Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 194 (2008) 57–76
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Coordination Models Orc and Reo Compared

José Proença[1,2] and Dave Clarke[2]

CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract

Orc and Reo are two complementary approaches to the problem of coordinating components or services. On one hand, Orc is highly asynchronous, naturally 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 (services). The question of how Orc and Reo relate to each other naturally arises. In this paper, we present a detailed comparison between 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 significantly more effort. We conclude with some general observations and comparisons between the two approaches.

Keywords: Reo, Orc, Coordination languages, Software Connectors.

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[3] 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. Web services are a common realization of this idea. Since the conception of SoC, research has focussed on developing languages to compose or coordinate services into either composite services or 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

[1] Supported by FCT grant 22485 – 2005, Portugal.

[2] Email: Jose.Proenca@cwi.nl and D.G.Clarke@cwi.nl

[3] We take the words choreography and orchestration to fall under the more general notion of coordination.

doi:10.1016/j.entcs.2008.03.099

the parts that coordinate the results and resources required to perform the computations. The original coordination language, Linda [17], played only a passive rôle in coordination, by providing a blackboard (tuple space) to which data could be written and read. Since then many coordination models have been proposed [22,3], and the trend is towards developing models that play a more active rôle in the coordination process. Two recent interesting coordination models, Orc [20] and Reo [4], sit diametrically opposite of each other in their approaches to coordinating services or components. This paper sets out to explore these in detail.

Orc is a simple orchestration language designed by Misra and Cook [20], based on a three connectives, and the simple notion of a site call to model computations—the external actions to be orchestrated. Central to 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.

Reo is a channel-based coordination language designed by Arbab [4] that is based on a simple notion of channel composition. It differs from existing models in that composition propagates synchronization and exclusion constraints through connectors. In combination with stateful channels, an expressive coordination language emerges.

This paper presents a comparison between Orc and 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 the two approaches, 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 Orc into Reo. We also discuss the difficulties of embedding in the other direction. Section 2 describes our encoding of Orc into Reo. Section 3 argues that the encoding in the other direction is not as trivial. 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 Orc and Reo, and give a small example.

1.1 Orc

In this section we present Orc's syntax and reduction semantics, and simple examples of Orc expressions. In work by Misra and others [20,18] Orc's semantics is described in more detail.

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:

$$\begin{aligned} f, g, h \in \mathit{Expr} &::= \mathbf{0} \;\mid\; M(\overline{p}) \;\mid\; E(\overline{p}) \;\mid\; f > x > g \;\mid\; f \,|\, g \;\mid\; f \textbf{ where } x :\in g \\ p \in \mathit{Actual} &::= x \;\mid\; v \\ \mathit{Definition} &::= E(\overline{x}) \overset{\text{def}}{=} f \end{aligned}$$

An Orc program consists on an Orc expression together with a set of definitions. Basic services, such as data manipulation, are assumed to be provided by primitive *sites*. An Orc expression can be a primitive site call, a reference to another Orc expression, or a composition of Orc expressions.

A site call is written as $M(\overline{p})$, where $\overline{p}$ is a tuple of arguments which can be constants or variables. When executed all variables have to be instantiated, that is, evaluation is strict, and the site returns at most one value. Examples of primitive sites are $\mathbf{0}$, which never responds, and $let(v)$, which responds value v. We use E to range over possibly recursive definitions of Orc expressions.

Three combinators exist for composing expressions f and g: symmetric composition, written as $f \mid g$; sequential composition, written as $f >x> g$; and asymmetric composition, written as f **where** $x :\in g$. The combinator $f \mid g$ executes f and g independently in parallel; $f >x> g$ executes f and several threads of g, one for each result returned by f (where each value of f is substituted for x in a new g); and f **where** $x :\in g$ executes f and g in parallel, replacing x in f by the first returned value of g, returning only the values of f.[4]

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 $f \xrightarrow{a} g$ and are presented below. Here a is a set of observations of the following type:

$$BaseEvent ::= \ \tau \mid M_k(v) \mid k?v \mid !v$$

We use the silent observation τ mainly to represent the binding of a variable to a value. $M_k(v)$ represents the call to site M, indexed by a fresh k and with arguments v. $k?v$ 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 $\xrightarrow{\emptyset}$ denotes $\xrightarrow{\tau}$. Here are the reduction rules:

$$\frac{k \text{ fresh}}{M(v) \xrightarrow{M_k(v)} ?k}\ (\textsc{SiteCall}) \qquad \frac{}{let(v) \xrightarrow{!v} \mathbf{0}}\ (\textsc{Let}) \qquad \frac{}{?k \xrightarrow{k?v} let(v)}\ (\textsc{SiteRet})$$

$$\frac{f \xrightarrow{a} f'}{f \mid g \xrightarrow{a} f' \mid g}\ (\textsc{Sym1}) \qquad \frac{g \xrightarrow{a} g'}{f \mid g \xrightarrow{a} f \mid g'}\ (\textsc{Sym2}) \qquad \frac{f \xrightarrow{a} f' \quad g \xrightarrow{b} g'}{f \mid g \xrightarrow{a,b} f' \mid g'}\ (\textsc{Sym3})$$

$$\frac{g \xrightarrow{a} g'}{g \textbf{ where } x :\in f \xrightarrow{a} g' \textbf{ where } x :\in f}\ (\textsc{Asym1N}) \qquad \frac{f \xrightarrow{b} f' \quad !w \notin b}{g \textbf{ where } x :\in f \xrightarrow{b} g \textbf{ where } x :\in f'}\ (\textsc{Asym2})$$

$$\frac{g \xrightarrow{a} g' \quad f \xrightarrow{b} f' \quad !w \notin b}{g \textbf{ where } x :\in f \xrightarrow{a,b} g' \textbf{ where } x :\in f'}\ (\textsc{Asym3N}) \qquad \frac{f \xrightarrow{!v,b} f'}{g \textbf{ where } x :\in f \xrightarrow{\tau} [v/x].g}\ (\textsc{Asym1V})$$

$$\frac{g \xrightarrow{a} g' \quad f \xrightarrow{!v,b} f'}{g \textbf{ where } x :\in f \xrightarrow{a} [v/x].g'}\ (\textsc{Asym2V}) \qquad \frac{f \xrightarrow{!v_1,..,!v_n,a} f' \quad !w \notin a \quad n \geq 0}{f >x> g \xrightarrow{a} f' >x> g \mid [v_1/x].g \mid .. \mid [v_n/x].g}\ (\textsc{Seq})$$

