



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Specification-oriented orchestration

Mercy N. Njima and J. W. Sanders

September 2009

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

Govindan Parayil, Director, a.i.



The United Nations
University

UNU-IIST

International Institute for
Software Technology

P.O. Box 3058
Macao

Specification-oriented orchestration

Mercy N. Njima and J. W. Sanders

Abstract

Orchestration of web services using low-level languages like BPEL is akin to performing Software Engineering with just the target programming language. Orc has been proposed as an elegant and more abstract design language. But much of its elegance derives from its functional style which must be matched, by the Software Engineer, with the state-based nature of systems as a whole. This paper explores the interface between the state-based and functional views of orchestration by studying a more abstract semantics for Orc. Sound laws are established and the resulting formalism is used on examples to demonstrate the top-down development of implementations.

Mercy Njima graduated from the University of Nairobi with a B.Sc (Mathematics) degree and obtained a Post-Graduate diploma in Mathematical Sciences from the African Institute for Mathematical Sciences in June, 2008. From July 2008 to January 2009 she was a UNU-IIST Fellow. She is currently pursuing a PhD at the Institutions Markets Technologies (IMT) Institute For Advanced Studies in Lucca.

Jeff Sanders is Principal Research Fellow at UNU-IIST. His interests lie largely in Formal Methods.

The authors acknowledge assistance from the Macao Science and Technology Development Fund under the PEARL project, grant number 041/2007/A3.

Contents

1	Introduction	1
2	Orc outlined	2
3	Specification-oriented semantics	3
4	Sequence notation	4
4.1	Interleaving	4
4.2	Bag interleaving	7
5	Semantics	8
5.1	Types	8
5.2	Sites	9
5.3	Basic sites	10
5.4	Expressions	11
5.5	Independent parallel invocation	11
5.6	Sequential invocation	12
5.7	Asymmetric parallel invocation	15
6	Examples	18
6.1	News	18
6.2	Fork-join	19
6.3	Mini network	20
7	Conclusion and further work	23

1 Introduction

Now that web services have become fundamental in defining business processes and transactions, twin approaches of ‘orchestration’ and ‘choreography’ have emerged for the composition of different services and for inter-enterprise collaboration. *Orchestration* is the coordination by one service of others; that service acts as ‘conductor’. *Choreography* refers to the decentralised, or distributed, coordination of several services in which no single service need be in control. In this paper we concentrate on the latter, of which the former is a special case.

There is much to be said for applying to web services our experience from programming: the use of higher-level languages makes design, programming and maintenance easier. The *de facto* standard BPEL [1], for all its advantages, can scarcely be called high-level. So languages like Orc [2, 9, 10, 11, 12] are of great interest. Orc elevates the level of design and implementation by using a functional style. That paradigm has already been found to be of great value in programming [3]; it is particularly useful in, though not restricted to, situations based on pattern matching. So it is of great interest to see how Orc combines a functional style with ideas from process algebra.

Our interest lies in the integration of Orc in the Formal Methods approach to Software Engineering. Orc’s natural partners there are perhaps process algebras (*e.g.* [13, 6]) because they largely abstract state for the sake of coordination and communication. However we study instead the approach of state-based specification and refinement, as exemplified by Z [14, 5]. Such methods are couched in terms of abstract data structures built from sets, relations, sequences *etc* and forsake executability for clarity. But being both state-based and application oriented, they do not match directly the functional, untyped, style of Orc. So of particular interest is how to specify service orchestration in a state-based manner and then find an Orc implementation. So we are interested in specifications in Z and implementations in Orc.

Several semantics have been given for Orc; naturally they have progressed with the language itself. Regardless of whether or not they incorporate more advanced features, like real time, their primary purpose has been correctness of implementation. That has determined the observables in terms of which each semantics is expressed. A fully abstract semantics of Orc that serves to guide implementors of the language must be founded on sufficient observables to ensure that equal Orc programs have the same semantics, and conversely. For example, an implementation is concerned with the values of bound variables, like x in the sequential invocation ($A \text{ >}x\text{ <} B$). Similarly it regards program that access different sites, or the same sites a different number of times, as different: site call is observable, just like the values published by the program.

But the user (and specifier) of web services just wants results. The sites called by a program are as irrelevant as the local variables used in a program. From that point of view, of importance is just the values that the program causes to be published. Binding of local variables are equally irrelevant. For such purposes, implementation-biased semantics are too fine; we are interested in more abstract observables.

Parameters are untyped in Orc. Abstracting types aids in the simplicity (and effectiveness) of the language; and if required types can always be restored by standard type-inference techniques (as used for example in Haskell [3]). But for purposes of specification, interfaces between modules provide essential

information and so types are important. Thus here we consider variables to be typed. Fortunately the result compromises the language scarcely at all.

2 Orc outlined

Orc is a language for the high-level orchestration of services. It coordinates interactions among basic subsystems, called sites, by use of a small number of combinators [2, 12]. The basis for its design is to allow integration of components and is founded on the premise that structured concurrent programs should be developed much like structured sequential programs, by decomposing a problem and combining the solutions with the combinators of the language.

An Orc program consists of a goal expression (either primitive or a combination of two expressions) and a set of definitions. The goal expression is evaluated in order to run the program. The definitions are used in the goal and in other definitions. A component is generally called a service; Orc adopts the more neutral term site which is the most primitive Orc expression. It represents an external program and is said to publish a value when a value is returned in response to a call.

An Orc expression B is *x-free*, for variable x , means that x does not appear in B as a free variable (and, as usual, bound occurrences do not matter because in them x can be replaced by a fresh variable if required).

A combinator in Orc combines two expressions to form another expression. Primitive expressions are merely site calls. By applying the combinators, Orc creates arbitrarily complex expressions and a hierarchical structure of the program which is amenable to inductive analysis, [10].

We consider the following combinators, using expressions A and B .

- Independent parallel invocation, $(A \mid B)$, allows independent concurrent execution of A and B . The sites called by A and B individually are called by $(A \mid B)$ and the values published by A and B are published by $(A \mid B)$.
- Sequential invocation, $(A \succ x \succ B)$, initiates a new instance of B for every value published by A whose value is bound to name x in that instance of B . The values published by $(A \succ x \succ B)$ are all instances of those published by B .
- Asymmetric parallel invocation, $(A \prec x \prec B)$, evaluates A and B independently, but the site calls in A that depend on x are suspended until x is bound to a value; the first value from B is bound to x , evaluation of B is then terminated and suspended calls in A are resumed; the values published by A are those published by $(A \prec x \prec B)$.

3 Specification-oriented semantics

Sites are nondeterministic: at the very least a site may publish either nothing, or a value; but typically the value itself may not be uniquely determined. Furthermore, when sites are combined independently in parallel, even if each is deterministic the result is not, because the order of publication is not determined and so interleavings occur. Thus each site, and the result of each Orc computation, is modelled as the *set* of what it may publish; that is, as a set of sequences of publishable values.

It might be hoped that a simpler, sequence-free, model would suffice. Then each site would be represented by a set of those values it can publish. Then the site $\mathbf{0}$ which fails to publish would have semantics $\{\}$. The semantics of the independent parallel invocation of A and B would be the union of the semantics of A and B . The semantics of the sequential invocation of A then B would be the union of the semantics of all invocations of B . However with asymmetric parallel invocation ($A \lt x \lt B$) we strike a problem. It is only the first publications of B that are taken as input by A , and the set model is not powerful enough to distinguish those. (Curiously enough, a traces (*i.e.* finite sequences) model was given for all but asymmetric parallel invocation, the very combinator which demands it.) Rather than augment the publications of a site with information indicating which are first, we simply consider sequences of publications.

