Computation Orchestration:
A Basis for Wide-Area Computing

Jayadev Misra*
The University of Texas at Austin
Austin, Texas 78712, USA
email: misra@cs.utexas.edu

May 20, 2005

Contents

*Work partially supported by the National Science Foundation grant CCR–0204323.

Abstract

We explore the following quintessential problem: given a set of basic computing elements how do we compose them to yield desired computation patterns. Our goal is to study composition operators which apply across a broad spectrum of computing elements, from sequential programs to distributed transactions over computer networks. Our theory makes very few assumptions about the nature of the basic elements; in particular, we do not assume that an element's computation always terminates, or that it is deterministic. We develop a theory which provides useful guidance for application designs, from integration of sequential programs to coordination of distributed tasks. The primary application of interest for us is orchestration of web services over the internet, which we describe in detail in this paper.

1 Introduction

The computational pattern inherent in many wide-area applications is this: acquire data from one or more remote services, calculate with these data, and invoke yet other remote services with the results. Additionally, it is often required to invoke alternate services for the same computation to guard against service failure. It should be possible to repeatedly poll a service until it supplies results which meet certain desired criteria, or to ask a service to notify the user when it acquires the appropriate data. And it should be possible to download an application and invoke it locally, or have a service provide the results directly to another service on behalf of the user.

We introduce *site* as a general term for a basic service. A web service is a site. More generally, a distributed transaction, which can be regarded as an atomic step of a larger computation, is a site. We sketch some of the requirements for sites later in this section and in greater detail in section 2.

We call the smooth integration of sites *orchestration*, and *Orc* is our theory of orchestration of sites. Orchestration requires a better understanding of the kinds of computations that can be performed efficiently over a wide-area network, where the delays associated with communication, unreliability and unavailability of servers, and competition for resources from multiple clients are dominant concerns.

Consider a typical wide-area computing problem. A client contacts two airlines simultaneously for price quotes. He buys a ticket from either airline if its quoted price is no more than \$300, the cheapest ticket if both quotes are above \$300, and any ticket if the other airline does not provide a timely quote. The client should receive an indication if neither airline provides a timely quote. Such problems are typically programmed using elaborate manipulations of low-level threads. We regard this as an orchestration problem in which each airline is a site; we can express such orchestrations very succinctly in Orc.

Our theory is built upon a small number of composition operators. We use alternation ($|$) for parallel composition and sequencing ($\gg$ and $>x>$) for sequential composition. Operator **where** allows selective pruning of parallel threads and data transfer across subcomputations. We show a variety of examples from web services and other domains to illustrate the power of these composition operators. Our theory is applicable to distributed application design in general, with particular emphasis on orchestration of web services.

1.1 An Overview of the Orchestration Theory

Starting an orchestration We propose a simple extension to a sequential programming language to permit orchestration. Introduce an assignment statement of the form

$$z :\in \; E(L)$$

where z is a variable, E is the name of an orchestration expression (abbreviated to *Orc expression*, or, simply *expression*)[1] and L is a list of actual parameters. Evaluation of $E(L)$ may entail a wide-area computation involving, possibly, multiple servers. The evaluation returns zero or more results, the first one of which (if there is one) is assigned to z. If the evaluation yields no result, the statement execution does not terminate. Additionally, the evaluation may initiate computations which have effects on other servers, and these effects may or may not be visible to the client.

Next, we give a brief introduction to the structure of Orc expressions.

Site The simplest Orc expression is a *site* name, possibly with parameters. Evaluation of the expression calls the site like a procedure. A site call elicits at most one response; it is possible that a site never responds to a call.

Consider the expression CNN, where CNN is a news service. A call may simply return the latest newspage. Calling $CNN(d)$, where d is a date, may download the newspage for the specified date. Let $Email(a,m)$ send message m to address a. Evaluation of the expression $Email(a,m)$ sends the message, causing permanent change in the state of the recipient's mailbox, and returns a signal to the client to denote completion of the operation. Let A be an airline flight-booking site. Evaluating the expression A returns the booking information and causes a state change in the airline database. A ticket is purchased only by making an explicit *commitment* later in the computation.

A site could be a function (say, to convert an XML file to a bit stream for transmission), a method of an object (say, to gain access to a password-protected object; in this case, the password, or an encrypted form of it, would be a parameter of the call), a monitor[16] procedure (such as read or write to a buffer, where the read responds only when the buffer is non-empty), or a web service (say, a stock quote service that delivers the latest quotes on selected stocks). A *transaction*[12] can be a site that returns a result and tentatively changes the states of some servers.

An orchestration may involve humans as sites. A program which coordinates the rescue efforts after an earthquake will have to accept inputs from the medical staff, firemen and the police, and direct them by sending commands and information to their hand-held devices. Humans communicate with the orchestration by sending digital inputs (key presses) and receiving output suitable for human consumption (print, display or audio).

A call to a site may not return a result if, for instance, the server or the communication link is down. This is treated as any other non-terminating computation. We show how time-outs can be used to alleviate this problem.

Composition Operators The simplest Orc expressions, as we have seen, are site calls. We use composition operators on expressions to form longer expressions. Orc has three composition operators; for expressions f and g: (1) $f >x> g$ is the sequential composition, (2) $f \mid g$ is the symmetric parallel composition, (3) (f **where** $x :\in \; g$) is asymmetric parallel composition. Additionally, we structure an expression by allowing expression definitions, and using names of expressions in other expressions. Naming also allows recursive definitions of expressions, which is essential in any substantive application design.

[1]The notation $:\in$ is due to Hoare. It neatly expresses, in analogy with the assignment operator $:=$, that the evaluation of the right side may yield a set of values one of which is to be assigned to z.

Below, we give a brief summary of the composition operators using a series of examples; a detailed description appears in section 3.

Evaluation of an expression calls some number of sites and produces a (possibly empty) stream of values.

- (Sequential Composition) Consider the expression $(CNN > m > Email(a, m))$. To evaluate it, first call CNN. The value returned is m, and $Email(a, m)$ is then called. The value returned by $Email(a, m)$ is the value of the expression. If CNN does not respond, the expression evaluation does not terminate. If CNN does respond but $Email(a, m)$ does not, then also the evaluation does not terminate. No value is returned in either case.

- (Symmetric Parallel Composition) For $(CNN \mid BBC)$, call the two sites simultaneously. The output stream consists of the values returned by the sites in time-order. Thus, there can be anywhere from zero to two values in the stream.

  Particularly interesting is an expression like

  $$(CNN \mid BBC) \ > m > \ Email(a, m)$$

  Here, $(CNN \mid BBC)$ may produce multiple values, and for each value v, we call $Email(a, m)$ setting m to v. Therefore, the evaluation can cause up to two emails to be sent, one with the newspage from CNN and the other from BBC.

- (Asymmetric Parallel Composition) The composition operators given so far only create threads. We introduce the **where** operator to prune threads selectively. For example, $(Email(a, m)$ **where** $m :\in (CNN \mid BBC))$ sends at most one email, with the first newspage received from either CNN or BBC. This expression is evaluated as follows. Start evaluation of both $Email(a, m)$ and $(CNN \mid BBC)$. Since m does not have a value initially, the call $Email(a, m)$ is not completed; it is deferred until m gets a value. Computing $(CNN \mid BBC)$, as described above (under Symmetric Parallel Composition), may yield many values; the first value is assigned to m and further evaluation of that expression is then terminated. At this point, $Email(a, m)$ is called and its response, if any, is the value of the whole expression.

