified for every element in \(\approx\).

- \(f' \triangleright x \triangleright g' \approx F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\), where \(f' \approx_f F\), \(g' \approx_{g,j} G_j\), \(F' \in F^{-1}\), and \(G_j \in G^{-1}\).

\[(i)\] Since \(F' \in F^{-1}\), then by Corollary 2.6 \(F'\) cannot produce any observation of the form \(\triangleright x\). This and the fact that \(f' \approx_f F'\) implies that possible reduction steps of \(f'\) can only be \(f' \triangleright a \rightarrow f''\), where \(a \notin a\). Therefore the only possible reduction step of \(f' \triangleright x \triangleright g'\) is by \(a\) to \(f'' \triangleright x \triangleright g'\).

Since \(f' \approx_f F'\), we know that exists \(F''\) and \(a'\) such that \(a' = a\), \(F' \triangleright a \rightarrow F''\) and \(f'' \approx_f F''\). With respect to \(F''\) we can consider two cases:

\[(i.1)\] \(F'' \in F^{-1}\): The input node is not fired, so \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n \rightarrow F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) and \(F'' \triangleright x \triangleright g' \approx F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) (because \(f'' \approx_f F'' \land g' \approx_{g,j} G_j \land F'' \in F^{-1} \land G_j \in G^{-1}\)).

\[(i.2)\] \(F'' \in F^{+1}\): The input node is fired, so \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n \rightarrow F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) and \(F'' \triangleright x \triangleright g' \approx F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) (because \(f'' \approx_f F'' \land g' \approx_{g,j} G_j \land F'' \in F^{-1} \land G_j \in G^{-1}\)).

\[(ii)\] The behaviour of \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) depends only on \(F'\).

The possible reduction steps of \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) are:

\[(ii.1)\] If \(F' \triangleright a \rightarrow F''\), then, with respect to \(F''\), the possible behaviour of the connector is:

\[(ii.1.1)\] \(F'' \in F^{-1}\): The input node is not fired, so \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n \rightarrow F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\). This corresponds to the reduction step \(f' \triangleright x \triangleright g' \rightarrow F'' * \triangleright x \triangleright g'\), and \(F'' \triangleright x \triangleright g' \approx F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) (because \(f'' \approx_f F'' \land g' \approx_{g,j} G_j \land F'' \in F^{-1} \land G_j \in G^{-1}\)).

\[(ii.1.2)\] \(F'' \in F^{+1}\): The input node is fired, so \(F' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n \rightarrow F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\). This corresponds to the reduction step \(f' \triangleright x \triangleright g' \rightarrow F'' * \triangleright x \triangleright g'\), and \(F'' \triangleright x \triangleright g' \approx F'' * \triangleright x \triangleright \{0, \ldots, 0\} * G'_1 * \ldots * G'_n\) (because \(f'' \approx_f F'' \land g' \approx_{g,j} G_j \land F'' \in F^{-1} \land G_j \in G^{-1}\)).

\[(ii.2)\] Let \(F' \triangleright a \rightarrow F''\), and let the arity of each \(G_j\) be \(\{I_g\} \cup \tilde{V}_g \rightarrow O_g\). As described in (i), \(a \notin a\), and none of the output nodes are fired (so the buffers will not change their values). The input node of each \(G_j\) has
not been fired yet \((G'_j \in G^{-1})\), therefore \(O_{aj}\) cannot be fired. The input node cannot be fired because the associated FIFO1 channel is empty. The only possible behaviour is then \(G'_j \xrightarrow{X} G''_j\), where \(\text{nodes}(X) \subseteq V_g\).

Note that, since each node in \(V_g\) is common to every connector \(G_j\), then they will be all triggered in the same step. We conclude that the possible behaviour, with respect to \(G_j\), is: \(F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G'' \xrightarrow{X} F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\).

Note that \(g' \approx_{aj} G'_j\) and \(G'_j \xrightarrow{X} G''_j\), where \(X = \{ \tau \}\), therefore \(g' \rightarrow g''\), and \(g'' \approx_{aj} G''_j\). We can then conclude that \(f' \rightarrow g'' \approx F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\) (because \(f' \approx f'_j \land g'' \approx_g G'_j \land F' \in F^{-1} \land G'_j \in G^{-1}\)).