Consider the following two examples, introduced by Kitchin *et al.* [18]:

$$EmailNews(d) \overset{\text{def}}{=} (CNN(uk,d) \mid BBC(uk)) >x> email(me,x)$$
$$EmailNewsOnce(d) \overset{\text{def}}{=} email(me,x) \textbf{ where } x :\in (CNN(uk,d) \mid BBC(uk))$$

4 These operations appear to be closely related to Friedman and Wise's *frons* constructor [16].

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

The expression $EmailNewsOnce$ reduces as follows. Here we assume a and b are fresh, v is the value returned by the BBC site, and v' is the value returned by the $Email$ site.

$$\begin{array}{rl}
 & email(me, x) \textbf{ where } x :\in (CNN(uk, d) \mid BBC(uk)) \\
\xrightarrow{BBC_a(uk)} & email(me, x) \textbf{ where } x :\in (CNN(uk, d) \mid ?a) \\
\xrightarrow{a?v} & email(me, x) \textbf{ where } x :\in (CNN(uk, d) \mid let(v)) \\
\xrightarrow{\tau} & email(me, v) \\
\xrightarrow{Email_b(me,v)} & ?b \\
\xrightarrow{b?v'} & let(v') \\
\xrightarrow{!v'} & 0
\end{array}$$

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

$$\begin{aligned}
Metronome &\stackrel{\text{def}}{=} Signal \mid (Rtimer(1) >x> Metronome) \\
EmailNewsFrequently(d) &\stackrel{\text{def}}{=} Metronome >x> EmailNewsOnce(d)
\end{aligned}$$

$Metronome$ is an Orc expression that sends a signal returned by $Signal$ every time unit. The site $Rtimer(t)$ waits t time units before returning a signal. Therefore, $EmainNewsFrequently$ calls $EmainNewsOnce$ every time unit, which in turn sends me an email from either CNN or BBC.

1.2 Reo

Reo is a powerful coordination model arising from the propagation of synchronisation and other constraints imposed by individual channels through connectors formed by plugging channels together, in combination with mutual exclusive data merging and synchronous data replication through nodes. A key characteristic of Reo is that synchrony is propagated through composition. We present the semantics of Reo connectors in an adaptation of the Q-automata model [11], which in turn extends constraint automata [8].

Firstly, we assume that connectors are defined over a denumerable set of port names, $\mathcal{P}ort$. Each connector C will have a set of input ports $\boldsymbol{I} \subseteq \mathcal{P}ort$, and

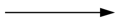
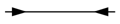
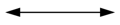
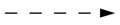
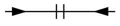
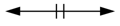
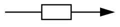
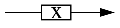
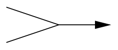
Visualisation	Representation	Arity	Axioms
	$Sync_{A,B}$	$A \rightarrow B$	$Sync_{A,B} \xrightarrow{A(v),B(v)} Sync_{A,B}$
	$SDrain_{A,B}$	$\{A,B\} \rightarrow \emptyset$	$SDrain_{A,B} \xrightarrow{A(v),B(w)} SDrain_{A,B}$
	$SSpout_{A,B}$	$\emptyset \rightarrow \{A,B\}$	$SSpout_{A,B} \xrightarrow{A(v),B(w)} SSpout_{A,B}$
	$Lossy_{A,B}$	$A \rightarrow B$	$Lossy_{A,B} \xrightarrow{A(v),B(v)} Lossy_{A,B}$ $Lossy_{A,B} \xrightarrow{A(v)} Lossy_{A,B}$
	$ADrain_{A,B}$	$\{A,B\} \rightarrow \emptyset$	$ADrain_{A,B} \xrightarrow{A(v)} ADrain_{A,B}$ $ADrain_{A,B} \xrightarrow{B(v)} ADrain_{A,B}$
	$ASpout_{A,B}$	$\emptyset \rightarrow \{A,B\}$	$ASpout_{A,B} \xrightarrow{A(v)} ASpout_{A,B}$ $ASpout_{A,B} \xrightarrow{B(v)} ASpout_{A,B}$
	$FIFO1_{A,B}$	$A \rightarrow B$	$FIFO1_{A,B} \xrightarrow{A(v)} FIFO1_{A,B}(v)$
	$FIFO1_{A,B}(v)$	$A \rightarrow B$	$FIFO1_{A,B}(v) \xrightarrow{B(v)} FIFO1_{A,B}$
	$Merger_{A,B,C}$	$\{A,B\} \rightarrow C$	$Merger_{A,B,C} \xrightarrow{A(v),C(v)} Merger_{A,B,C}$ $Merger_{A,B,C} \xrightarrow{B(v),C(v)} Merger_{A,B,C}$
0	0-out$_A$	$\emptyset \rightarrow A$	–
0	0-in$_A$	$A \rightarrow \emptyset$	–
1	1Drain$_A$	$A \rightarrow \emptyset$	1Drain$_A \xrightarrow{A(v)}$ 1Drain$_A$

Table 1
Arities and behaviour of some Reo primitives

a (disjoint) set of output ports, $\boldsymbol{O} \subseteq \mathcal{P}ort$.[5] The input and output ports of a connector define its *arity*, denoted $C : \boldsymbol{I} \rightarrow \boldsymbol{O}$. We define $\mathcal{N}ames(C)$ to be $\boldsymbol{I} \cup \boldsymbol{O}$.

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 $\mathsf{nodes}(N)$ to denote the domain of N. We say that C evolves to C' and fires nodes $\mathsf{nodes}(N)$. 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.

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:

$$\frac{\begin{array}{c} C : \boldsymbol{I} \rightarrow \boldsymbol{O} \qquad C' : \boldsymbol{I'} \rightarrow \boldsymbol{O'} \\ \boldsymbol{I''} \overset{\text{def}}{=} \boldsymbol{I} \cup \boldsymbol{I'} \quad \boldsymbol{O''} \overset{\text{def}}{=} \boldsymbol{O} \cup \boldsymbol{O'} \quad \boldsymbol{O} \cap \boldsymbol{O'} = \emptyset \end{array}}{C * C' : (\boldsymbol{I''} \setminus \boldsymbol{O''}) \rightarrow \boldsymbol{O''}}$$

This rule expresses that output and input nodes are plugged $1 : n$, *i.e.*, each output node can be plugged into multiple input nodes. Regarding the behaviour, output

[5] For the purpose of this paper, we assume that primitive connectors are not plugged into themselves.