Sequencing allows results from one expression to be used as input to another; for instance, we may contact a discovery service and pipe its output —the name of an application— to another service which downloads the application and executes it on some given data. Operator $\mid$ allows us to receive data from mirror sites or to compute a result by calling alternate services. And **where** allows selective pruning of the computation.

Expression Definition To structure an orchestration, we allow expression definitions. An expression is defined like a procedure, with a name and possible parameters. Below, $MailOnce(a)$ emails the first newspage from CNN or BBC to address a.

$$MailOnce(a) \ \underline{\Delta}$$
$$Email(a, m) \ \textbf{where} \ m :\in (CNN \mid BBC)$$

An expression, such as $MailOnce$, may be called from another expression, as in

$$MailOnce(a) \ > x > \ MailOnce(b)$$

to send two newspages, to addresses a and b in succession. An orchestration is started by the host language program by calling an expression.

An expression may call itself, as in

$$MailForever(a) \ \underline{\Delta}$$
$$MailOnce(a) \ > x > \ MailForever(a)$$

which keeps sending newspages to a. A more interesting expression is $Ticker$ which emails a newspage to a, receives a confirmation from $Email$, waits for t time units, and then repeats these steps forever.

$$Ticker(a, t) \ \underline{\Delta}$$
$$MailOnce(a) \ > x > \ Rtimer(t) \ > y > \ Ticker(a, t)$$

Here $Rtimer(t)$, a site call, returns a value after t time units (the value itself is of no significance, only the time delay is). We will see more sophisticated orchestration schemes which allow time-outs, interruptions, eager evaluations (such as calling $Rtimer$ as soon as $Email$ is called but before it responds) in this paper.

1.2 Power of the Orc computation model

The proposed programming model is quite minimal. It has no inherent computational power; it has to rely on external sites for doing even arithmetic. However, this apparent limitation permits us to study orchestration in isolation and to combine sites of arbitrary complexity in a computation, without making any assumptions about their behavior. Our model includes no explicit constructs for time-out or thread synchronization and communication, features which are common in thread-based languages. We show in section 4 how such constructs are easily implemented in Orc. As a special case, single-threaded computations (as in sequential computing) are also easy to code in Orc. We program arbitrary process-network-style computations by having expressions correspond to processes, and letting them communicate through sites that implement channels.

In many distributed applications, no value is returned from a computation because the computation never terminates by design. An Orc computation can start and spawn (a bounded or unbounded number of) threads: some may terminate on their own (thus, returning values), some are terminated using a **where** clause (still returning a value), and others continue to run forever (without returning a value), though they affect the states of sites. This generality permits us to program a variety of thread-based applications using a small number of concepts.

Structure of the paper The goal of this paper is to introduce the Orc programming model and to illustrate its application in diverse areas of programming. In forthcoming papers we propose semantic models, describe an implementation of Orc, and develop strategies to commit the computation of a specific thread (if the thread makes calls to transactions which need commitment). A preliminary version of the semantics appears in Hoare et. al. [18].

We discuss several issues related to sites in section 2. In particular, we state some assumptions we do *not* make about sites. We define a few sites which are fundamental to effective programming in Orc. We describe the syntax of Orc in section 3.1 and an operational semantics in section 3.2. Most programming is done by learning certain idioms. We develop a number of idioms in section 4, which show the programming strategy for sequential computing, time-out, and communication and synchronization among threads. Section 5 contains a few laws, describing equivalences over Orc expressions.

We develop some longer examples in section 6. These are motivated by the intended application domain of Orc, web services orchestration. Our examples illustrate that Orc provides succinct representation for a variety of distributed applications.

2 Sites

2.1 Properties of sites

The most basic Orc expression is a site call. A site call has the same form as a function call: the name of a site followed by an optional list of parameters. Therefore, the simplest Orc expression is the name of a site. A parameter is a constant, variable, or a site name.

In this paper, we do not specify exactly how a site is to be called; the kinds of communication protocols to be used and the servers on which the computations of a site take place are not relevant to our theory. It is possible to designate a site as being downloadable —as is the case with most Java applets— which causes a site call to result in a download and execution of the application on the client's machine. More elaborate schemes for migration and execution may be specified for certain sites. In general, calling a site causes execution of the corresponding procedure at the appropriate servers.

A site is different in several ways from a mathematical function. First, a site call may have side-effects, changing the state of some object. Second, a site call may elicit no response, or produce different results with the same input at different times. In particular, a site may return no result for one call and a result for an identical call (with the same inputs) at a different time. This is because the server or the communication link may have failed during the former call. Third, the response delay of a site is unpredictable.

2.2 Types of results produced by sites

A site is called with values of certain types and it returns typed values. The internet already supports a number of esoteric data types, such as newspages, downloadable files, images, animation and video, url strings, email lists, order forms, etc. The result returned by a discovery service is of type *site*. We expect the variety of types to proliferate in the coming years. Many of these types will be XML document types[10]; see Cardelli [5] for an interesting presentation on this and related topics. Even though it is a fascinating area, we will not pursue the question of how various types will be handled within a traditional sequential programming language. We merely assume that a result produced by a site can be assigned to a program variable.

We introduce a type, called *signal*, which has exactly one value. Its purpose is to indicate the termination of some expression evaluation.

2.3 States changed by site calls

A site call can potentially affect the state of the external world in addition to returning a value to the client. The state changes could be one of the following: (1) no (discernible) state change (2) a permanent state change, or (3) a tentative state change.

A site which is a function (in the strict mathematical sense) causes no state change. (Although its execution consumes resources, such aspects are not relevant to our work.) Similarly, a query on a database does not cause visible state change, though it may have the benign side-effect of caching the data for faster access in the future.

A call to an *Email* site causes a permanent state change in the mailbox of the intended recipient. This state change cannot be rolled back. Any roll-back strategy is application dependant, say, by sending a cancellation message, which has to be interpreted appropriately by the recipient.

A call to a site that implements a transaction will usually cause a tentative state change. Imagine booking an airline ticket through its web site or trading stocks online at a brokerage service. The tentative state changes are made permanent only by explicit commitment (i.e., the user confirms the purchase of the airline ticket or buys the stock). If the transaction is not confirmed in a timely manner, the state changes are rolled back.

A transaction can be regarded as an atomic instruction which is either executed completely or not at all. This permits us to build larger computational

Table 1: Fundamental Sites

$\mathbf{0}$	never responds.
$let(x, y, \cdots)$	returns a tuple consisting of the values of its arguments.
$Clock$	returns the current time at the server of this site as an integer.
$Atimer(t)$	where t is integer and $t \geq Clock$, returns a signal at time t.
$Rtimer(t)$	where t is integer and $t \geq 0$, returns a signal after exactly t time units.
$Signal$	returns a signal immediately. It is same as $Rtimer(0)$.
$if(b)$	where b is boolean, returns a signal if b is true, and it remains silent (i.e., does not respond) if b is false.

units by composing the atomic instructions in various ways. And a transaction has no permanent effect unless it is committed. This permits us to explore alternative computations, each computation being a series of transactions, and to commit to a specific computation (i.e., all transactions in it), only after observing the results of different computations. For example, a client may book tickets at different airlines, compare their prices and then confirm the cheapest one. In a forthcoming paper, we describe a protocol to select and commit an appropriate subset of transactions that are invoked during a computation.