\(- f' \rightarrow g'' \mid (\nu_i/x) \mid g \mid \ldots \mid (\nu_i/x) \mid g \approx F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\), where \(f' \approx f'_j \land F' \in F^{-1} \land \nu_1 \ldots \nu_n\) are values \(\forall m \in \{1, \ldots, r\}.[\nu_i/x].g \approx_{gm} G_m \land G_m \in G^{-1} \Leftrightarrow \alpha_m = 0) \land \forall m \in \{r+1, \ldots, s\}.(g' \approx_{gm} G_m \land G_m \in G^{-1} \land \alpha_m = 0)\). Let \(h'\) and \(h''\) be these two elements of the composition. We have to prove that for every possible behaviour of \(h'\) and \(h''\), the bisimulation conditions still apply.

(i) Since \(h''\) is a parallel composition of several Orc expressions, the possible behaviour of \(h''\) is:

(i.1) if \(f' \rightarrow \nu_i \rightarrow \ldots \rightarrow \nu_i \rightarrow \nu_i \rightarrow f'' \rightarrow g'' \mid \nu_i / x \mid g \mid \ldots \mid \nu_i / x \mid g\). Since \(f' \approx f'_j\), then exists \(a'\) such that \(a' = \nu_i \rightarrow \nu_i \rightarrow \ldots \nu_i \rightarrow \nu_i \rightarrow \nu_i \rightarrow a''\), and \(a'' = a\). Then let \((\alpha_1, \ldots, \alpha_n)\) be the new buffer content after the values \(\nu_1 \ldots \nu_n\) flow into the corresponding buffer. Let also, for each output \(v_i\), \(G'_i\) be such that \(G'_i \xrightarrow{X_i} G'_i \ast \ldots \ast G'_n \ast \ldots \ast G'_n\), where \(X_i\) corresponds to the variable \(x\) in \(\nu_i / x\). We know that \(F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\). We now need to show that the resulting connector is bisimilar to \(f' \rightarrow g'' \mid \nu_i / x \mid g \mid \ldots \mid \nu_i / x \mid g\). Since \(F' \in F^{-1}\), then \(F' \in F^{-1}\). Since the input node of none of \(G'_j\) was fired, and \(\forall m \in \{r+1, \ldots, s\}.(g' \approx_{gm} G_m \land G_m \in G^{-1} \Leftrightarrow \alpha_m = 0) \land \forall m \in \{r+1, \ldots, s\}.(g' \approx_{gm} G_m \land G_m \in G^{-1} \land \alpha_m = 0)\).

(ii.2) if \(\nu_i / x\).g' \rightarrow \nu_i / x\).g', then \(h' \rightarrow f' \rightarrow g'' \mid \nu_i / x\).g' \mid \ldots \mid \nu_i / x\).g. Since \(\nu_i / x\).g' \approx_{gm} G_m\), then exists \(b'\) such that \(b' = a\), \(G'_m \xrightarrow{X_m} G'_m\), and \(\nu_i / x\).g' \approx_{gm} G'_m\). There are two cases with respect to \(G'_m\):

(i.2.1) if \(G'_m \in G^{-1}\), then also \(G'_m \in G^{-1}\) (by Corollary 2.6). Therefore \(F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\). Furthermore, \(f' \rightarrow g'' \mid \nu_i / x \mid g \mid \ldots \mid \nu_i / x \mid g \approx_{gm} G_m \land G_m \in G^{-1} \land \alpha_m = 0) \land \forall m \in \{r+1, \ldots, s\}.(g' \approx_{gm} G_m \land G_m \in G^{-1} \land \alpha_m = 0)\).

(i.2.2) if \(G'_m \in G^{+1}\), then there are two more cases with respect to \(G'_m\):

(i.2.2.1) if \(G'_m \in G^{+1}\), then \(F' \ast x_0 \ast \ldots \ast G' \ast \ldots \ast G''\). Furthermore, \(f' \rightarrow g'' \mid \nu_i / x \mid g \mid \ldots \mid \nu_i / x \mid g \approx_{gm} G_m \land G_m \in G^{+1} \land \alpha_m = 0) \land \forall m \in \{r+1, \ldots, s\}.(g' \approx_{gm} G_m \land G_m \in G^{+1} \land \alpha_m = 0)\).