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, without fundamentally changing anything, in order to simplify our formal results.

Notation 1 *Given a map N and a set P. With a slight abuse of notation, define $N \cap P \overset{def}{=} \{(n, d) \in N \mid n \in P\}$ and $N \setminus P \overset{def}{=} \{(n, d) \in N \mid n \notin P\}$.*

The following two rules give the semantics for the composition of connectors $C_1 : \boldsymbol{I_1} \to \boldsymbol{O_1}$ and $C_2 : \boldsymbol{I_2} \to \boldsymbol{O_2}$. 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.

$$\frac{C_1 \xrightarrow{N_1} C_1' \quad C_2 \xrightarrow{N_2} C_2' \quad N_1 \cap \mathcal{N}ames(C_2) = N_2 \cap \mathcal{N}ames(C_1)}{C_1 * C_2 \xrightarrow{N_1 \cup N_2} C_1' * C_2'} \qquad \frac{C_1 \xrightarrow{N_1} C_1' \quad N_1 \cap \mathcal{N}ames(C_2) = \emptyset}{C_1 * C_2 \xrightarrow{N_1} C_1' * C_2}$$

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 the output nodes of a connector. Given a connector $C : \boldsymbol{I} \to \boldsymbol{O}$ and a set of nodes Ω, define $C \restriction_{\Omega} = C : \boldsymbol{I} \to (\boldsymbol{O} \backslash \Omega)$.

Consider the services *Politics*, *Sports* and *Email*, that return news about politics or news about sport, or sends an email of a given message, respectively. In Fig. 1 we present a connector that coordinates these three services. Initially the connector receives data from the *Politics* and then 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 there is also sports news.

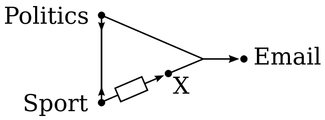


Fig. 1. Example of a Reo connector

Formally, we consider two different states of the connector:
$Ord = (FIFO1_{Sport,X} * SDrain_{Politics,Sport} * Merger_{Politics,X,Email}) \restriction_{Email}$ and
$Ord(x) = (FIFO1_{Sport,X}(x) * SDrain_{Politics,Sport} * Merger_{Politics,X,Email}) \restriction_{Email}$.
The former corresponds to the connector depicted in Fig 1, while the latter corresponds to the same connector when the $FIFO1$ channel has data is full with data x. Using the rules above, we can calculate the connector's arity, $Ord : \{Politics, Sport\} \to Email$, and behaviour:

$$Ord \xrightarrow{Politics(v), Sport(w), Email(v)} Ord(w)$$
$$Ord(x) \xrightarrow{X(w), Email(w)} Ord.$$

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*.

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 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.[6] The connectors are defined in Table 2.

Table 2 is divided into two parts. In the upper part we present three connectors, which play a rôle similar to nodes in that they connect a single output node to multiple input nodes, except that their behaviour differs. Firstly, an *Exclusive Router* receives data in the input node and sends data to exactly one of the output nodes synchronously. 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 which 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 Var and $\mathsf{Var}(x)$ 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 one or more input nodes and the output node can flow.

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, all at

[6] The tupling connector is an exception, as none of our primitives are capable of data manipulation.

Representation	Arity	Axioms
$\otimes$	$I \to \{O_1,\ldots,O_n\}$	$\dfrac{O \in \{O_1,\ldots,O_n\}}{\otimes \xrightarrow{I(x),O(x)} \otimes}$
$\obar$	$I \to \{O_1,\ldots,O_n\}$	$\dfrac{\emptyset \subsetneq \{P_1,\ldots,P_m\} \subseteq \{O_1,\ldots,O_n\}}{\obar \xrightarrow{I(x),P_1(x),\ldots,P_m(x)} \obar}$
$\textcircled{0}$	$I \to \{O_1,\ldots,O_n\}$	$\textcircled{0} \xrightarrow{I(x),O_1(x_1),\ldots,O_n(x_n)} \textcircled{1}$
$\textcircled{1}$	$I \to \{O_1,\ldots,O_n\}$	–
T_n	$\{I_1,\ldots,I_n\} \to O$	$\mathsf{T}_n \xrightarrow{I_1(x_1),\ldots,I_n(x_n),O(x_1,\ldots,x_n)} \mathsf{T}_n$
C_p	$\emptyset \to O$	$\mathsf{C}_p \xrightarrow{O(p)} \mathsf{C}_p$
Var	$I \to O$	$\mathsf{Var} \xrightarrow{I(x)} \mathsf{Var}(x)$
$\mathsf{Var}(x)$	$I \to O$	$\mathsf{Var}(x) \xrightarrow{O(x)} \mathsf{Var}(x)$ $\mathsf{Var}(x) \xrightarrow{I(y)} \mathsf{Var}(y)$
P_n	$\{I_1,\ldots,I_n\} \to O$	$\dfrac{\emptyset \subsetneq \{P_1,\ldots,P_m\} \subseteq \{I_1,\ldots,I_n\} \quad x \in \{x_1,\ldots,x_m\}}{\mathsf{P}_n \xrightarrow{P_1(x_1),\ldots,P_m(x_m),O(x)} \mathsf{P}_n}$

Table 2
Definition of some Reo connectors.

the same time or just some at each time. To coordinate access to a variable, we propose two different approaches in Fig. 2: (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.

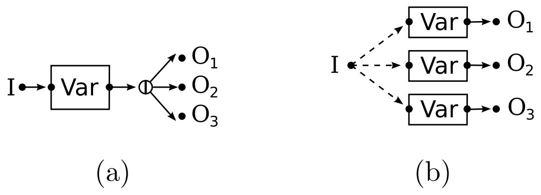


Fig. 2. (a) Replication *after* the storage of a variable. (b) Replication *of* the storage of a variable.

2.2 Merged-Output Encoding

This section presents an encoding of an 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 $h \in \mathsf{Orc}^-$ is encoded as a connector with arity $\{I, X_1, \ldots, X_n\} \to O$, depicted in Fig. 3(a), where the X_i corresponds to the free variables of h. For example, the encoding of the expression $(CNN(uk,d) \mid BBC(uk)) >x> email(me,x)$

is presented in Fig. 3(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 BBC can then be called, while site CNN needs to wait until data is available on node D to be called. The results from the site calls are stored in the RVar component one at a time, which subsequently provides the value to site *email*, once for each value returned by BBC and CNN.