2.4 Some Fundamental Sites

We define a few sites in Table 1 that are fundamental to effective programming in Orc. The *Zero* site, written as $\mathbf{0}$, never responds. Sites *let*, *Clock*, *Signal* and *if* respond immediately (or may not respond at all, in the case of *if*). The timer sites —*Clock*, *Atimer* and *Rtimer*— are used for computations involving time-outs. Time is measured locally by the server on which the client (and the timer) reside. Since the timer is a local site, the client experiences no network delay in calling the timer or receiving a response from it; this means that the signal from the timer can be delivered at exactly the right moment. With $t = 0$, *Rtimer* responds immediately. Sites *Atimer* and *Rtimer* differ only in having absolute and relative values of time as their arguments, respectively. They are related as follows, where the current clock value is c.

$$Atimer(t) \equiv Rtimer(t - c)$$
$$Rtimer(u) \equiv Atimer(u + c)$$

3 Syntax and Semantics

We describe the syntax and operational semantics of Orc in this section. The notation, which we have outlined in section 1.1, is quite simple, and can be adapted easily for many sequential host languages.

Table 2: Syntax of Orc

E	$\in$	Expression Name	
M	$\in$	Site	
x, z	$\in$	Variable	
c	$\in$	Constant	
P	$\in$	List of Actual Parameter	
Q	$\in$	List of Formal Parameter	
$p \in Actual\ Parameter$	$::=$	M	Site
	$\mid$	x	Variable or structure
	$\mid$	c	Constant
$q \in Formal\ Parameter$	$::=$	M	Site
	$\mid$	x	Variable or structure
$Orc\ Statement$	$::=$	$z{:}\in\ E(P)$	Evaluate $E(P)$ and assign to z
$Expression\ Defn$	$::=$	$E(Q)\ \underline{\Delta}\ f$	Define expression E
$f, g \in Expression$	$::=$	$\mathbf{0}$	Zero site; never responds
	$\mid$	$M(P)$	Site call
	$\mid$	$E(P)$	Expression call
	$\mid$	$f \mid g$	Symmetric Parallel Composition
	$\mid$	$f > x > g$	Sequential Composition
	$\mid$	f **where** $x{:}\in g$	Asymmetric Parallel Composition

3.1 Syntax

The syntax of Orc appears in Table 2. Note that a parameter can be a site name; therefore, a site call may be made to a parameter. Henceforth, we abbreviate $f > x > g$ to $f \gg g$ if x is of no concern (i.e., there is no reference to x in g).

The following are examples of Orc expressions. Here, M, N and R are sites, x and y are variables and E is an expression defined separately.

$$N(x),\ M \gg N(x),\ M\ > x >\ N(x),\ E(x, y) \gg N(x)$$
$$(M \gg \mathbf{0}\ \mid\ (N(x)\ \textbf{where}\ x{:}\in\ R \mid N(y)))$$

Binding powers of the operators We list the operators in increasing order of precedence (binding power):
$\underline{\Delta}$, $:\in$, **where** , $\mid$, $> x >$.

Operator $> x >$ is right associative. So

$$M\ > x >\ (N(x) \mid R)\ > y >\ S(y)\ \text{ is}$$
$$M\ > x >\ ((N(x) \mid R)\ > y >\ S(y)).$$

Well-formed expressions The *free* variables of an expression are defined as follows, where M is a site or an expression name and L is a list of its actual parameters.

$$\begin{aligned}
&\mathit{free}(\mathbf{0}) \;=\; \{\}\\
&\mathit{free}(M(L)) \;=\; \{x \mid x \in L\}\\
&\mathit{free}(f \mid g) \;=\; \mathit{free}(f) \,\cup\, \mathit{free}(g)\\
&\mathit{free}(f \;>x>\; g) \;=\; \mathit{free}(f) \,\cup (\mathit{free}(g) - \{x\})\\
&\mathit{free}(f \;\mathbf{where}\; x{:\in}\; g) \;=\; (\mathit{free}(f) - \{x\}) \,\cup\, \mathit{free}(g)
\end{aligned}$$

Variable x is *bound* in f if it is named in f and is not free. In the host program, Orc statement $y{:\in}\; E(L)$ is *well-formed* if the variable parameters in L are variables of the host language program. Expression definition $E(L) \;\underline{\Delta}\; f$ is well-formed if the set of free variables of f is a subset of L.

Notational conventions We write

$$(f \;\mathbf{where}\; x{:\in}\; g) \;\mathbf{where}\; y{:\in}\; h$$

also as

$$\begin{aligned}
&f\\
&\mathbf{where}\; x :\in g\\
&\qquad\;\; y :\in h
\end{aligned}$$

or, $(f \;\mathbf{where}\; x{:\in}\; g,\; y{:\in}\; h)$.

We use two forms of brackets, () and {}, in writing expressions so that they are easier to read. They are interchangeable.

3.2 Operational semantics

In this section, we describe the semantics of Orc in operational terms.

Evaluation of an expression (for a certain set of global variable values) yields a stream of values; additionally, the evaluation may call certain sites causing changes in their states.

3.2.1 Site call

The simplest expression is a site name without parameters. To evaluate the expression, call the site and the value returned by the site becomes the (only) value of the expression.

A site call with parameters is *strict*; that is, the site is called only when all its parameters are defined. The parameters and return value of a site can be of any type (see section 2.2), including a site name which can be called later in the Orc expression.

3.2.2 Expression call

An expression call is syntactically similar to a site call, with the name of an expression replacing a site name. However, there are several semantic differences.

First, a site call produces at most one value whereas an expression may produce many.

Second, calling an expression starts evaluation of a *new instance* of that expression; that is $f \gg f$ refers to two different instances of f. A site call, typically, will not create new instances of the site, but will queue its callers and serve them in some order.

Third, a site call is strict in that its actual parameter values are defined before the call. An expression call is non-strict; evaluation of an expression begins *when it is called*, even if some of its actual parameters are undefined. See sections 3.2.6 and 3.3.3 for elaboration.

3.2.3 Sequential composition

Operator $\gg$ and its more general form $>x>$ allow sequencing of site calls. We first take up the simpler case, $\gg$. Expression $M \gg N$ first calls M, and on receiving the response from M calls N. The value of the expression is the value returned by N. Site N cannot reference the value returned by M. Operator $\gg$ is associative.

Consider

$$\mathit{Rtimer}(1) \gg \mathit{Email}(\mathit{address}, \mathit{message})$$

which sends an email after unit delay and returns a signal (the value from Email). And $\mathit{Rtimer}(1) \gg \mathit{Rtimer}(1)$ has the same effect as $\mathit{Rtimer}(2)$. Expression

$$\begin{aligned}
&\quad\mathit{Email}(\mathit{address}1, \mathit{message})\\
&\gg \mathit{Email}(\mathit{address}2, \mathit{message})\\
&\gg \mathit{Notify}
\end{aligned}$$

sends two emails in sequence and then calls Notify.

The examples we have shown so far each produce at most one value. In this case, $\gg$ has the same meaning as the sequencing operator in a conventional sequential language (like ";" in Java). For expression $f \gg g$, where f and g are general Orc expressions, f produces a stream of values, and each value causes a fresh evaluation of g. The values produced by all instances of g in the time-order is the stream produced by $f \gg g$. Note that during the evaluation of $f \gg g$, threads for both f and g may be executing simultaneously. We elaborate on this below and after introducing more general Orc expressions.