We consider two possibilities for that set. In the first, following trace semantics in process algebra [7], the result of an Orc computation consists of a prefix-closed set of sequences of published values. The program is thought of as publishing the values successively (even though it has a single initial input) and prefix closure reflects the fact that values accumulate with time. This view is reflected in existing Orc semantics [8, 9].

In the second view, the nondeterminism is again captured by a set of sequences of values, but prefixes are excluded: only maximal sequences (infinite if necessary) are observed. That is because no environment in Orc is able to block outputs and so there is no reason (as there is in process algebra) to include all prefixes to which different behaviours can be attached (by failures [7], for example). That is the view taken here.

Thus the result of an Orc program is a set of sequences of published values. The only healthiness condition is that it must contain the empty sequence (corresponding to lack of response); in principle any other sequence could be published. To state that, some notation is required.

If there were an accepted, definitive, semantics of Orc then we would expect to prove that ours is more abstract: any law sound with respect to the ‘canonical’ semantics would be sound in ours. But in spite of various semantic models [2] consensus does not seem to have been reached. Thus we instead verify soundness from first principles.

This paper gives a specification-oriented semantics for Orc, proves laws showing how close the result is to being a Kleene algebra, and works various examples.

4 Sequence notation

We use list notation [3] for sequences, though we write $\text{seq } \mathbb{T}$ for the type of sequences of type \mathbb{T} . Sequence comprehension is written $[\dots]$ and the empty sequence written $[]$; we write $xs[i]$ for the i th element of sequence xs (starting from 0); we use $x:xs$ for a sequence with head x and tail xs ; we write $(x:)$ for the function which appends x to the front of a sequence; and use $++$ for catenation of sequences. We write $f \langle E \rangle$ for the pointwise promotion of a function f to a subset of its domain,

$$f \langle E \rangle := \{f(e) \mid e \in E\}.$$

Thus, for example,

$$(0:)\langle \{ [], [1,2], [0,1] ++ [1] \} \rangle = \{ [0], [0,1,2], [0,0,1,1] \}.$$

The relation *in* relates an element to any sequence containing it

$$\begin{aligned} \underline{in} &: E \leftrightarrow \text{seq } E \\ e \underline{in} es &:= \exists as, bs. es = as ++ [e] ++ bs. \end{aligned}$$

4.1 Interleaving

The *interleaving* of two finite sequences consists of the set of sequences in which the two sequences appear as subsequences, and the frequency of elements in the resulting sequence equals the combination of those in the two subsequences. Thus the sequence $[0,0,2,1]$ is an interleaving of $[0,1]$ and $[0,2]$, but $[0,2,1]$ is not. From that informal definition follows a formal explicit one. But instead we give a recursive definition because it is perhaps clearer and anyway executable as a functional program.

$$\begin{aligned} \text{inter} &:: \text{seq } \mathbb{T} \rightarrow \text{seq } \mathbb{T} \rightarrow \mathbb{P} \text{seq } \mathbb{T} \\ \text{inter } [] \ ys &= \{ys\} \\ \text{inter } (x:xs) [] &= \{x:xs\} \\ \text{inter } (x:xs) (y:ys) &= (x:)\langle \text{inter } xs (y:ys) \rangle \\ &\cup \\ &(y:)\langle \text{inter } (x:xs) ys \rangle \end{aligned}$$

A version in Haskell replaces the resulting set by a sequence and union by catenation.

```
> inter :: [a] -> [a] -> [[a]]
> inter [] bs          = [bs]
> inter (a:as) []      = [a:as]
> inter (a:as) (b:bs) = (map (a:)) (inter as (b:bs))
                        ++
                        (map (b:)) (inter (a:as) bs)
```

Interleaving is extended from two sequences to a *sequence* of sequences, and converted to Haskell, as follows.

$$\begin{aligned} iinter &:: \text{seq seq } \mathbb{T} \rightarrow \mathbb{P} \text{ seq } \mathbb{T} \\ iinter [] &= \{[]\} \\ iinter (xs:yss) &= \cup\{inter\ xs\ ys \mid ys \in iinter\ yss\} \end{aligned}$$

```
> iinter :: [[a]] -> [[a]]
> iinter [] = [[]]
> iinter (xs:yss) = concat [ inter xs ys | ys <- iinter yss ]
```

The following result establishes simple properties of those functions. The fifth relates the two definitions, whilst 6 and 7 enumerate interleavings. We write $\#E$ for the cardinality of E and $n!$ for the factorial of n .

Interleaving lemma

1. $inter [] [] = \{[]\}$
2. $inter\ xs\ ys = inter\ ys\ xs$
3. $iinter\ [xs] = \{xs\}$
4. For any permutation yss of xss , $iinter\ xss = iinter\ yss$. Thus $iinter$ is well defined on bags.
5. $inter\ xs\ ys = iinter\ [xs,ys]$
6. $\#(inter\ xs\ ys) = (\#xs + \#ys)! / (\#xs)! (\#ys)!$
7. $\#(iinter\ xss) = (\sum_{xs \in xss} \#xs)! / (\prod_{xs \in xss} (\#xs)!)$
8. For any $as, bs : \text{seq } \mathbb{T}$, $\exists ys : (inter\ as\ bs) \cdot y \underline{in}\ ys$ iff $(y \underline{in}\ as) \vee (y \underline{in}\ bs)$.

Proof

The first five claims result from the definitions and simple calculus; we prove just two of them. For 3,

$$\begin{aligned} &iinter\ [xs] \\ &= &&\text{calculus} \\ &iinter\ (xs : []) \\ &= &&\text{definition of } iinter \\ &\cup\{inter\ xs\ ys \mid ys \in iinter\ []\} \\ &= &&\text{definition of } iinter \\ &\cup\{inter\ xs\ ys \mid ys \in \{[]\}\} \\ &= &&\text{calculus} \end{aligned}$$

$$\begin{aligned}
& \cup\{inter\ xs\ []\} \\
& = \hspace{15em} \text{definition of } inter \text{ (cases } xs \text{ empty or not)} \\
& \cup\{\{xs\}\} \\
& = \hspace{15em} \text{calculus} \\
& \{xs\}.
\end{aligned}$$

For part 5,

$$\begin{aligned}
& iinter\ [xs,ys] \\
& = \hspace{15em} \text{calculus} \\
& iinter\ (xs : [ys]) \\
& = \hspace{15em} \text{definition of } iinter \\
& \cup\{inter\ xs\ zs \mid zs \in iinter\ [ys]\} \\
& = \hspace{15em} \text{Part 3} \\
& \cup\{inter\ xs\ zs \mid zs \in \{ys\}\} \\
& = \hspace{15em} \text{calculus} \\
& inter\ xs\ ys.
\end{aligned}$$

The enumerations are standard. An interleaving of two sequences, of lengths m and n , may conveniently be visualised as a path in the discrete plane \mathbb{N}^2 from the origin $(0,0)$ to (m,n) that takes a horizontal or vertical unit jump at each of its $m+n$ steps. The number of interleavings thus equals the number of such paths. More generally, for k sequences of lengths n_i , $0 \leq i < k$, the plane is replaced by k -dimensional discrete space \mathbb{N}^k and the path goes from the origin there to the point (n_0, \dots, n_{k-1}) . Claims 6 and 7 are then straightforward.

Alternatively, from the definition of *inter*, letting $T(m,n)$ denote the number of interleavings of sequences of length m and n ,

$$\begin{aligned}
T(0,n) &= 1 \\
T(m,0) &= 1 \\
T(m+1,n+1) &= T(m,n+1) + T(m+1,n)
\end{aligned}$$

which has generating function

$$F_m(x) = (1-x)^{-m}$$

whence the result follows using elementary calculus, since

$$T(m, n) = \frac{1}{n!} \frac{d^n F_m}{dx^n} \Big|_{x=0}$$

isolates the coefficient of x^n . □

The enumerations have been found useful in checking small examples, as have the Haskell implementations.