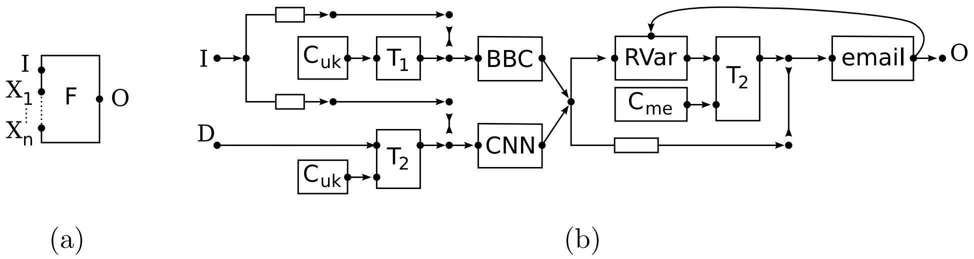


Fig. 3. Encodings into Reo connectors with a single output: (a) a general Orc expression; (b) a specific Orc expression $(CNN(uk,d) \mid BBC(uk)) >x> email(me,x)$. RVar is a resettable variable. It acts like a variable (*e.g.*, Var from Section 2.1), but it cannot be updated until the reset (top) node is fired, removing the value of the variable.

The example encoding reveals the main problem of this approach. The outputs of $CNN(uk,d) \mid BBC(uk)$ are forwarded to a single instance of *email*, serializing the execution of *email*. As a consequence, it is possible that CNN finishes before BBC, but that site *email* hangs on the result of CNN, preventing *email* from even getting the result from BBC. The semantics of Orc [18], however, dictate that $(CNN(uk,d) \mid BBC(uk)) >x> email(me,x)$ is strongly bisimilar to $(CNN(uk,d) >x> email(me,x)) \mid (BBC(uk) >x> email(me,x))$, which means that *email* is not serialized and could respond to either results from CNN or 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 cannot return any value, as done by Bruni *et al.* in their encoding of Orc into Petri Nets [7]. This could be achieved, for example, by adding timeouts to each primitive site call. An extension for Reo that includes connectors capable of dealing explicitly with time was proposed by Arbab *et al.* [1]. The authors introduce the *Timed Constraint Automata*, which can be used to formally model the timeouts in 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 FIFO1 channel, which loses the contents of the buffer after a certain time. Using these ideas 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 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. The following lemma captures this property.

Lemma 2.1 *Define function (#) on* Orc^- *expressions (and internal representations:* $?k$*,* $let(v)$ *and* $\mathbf{0}$*), and on sets of output actions as follows:*

$$\begin{array}{llll}
\#(f \mid g) = \#(f) + \#(g) & & \#(?k) = 1 \\
\#(f >x> g) = \#(f) \times \#(g) & & \#(let(v)) = 1 \\
\#(g \ \textbf{\textit{where}}\ x :\in f) = \#(g) & & \#\mathbf{0} = 0 \\
\#(M(v_1, \ldots, v_n)) = 1 & & \#(!v_1, \ldots, !v_n, a) = n \\
 & & \quad \textit{where } !w \notin a
\end{array}$$

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 Outline. Straightforward induction, observing that substitution of values for variables has no effect on the maximum number of outputs produced. □

Corollary 2.2 *Let* $f \in \mathsf{Orc}^-$*, and* $f \xrightarrow{a_1} f' \xrightarrow{a_2} \cdots \xrightarrow{a_n} f^{(n)}$ *be a possible trace. Then* $\#f \geq \#a_1 + \cdots + \#a_n + \#f^{(n)}$.

We now define a function $[\![\cdot]\!]$ which converts an expression $f \in \mathsf{Orc}^-$ into a Reo connector. The arity of resulting connector will be $\{I\} \cup \boldsymbol{V} \to \boldsymbol{O}$, where I denotes the main input node, $\boldsymbol{V}$ denotes a set of nodes corresponding to the free variables of f, and $\boldsymbol{O}$ is the set of output nodes. Node I is used to initiate the connector, though nodes in $\boldsymbol{V}$ can be fired beforehand, which corresponds to the setting of these variables. The function $[\![\cdot]\!]$ is defined inductively on the shape of Orc expressions, in such a way that the number of output nodes is given by function (#) defined above. The encoding is defined in Fig. 4, based on the following primitives.

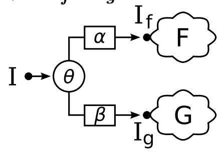
Symmetric Parallel Composition:

$$\|_{\theta,\alpha,\beta} : I \to \{I_f, I_g\}$$

Initially $\theta = \alpha = \beta = 0$. The intuition behind the connector $\|_{\theta,\alpha,\beta}$, illustrated in Fig. 4(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 fired nodes I_f and I_g as soon as they are ready to be fired. As $[\![f \mid g]\!] = \|_{0,0,0} * \mathsf{F} * \mathsf{G}$, 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.

$[\![f \mid g]\!] = (\mathsf{F} * \|_{0,0,0} * \mathsf{G}) \restriction_{\boldsymbol{O_f} \cup \boldsymbol{O_g}}$ where

$\|_{\theta,\alpha,\beta} : I \to \{I_f, I_g\} :=$

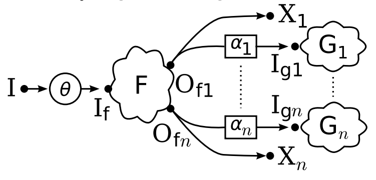


$\mathsf{F} := [\![f]\!] : \{I_f\} \cup \boldsymbol{V_f} \to \boldsymbol{O_f}$
$\mathsf{G} := [\![g]\!] : \{I_g\} \cup \boldsymbol{V_g} \to \boldsymbol{O_g}$

(a)

$[\![f >x> g]\!] = (\mathsf{F} * \rangle\!\rangle x \rangle\!\rangle_{0,\langle 0,\ldots,0\rangle} * \mathsf{G}_1 * \cdots * \mathsf{G}_n) \restriction_{\bigcup_{i=1}^{n} \boldsymbol{O_{gi}}}$
where