3.2.4 Value passing

In $M \gg N$, we have merely specified an order of site calls without showing how N may reference the value produced by M. We write $M >x> N(x)$ to assign name x to the value produced by M, which allows N to reference this value. Operator $>x>$ is right associative; so

$$M >x> (N(x) \mid R) >y> S(x,y) \text{ is}$$
$$M >x> \{(N(x) \mid R) >y> S(x,y)\}.$$

That is, the scope of x is as far to the right as possible over a chain of $\gg$. We can show that $>x>$ is associative, i.e.,

$$(f \ >x> \ g) \ >y> \ h \ = \ f \ >x> \ (g \ >y> \ h)$$

if both sides of the identity are well-formed, i.e., if x is not a free variable of h.

For general Orc expressions f and g, $f >x> g$ assigns name x to *every* value produced by f. Each value is referenced in a different thread (an instance of g) as x. For example, suppose f produces three values, 0, 1 and 2. We show the computation of $f >x> M(x)$ schematically in figure 1. Here, each path in the tree is an independent thread.

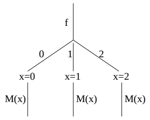

Figure 1: Computation of $f \ >x> \ M(x)$

The same variable name may be used more than once as in

$$M \ >x> \ N(x) \ >x> \ R(x), \text{ which is}$$
$$M \ >x> \ (N(x) \ >x> \ R(x)).$$

It is best to avoid reuse of x, instead writing $M \ >x> \ N(x) \ >y> \ R(y)$.

3.2.5 Symmetric parallel composition

Using the sequencing operator, we can only create single-threaded computations. We introduce $\mid$ to permit symmetric creations of multiple threads. Evaluation of $(M \mid N)$ creates two parallel threads (one for M and one for N), and produces values returned by both threads in the order in which they are computed. Given that *CNN* and *BBC* are two sites that return newspages, *CNN* $\mid$ *BBC* may potentially return two newspages. (It may return zero, one or two values depending on how many sites respond.)

In general, evaluation of $f \mid g$, where f and g are Orc expressions, creates two threads to compute f and g, which may, in turn, spawn more threads. The result from each thread is a stream of values. The result from $f \mid g$ is the merge of these two streams in time order. If both threads produce values simultaneously, their merge order is arbitrary. Operator $\mid$ is commutative and associative.

In traditional thread-based languages, $f \mid g$ returns a single value, either the first value computed by one of the threads or a tuple of values combining the first one from each thread. The first strategy makes a commitment to the first value, discarding all other values. The second strategy, often called fork-join parallelism, requires both threads to deliver results before proceeding with further computation. In Orc, each value from either thread is treated independently in further computations. Therefore, $(f \mid g) \gg h$ creates multiple threads of h, one for each value from $f \mid g$. The two traditional computation styles can be expressed in Orc, as we will show later.

Consider the expression $(M \mid N) >x> R$. The evaluation starts by creating two threads to evaluate M and N. Suppose M returns a value first. This value is called x and R is called. If N returns a value next, R is called again with a new value of x. That is, each value from $(M \mid N)$ spawns a thread for evaluating the remaining part of the expression.

Expressions $M \mid M$ and M are different; the former makes two parallel calls to M, and the latter makes just one. And $M \gg (N \mid R)$ is different from $M \gg N \mid M \gg R$. In the first case, exactly one call is made to M, and N and R are called after M responds. In the second case, two parallel calls are made to M, and N and R are called only after the corresponding calls respond. The difference is significant when M returns different values for each call, and N and R use those values. The two computations are depicted pictorially in figure 2.

Figure 2: (a) $M \gg (N \mid R)$ and (b) $M \gg N \ \mid \ M \gg R$

Earlier, we wrote

$$\begin{array}{l} \quad Email(address1, message) \\ \gg Email(address2, message) \\ \gg Notify \end{array}$$

to send two emails and then call *Notify*. Below, we send the emails in parallel and call *Notify* on receiving each response.

$$\begin{array}{l} \{ \quad Email(address1, message) \\ \quad \mid Email(address2, message)\} \\ \gg Notify \end{array}$$

Here, *Notify* is potentially called twice, once for each response from $Email$.

The operators $\gg$, $>x>$ and $\mid$ can only create threads, not destroy them. Our next operator permits us to terminate parts of an expression evaluation selectively.

3.2.6 Asymmetric parallel composition

An expression with a **where** clause (henceforth, called a **where** expression), has the form $\{f \textbf{ where } x :\in g\}$. Expression f may name x as a parameter in some of its site calls. Evaluation of the **where** expression proceeds as follows. Evaluate f and g in parallel. When g returns a result, assign the value to x and terminate evaluation of g. During evaluation of f, any site call which does not name x as a parameter may proceed, but site calls in which x is a parameter are deferred until x acquires a value. The stream of values produced by f under this evaluation strategy is the stream produced by $\{f \textbf{ where } x :\in g\}$.

A useful application of **where** is in pruning the computation selectively, by destroying certain threads. Consider $(M \mid N) \;\; >x> \;\; R(x)$ where each value produced by $(M \mid N)$ creates an instance of $R(x)$. To create just one thread for $R(x)$, corresponding to the first value produced by $(M \mid N)$, use

$$\{R(x) \textbf{ where } x :\in (M \mid N)\}$$

In section 3.2.5, expression

$$\begin{aligned} \{ \quad & Email(address1, message) \\ \mid \; & Email(address2, message)\} \\ \gg \; & Notify \end{aligned}$$

causes *Notify* to be (potentially) called twice. Below, *Notify* is called just once after both calls to *Email* respond.

$$\begin{aligned} &\{let(u, v) \gg Notify \\ &\quad \textbf{where} \\ &\quad\; u :\in Email(address1, message) \\ &\quad\; v :\in Email(address2, message)\} \end{aligned}$$

Expression calls are non-strict because the semantics of **where** demand it. In $\{F(x) \textbf{ where } x :\in g\}$, where F is the name of an expression, the semantics of **where** require that we start the evaluation of $F(x)$ and g simultaneously, i.e., before x has a value. An implementation has to pass x by reference (where the value of x will be stored when it is defined) to F.

3.2.7 Zero Site

The *Zero* site, written as $\mathbf{0}$, never responds (so, an implementation need not call the site). This is the only specific site which appears in the theory.

Use $\{Email(address1, message) \gg \mathbf{0} \;\mid\; Notify\}$ to send an email and call *Notify* simultaneously. The first alternative never returns a value. (A site like *Email* is called an *asynchronous* procedure in polyphonic $C\#$ [1]; no response is needed from it to proceed with the main computation.)