In passing we note that Part (8), which has the effect of distributing y *in* into two interleaved sequences, is not true if y is replaced by a sequence of length greater than 1. It is thus analogous to the division law for primes: if a prime divides a product of two numbers then it divides either one number or the other ($p \mid mn \Rightarrow p \mid m \vee p \mid n$); but that is not true for composite p .

4.2 Bag interleaving

Part 4 of the Interleaving lemma enables us to extend *iinter* from sequences of sequences (*i.e.* $\text{seq seq } \mathbb{T}$) to bags of sequences, by choosing some ordering of the bag (*i.e.* some sequence whose bag of values equals that of the bag). But what we really need is the extension of such a function (having domain ‘bags of sequences’) to one having domain ‘bags of bags of sequences’; it is defined to return the set of all interleavings of sequences, one from each bag in the bag. For example, writing bag comprehension as $\wr \dots \wr$, it acts

$$\wr \wr [], [0] \wr, \wr [], [a] \wr, \wr [], [0] \wr \wr \mapsto \{ [], [0], [a], [0, a], [a, 0], [0, 0], [0, a, 0], [a, 0, 0], [0, 0, a] \}.$$

We write $\mathbb{B}\mathbb{T}$ for the type of bags of type \mathbb{T} . The required definition is

$$\begin{aligned} \text{binter} &:: \mathbb{B}\mathbb{B} \text{ seq } \mathbb{T} \rightarrow \mathbb{P} \text{ seq } \mathbb{T} \\ \text{binter } \wr \wr &= \{ [] \} \\ \text{binter } bgg &= \cup \{ \text{iinter } [bs_0, bs_1, \dots, bs_{n-1}] \mid bs_i \in bss[i] \} \quad \text{if } bgg \neq \wr \wr \end{aligned}$$

where bss is some ordering of the bag bgg and n is the size of bgg (and so also the length of sequence bss). The observations above show that the definition is independent of the choice of ordering bss .

It may seem that a more satisfactory definition would replace the result set by a bag. But the present definition suits our purposes because it has the property of suppressing repetitions in the result. We shall also use *binter* on a bag of sets of sequences, treating the set as a bag in which each member occurs just once. Our applications ensure that each ‘inner’ bag (or set) of sequences contains at least the empty sequence; however it may be that the ‘outer’ bag is empty.

In Haskell,

```

> binter :: [[a]] -> [a]
> binter []      = []
> binter (xss:ysss) = concat [ inter xs ys | xs <- xss, ys <- binter ysss ] .

```

Elementary properties of the bag-interleaving function *binter* are as follows. The bag interleaving of a singleton bag $\langle bg \rangle$ consists of the set of all sequences in that single bag *bg*. The bag interleaving function is related to the previous interleaving functions by taking the argument to be a bag with two elements; see part 3. The bag interleaving of a disjunction is the bag interleaving of the bag interleavings of the disjuncts (themselves sets, treated as bags). Finally bag interleaving preserves prefix closure and so, in particular, containment of the empty sequence.

Bag-interleaving lemma

1. Function *binter* is well defined.
2. For any bag *bg* of sequences, $binter \langle bg \rangle = \{bs \mid bs \in bg\}$.
3. For $As, Bs : \mathbb{PT}$, $binter \langle As, Bs \rangle = \cup \{inter \ as \ bs \mid as \in As, bs \in Bs\}$. In particular

$$binter \langle \langle as \rangle, \langle bs \rangle \rangle = iinter [as, bs] = inter \ as \ bs$$
 and, for bag *bg* of sequences,

$$binter \langle \langle [] \rangle, bg \rangle = \{bs \mid bs \in bg\}.$$
4. $binter \langle D \mid p \vee q \rangle = binter \langle binter \langle D \mid p \rangle, binter \langle D \mid q \rangle \rangle$
5. If each $bg \in bgg$ contains $[]$ then so does *binter bgg*.
6. More generally, if each $bg \in bgg$ is prefix closed (considered as a set: if $bs \in bg$ then each prefix of *bs* is also in *bg*), so is *binter bgg*. \square

5 Semantics

5.1 Types

For any type \mathbb{T} , the type of published values from \mathbb{T} includes all possible sequences (finite \mathbb{T}^* or infinite \mathbb{T}^∞) of values from \mathbb{T} in any order. The empty sequence corresponds to ‘no return’ and an infinite sequence occurs if a site publishes *ad infinitum*.

$$pubs \mathbb{T} := \mathbb{T}^* \cup \mathbb{T}^\infty.$$

(Notice that we do not use prefix-closed sets of finite sequences as would be the case for instance in most models of process algebra [7].) It is also convenient to have notation for sequences of length at most a natural number *n*

$$(1) \quad pubs^n \mathbb{T} := \{ts : pubs \mathbb{T} \mid \#ts \leq n\}$$

Each Orc program A is denoted, as discussed below, by a function from its input type AIn to a set of sequences of its publication type, $AOut$, with the property that the empty sequence is included (because it models failure to publish). Thus, for any type \mathbb{T} , we define the type

$$\langle \mathbb{T} \rangle := \{E : \mathbb{P} \text{pubs } \mathbb{T} \mid [] \in E\}.$$

5.2 Sites

The Orc philosophy is to concentrate on orchestration and abstract details wherever possible of constituent sites. It is supposed that any site may fail to publish; thus sites behave nondeterministically. When giving an informal description of a site's behaviour we shall implicitly assume that failing to publish is one possibility. (For emphasis we use the longer alternative: 'a site *may* publish p ' (as well as failing to publish).) When sites are combined, for example in independent parallel invocation, nondeterministic choice of publication becomes greater. So it makes little difference to the theory to assume that a site may publish more than a single value; for uniformity of our theory, we suppose so.

A site may be invoked with, or in the absence of, input values. If A is a site name and $x : AIn$ a list of its inputs required for A 's invocation, then $A(x)$ denotes the invocation of A with those inputs. It occurs only when all inputs have been supplied, in which case the result is presumed to be a set of sequences of type $AOut$. The empty sequence represents failure to publish. Publication of a single value p is represented by the singleton sequence $[p]$. The reason for using sequences (rather than some virtual element \perp for failure to publish, and then a set of 'published' values $\{\perp, p, \dots\}$) is to facilitate the interleaving of publications when sites are combined. (So instead we use $\{[], [p], \dots\}$.)

Specific sites define the relationship between their input parameters and their publications. We denote the semantics of a call to site A by $\llbracket A(x) \rrbracket$. Thus it is a set of sequences of published values which contains the empty sequence

$$(2) \quad \llbracket A(x) \rrbracket :: \langle AOut \rangle.$$

The semantics of a site A whose invocation requires inputs is a function from AIn to such sets of sequences.

$$\begin{aligned} & \llbracket A \rrbracket :: AIn \rightarrow \langle AOut \rangle \\ (3) \quad & \llbracket A \rrbracket(x) := \llbracket A(x) \rrbracket \end{aligned}$$

If invocation does not require input, rather than consider the constant function over some unnecessary set of inputs, we use just (2) for the semantics $\llbracket A \rrbracket$ of the site A itself.