$\rangle\!\rangle x \rangle\!\rangle_{\theta,\langle \alpha_1,\ldots,\alpha_n\rangle} : \{I, O_{f1}, \ldots, O_{fn}\} \to \{I_f, I_{g1}, \ldots, I_{gn}, X_1, \ldots, X_n\} :=$

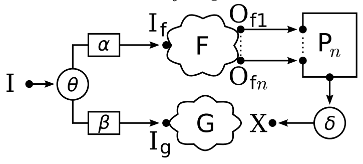


$\mathsf{F} := [\![f]\!] : \{I_f\} \cup \boldsymbol{V_f} \to \{O_{f1}, \ldots, O_{fn}\}$
for $j \in \{1, \ldots, n\}$:
$$\begin{cases} \mathsf{G}_j := [\![[x_j/x].g]\!] : \{I_{gj}\} \cup \boldsymbol{V_g} \to \boldsymbol{O_{gj}} \\ x_j \text{ is a fresh variable name} \end{cases}$$

(b)

$[\![g \textbf{ where } x :\in f]\!] = (\mathbb{W}^x_{0,0,0,0} * \mathsf{F} * \mathsf{G}) \restriction_{\boldsymbol{O_g}}$ where

$\mathbb{W}^x_{\theta,\alpha,\beta,\delta} : \{I, O_{f1}, \ldots, O_{fn}\} \to \{I_f, I_g, X\} :=$

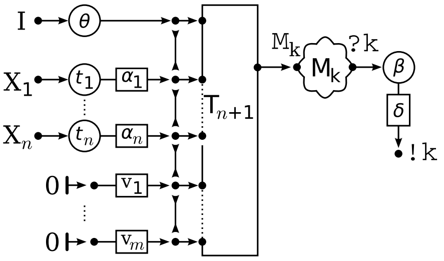


$\mathsf{F} := [\![f]\!] : I_f \cup \boldsymbol{V_f} \to \{O_{f1}, \ldots, O_{fn}\}$
$\mathsf{G} := [\![g]\!] : I_g \cup \boldsymbol{V_g} \to \boldsymbol{O_g}$

(c)

$[\![M(x_1, \ldots, x_n, v_1, \ldots, v_m)]\!] = (\mathbb{M}_{0,\langle 0,\ldots,0\rangle,V,0,0} * \mathsf{M_k}) \restriction_{!\mathtt{k}}$
where

$\mathbb{M}_{\theta,\langle \alpha_1,\ldots,\alpha_n\rangle,V,\beta,\delta} : \{I, X_1, \ldots, X_n, ?\mathtt{k}\} \to \{\mathtt{M_k}, !\mathtt{k}\} :=$



$V = \langle v_1, \ldots, v_m\rangle$
$x_1, \ldots, x_n$ are variables
$v_1, \ldots, v_m$ are values
for $j \in \{1, \ldots, n\}$:
$$t_j := \begin{cases} 0 \text{ if } \theta = \alpha_j = 0 \\ 1 \text{ otherwise} \end{cases}$$
$\mathsf{M_k} : \mathtt{M_k} \to ?\mathtt{k}$
:= Reo component of site M
k is fresh

(d)

Fig. 4. Definition of the encoding function $[\![\cdot]\!]$ from Orc into Reo, where α, β, θ, and δ 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.

$$\begin{tikzcd}
\boxed{\|_{0,0,0}} \arrow[r, "I(v)"] & \boxed{\|_{1,v,v}} \arrow[r, "I_f(v)"] \arrow[d, "I_g(v)"] \arrow[dr, "{I_f(v),I_g(v)}"] & \boxed{\|_{1,0,v}} \arrow[d, "I_g(v)"] \\
& \boxed{\|_{1,v,0}} \arrow[r, "I_f(v)"] & \boxed{\|_{1,0,0}}
\end{tikzcd}$$

Sequential Composition:

$$\rangle\!\rangle x \rangle\!\rangle_{\theta,\langle\alpha_1,\ldots,\alpha_n\rangle} : \{I, O_{f1}, \ldots, O_{fn}\} \to \{I_f, I_{g1}, \ldots, I_{gn}, X_1, \ldots, X_n\}$$

The connector is illustrated in Fig. 4(b). The main idea is to execute the encoding of f when data flows though the input node I, and to buffer each of its outputs in a different FIFO1 channel. Each of these FIFO1 channels is connected to a different instance of the encoding of g, which can be executed in parallel after the corresponding FIFO1 channel is filled.

To make the behaviour easier to describe, we partition $\rangle\!\rangle x \rangle\!\rangle_{\theta,\langle\alpha_1,\ldots,\alpha_n\rangle}$ into $n+1$ different connectors corresponding to unconnected parts of the main connector:

$$\rangle\!\rangle x \rangle\!\rangle_{\theta,\langle\alpha_1,\ldots,\alpha_n\rangle} = \rangle\!\rangle x \rangle\!\rangle^{\mathsf{F}}_{\theta} * \rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}1}_{\alpha_1} * \cdots * \rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}n}_{\alpha_n},$$

where $\rangle\!\rangle x \rangle\!\rangle^{\mathsf{F}}_{\theta} : I \to I_f$, $\rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}j}_{\alpha_j} : O_{fj} \to \{I_{gj}, X_j\}$, 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:

$$\begin{array}{l}
\rangle\!\rangle x \rangle\!\rangle^{\mathsf{F}}_{0} \xrightarrow{I(v),I_f(v)} \rangle\!\rangle x \rangle\!\rangle^{\mathsf{F}}_{1} \\
\rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}j}_{0} \xrightarrow{O_{fj}(v),X_j(v)} \rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}j}_{v} \xrightarrow{I_{gj}(v)} \rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}j}_{0},
\end{array}$$

where $1 \leq j \leq n$. This means that $\rangle\!\rangle x \rangle\!\rangle_{0,\langle 0,\ldots,0\rangle}$, when triggered by node I, also triggers the input node of F. For each output of F (in node O_{fj}), the connector $\rangle\!\rangle x \rangle\!\rangle^{\mathsf{G}j}_{0}$ fires also 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 when possible.