3.2.8 Expression Definition

Essential to program structuring is the ability to write a long expression in terms of other expressions that are defined separately. In Orc, an expression is defined by its name, a list of parameters which serve as its global variables, and an expression which serves as its body. As an example, consider the definition

$$Asynch(M, N) \;\; \underline{\Delta} \;\; M \gg \mathbf{0} \mid N$$

which defines the name *Asynch*, specifies its formal parameters (sites M and N) and its body. Another expression may call it, for example, in

$$Asynch(Email(address1, message), Notify).$$

As another example, sites P and Q manage the calendars of two different professors. Calling $P(t)$, where t is a time, returns t if the corresponding professor can attend a meeting at t, and it is *silent* (i.e., returns no value), otherwise. Expression *PmeetQ* has two parameters, u and v, which are two possible meeting times, and it outputs the times (out of u and v) when *both* P and Q can meet. So, it may produce 0, 1 or 2 outputs.

$$\begin{aligned} &PmeetQ(u, v) \;\; \underline{\Delta} \\ &\qquad \{P(u) \gg let(x) \textbf{ where } x :\in Q(u)\} \\ &\quad \mid \{P(v) \gg let(x) \textbf{ where } x :\in Q(v)\} \end{aligned}$$

3.2.9 Recursive definitions of expressions

Naming expressions has the additional benefit that we can use the name of an expression in its own definition, getting a recursive definition. Below is an expression which emits a signal every time unit, starting immediately.

$$Metronome \;\; \underline{\Delta} \;\; Signal \mid Rtimer(1) \gg Metronome$$

Parameters may appear in recursive calls in the usual fashion. Define a bounded metronome to generate n signals at unit intervals, starting immediately. We permit pattern matching over parameter values in the same style as Haskell[15].

$$\begin{aligned} &BMetronome(0) && \underline{\Delta} \;\; \mathbf{0} \\ &BMetronome(n + 1) && \underline{\Delta} \\ &\; Signal \mid Rtimer(1) \gg BMetronome(n) \end{aligned}$$

In the following example, we use *Metronome* to implement an iterative computation. Let site *Query* return a value, perhaps different ones at different times. And, site $Accept(x)$ returns x if x is acceptable according to some criterion, and is silent otherwise. It is required to produce all acceptable values by calling *Query* at unit intervals forever.

$$\begin{aligned} &RepeatQuery \;\; \underline{\Delta} \\ &\; Metronome \;\; >x> \;\; Query \;\; \gg \;\; Accept(x) \end{aligned}$$

Or, produce all acceptable values by calling $Query$ at unit intervals n times.

$RepeatQuery(n)\ \underline{\Delta}$
$BMetronome(n)\ >x>\ Query \gg Accept(x)$

Using only the basic composition operators, an expression can produce only a bounded number of values. As we see in *Metronome*, recursive definitions allow unbounded computations. Many more examples of the use of recursion appear throughout this paper.

3.2.10 Starting and ending a computation

A computation is started from a host language program by executing an *Orc statement*

$z{:}\in\ E(L)$

where z is a variable of the host program, E is the name of an expression and L is a list of actual parameters. All variable parameters in L are variables of the host language program, and they have values before E's evaluation starts. (This is unlike calls to expressions made during evaluation of an Orc expression. Then, the parameters may not have values when the expression evaluation begins.) To execute this statement, start the evaluation of a fresh instance of E with actual parameter values substituted for the formal ones, assign the *first* value produced to z, and then terminate the evaluation. If expression evaluation produces no value, the execution of the statement does not terminate.

In many distributed programming applications, expression evaluation never produces a value though it has effects on the external world through its site calls. Several such examples appear in sections 4 and 6. In such cases, the Orc statement should be placed within a thread of the host language program with the expectation of non-termination.

3.3 Non-determinism and Referential Transparency

3.3.1 Angelic non-determinism

In evaluating $(M \mid N) \gg R$, it is tempting to accept the first value computed for $(M \mid N)$ and call R only with this input, a form of demonic choice. But we reject this strategy, because we would like to explore all possible computation paths denoted by the expression. That is, we employ *angelic* non-determinism. Therefore, we call R with all values returned by M *and* N. And R may respond after, say, N has returned its value, but fail to respond after M. One pleasing outcome of this evaluation strategy is that we have the identity (see section 5),

$$(M \mid N) \gg R\ =\ M \gg R\ \mid\ N \gg R,$$

and, more generally, the following distributivity law over expressions f, g and h.

(Right Distributivity of $\gg$ over $\mid$)
$(f \mid g) \gg h\ =\ (f \gg h \mid g \gg h)$

See section 4.10 for a solution to the eight queens problem which exploits angelic non-determinism.

3.3.2 Demonic Nondeterminism

In a functional programming language like Haskell[15], the **where** construct provides a convenient mechanism for program structuring and efficient evaluation of expressions. It is not a necessity because of *referential transparency*: a variable defined by a **where** clause can be eliminated from an expression by replacing its occurrence by its definition.

In Orc, the **where** clause is essential to implement *demonic* nondeterminism: to accept a single value of an Orc expression and discard the remaining ones. Therefore,

$let(x) \gg M$
where
$x{:}\in\ N \mid R$

is *not* equivalent to

$(N \mid R)\ >x>\ M$

In the first case, evaluation of $(N \mid R)$ is terminated after it returns a value. Therefore, M is called at most once. In the second case, each value returned by $(N \mid R)$ forces a fresh evaluation of M. The second form of programming (angelic) allows us to explore all possible computation paths, and the first form (demonic) permits a more efficient evaluation strategy, used when only some of the paths need to be explored.

3.3.3 Referential Transparency

Orc is referentially transparent: the name of an expression can be replaced by its body in any context to yield an equivalent expression. We list some equivalences in section 5.

We show that referential transparency and the semantics of **where** expressions force us to implement non-strict evaluation of expressions. Consider the example of parallel-or (section 4.8) which we reproduce below.

$Parallel\_or\ \underline{\Delta}$
$\{if(x)\ \mid\ if(y)\ \mid\ or(x,y)$
where
$x{:}\in\ M$
$y{:}\in\ N\}$

Define

$$Por(u,v) \;\underline{\Delta}\; \{if(u) \mid if(v) \mid or(u,v)\}$$

Under referential transparency,

$$\{Por(x,y) \;\textbf{where}\; x :\in M \;\; y :\in N\}$$

has the same semantics as $Parallel\_or$. This requires us to call Por as soon as the evaluation of the **where** expression starts, i.e., before x and y have values.

3.4 Small Examples

We give a number of small examples to familiarize the reader with the programming notation. Some fundamental programming idioms appear in the next section and a few longer examples appear in section 6.

Timing thread creations Make four requests to site M, in intervals of one time unit each.

$$M \mid Rtimer(1) \gg M \mid Rtimer(2) \gg M \mid Rtimer(3) \gg M$$

If site M returns result v before t time units, set z to v; if after t (or never), set z to 0; if at t, set z to either value.

$$z :\in M \mid Rtimer(t) \gg let(0)$$

If the computation shown above is to be embedded as part of a larger expression evaluation, we write

$$\{let(z) \;\textbf{where}\; z :\in M \mid Rtimer(t) \gg let(0)\}$$

Selective timing with threads Receive N's response as soon as possible, but no earlier than one time unit from now. Expression $Rtimer(1) \gg N$ delays calling N for a time unit and expression $\{N > x > Rtimer(1) \gg let(x)\}$ delays producing the response for a unit after it is received. What we want is to call N immediately but delay its response until a time unit has passed. The following expression achieves this.

$$DelayedN \;\underline{\Delta}\; \{Rtimer(1) \gg let(u) \;\textbf{where}\; u :\in N\}$$

We can use this expression to give priority to M over N in the following manner: Request M and N for values, but give priority to M by allowing its response to overtake N's response provided it arrives within the first time unit. We write

$$x :\in M \mid DelayedN$$

Flow rate calculation Count the number of values produced by expression f in 10 time units. We use a local site *count* which implements a counter. The initial value of the counter is 0; calling *count.inc* increments the counter and returns a signal, and *count.read* returns the counter value. In this solution, the value returned by *count.inc* is explicitly ignored, because we are only interested in producing a single value after 10 time units.

$$f \gg count.inc \gg \mathbf{0} \mid Rtimer(10) \gg count.read$$

This expression can be used to compare the rate at which two sources (say, expressions f and g) are producing values. We may then choose one source over another when both are producing the same stream of values. Flow rate computation is important in many applications. Cardelli and Davies [6] introduces a basic language construct to compute flow rates for bit streams.