5.3 Basic sites

The site $(\mathbf{if}_{\mathbb{T}} c)$ publishes a value of \mathbb{T} iff predicate c holds. Its input is the state on which condition c is defined, which for simplicity we omit.

$$\begin{aligned} & \llbracket \mathbf{if}_{\mathbb{T}} c \rrbracket :: \langle \mathbb{T} \rangle \\ (4) \quad & \llbracket \mathbf{if}_{\mathbb{T}} c \rrbracket := \bigcup \{ \{ [], [t] \} \mid t : \mathbb{T} \} \text{ if } c \text{ else } \{ [] \} \end{aligned}$$

The site $\mathbf{0}$ fails to publish (whatever the type \mathbb{T}). It is the same as $(\mathbf{if}_{\mathbb{T}} \text{false})$.

$$\begin{aligned} (5) \quad & \llbracket \mathbf{0}_{\mathbb{T}} \rrbracket := \{ [] \} \\ & \mathbf{0}_{\mathbb{T}} = \mathbf{if}_{\mathbb{T}} \text{false} \end{aligned}$$

The site $\mathbf{signal}_{\mathbb{T}}$ publishes a value immediately. It is the same as $(\mathbf{if}_{\mathbb{T}} \text{true})$.

$$\begin{aligned} & \llbracket \mathbf{signal}_{\mathbb{T}} \rrbracket :: \langle \mathbb{T} \rangle \\ (6) \quad & \llbracket \mathbf{signal}_{\mathbb{T}} \rrbracket := \bigcup \{ \{ [], [t] \} \mid t : \mathbb{T} \} \\ & \mathbf{signal}_{\mathbb{T}} = \mathbf{if}_{\mathbb{T}} \text{true} \end{aligned}$$

Those sites publish arbitrary values; the site $(\mathbf{let}_{\mathbb{T}} t)$ publishes $t : \mathbb{T}$.

$$\begin{aligned} & \llbracket \mathbf{let}_{\mathbb{T}} t \rrbracket :: \langle \mathbb{T} \rangle \\ (7) \quad & \llbracket \mathbf{let}_{\mathbb{T}} t \rrbracket := \{ [], [t] \} \end{aligned}$$

Invocation of those sites does not require input. The site $\mathbf{echo}_{\mathbb{T}}$ publishes its input. Thus it equals $\mathbf{let}_{\mathbb{T}} t$ where t is the input.

$$\begin{aligned} & \llbracket \mathbf{echo}_{\mathbb{T}} \rrbracket :: \mathbb{T} \rightarrow \langle \mathbb{T} \rangle \\ (8) \quad & \llbracket \mathbf{echo}_{\mathbb{T}}(t) \rrbracket := \{ [], [t] \} \\ & \mathbf{echo}_{\mathbb{T}}(t) = \mathbf{let}_{\mathbb{T}} t \end{aligned}$$

When appropriate, in any of those sites the subscript \mathbb{T} will be suppressed.

5.4 Expressions

As usual, we call an expression $A = B$ between Orc expressions A and B a *law* iff it is sound with respect to the semantics: $\llbracket A \rrbracket = \llbracket B \rrbracket$. Whilst there is much to be said for eliding the semantic brackets and identifying an expression with its semantics, we have chosen here to take the more rudimentary approach for conceptual clarity and to emphasise that our semantic model is more abstract.

5.5 Independent parallel invocation

The *independent parallel invocation* of expressions A and B publishes anything published by A or B independently, preserving the order in which each publishes. The semantics thus consists of a function from pairs of inputs (if they are required) to sets of sequences in $\langle AOut \cup BOut \rangle$ obtained by interleaving sequences in the semantics of the calls $A(x)$ and $B(y)$ since they can be published at the same time.

$$\begin{aligned} (A \mid B) &:: AIn \times BIn \rightarrow \langle AOut \cup BOut \rangle \\ \llbracket A \mid B \rrbracket(a, b) &:= binter \wr \llbracket A(a) \rrbracket, \llbracket B(b) \rrbracket \wr \\ &= \bigcup \{ inter \text{ as } bs \mid as \in \llbracket A(a) \rrbracket, bs \in \llbracket B(b) \rrbracket \} \end{aligned}$$

The equivalence of the two expressions follows by part (3) of the Bag-interleaving lemma. In the first expression, the function *binter* is applied to a bag of sets (as anticipated in Section 4.2); alternatives are for the bag to be a set, or the set to be a bag. But the ‘outer’ bag cannot be replaced by a set because that might lose interleavings; for example, the sites A and B might be the same. And the inner sets do not need to be bags, since frequency is not necessary in expressing the result of a call $\llbracket A(a) \rrbracket$.

The order in which multiple independent parallel invocations are combined does not matter—operator \mid is commutative and associative—and independent parallel invocation of A with a site that fails to publish results in A :

Independent parallel theorem

The semantics of $(A \mid B)$ is well defined and \mid is commutative, associative and has unit $\mathbf{0}$.

$$(9) \quad (A \mid B) = (B \mid A)$$

$$(10) \quad (A \mid (B \mid C)) = ((A \mid B) \mid C)$$

$$(11) \quad (\mathbf{0} \mid A) = A$$

Proof

Well definedness here means simply that $\llbracket \]$ belongs to each element of $\llbracket A \mid B \rrbracket$; that follows by part 5 of the Bag-interleaving lemma. Both the commutativity law, (9), and the associativity law, (10), follow by well-definedness of *binter*.

For the left-unit law, (11), we reason interpreting a set which is an argument of *binter* as a bag in which its elements occur just once. If $a : AIn$,

$$\begin{aligned}
& \llbracket \mathbf{0} \mid A \rrbracket(a) \\
& = \text{semantics of independent parallel invocation} \\
& \text{binter } \llbracket \llbracket \mathbf{0} \rrbracket, \llbracket A(a) \rrbracket \rrbracket \\
& = \text{Law (5)} \\
& \text{binter } \llbracket \{\} \rrbracket, \llbracket A(a) \rrbracket \rrbracket \\
& = \text{Part 3 of the Bag-interleaving lemma (for the case of sets rather than bags)} \\
& \llbracket A(a) \rrbracket.
\end{aligned}$$

□

Laws (9), (10) and (11) justify use of a prefix form, $(\mid Bg)$, of independent parallel invocation applied to any finite bag Bg of Orc expressions. (Law (11) means that $(\mid \llbracket \rrbracket) = \mathbf{0}$.) However then we need notation for its invocation, ensuring that input parameters apply to the correct expressions. By $(\mid Bg)(bs)$ we mean that some ordering of the bag Bg is given and bs is a sequence of inputs in which $bs[i]$ belongs to the input type of the i^{th} Orc process $Bg[i]$.

5.6 Sequential invocation

If A fails to publish so does $(A \succ x \succ B)$; but otherwise the publications of $(A \succ x \succ B)$ are those of the independent parallel invocations of the invoked $B(b)$ where b is published by A . (Thus the case in which A fails to publish is the empty independent parallel invocation.)

$$\begin{aligned}
(A \succ x \succ B) &:: AIn \rightarrow \langle BOut \rangle \\
\llbracket A \succ x \succ B \rrbracket(a) &:= \bigcup \{ \text{binter } \llbracket \llbracket B(b) \rrbracket \mid b \text{ in } bs \rrbracket \mid bs \in \llbracket A(a) \rrbracket \}
\end{aligned}$$

When x is not free in B the abbreviation $(A \gg B)$ is used and referred to as *sequential composition*.

Sequential invocation theorem

The semantics of $(A \succ x \succ B)$ is well defined, and $(\succ x \succ)$ has $\mathbf{0}$ as left and right zero, **echo** as left unit, is typically¹ associative and is distributed by \mid on the left.

$$\begin{aligned}
(12) \quad & (\mathbf{0} \succ x \succ A) = \mathbf{0} \\
(13) \quad & (A \succ x \succ \mathbf{0}) = \mathbf{0} \\
(14) \quad & (\text{echo}_{AIn} \succ x \succ A) = A \\
(15) \quad & (A \succ x \succ (B \succ y \succ C)) = ((A \succ x \succ B) \succ y \succ C) \text{ provided } C \text{ is } x\text{-free} \\
(16) \quad & ((A \mid B) \succ x \succ C) = ((A \succ x \succ C) \mid (B \succ x \succ C))
\end{aligned}$$

Proof

Well definedness —that if $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$ contain the empty sequence then so does $\llbracket A \succ x \succ B \rrbracket$ — follows immediately from part 5 of the Bag-interleaving lemma.