Asymmetric Parallel Composition:

$$\mathbb{W}^{x}_{\theta,\alpha,\beta,\delta} : \{I, O_{f1}, \ldots, O_{fn}\} \to \{I_f, I_g, X\}$$

The connector is illustrated in Fig. 4(c). The intuition is that the encodings of f (F) and g (G) are executed in parallel, as described in the symmetric parallel composition. The output nodes of G are merged in such a way that only the first output value will flow through node X, which will be connected to F where the value of x is used. The output nodes of the connector $\mathbb{W}^{x}_{\theta,\alpha,\beta,\delta}$ are only the output nodes of F.

To make the behaviour easier to describe, we partition $\mathbb{W}^{x}_{\theta,\alpha,\beta,\delta}$ into two different connectors corresponding to unconnected parts of the main connector:

$$\mathbb{W}^{x}_{\theta,\alpha,\beta,\delta} = \mathbb{W}^{\leftarrow}_{\theta,\alpha,\beta} * \mathbb{W}^{\rightarrow}_{\delta},$$

where $\mathbb{W}^{\leftarrow}_{\theta,\alpha,\beta} : I \to \{I_f, I_g\}$ and $\mathbb{W}^{\rightarrow}_{\delta} : \{O_{f1}, \ldots, O_{fn}\} \to \{X\}$. Initially $\theta = \alpha = \beta = \delta = 0$. The behaviour of $\mathbb{W}^{\leftarrow}_{0,0,0}$ is the equivalent to the behaviour of $\|_{0,0,0}$. The possible behaviour of $\mathbb{W}_0^{\rightarrow}$ is the following:

$$\mathbb{W}_0^{\rightarrow} \xrightarrow{\boldsymbol{O},X(v_k)} \mathbb{W}_1^{\rightarrow},$$

where $\boldsymbol{O} \subseteq \{O_{f1}(v_1), \ldots, O_{fn}(v_n)\}$, and $v_k \in \{v_1, \ldots, v_n\}$ such that $O_{fk}(v_k) \in \boldsymbol{O}$. 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_n (see Table 2). This means that $\mathbb{W}^x_{0,0,0,0} * \mathsf{F} * \mathsf{G}$ behaves similarly to $\|_{0,0,0} * \mathsf{F} * \mathsf{G}$, except that the output nodes of F trigger the connector $\mathbb{W}_\delta^{\rightarrow}$. The output nodes of $\mathbb{W}^x_{\theta,\alpha,\beta,\delta}$ 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.

Site call:

$$\mathbb{M}_{\theta,\Sigma,V,\beta,\delta} : \{I, X_1, \ldots, X_n, ?\mathtt{k}\} \rightarrow \{\mathtt{M_k}, !\mathtt{k}\}$$

The connector is illustrated in Fig. 4(d). The main idea is to tuple all the arguments required by site M before the site is executed. As in previous cases, we partition this connector into two different connectors corresponding to unconnected parts of $\mathbb{M}$ to make the behaviour easier to describe:

$$\mathbb{M}_{\theta,\Sigma,V,\beta,\delta} = \mathbb{M}^{\leftarrow}_{\theta,\Sigma,V} * \mathbb{M}^{\rightarrow}_{\beta,\delta}.$$

Initially $\theta = \beta = \delta = 0$ and $\Sigma = \langle 0, \ldots, 0\rangle$. The behaviour of each of the subparts is described below:

$$\begin{array}{l}\mathbb{M}^{\leftarrow}_{0,\Sigma_0,V} \xrightarrow{N_1} \mathbb{M}^{\leftarrow}_{0,\Sigma_1,V} \xrightarrow{N_2} \cdots \xrightarrow{N_j} \mathbb{M}^{\leftarrow}_{0,\Sigma_j,V} \xrightarrow{I(v),\mathtt{M_k}(v)} \mathbb{M}^{\leftarrow}_{1,\emptyset} \\ \mathbb{M}^{\rightarrow}_{0,0} \xrightarrow{\mathtt{M_k}(v)} \mathbb{M}^{\rightarrow}_{1,v} \xrightarrow{!\mathtt{k}(v)} \mathbb{M}^{\rightarrow}_{1,0}\end{array}$$

where $v = \langle v_1, \ldots, v_n\rangle$ is a tuple of data values, and for each $i \in \{0, \ldots, j\}$, $N_i \subseteq \{X_1(v_1), \ldots, X_n(v_n)\}$, $\cup_i N_i = \{X_1(v_1), \ldots, X_n(v_n)\}$, $\Sigma_j = \langle \alpha'_1, \ldots, \alpha'_n\rangle$ such that, for $i \in \{1, \ldots, n\}$, $\alpha'_i \neq 0$, and for each $X_m(v_m) \in N_i$, $\Sigma_i = [v_m/\alpha_m].\Sigma_{i-1}$. This means that initially the empty FIFO1 channels in $\mathbb{M}^{\leftarrow}_{\theta,\Sigma,V}$ need to become full by the firing of the corresponding nodes. Only then node I can be fired, together with node $\mathtt{M_k}$ which triggers component $\mathsf{M_k}$. When this component returns data on node `?k`, the value is stored in a FIFO1 channel, and in the next step the value is output by node `!k`.

Example revisited

Recall the Orc expression $(CNN(uk, d) \mid BBC(uk)) > x > email(me, x)$ presented in Section 1.1. We presented its encoding with merged-outputs in Section 2.2. Fig. 5 presents the connector $[\![(CNN(uk, d) \mid BBC(uk)) > x > email(me, x)]\!]$. Data flowing through the input node I corresponds to the start of execution of the Orc expression, and flowing data through the input node D corresponds to the binding of variable d.

Note that the resulting connector is not the most simple one, in the sense that there are consecutive FIFO1 channels that could be merged into a single one, and

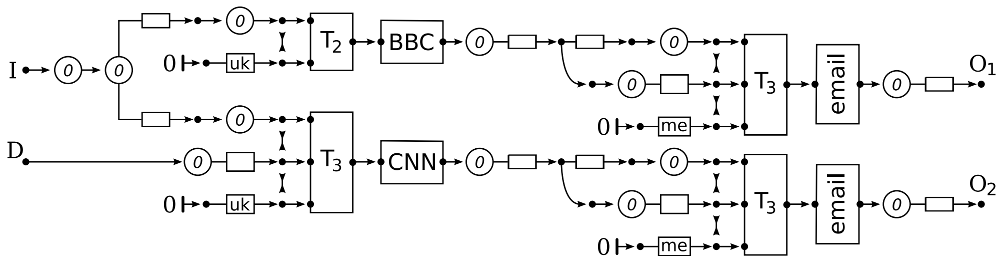


Fig. 5. Example of the encoding of $(CNN(uk, d) \mid BBC(uk)) >x> email(me, x)$

there are some redundant One Time Nodes. If we wanted to actually run the encoding of an Orc expression we could remove the One Time Nodes, relying on the assumption that site calls only return once, and the encoded connector is only executed once. Without these assumptions, the One Time Nodes are needed to derive a bisimulation between Reo and Orc$^-$.