Recursive definition with time-out Call a list of sites and tally the number of responses received in a certain time interval. Below, $tally(L)$ implements this specification where L is a list of sites, m is a (fixed) argument for each site call, and the time interval is 10 units. This example illustrates the use of recursion over a list. We use the Haskell [15] notation for lists, denoting an empty list by $[]$, and a list with head x and tail xs by $(x : xs)$. Below, site call $add(u,v)$ returns the sum of u and v.

$$\begin{aligned}
&tally([]) \;\underline{\Delta}\; let(0)\\
&tally(x:xs) \;\underline{\Delta}\\
&\quad \{add(u,v)\\
&\quad\quad \textbf{where}\\
&\quad\quad\quad u :\in x(m) \gg let(1) \mid Rtimer(10) \gg let(0)\\
&\quad\quad\quad v :\in tally(xs)\}
\end{aligned}$$

4 Programming Idioms

Lexical conventions Orc does not include any facility for doing primitive operations on data, such as arithmetic or predicate evaluation. We have to call specific sites to carry out such operations. For example, to add x and y we need to call $add(x,y)$, which returns their sum. In our examples, we take the liberty of writing $x + y$ as an arithmetic expression; it is easily converted to an Orc expression by a compiler. Similarly, we write expressions over booleans, lists and other data types. And we use the fundamental sites defined in Table 1 (page 9).

We use quantification in the following form: $(\mid i : 0 \le i \le 2 : P_i)$ is an abbreviation for $(P_0 \mid P_1 \mid P_2)$. Similarly, $\{f \;\textbf{where}\; (\forall i : 0 \le i \le 2 : x_i :\in g_i)\}$ is $\{f \;\textbf{where}\; x_0 :\in g_0,\; x_1 :\in g_1,\; x_2 :\in g_2\}$. We omit the range of i when it is clear from the context.

4.1 Sequential computing

Orc is not intended as a replacement for sequential programming. Yet its constructs can be used to simulate control structures of sequential programming languages, as we show in this section.

Sequencing The sequential program fragment $(S;\ T)$ is $(S \gg T)$ in Orc. If S is an assignment statement $x := e$, the Orc code is $(E > x > T)$ where Orc expression E returns the (single) value of e. This encoding also supports reassignments of variables.

Conditional execution A typical if-then-else statement,

$$\textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ T$$

where b is a predicate, is coded in Orc as

$$\mathit{if}(b) \gg S \ \mid\ \mathit{if}(\neg b) \gg T$$

Note that of the two threads created here, only one can proceed to compute a value. As a specific example, the following expression returns the absolute value of its numerical argument.

$$\begin{array}{l} \mathit{absolute}(x) \ \underline{\Delta} \\ \ \mathit{if}(x \geq 0) \gg \mathit{let}(x) \ \mid\ \mathit{if}(x < 0) \gg \mathit{let}(-x) \end{array}$$

Iteration A typical loop in an imperative program has the form

$$\textbf{while}\ b\ \textbf{do}\ x := S(x)$$

where x may be a set of variables. We simulate this code fragment in Orc as follows; the value returned by the Orc expression is that of x.

$$\begin{array}{l} \mathit{loop}(x) \ \underline{\Delta} \\ \ \mathit{if}(b) \gg S(x) \ >y>\ \mathit{loop}(y) \ \mid\ \mathit{if}(\neg b) \gg \mathit{let}(x) \end{array}$$

Consider a typical program which starts with an initialization, followed by a loop and a terminating computation.

$$\begin{array}{l} x := x_0; \\ \textbf{while}\ b\ \textbf{do}\ x := S(x); \\ \text{return } T(x) \end{array}$$

This is equivalent in Orc to $\{\mathit{loop}(x_0) \ >x>\ T(x)\}$.

4.2 Kleene Star and Primitive Recursion

In the theory of regular expressions, M^* denotes the set of strings formed by concatenating zero or more M symbols. By analogy, we would like to define an expression, $\mathit{Mstar}(x)$, which returns the stream of results

$$\begin{array}{l} x,\ M(x),\ M(x) >y> M(y), \\ \ M(x) >y> M(y) >z> M(z),\ \ldots \end{array}$$

Our definition of this expression is

$$\mathit{Mstar}(x) \ \underline{\Delta}\ \mathit{let}(x) \mid M(x) >y> \mathit{Mstar}(y)$$

Closely related $\mathit{Mstar}(x)$ is $\mathit{Mplus}(x)$ which returns the same stream as $\mathit{Mstar}(x)$ except its very first value, i.e., the stream

$$\begin{array}{l} M(x),\ M(x) >y> M(y), \\ \ M(x) >y> M(y) >z> M(z),\ \ldots \end{array}$$

We define

$$\mathit{Mplus}(x) \ \underline{\Delta}\ M(x) >y> (\mathit{let}(y) \mid \mathit{Mplus}(y))$$

More general expressions which take M as parameter are,

$$\begin{array}{l} \mathit{Star}(M,x) \ \underline{\Delta}\ \mathit{let}(x) \mid M(x) >y> \mathit{Star}(M,y) \\ \mathit{Plus}(M,x) \ \underline{\Delta}\ M(x) >y> (\mathit{let}(y) \mid \mathit{Plus}(M,y)) \end{array}$$

Creating a stream of successive approximations Consider a numerical analysis program which computes its final value by successive approximations from an initial value. It checks each produced value for a convergence criterion, and stops the computation once a *convergent* value is found (i.e., one that meets the convergence criterion).

Site $\mathit{Refine}(x)$ returns a refined approximation of x; and $\mathit{Converge?}(x)$ returns x if x is a convergent value, it is silent otherwise. We define $\mathit{RefineStream}(x)$ which returns a stream of successive approximations starting from x, and $\mathit{RefineConverge}(x)$ which returns the substream of $\mathit{RefineStream}(x)$ of convergent values.

$$\begin{array}{l} \mathit{RefineStream}(x) \ \underline{\Delta} \\ \ \mathit{Star}(\mathit{Refine}, x) \\ \mathit{RefineConverge}(x) \ \underline{\Delta} \\ \ \mathit{RefineStream}(x) \ >y>\ \mathit{Converge?}(y) \end{array}$$

Use $\{\mathit{let}(z)\ \textbf{where}\ z :\in \mathit{RefineConverge}(x)\}$ to stop the computation after finding the first convergent value.

4.3 Arbitration

A fundamental problem in concurrent computing is *arbitration*: to choose between two threads and let only one proceed. Arbitration is the essence of mutual exclusion. In process algebras like CCS and CSP, specific operators are included to allow arbitration; in very simple terms, $\alpha.P + \beta.Q$ is a process which behaves as process P if action α happens and as Q if β happens.