For the left-zero law, (12), we reason

¹We adopt ‘typically’ to describe the condition, in the context of (15), that C is x -free.

$$\begin{aligned}
& \llbracket \mathbf{0} \succ x \succ A \rrbracket \\
& = \text{semantics of sequential invocation} \\
& \cup \{ \text{binter } \lambda \llbracket A(a) \rrbracket \mid a \text{ in } as \mid as \in \llbracket \mathbf{0} \rrbracket \} \\
& = \text{Law (5)} \\
& \cup \{ \text{binter } \lambda \llbracket A(a) \rrbracket \mid a \text{ in } [] \} \\
& = \text{calculus} \\
& \cup \{ \text{binter } \lambda \} \\
& = \text{definition of } \text{binter} \\
& \cup \{ \{ [] \} \} \\
& = \text{calculus and Law (5) again} \\
& \llbracket \mathbf{0} \rrbracket.
\end{aligned}$$

For the right-zero law, (13), we reason that for any input a to A ,

$$\begin{aligned}
& \llbracket A \succ x \succ \mathbf{0} \rrbracket(a) \\
& = \text{semantics of sequential invocation} \\
& \cup \{ \text{binter } \lambda \llbracket \mathbf{0}(o) \rrbracket \mid o \text{ in } os \mid os \in \llbracket A(a) \rrbracket \} \\
& = \text{Law (5)} \\
& \cup \{ \text{binter } \lambda \{ [] \} \mid o \text{ in } os \mid os \in \llbracket A(a) \rrbracket \} \\
& = \text{property of } \text{binter} \text{ and calculus} \\
& \{ [] \} \\
& = \text{Law (5) again} \\
& \llbracket \mathbf{0} \rrbracket.
\end{aligned}$$

For the left-unit law (14), we reason that for e of type Aln ,

$$\begin{aligned}
& \llbracket \mathbf{echo}_{Aln} \succ x \succ A \rrbracket(e) \\
& = \text{semantics of sequential invocation} \\
& \cup \{ \text{binter } \lambda \llbracket A(a) \rrbracket \mid a \text{ in } as \mid as \in \llbracket \mathbf{echo}_{Aln}(e) \rrbracket \} \\
& = \text{Law (8)} \\
& \cup \{ \text{binter } \lambda \llbracket A(a) \rrbracket \mid a \text{ in } as \mid as \in \{ [], [e] \} \} \\
& = \text{calculus} \\
& \cup \{ \text{binter } \lambda \llbracket A(e) \rrbracket \}
\end{aligned}$$

$$= \text{calculus and Part (2) of the Bag-interleaving lemma} \\ \llbracket A(e) \rrbracket .$$

For the associativity law, (15), we reason that if C is x -free then for any input a to A ,

$$\begin{aligned} & \llbracket (A \succ x \succ B) \succ y \succ C \rrbracket (a) \\ = & \text{semantics of sequential invocation} \\ & \bigcup \{ \text{binter } \wr \llbracket C(c) \rrbracket \mid c \underline{\text{in}} cs \mid cs \in \llbracket (A \succ x \succ B) \rrbracket (a) \} \\ = & \text{semantics of sequential invocation again} \\ & \bigcup \{ \text{binter } \wr \llbracket C(c) \rrbracket \mid c \underline{\text{in}} cs \mid cs \in \bigcup \{ \text{binter } \wr \llbracket B(b) \rrbracket \mid b \underline{\text{in}} bs \mid bs \in \llbracket A(a) \rrbracket \} \} \\ = & \text{Part (8) of the Interleaving lemma, calculus and } C \text{ independent of } c \\ & \bigcup \{ \text{binter } \wr \llbracket C \rrbracket \wr \mid \exists bs : \llbracket A(a) \rrbracket \cdot \exists b \underline{\text{in}} bs \cdot \exists cs : \llbracket B(b) \rrbracket \cdot c \underline{\text{in}} cs \} \\ = & \text{similarly} \\ & \bigcup \{ \text{binter } \bigcup \{ \text{binter } \wr \llbracket C(c) \rrbracket \mid c \underline{\text{in}} cs \mid cs \in \llbracket B(b) \rrbracket \} \mid \exists bs : \llbracket A(a) \rrbracket \cdot b \underline{\text{in}} bs \} \\ = & \text{semantics of sequential invocation again} \\ & \bigcup \{ \text{binter } \wr \llbracket (B \succ y \succ C)(b) \rrbracket \mid b \underline{\text{in}} bs \mid bs \in \llbracket A(a) \rrbracket \} \\ = & \text{semantics of sequential invocation yet again} \\ & \llbracket A \succ x \succ (B \succ y \succ C) \rrbracket (a) . \end{aligned}$$

For the left-distributivity law, (16), we reason that for any input a to A and b to B ,

$$\begin{aligned} & \llbracket (A \mid B) \succ x \succ C \rrbracket (a, b) \\ = & \text{semantics of sequential invocation} \\ & \bigcup \{ \text{binter } \wr \llbracket C(c) \rrbracket \mid c \underline{\text{in}} cs \mid cs \in \llbracket (A \mid B) \rrbracket (a, b) \} \\ = & \text{semantics of independent parallel invocation} \\ & \bigcup \{ \text{binter } \wr \llbracket C(c) \rrbracket \mid c \underline{\text{in}} cs \mid cs \in \bigcup \{ \text{inter } as \ bs \mid as \in \llbracket A(a) \rrbracket, bs \in \llbracket B(b) \rrbracket \} \} \\ = & \text{calculus} \\ & \text{binter } \wr \llbracket C(c) \rrbracket \mid \exists as : \llbracket A \rrbracket \cdot \exists bs : \llbracket B \rrbracket \cdot \exists cs : \text{inter } as \ bs \cdot c \underline{\text{in}} cs \} \\ = & \text{Part (8) of the Interleaving lemma, calculus and } C \text{ independent of } c \text{ again} \\ & \text{binter } \wr \llbracket C(c) \rrbracket \mid \exists as : \llbracket A \rrbracket \cdot c \underline{\text{in}} as \vee \exists bs : \llbracket B \rrbracket \cdot c \underline{\text{in}} bs \} \\ = & \text{Bag-interleaving lemma, part 4} \\ & \text{binter } \wr \text{binter } \wr \llbracket C(a) \rrbracket \mid \exists as : \llbracket A \rrbracket \cdot a \underline{\text{in}} as \} \wr, \text{binter } \wr \llbracket C(b) \rrbracket \mid \exists bs : \llbracket B \rrbracket \cdot b \underline{\text{in}} bs \} \} \\ = & \text{calculus} \end{aligned}$$

$$\begin{aligned}
& \text{binter } \lambda \{ \text{binter } \lambda \{ \llbracket C(a) \rrbracket \mid a \text{ in } as \} \mid as \in \llbracket A \rrbracket \}, \cup \{ \text{binter } \lambda \{ \llbracket C(b) \rrbracket \mid b \text{ in } bs \} \mid bs \in \llbracket B \rrbracket \} \} \\
& = \text{semantics of sequential invocation} \\
& \text{binter } \lambda \{ \llbracket A \langle x \rangle C \rrbracket, \llbracket B \langle x \rangle C \rrbracket \} \\
& = \text{semantics of independent parallel invocation} \\
& \llbracket (A \rangle x \rangle C) \mid (B \rangle x \rangle C) \rrbracket.
\end{aligned}$$

□

Law (13) fails in ‘standard’ Orc semantics [8, 9] and so is a reflection of our more abstract semantics. Indeed on the left the parameter x is instantiated but on the right it is not. So an implementation-oriented semantics distinguishes the two sides; but ours does not because from a user’s point of view they are equivalent in failing to output.