2.4 Soundness

In this section we provide several important results about the translation presented in Fig. 4 that are required to understand and prove the main result, namely that every $h \in$ Orc$^-$ is weakly bisimilar to its encoding in Reo.

Define the function $\widehat{\cdot}$ to map labels in Reo's operational semantics to base events of Orc as follows:

$$\widehat{\mathtt{M_k}(v)} = M_k(v) \qquad \widehat{\mathtt{?k}(v)} = k?v \qquad \widehat{\mathtt{!k}(v)} = !v$$
$$\widehat{\emptyset} = \emptyset \qquad \widehat{a, b} = \widehat{a} \cup \widehat{b} \qquad \widehat{a} = \tau \quad \textit{otherwise},$$

where $\mathtt{M_k}$, $\mathtt{?k}$ and $\mathtt{!k}$ correspond to nodes in the Reo connector obtained from the translation of a site call M.

Lemma 2.3 *Let $h \in$ Orc$^-$. Each node in $\mathcal{N}ames(\llbracket h \rrbracket)$ can be fired at most once.*

This result can be proved by structural induction. It follows from the presence of One Time Nodes connected to input and output nodes, and from the dependence between input and output nodes.

Using this property, we relate the order in which input and output nodes are fired in Lemma 2.4.

Lemma 2.4 *Let $h \in$ Orc$^-$ and $\llbracket h \rrbracket = H : \{I\} \cup \boldsymbol{V} \to \boldsymbol{O}$. For any trace $\langle a_0, a_1, a_2, \ldots \rangle$ of sets of fired boundary nodes of $\llbracket h \rrbracket$, we claim that:*

$$I \in a_n \quad \Rightarrow \quad \boldsymbol{O} \cap a_n = \emptyset, \text{ and for } 0 \leq j < n,\ a_j \subseteq \boldsymbol{V} \text{ and } \boldsymbol{O} \cap a_j = \emptyset.$$

This lemma can also be proved by structural induction on h. It is enough to verify that, for each case of the encoding function, the firing of the main input must precede the firing of the output node. By Lemma 2.4, the input and output nodes can be fired only once, so every action a occurring before the input node is fired is such that $\widehat{a} = \emptyset$ or $\widehat{a} = \tau$, because a can only refer to input or output nodes.

Since the main input node of the encoding of an Orc^- expression can only be fired once, we introduce some notation to distinguish the states of the connector before and after the input node is fired. This simplifies the comparison of the evolution of Orc^- expressions with different configurations of the encoded connector.

Definition 2.5 Let $f \in \mathsf{Orc}^-$ and $F = [\![f]\!] : \{I_f\} \cup \boldsymbol{V_f} \to \boldsymbol{O_f}$. We define two partitions of reachable configurations of F:

$$F^{-I} = \left\{F' \mid F \xrightarrow{a_1} \cdots \xrightarrow{a_n} F' \wedge I_f \notin \mathsf{nodes}(a_1 \cup \ldots \cup a_n) \wedge n \geq 0\right\}$$
$$F^{+I} = \left\{F' \mid F \xrightarrow{a_1} \cdots \xrightarrow{a_n} F' \wedge I_f \in \mathsf{nodes}(a_1 \cup \ldots \cup a_n) \wedge n \geq 1\right\}.$$

The first set consists on the configurations of F after zero or more steps until the input node is fired, and the second set consists on the possible configurations after the input node is fired. Combining Definition 2.5 with Lemmas 2.3 and 2.4, we arrive at the following corollary.

Corollary 2.6 *Let $f \in \mathsf{Orc}^-$ and $F = [\![f]\!] : \{I_f\} \cup \boldsymbol{V_f} \to \boldsymbol{O_f}$. Then:*

- *If $H \in F^{-I}$, then $H \xrightarrow{a} H'$ implies $\mathsf{nodes}(a) \cap \boldsymbol{O_f} = \emptyset$, and for $H'' \in F^{-I}$, $H'' \xrightarrow{a} H$ implies that either $\widehat{a} = \emptyset$ or $\widehat{a} = \tau$, and $I_f \notin \mathsf{nodes}(a)$.*
- *If $H \in F^{+I}$, then $H \xrightarrow{a} H'$ implies $I \notin \mathsf{nodes}(a)$ 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 [21].

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

Definition 2.8 Let f be an Orc expression, C be a a connector, and $a \subseteq$ *BaseEvents*. Define weak bisimulation $\sim$ as a relation between Orc expressions and connector (configurations), such that $f \sim C$ whenever:

(i) if $f \xrightarrow{a} f'$, then $\exists b, C'$ such that $\widehat{b} = a$, $C \overset{b}{\Longrightarrow} C'$ and $f' \sim C'$; and

(ii) if $C \xrightarrow{a} C'$, then there is an expression f' such that $f \overset{\widehat{a}}{\Longrightarrow} f'$ and $f' \sim C'$.

We introduce Lemma 2.9 to capture 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 $h \in \mathsf{Orc}^-$ and $h_v \overset{def}{=} [v/x].h$, 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.,

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

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 $h = M(\overline{p})$, and $x \in \overline{p}$, because that is the only place where x can be used. We prove that, in this case, the possible behaviour of $[\![h]\!]$ 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 base case.

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

Proof Outline. This theorem follows by induction on the structure of h. For each case, we define the relation $\sim$, and prove that it is a bisimulation. □

3 Encoding Reo into Orc

The encoding from Reo connectors into Orc expressions is more complex and, unlike the dual encoding, cannot be achieved in a compositional manner. Due to lack of space, we only present a brief discussion of this encoding, providing enough intuition for the limitations.