In Orc terms, α and β correspond to sites *Alpha* and *Beta* and P and Q are expressions. We have the expression $Alpha \gg P \mid Beta \gg Q$, though we evaluate only one of the threads, P or Q, depending on which site, *Alpha* or *Beta*, responds first. (This is similar, though not identical, to the process algebra expression, where only one of α or β succeeds; here, we have to attempt both *Alpha* and *Beta*, and choose one when both succeed.) Below, *flag* records which of *Alpha* and *Beta* responds first.

$$
\begin{array}{l}
if(flag) \gg P \mid if(\neg flag) \gg Q \\
\quad \textbf{where} \\
\qquad flag :\in (Alpha \gg let(true)) \mid (Beta \gg let(false))
\end{array}
$$

If P and Q use the values returned by *Alpha* and *Beta*, modify the program:

$$
\begin{array}{l}
if(flag) \gg let(x) \gg P \mid if(\neg flag) \gg let(x) \gg Q \\
\quad \textbf{where} \\
\qquad (x, flag) :\in \\
\qquad\quad Alpha\ >y>\ let(y, true) \\
\qquad \mid Beta\ >y>\ let(y, false)
\end{array}
$$

An important special case of arbitration involves time-out: run P if *Alpha* responds within 1 time unit, otherwise run Q. This amounts to encoding *Beta* as $Rtimer(1)$. A more detailed treatment of time-out appears next.

The Orc model permits more complex arbitration protocols, such as, execute one of P, Q and R, depending how many sites out of *Alpha*, *Beta* and *Gamma* respond within 10 time units.

4.4 Time-out

For time-out with f, write $\{f \mid Rtimer(t) \gg let(x)\}$, which either returns a result from f, or times out after t units and returns x. A typical programming paradigm is to call site M and return a pair (x, b) as the value, where b is true if M returns x before the time-out, and false if there is a time-out. In the latter case, x is irrelevant.

$$
\begin{array}{l}
let(z) \\
\quad \textbf{where} \\
\qquad z :\in \\
\qquad\quad M >y> let(y, true) \\
\qquad \mid Rtimer(t) >y> let(y, false)
\end{array}
$$

As a more involved example, call *Refine* repeatedly (starting with initial value x_0) and return the last value (the most refined) before time t. Below, $BestRefine(t, x)$ implements this specification. It returns x if there is a time-out; else it returns $BestRefine(t, y)$, where y is the value returned by *Refine* before the time-out.

$$
\begin{array}{l}
BestRefine(t, x)\ \underline{\Delta} \\
\quad if(b) \gg BestRefine(t, y) \mid if(\neg b) \gg let(x) \\
\qquad \textbf{where} \\
\qquad\quad (y, b) :\in \\
\qquad\qquad Refine(x)\ >y>\ let(y, true) \\
\qquad\quad \mid Atimer(t)\ >y>\ let(y, false)
\end{array}
$$

The parameter t of *BestRefine* is an absolute time. To modify the argument to a relative time h, define $BestRefineRel(h, x)$ as follows.

$$
\begin{array}{l}
BestRefineRel(h, x)\ \underline{\Delta} \\
\quad clock >y> BestRefine(y + h, x)
\end{array}
$$

4.5 Fork-join Parallelism

In concurrent programming, we often need to spawn two independent threads at a point in the computation, and resume the computation after both threads complete. Such an execution style is called *fork-join* parallelism. There is no special construct for fork-join in Orc, but it is easy to code such computations. The following code fragment calls sites M and N in parallel and returns their values as a tuple after they both complete their executions.

$$
\begin{array}{l}
\{\ let(u, v) \\
\qquad \textbf{where}\ u :\in M \\
\qquad\qquad\quad v :\in N \\
\}
\end{array}
$$

As a simple application of fork-join, consider refreshing a display device at unit time intervals. The display is drawn by calling site *Draw* with a triple: a given screen image, keyboard inputs and the mouse position. We use *Metronome* (see section 3.2.9, page 16) to generate a signal at every unit, then start independent threads to acquire the image, keyboard inputs and the mouse position, and on completion of all three threads, call *Draw*. We code this as

$$
\begin{array}{ll}
 & Metronome \\
\gg & \{\ let(i, k, m) \\
 & \quad \textbf{where}\ i :\in Image \\
 & \qquad\qquad k :\in Keyboard \\
 & \qquad\qquad m :\in Mouse \\
 & \} \\
\multicolumn{2}{l}{>(i, k, m)>} \\
 & Draw(i, k, m)
\end{array}
$$

The implicit assumption in this code is that i, k and m are evaluated faster than the refresh rate of one time unit.

4.6 Synchronization

Synchronization of threads is fundamental in concurrent computing. There is no special machinery for synchronization in Orc; a **where** expression provides the necessary ingredients for programming synchronizations. Consider two threads $M \gg f$ and $N \gg g$; we wish to execute them independently, but synchronize f and g by starting them only after *both* M and N have completed.

$$\begin{array}{l} \{\ let(u,v) \\ \quad \textbf{where}\ u :\in\ M \\ \qquad\qquad v :\in\ N\} \\ \gg\ (f \mid g) \end{array}$$

If the values returned by M and N have to be passed on to f and g, respectively, we modify the expression to

$$\begin{array}{ll} & \{\ let(u,v) \\ & \quad \textbf{where}\ u :\in\ M \\ & \qquad\qquad v :\in\ N\} \\ >(u,v)> & \\ & (f \mid g) \end{array}$$

Barrier synchronization The form of synchronization we have shown is known in the literature as *barrier* synchronization. In the general case, each independent thread executes a sequence of phases. The $(k+1)^{th}$ phase of a thread is begun only if *all* threads have completed their k^{th} phases. A straightforward generalization of the given expression solves the barrier synchronization problem.

Barrier synchronization is common in scientific computing. For example, Gauss-Siedel iteration proceeds in phases where the $(k+1)^{th}$ approximation for all variables are computed from their k^{th} approximations. In heat transfer computation over a grid, the temperature at point (i,j) at moment $k+1$ is the average temperature over its neighboring points at moment k. The computation proceeds until some convergence criterion is met (we assume that the boundary points have constant temperature). We give a sketch of heat transfer computation in Orc.

Given the temperature matrix x for some moment, where x_{ij} is the temperature at grid point (i,j), *Refine*(x) produces matrix y, the temperature at the next moment. Site *Next* computes the temperature at a point from its and its neighbors' previous temperatures. Typically, it would return the average temperature of the neighboring points of (i,j) if (i,j) is not a boundary point, but it may implement more sophisticated strategies. For a boundary point, the neighboring temperatures are irrelevant and it returns the previous temperature. The notation used below for enumerating y_{ij} should be self-explanatory.

$$\begin{array}{l} \mathit{Refine}(x)\ \underline{\Delta} \\ \quad \{let(y) \\ \qquad \textbf{where} \\ \qquad (\forall i,j ::\ y_{ij} :\in \\ \qquad\qquad\quad Next(x_{ij}, x_{(i-1)j}, x_{(i+1)j}, x_{i(j-1)}, x_{i(j+1)})) \\ \quad \} \end{array}$$

As in section 4.2, we can get a convergent value by using *RefineConverge*. Using this strategy, the heat transfer computation is run by

$$z :\in\ \mathit{RefineConverge}(I)$$

where I is the initial temperature matrix.

4.7 Interrupt

Consider an Orc expression which orchestrates the vacation planning for a family. It makes airline and hotel reservations by contacting several sites and choosing the most suitable ones according to the criteria set by the client. Suppose the client decides to cancel vacation plans while the Orc program is still executing. There is no mechanism for the client to interrupt the program because an Orc expression is evaluated like an arithmetic expression, not as a process which waits to receive messages. In this section, we show how an expression evaluation can be interrupted and, more importantly, how a different computation (such as roll back) can be initiated in case of interruption. This is important in many practical applications, such B2B transactions, where clients of a company may interrupt its computations by specifying new requirements, and vendors may wish to renegotiate their promises about parts delivery. For the vacation planner, an interruption by the client may require it to cancel any reservations it may have made.