Sequential invocation ($\rangle x \rangle$) does not have a right unit because if invocation of the left-hand argument were to publish several values then several independent copies of the right-hand argument would be invoked and so their outputs would be interleaved; thus the order of the left-hand argument’s publications could not be preserved. But since that interleaving includes the left-hand argument’s publications, we have an inclusion when the right-hand expression is the site **echo**:

$$\llbracket A \rangle x \rangle \mathbf{echo}_{AOut} \rrbracket(a) \supseteq \llbracket A(a) \rrbracket.$$

Equality holds if A publishes at most one value or, more generally, if its set of publications is invariant under interleavings.

5.7 Asymmetric parallel invocation

If B fails to publish, so does $(A \langle x \rangle B)$; but otherwise $(A \langle x \rangle B)$ publishes the result of A ’s invocation with a first publication of B .

$$\begin{aligned}
(A \langle x \rangle B) &:: BIn \rightarrow \langle AOut \rangle \\
\llbracket A \langle x \rangle B \rrbracket(b) &:= \{ \langle \rangle \} \cup \cup \{ \llbracket A(a) \rrbracket \mid [a] \in \llbracket B(b) \rrbracket \}
\end{aligned}$$

The empty sequence is added to ensure healthiness; it is necessary because the right-hand union is empty if B fails to publish.

Asymmetric parallel theorem

The semantics of $(A \langle x \rangle B)$ is well defined and $(\langle x \rangle)$ has $\mathbf{0}$ as left and right zero, satisfies associativity and various exchange laws, including a semi-distributivity of \mid from the left and an exchange law with sequential invocation.

$$(17) \quad (\mathbf{0} \langle x \rangle A) = \mathbf{0}$$

$$(18) \quad (A \langle x \rangle \mathbf{0}) = \mathbf{0}$$

$$(19) \quad ((A \langle x \rangle B) \langle y \rangle C) = (A \langle x \rangle (B \langle y \rangle C))$$

$$(20) \quad ((A \langle x \rangle B) \langle y \rangle C) = ((A \langle y \rangle C) \langle x \rangle B) \quad \text{provided } B \text{ is } y\text{-free and } C \text{ is } x\text{-free}$$

$$(21) \quad ((A \mid B) \langle x \rangle C) = ((A \langle x \rangle C) \mid B) \quad \text{provided } B \text{ is } x\text{-free}$$

$$(22) \quad ((A \rangle y \rangle B) \langle x \rangle C) = ((A \langle x \rangle C) \rangle y \rangle B) \quad \text{provided } B \text{ is } x\text{-free.}$$

Proof

For the left-zero law, (17), we reason that for any input a to A ,

$$\begin{aligned} & \llbracket \mathbf{0} \langle x \rangle A \rrbracket(a) \\ &= \text{semantics of asymmetric parallel invocation} \\ & \{ \{\} \} \cup \cup \{ \llbracket \mathbf{0}(o) \rrbracket \mid [o] \in \llbracket A(a) \rrbracket \} \\ &= \text{Law (5)} \\ & \{ \{\} \} \cup \cup \{ \{\{\} \} \mid [o] \in \llbracket A(a) \rrbracket \} \\ &= \text{calculus} \\ & \{\{\} \} \\ &= \text{Law (5) again} \\ & \llbracket \mathbf{0} \rrbracket. \end{aligned}$$

For the right-zero law, (18),

$$\begin{aligned} & \llbracket A \langle x \rangle \mathbf{0} \rrbracket \\ &= \text{semantics of asymmetric parallel invocation} \\ & \{ \{\} \} \cup \cup \{ \llbracket A(a) \rrbracket \mid [a] \in \llbracket \mathbf{0} \rrbracket \} \\ &= \text{Law (5) and calculus} \\ & \{ \{\} \} \cup \{ \} \\ &= \text{Law (5) again} \\ & \llbracket \mathbf{0} \rrbracket. \end{aligned}$$

For the associative law, (19), we reason that for any input c to C ,

$$\begin{aligned} & \llbracket (A \langle x \rangle B) \langle y \rangle C \rrbracket(c) \\ &= \text{semantics of asymmetric parallel invocation} \\ & \{ \{\} \} \cup \cup \{ \llbracket A \langle x \rangle B \rrbracket(b) \mid [b] \in \llbracket C(c) \rrbracket \} \\ &= \text{semantics of asymmetric parallel invocation} \\ & \{ \{\} \} \cup \cup \{ \cup \{ \llbracket A(a) \rrbracket \mid [a] \in \llbracket B(b) \rrbracket \} \mid [b] \in \llbracket C(c) \rrbracket \} \\ &= \text{calculus} \end{aligned}$$

$$\begin{aligned}
& \{\{\}\} \cup \cup\{\llbracket A(a) \rrbracket \mid \exists[b] : \llbracket C(c) \rrbracket \cdot [a] \in \llbracket B(b) \rrbracket\} \\
& = \text{calculus} \\
& \{\{\}\} \cup \cup\{\llbracket A(a) \rrbracket \mid [a] \in \cup\{\llbracket B(b) \rrbracket \mid [b] \in \llbracket C(c) \rrbracket\}\} \\
& = \text{semantics of asymmetric parallel invocation} \\
& \{\{\}\} \cup \cup\{\llbracket A(a) \rrbracket \mid [a] \in \llbracket B <y< C \rrbracket(c)\} \\
& = \text{semantics of asymmetric parallel invocation yet again} \\
& \llbracket A <x< (B <y< C) \rrbracket(c).
\end{aligned}$$

For the context commutative law, (20), we reason just as in the previous argument; for any input c to C ,

$$\begin{aligned}
& \llbracket (A <x< B) <y< C \rrbracket(c) \\
& = \text{semantics of asymmetric parallel invocation} \\
& \{\{\}\} \cup \cup\{\llbracket A <x< B \rrbracket(b) \mid [b] \in \llbracket C(c) \rrbracket\} \\
& = \text{semantics of asymmetric parallel invocation and } B \text{ independent of } b \\
& \{\{\}\} \cup \cup\{\cup\{\llbracket A(a) \rrbracket \mid [a] \in \llbracket B \rrbracket\} \mid [b] \in \llbracket C(c) \rrbracket\} \\
& = \text{calculus} \\
& \{\{\}\} \cup \cup\{\llbracket A(a) \rrbracket \mid [a] \in \llbracket B \rrbracket\} \\
& = \text{calculus} \\
& \{\{\}\} \cup \cup\{\cup\{\llbracket A(a) \rrbracket \mid [a] \in \llbracket C \rrbracket\} \mid [c] \in \llbracket B(b) \rrbracket\} \\
& = \text{semantics of asymmetric parallel invocation and } C \text{ independent of } c \\
& \{\{\}\} \cup \cup\{\llbracket A <y< C \rrbracket(c) \mid [c] \in \llbracket B(b) \rrbracket\} \\
& = \text{semantics of asymmetric parallel invocation yet again} \\
& \llbracket (A <y< C) <x< B \rrbracket(b).
\end{aligned}$$