(i.2.2.2) if \(G'_m \in G^{+1}\), then \(I \in \text{nodes}(b)\), where \(I\) is the main input node of \(G'_m\). The firing of node \(I\) makes the FIFO1 channel attached to it to become empty, i.e., the corresponding \(\alpha\) value becomes zero. The FIFO1 must be full since the firing
of node $I$ is guaranteed by assuming that $\approx_{gm}$ is a bisimulation. Let $\langle^\alpha_1,..^\alpha_n\rangle$ be the new $\alpha$ values after replacing $\alpha_m$ by zero. We can then conclude that $F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_n \nleq F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_m * \ldots * G'_n$. Furthermore, $f' > x > g' | [v_1/x].g' | \ldots | [v_n/x].g' \approx F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_m * \ldots * G'_n$ (because $f' \approx_{F'} F' \land F'' \in F^+I \land \forall m \in \{1,..,r\}.([v_m/x].g' \approx_{gm} G'_m \land G'_m \in G^{-I} \Leftrightarrow \alpha_m = 0) \land \forall m \in \{r+1,..,n\}.(g' \approx_{gm} G'_m \land G'_m \in G^{-I} \land \alpha_m = 0)).$

(iii) any combination of (i.1) and (i.2), for which the proof is identical.

(ii) The possible behaviour of $F'$ is $\approx F''$. We can then conclude that $F'' \in F^+I$. Let $\bar{O}_f$ be the output nodes of $F'$. There are two possible cases:

(ii.1) If $\bar{O}_f \cap \text{nodes}(a) = \emptyset$, then we know that $F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_n \nleq F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_n$. Since $f' \approx_{F'} F'$, then $f' \nleq f''$ and $f'' \approx_{F'} F''$. Therefore $f' > x > g' | [v_1/x].g' | \ldots | [v_n/x].g'$.

We conclude that $f'' \approx_{F''} F'' \land F'' \in F^+I \land \forall m \in \{1,..,r\}.([v_m/x].g' \approx_{gm} G'_m \land G'_m \in G^{-I} \Leftrightarrow \alpha_m = 0) \land \forall m \in \{r+1,..,n\}.(g' \approx_{gm} G'_m \land G'_m \in G^{-I} \land \alpha_m = 0))$.

(ii.2) If $\bar{O}_f \cap \text{nodes}(a) \neq \emptyset$, then some output nodes of $F'$ are fired. Let $a = a' \cup \{X_f\}$ correspond to the partition of $a$ into the output nodes (a') and the input variable nodes (X_f). Note that, by Lemma 2.3, the input node cannot be fired a second time. Let $a' = \{G_{r+1}(v_{r+1}),..,O_a(v_a)\}$. In this case $\bar{a} = \bar{a}' \cup \{\tau\}$. Then we know that $f' \nleq f''$. The firing of the output nodes will fill some FIFO1 channels. Let $\langle^\alpha_1,..^\alpha_n\rangle$ be the values of the FIFO1 channels after the output node are fired. Since for each connector $G_m$ connected to these FIFO1 channels $G_m \in G^{-I}$, then the conditions over the $\alpha$'s will still hold. Moreover, the firing of the output nodes will also trigger the actions $X = \{X_{r+1}(v_{r+1}),..,X_n(v_n)\}$ corresponding to the variable $x$ in the connectors $G'_{r+1},..,G'_n$, who evolve to $G''_{r+1},..,G''_n$, respectively. We can then conclude that $F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G'_n \nleq F' * \{x\}_{1,\langle^\alpha_1,..^\alpha_n\rangle} * G'_1 * \ldots * G''_{r+1} * \ldots * G''_n$. We also conclude that $\forall m \in \{r+1,..,n\}.(g' \approx_{gm} G'_m)$, and by Lemma 2.3 $\forall m \in \{r+1,..,n\}.(g' \approx_{gm} G'_m)$. Therefore $f'' > x > g' | [v_1/x].g' | \ldots | [v_n/x].g'$.
corresponds to the application of that substitution of all occurrences of $f'$ and $g'$. Since $f' \approx f_1 F'$ and $\forall m \in \{1, \ldots, n\}. (g' \approx_g G'_m)$, then by Lemma 3.1, $\sigma f' \approx f_1 F''$ and $\forall m \in \{1, \ldots, n\}. (\sigma g' \approx_g G''_m)$. We can conclude that $\sigma f' \approx f_1 F'' | [v_1/x], \sigma g' | \ldots | [v_n/x]. \sigma g' \approx F'' | [v_1/\alpha_{1, \ldots, n}], G'_1 \ast \ldots \ast G'_n$. Note that the application of the substitution is considered to be an internal transition.