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.* [14], where the authors encode in Orc the set of workflow patterns proposed by Van der Aalst [2]. A similar approach could be attempted to encoding Reo into Orc, using higher level primitive site calls that can synchronize with each other, but the encoding will not be compositional. For example, $\mathcal{C}(Sync_{A,B} * Merger_{B,C,D})$ does not correspond to $\mathcal{C}(Sync_{A,B}) \mid \mathcal{C}(Merger_{B,C,D})$, 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.

In any case, the encoding would correspond roughly to the implementation of one of the known algorithms to combine the synchronous constraints imposed by Reo primitives, such as Connector Colouring [9]. The main troublesome issues are: *inversion of control*: in Orc the sites cannot initiate contact with the orchestrator, which is the opposite of how Reo is; *data structures for colouring tables, channels, connector topology, etc.*: Orc provides no data structures which could be required to manage the possible behaviours in a Reo connector; *propagation of synchrony*: as Orc is highly asynchronous, implementing the synchrony propagation in Reo either needs transactions or global consensus, which cannot be implemented without adding sufficiently expressive primitives to Orc, and both are fragile in the presence of failure; *handling failure*: if external sites implement key ingredients required to encode Reo, these sites must be responsive and must avoid failure.

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 the composition yields the encoding of the main expression. On the other hand, we anticipate that encoding of Reo into Orc would be global, since each Reo connector needs to be considered as whole, and the encoding becomes an implementation of a Reo engine. An interesting question is to determine precisely which set of Orc primitives would be needed to give a local encoding of Reo in Orc.

Cook *et. al.* presented a synchronous semantics to Orc [13] where all events other than external response are processed as soon as possible. This allows, for example, to impose an order by which 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 Reo semantics.

These issues regarding synchronous and asynchronous communication can also be found in the context of the π-calculus. To have synchrony in the π-calculus means that it is possible to constrain a fixed-sized tuple of more than one channel such that each element can only be executed if all the other elements of this tuple can also be executed. This notion of synchrony is closely related to synchrony in Reo, since Reo allows for the definition of constraints regarding the firing of more than one port in the same step. Unlike Reo, the π-calculus does not propagate synchrony through composition. In this area Palamidessi [23] compared the expressiveness of synchronous and asynchronous π-calculus, saying that the synchronous π-calculus cannot be encoded in the asynchronous π-calculus. Carbone and Maffeis [12] extended this result proving that the expressive power of synchronous π-calculus that can synchronize at most n channels is less than of the one that can synchronize at most $n + 1$ channels. These results suggest that Reo cannot be encoded in Orc, because Orc is asynchronous whereas Reo can synchronize an arbitrary number of ports.

4 Discussion

We now compare Orc and Reo on some issues of philosophy and design.

Focus of Control In Orc control lies with the orchestrator: an Orc expression initiates contact with external sites. On the other hand, Reo assumes that control is initiated externally to a connector by a component. The take or write is subsequently handled by the connector. This is how Reo coordinates, by controlling when takes and writes succeed, though from the perspective of web services, control is inverted.

Component/Service Instantiation In Reo components are attached externally to a connector, whereas Orc can dynamically initiate contact with services. 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 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, 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, $f >x> g$, 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 circuitry that is used only once. Reo does offer some operations for plugging and unplugging primitive connectors, but no decent high-level abstractions for dynamic reconfiguration and subsequent garbage collection of connectors exist. In recent work by Koehler *et al.* [19] the authors present a high level approach to the rewriting of connectors, which can be the basis for more 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. Much of this work remains to be done. In fact, the jury is still out regarding whether synchrony is a good idea in a distributed setting, even though it has the potential to offer better abstractions.

5 Related Work

Bruni *et al.* [7] present a static encoding of Orc into Petri nets. Their encoding is not, however, 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 faithfully 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 based on the homonymous process calculus. 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 be all present so the pattern can be recognized. However, the Join-calculus is not highly synchronous like Reo, as it does not propagate synchrony through composition. The relation between the Join calculus and Reo is left for future work.

Many other coordination languages exist, and these are compared in some earlier surveys [22,3]. 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 [5]. Such languages are useful for programming reactive systems, though seem not to be directly useful for coordinating distributed systems. To remedy this situation, the GALS (globally asynchronous, locally synchronous) model [10] 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 simple form of transaction [4]. This has not yet been convincingly demonstrated.

A method for comparing expressiveness was proposed by de Boer and Palamidessi [15], 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 [6]. However, it is not clear how Reo or Orc would fit this setting.

6 Conclusion and Future Work

We have briefly compared Orc and Reo, by encoding Orc^- expressions into Reo, by discussing the encoding the other way, and by comparing a number of design decisions. Orc is highly asynchronous and deals well with failure. Reo supports a high degree of synchrony, and potentially high-level abstractions. There is a lot more we could have said. 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. To extend our encoding to full Orc requires either recursively-defined or dynamically reconfigurable Reo connectors. These extensions to Reo are interesting on their own, and are the subject of future work.

References

[1] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logical specifications for timed component connectors. *Software and Systems Modeling (SoSyM)*, 6(1):59–82, March 2007.

[2] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[3] Jean-Marc Andreoli, Chris Hankin, and Daniel Le Metayer, editors. *Coordination Programming: Mechanisms, Models and Semantics.* Imperial College Press, 1996.

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

[5] G. Berry. *The Foundations of Esterel.* MIT Press, 2000.

[6] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination via shared dataspaces. *Sci. Comput. Program.*, 46(1-2):71–98, 2003.

[7] Roberto Bruni, Hernán C. Melgratti, and Emilio Tuosto. Translating Orc features into Petri nets and the Join calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2006.

[8] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[9] Dave Clarke, David Costa, and Farhad Arbab. Connector Colouring I: Synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007.

[10] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[11] Tom Chothia and Jetty Kleijn. Q-automata: Modelling the resource usage of concurrent components. In *The 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2006)*, 2006.

[12] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nord. J. Comput.*, 10(2):70–98, 2003.

[13] William R. Cook and Jayadev Misra. Computation orchestration, a basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, 2007.

[14] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006.

[15] Frank S. de Boer and Catuscia Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128–157, 1994.

[16] Daniel P. Friedman and David S. Wise. An indeterminate constructor for applicative programming. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada*, pages 245–250, January 1980.

[17] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[18] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006.

[19] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. In *The 6th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2007)*, 2007.

[20] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, March 2007.

[21] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.

[22] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *M. Zelkowitz (Ed.), The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998.

[23] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265, 1997.