For the distributive law, (21), we reason that for input c to C ,

$$\begin{aligned}
& \llbracket (A \mid B) <x< C \rrbracket(c) \\
& = \text{semantics of asymmetric parallel invocation} \\
& \{\{\}\} \cup \cup\{\llbracket A \mid B \rrbracket(a) \mid [a] \in \llbracket C(c) \rrbracket\} \\
& = \text{semantics of independent parallel invocation and } B \text{ independent of } a \\
& \{\{\}\} \cup \cup\{binter \ \? \ \llbracket A(a) \rrbracket, \llbracket B \rrbracket \} \mid [a] \in \llbracket C(c) \rrbracket\} \\
& = \text{calculus and } B \text{ independent of } a \\
& binter \ \? \ \cup\{\llbracket A(a) \rrbracket \mid [a] \in \llbracket C(c) \rrbracket\}, \llbracket B(b) \rrbracket
\end{aligned}$$

$$\begin{aligned}
&= \text{semantics of sequential invocation} \\
&\text{binter } \lambda \llbracket A \text{ <}x\text{<} C \rrbracket(c), \llbracket B(b) \rrbracket \} \\
&= \text{semantics of independent parallel invocation} \\
&\llbracket (A \text{ <}x\text{<} C) \mid B \rrbracket(c, b).
\end{aligned}$$

□

6 Examples

In this section we consider examples that have already been published to demonstrate various features of Orc. For us they serve the purpose of benchmarks on which to test our abstract semantics.

The first demonstrates semantic reasoning from specification to Orc implementation, and the way in which variations on the theme are reflected in the semantics. The second involves slightly more complex reasoning and the use of **let**. The third combines reasoning by law and by semantics to simplify the otherwise entirely semantic—and as a result lengthy—reasoning.

6.1 News

From [9, 10], the evaluation of $BBC(d)$, where BBC is a news service site and d is a date, calls BBC with parameter value d ; if BBC responds (with the news page for the specified date), the response $c(d)$ is published

$$\llbracket BBC \rrbracket(d) := \{[], [b(d)]\}.$$

Similarly for site CNN ,

$$\llbracket CNN \rrbracket(d) := \{[], [c(d)]\}.$$

Now we are interested in the independent parallel invocation of those two sites. It publishes either interleaving of those two values, or just one (in the case that the other fails to publish), or neither (in the case that they both fail to publish). Thus, recalling the notation (1),

$$\text{NewsSpec} := \text{pubs}^2 \{b(d), c(d)\}.$$

An Orc implementation is simply

$$\text{News}(d) := (BBC(d) \mid CNN(d)).$$

In practice the specification will be given and an Orc implementation required to meet it. A standard, but particularly powerful and flexible, method of verifying an implementation is its top-down derivation. In this case that is simple: for any input date d ,

$$\begin{aligned}
\text{NewsSpec} & \\
= & \text{Definition (1)} \\
\{[], [b(d)], [c(d)], [b(d), c(d)], [c(d), b(d)]\} & \\
= & \text{definition of } \textit{inter} \\
\bigcup \{ \textit{inter } bs \ cs \mid bs \in \{[], [b(d)]\}, cs \in \{[], [c(d)]\} \} & \\
= & \text{semantics of sites } \textit{BBC}, \textit{CNN} \\
\bigcup \{ \textit{inter } bs \ cs \mid bs \in \llbracket \textit{BBC}(d) \rrbracket, cs \in \llbracket \textit{CNN}(d) \rrbracket \} & \\
= & \text{semantics of independent parallel invocation} \\
\llbracket \textit{BBC}(d) \mid \textit{CNN}(d) \rrbracket & \\
= & \text{definition} \\
\llbracket \textit{News}(d) \rrbracket . &
\end{aligned}$$

Modifications [12] of that example incorporate further combinators by binding the responses of the calls above to a parameter x then calling a site *Email* which sends the information to an address a . The first modification emails at most the first page to be published by the two news sites, whilst the second emails at most each. Accordingly, the specifications are

$$\begin{aligned}
\llbracket \textit{EmNews1}(d) \rrbracket &= \textit{pubs}^1 \{b(d), c(d)\} = \{[], [b(d)], [c(d)]\} \\
\llbracket \textit{EmNews2}(d) \rrbracket &= \textit{pubs}^2 \{b(d), c(d)\} = \{[], [b(d)], [c(d)], [b(d), c(d)], [c(d), b(d)]\}.
\end{aligned}$$

Reasoning analogous to that above shows that the first may be implemented using asymmetric parallel invocation to invoke *News* whilst the second may be implemented using sequential invocation

$$\begin{aligned}
\textit{EmNews1}(d) &:= (\textit{Email}(a, x) <x< (\textit{BBC}(d) \mid \textit{CNN}(d))) \\
\textit{EmNews2}(d) &:= ((\textit{BBC}(d) \mid \textit{CNN}(d)) >x> \textit{Email}(a, x)).
\end{aligned}$$

6.2 Fork-join

A computation may require that two independent threads be spawned at some point, and the computation resumed only after both threads are complete. Assume that sites A and B accept inputs a and b and may

compute results $x(a)$ and $y(b)$ respectively, just like the news sites of the previous section; thus

$$\begin{aligned} \llbracket A \rrbracket &:= \{\{\}, [x(a)]\} \\ \llbracket B \rrbracket &:= \{\{\}, [y(b)]\}. \end{aligned}$$

The specification ensures that both have completed by publishing their publications as a pair:

$$\text{ForkjoinSpec} := \{\{\}, [(x(a), y(b))]\}.$$

To achieve that, Orc defines *Forkjoin* [9, 10] to call sites A and B in parallel; the first result to be published is held in the site **let** (x, y) which publishes the pair when it receives the second result

$$\text{Forkjoin} := ((\mathbf{let} (x, y) <x< A(a) <y< B(b)).$$

A derivation is as follows. For any inputs a to A and b to B ,

$$\begin{aligned} &\text{ForkjoinSpec} \\ &= &&\text{definition} \\ &\{\{\}, [(x(a), y(b))]\} \\ &= &&\text{calculus} \\ &\cup\{\{\{\}, [(x(a), y(b))]\}\} \\ &= &&\text{semantics (7) of } \mathbf{let} \\ &\{\{\}\} \cup \cup\{\llbracket \mathbf{let} (x, y(b)) \rrbracket \mid [x] \in \{\{\}, [x(a)]\}\} \\ &= &&\text{semantics of asymmetric parallel invocation and of site } A \\ &\{\{\}\} \cup \cup\{\llbracket \mathbf{let} (x, y) <x< A(a) \rrbracket \mid [y] \in \{\{\}, [y(b)]\}\} \\ &= &&\text{semantics of asymmetric parallel invocation and of site } B \\ &\llbracket (\mathbf{let} (x, y) <x< A(a) <y< B(b)) \rrbracket \\ &= &&\text{definition} \\ &\llbracket \text{Forkjoin} \rrbracket. \end{aligned}$$

6.3 Mini network

The use of Orc to model networks of sites that communicate values is demonstrated by the following simple example based an example from [10], Section 3.

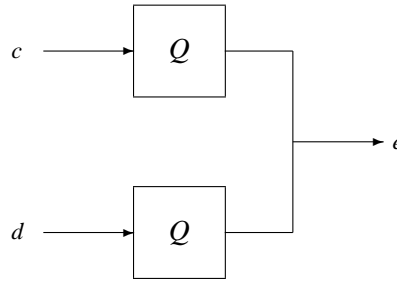


Figure 1: A mini network consisting of two input wires, a computation Q on each and a single output wire e .