(ii.2.1.2) If $G'' \in G^{-1}$, then $I_{gm} \in \text{nodes}(a)$, where $I_{gm}$ is the main input node of $G''$. This also triggers the FIFO channel connected to $I_{gm}$, changing $\alpha_m$ to 0. Let $(a_1', \ldots, a_n')$ be the $\alpha$ values after the step. Then $F'' | [x_1], (a_1', \ldots, a_n') * G'_1 \ast \ldots \ast G''_n$. Since $g' \approx_g G'_m$, then $g' \Rightarrow g''$, and we conclude that $f' > x > g' | [v_1/x], \ldots | [v_n/x]. g' \Rightarrow g'' | [v_1/x]. g' | \ldots | [v_n/x]. g'' | [v_1/x]. g' \approx F'' | [v_1/\alpha_{1, \ldots, n}], G'_1 \ast \ldots \ast G''_n$.

(ii.2.2) If $G'' \in G^{-1}$, then $G'' \Rightarrow G'' \ast G''$, and $G'' \Rightarrow G^{-1}$. If a contains a node corresponding to an input variable other than $x$, then the situation is equivalent to case (ii.2.1.1). Otherwise, since $[v_n/x]. g' \approx_g G''_m$, then $[v_n/x]. g' \Rightarrow g''$, and $[v_n/x]. g'' \approx_g G''_m$. Therefore $F'' | [x_1], (a_1, \ldots, a_n) * G'_1 \ast \ldots \ast G''_m \Rightarrow F'' | [x_1], (a_1, \ldots, a_n) * G'_1 \ast \ldots \ast G''_m$.

Since $f' \approx f_1 F' \land F' \in F^{-1}$, and $\forall m \in \{1, \ldots, n\}. ([v_m/x]. g' \approx_g G''_m \land G''_m \in G^{-1}) \Rightarrow \alpha_m = 0$ and $\forall m \in \{1, \ldots, n\}. (g' \approx_g G''_m \land G''_m \in G^{-1}) \Rightarrow \alpha_m = 0$.

(ii.3) With respect to $G''_m$, where $r + 1 \leq m \leq n$, we know that $G''_m \in G^{-1}$. Let $G'' \Rightarrow G''_m$. Since $\alpha_m = 0$, then the input node cannot be fired, and therefore $G''_m \in G^{-1}$. The possible behaviour for $G''_m$ is then to fire input nodes associated with variables. If node $x$, associated with variable $x$, is fired, then the corresponding output node of $F'$ is also fired, which corresponds to the case proven in (ii.1.1). If another node is fired, then this corresponds to the case proven in (ii.2.1.1). Combination of these cases follow a similar prove.

(ii.4) Any combination of the (ii.1), (ii.2) and (ii.3), for which the proofs are identicals.

- $h = q$ where $x \in f$

Let $n = \#f$, and $1 \leq j \leq n$. Also let $F = [f]$ and $G = [g]$. We know by the induction hypothesis that there exist 2 bisimulations, $\approx_f$ and $\approx_g$, such that $f \approx_f F$ and $g \approx_g G$.

Let $v$ be any data value. Based on these bisimulations, we define $\approx$ such that $h \approx [h]$:

\[
\approx = \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \cup (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\cup \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\cup \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\cup \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\cup \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\cup \{ (g' where x \in f, F' * W_{0,0,0,0} * G') \}
\text{The proof that } \approx \text{ is in fact a bisimulation follow similar lines to the previous cases, and is omitted.}