A mini network is shown in Figure 1 and realised in terms of orchestration as follows. After accepting two input values using the methods $c.get()$ and $d.get()$, the results $y(c)$ and $y(d)$ are computed by Q evaluated on those inputs and published via the method $e.put()$. Thus the orchestrating program is specified

$$NetSpec := pubs^2 \{e.y(c), e.y(d)\}.$$

An Orc implementation treats those methods as sites and orchestrates them thus:

$$\begin{aligned} P(c,e) &:= ((c.get() >x> Q(x)) >y> e.put(y)) \\ Net(c,d,e) &:= (P(c,e) | P(d,e)). \end{aligned}$$

Part of the interest in this example is the manner in which the design is reasoned about top-down. Recognising that the specification is going to be achieved by the interleaving of sequences of length at most 1, we reason first:

$$\begin{aligned} &\{[], [y(c)], [y(d)], [y(c), y(d)], [y(d), y(c)]\} \\ &= \text{properties of interleaving and calculus} \\ &\cup \{inter\ as\ bs\ | \ as \in \{[], [y(c)]\}, \ bs \in \{[], [y(d)]\}\} \\ &= \text{semantics of sequential invocation evaluated above} \\ &\cup \{inter\ as\ bs\ | \ as \in \llbracket (c.get(a) >x> Q(x)) \rrbracket, \ bs \in \llbracket (d.get(a) >x> Q(x)) \rrbracket\} \\ &= \text{semantics of independent parallel invocation} \\ &\llbracket (c.get(a) >x> Q(x)) | (d.get(a) >x> Q(x)) \rrbracket. \end{aligned}$$

Now we make a side calculation to investigate how to achieve the sequences of length at most 1. For any input value a ,

$$\begin{aligned}
& \{ [], [y(c)] \} \\
& = \text{calculus} \\
& \cup \{ \text{binter } \llbracket [], [y(c)] \rrbracket \} \\
& = \text{semantics of site invocation for } Q \text{ and calculus} \\
& \cup \{ \text{binter } \llbracket \llbracket Q(c.get(a)) \rrbracket \rrbracket \} \\
& = \text{calculus} \\
& \cup \{ \text{binter } \llbracket \llbracket Q(b) \rrbracket \mid b \text{ in } bs \rrbracket \mid bs \in \{ [], [c.get(a)] \} \} \\
& = \text{semantics of input site and calculus} \\
& \cup \{ \text{binter } \llbracket \llbracket Q(b) \rrbracket \mid b \text{ in } bs \rrbracket \mid bs \in \llbracket c.get(a) \rrbracket \} \\
& = \text{definition of sequential invocation} \\
& \llbracket c.get(a) \text{ >}x\text{>} Q(x) \rrbracket .
\end{aligned}$$

Finally we have

$$\begin{aligned}
& \text{NetSpec} \\
& = \text{definition} \\
& \{ [], [e.y(c)], [e.y(d)], [e.y(c), e.y(d)], [e.y(d), e.y(c)] \} \\
& = \text{properties of } \text{binter}, \text{ semantics of output site and calculus} \\
& \cup \{ \text{binter } \llbracket \llbracket e.put(y) \rrbracket \mid y \text{ in } bs \rrbracket \mid bs \in \{ [], [y(c)], [y(d)], [y(c), y(d)], [y(d), y(c)] \} \} \\
& = \text{semantics of sequential invocation and the evaluation above} \\
& \llbracket ((c.get() \text{ >}x\text{>} Q(x)) \mid (d.get() \text{ >}x\text{>} Q(x)) \text{ >}y\text{>} e.put(y)) \rrbracket \\
& = \text{Law (16)} \\
& \llbracket ((c.get() \text{ >}x\text{>} Q(x)) \text{ >}y\text{>} e.put(y)) \mid ((d.get() \text{ >}x\text{>} Q(x)) \text{ >}y\text{>} e.put(y)) \rrbracket \\
& = \text{definition of } P \\
& \llbracket P(c, e) \mid P(d, e) \rrbracket \\
& = \text{definition of } Net \\
& \llbracket Net(c, d, e) \rrbracket .
\end{aligned}$$

That mini circuit invokes $c.get()$ and $d.get()$ just once and so inputs just once on each input channel. A more realistic version, in which an unending sequence of inputs is allowed, is specified

$$NettSpec := pubs \{ e.y(c), e.y(d) \}$$

and shown to be implemented by the variant in which each P is recursive

$$P(c, e) := (((c.get() >x> Computer(x)) >y> e.put(y)) \gg P(c, e)).$$

7 Conclusion and further work

Accountably correct orchestration of web services appears to be important if not vital in order for web services to provide the same level of accuracy and support as its earlier counterpart, the ‘human navigable’ web. Orc, being a language for orchestration at a moderately abstract level, provides important support for the Software Engineer. In this paper we have studied the way in which Orc integrates with another of the Software Engineer’s standard techniques: that of state-based specification.

We have proposed a slightly more abstract semantics of Orc that corresponds directly to the user’s requirements—and specifier’s view—of web orchestration. The semantics is not meant as guidance to an implementor of Orc, but is aimed at the higher levels of specification and development. Law (13) is an example of one which holds in our semantics but not in an implementation-oriented semantics of Orc. It is perhaps surprising how similar the resulting model is to one which exposes values of parameters within combinators; that is due to the extent to which the language is captured by sequences of publications. Laws have been proved sound with respect to the semantics and examples studied to confirm that the semantic model is realistic.

We conclude that further work appears to be justified. The next step is perhaps a case study, preferably one already been implemented in BPEL (like the Virtual Travel Agent from [15]); it might be specified and reasoned about using the laws and semantics given here, adopting the style used in the examples here.

In this paper (strict) refinement has not been necessary: all reasoning has involved equalities. But in general, nondeterminism will be restricted in the implementation compared with the specification, and then an subset-based definition of refinement will be necessary and, with it, weaker laws that are merely inclusions.

References

- [1] Web Services Business Process Execution Language Version 2.0.
<http://www.oasis-open.org/apps/org/workgroup/wsbpel/>
- [2] The Orc web site. <http://orc.csres.utexas.edu/>
- [3] R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science, 1998.

- [4] W. R. Cook, S. Patwardhan and J. Misra. Workflow patterns in Orc. In *Proceedings of the International Conference on Coordination Models and Languages*, Bologna, Italy, June 14-16, 82–96, 2006.
- [5] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan Press, 2000.
- [6] R. Eshuis and J. Dehnert. Reactive Petri Nets for workow modeling. In *International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*. LNCS, **2679**:296–315, Springer-Verlag, 2003.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [8] C. A. R. Hoare, G. Menzel and J. Misra. A tree semantics of an orchestration language. In M. Broy, J. Gruenbauer, D. Harel and C. A. R. Hoare, editors, *Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, **195**, 2004.
- [9] A language for task orchestration and its semantic properties. D. Kitchin, W. R. Cook and J. Misra. In *Proceedings of the International Conference on Concurrency Theory (CONCUR) 2006*, 477–491, 2006.
- [10] D. Kitchin, E. Powell and J. Misra. Simulation using orchestration (Extended abstract). In *Algebraic Methodology and Software Technology, SLNCS*, **5140**:2–15, Springer Verlag, 2008.
- [11] J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, J. Gruenbauer, D. Harel and C. A. R. Hoare, editors, *Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, **195**, 2004.
- [12] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, **6**(1):83–110, 2007.
- [13] F. Puhlmann and M. Weske. Using the π -calculus for formalizing workow patterns. In *Proceedings of the 3rd International Conference on Business Process Management*. SLNCS, **3649**, Springer Verlag, 2005.
- [14] J. M. Spivey. *The Z Notation: a reference manual*, second edition. Prentice-Hall International, 1992.
- [15] R. Kazhamiakin, M. Pistore and L. Santuari. Analysis of communication models in web service compositions. In *Proceedings of the 15th International Conference on World Wide Web*, Edinburgh, Scotland. 267–276, 2006.