We have already seen a form of interrupt: time-out. To allow for general interrupts, set up sites *Interrupt.set* and *Interrupt.get*. An external agent calls *Interrupt.set* to interrupt the evaluation of an expression. And, calling *Interrupt.get* returns a signal only if *Interrupt.set* has been called earlier. Note the similarity of *Interrupt* to a semaphore, where *set* and *get* are the V and P operations on the semaphore.

If a call on site M can be interrupted, use

$$let(z)\ \textbf{where}\ z :\in\ M \ \mid\ \mathit{Interrupt.get}$$

where z acquires a value from M or *Interrupt.get*. It is easy to extend this solution to handle different types of interrupts, by waiting to receive from many possible interruption sites, and returning specific codes for each kind of interrupt.

Often we wish to determine if there has been an interrupt. Then we return a tuple whose first component is the value from M (if any) and the second component is a boolean to indicate whether there has been an interrupt:

$$
\begin{array}{l}
let(z,b)\\
\quad \textbf{where}\\
\quad\quad (z,b) :\in\\
\quad\quad\quad M \ {>}y{>}\ let(y,true)\\
\quad\quad\ |\ Interrupt.get\ \ {>}y{>}\ let(y,false)
\end{array}
$$

An easy generalization is to interrupt a stream. Below, expression $callM$ calls M repeatedly until it is interrupted. It produces a stream of tuples: $(x, true)$ for value x received from M and $(-, false)$ for interrupt. It terminates computation after receiving an interrupt.

$$
\begin{array}{l}
callM\ \underline{\Delta}\\
\quad let(x,b)\ \ |\ \ if(b) \gg callM\\
\quad\quad \textbf{where}\\
\quad\quad\quad (x,b) :\in\\
\quad\quad\quad\quad M\ \ {>}y{>}\ let(y,true)\\
\quad\quad\quad\ |\ Interrupt.get\ \ {>}y{>}\ let(y,false)
\end{array}
$$

Typically, occurrence of an interrupt is followed by interrupt processing. An expression which processes the values from M and the interrupt differently is shown below.

$$
\begin{array}{ll}
 & callM\\
{>}(x,b){>} & \\
 & \{\ if(b)\ \ \gg \text{“Normal processing using } x\text{”}\\
 & \ |\ if(\neg b)\ \gg \text{“Interrupt Processing”}\ \}
\end{array}
$$

4.8 Non-strict Evaluation; Parallel-or

A classic problem in non-strict evaluation is *Parallel-or*: computation of $x \vee y$ over booleans x and y. The value of $x \vee y$ is *true* if either variable value is *true*; therefore, the expression evaluation may terminate even when one of the variable values is unknown. In this section, we state the problem in Orc terms, give a simple solution, and show examples of its use in web services orchestration.

Suppose sites M and N return booleans. Compute the *parallel-or* of the two booleans, i.e., (in a non-strict fashion) return *true* as soon as either site returns *true* and *false* only if both sites return *false*. In the following solution, site $or(x, y)$ returns $x \vee y$.

$$
\begin{array}{l}
\{if(x)\ \ |\ \ if(y)\ \ |\ \ or(x,y)\\
\quad \textbf{where}\\
\quad\quad x :\in\ M\\
\quad\quad y :\in\ N\}
\end{array}
$$

This solution may return up to three different values, depending on how many of x and y are *true*. To return just one value, use

$$
\begin{array}{l}
\{let(z)\\
\quad \textbf{where}\\
\quad\quad z :\in\ if(x)\ \ |\ \ if(y)\ \ |\ \ or(x,y)\\
\quad\quad x :\in\ M\\
\quad\quad y :\in\ N\}
\end{array}
$$

A generalization of this expression for a list of sites is as follows.

$$
\begin{array}{ll}
Paror([\,]) & \underline{\Delta}\ \ let(false)\\
Paror(u:us) & \underline{\Delta}\\
 & \quad \{let(z)\\
 & \quad\quad \textbf{where}\\
 & \quad\quad\quad z :\in\ if(x)\ \ |\ \ if(y)\ \ |\ \ or(x,y)\\
 & \quad\quad\quad x :\in\ u\\
 & \quad\quad\quad y :\in\ Paror(us)\}
\end{array}
$$

We can use the strategy of parallel-or to evaluate any function f of the form

$$
f(x,y) = \begin{cases} p(x) & \text{if } \ c(x)\\ q(y) & \text{if } \ d(y)\\ r(x,y) & \text{otherwise} \end{cases}
$$

where x and y are received from different sites. Many search problems over partitioned databases have this structure.

Airline Booking We show a typical orchestration example in which parallel-or plays a prominent role in one of the solutions.

There are two airlines A and B each of which returns a *quote*, i.e., the price of a ticket to a certain destination. We show several variations in choosing a quote.

First, compute the cheapest quote. Below, *Min* is a site which returns the minimum of its arguments.

$$
\{Min(x,y)\ \ \textbf{where}\ \ x :\in\ A,\ y :\in\ B\}
$$

Our next solution returns each quote that is below some threshold value c, and there is no response if neither quote is below c. Assume that site *threshold* returns the value of its argument provided it is below c, and it is silent otherwise. Expression

$$
(A\ \ |\ \ B)\ \ {>}y{>}\ \ threshold(y)
$$

returns each quote that is below the threshold. To obtain at most one such quote, we write

$$
\{let(z)\ \ \textbf{where}\ \ z :\in\ (A\ \ |\ \ B)\ \ {>}y{>}\ \ threshold(y)\}
$$

To return any quote if it is below c as soon as it is available, otherwise return the minimum quote, we use the strategy of parallel-or.

$\{threshold(x) \ \mid \ threshold(y) \ \mid \ Min(x,y)$
where
$x{:}\in \ A$
$y{:}\in \ B\}$

4.9 Communicating Processes

Orchestration is closely tied to distributed computing. Traditional distributed computing is structured around a network of processes, where the processes communicate by participating in *events*, or reading and writing into common channels. Processes are usually long-lived entities. In many cases, we do not expect a distributed computation to terminate. Programming constructs of Orc, as we have seen, can implement essential distributed computing paradigms, such as arbitration, synchronization and interrupt. We argue that they are also well-suited for encoding process-based computations.

As a small example, consider a light bulb which is controlled by two switches. Flipping either switch changes the state of the bulb, from *off* to *on* and *on* to *off*. This behavior is captured by

$$
\begin{array}{ll}
Light \ \underline{\Delta} & \\
& \{let(x) \ \textbf{where} \ x{:}\in \ switch1 \mid switch2\} \\
\gg & ChangeBulbState \\
\gg & Light
\end{array}
$$

This expression never returns a value, but causes the light bulb to change state (using site $ChangeBulbState$). Note that only one of the switch flips is recognized if both switches are flipped before the bulb state changes.

Channel We introduce channels for communication among processes. It is not an Orc construct; each channel has to be implemented by sites outside Orc. We assume in our examples that channels are FIFO and unbounded, though other kinds of channels (including rendezvous-based communications) are easily implemented through sites.

Channel c has two methods, $c.get$ and $c.put$, which are called as sites from an Orc expression. Calling $c.put(m)$ adds item m to the end of the channel and returns a signal. Calling $c.get$ returns the value at the head of c and removes it from c; if the channel is empty, $c.get$ queues the caller until it becomes non-empty.

Fairness We make no fairness assumption about the queuing discipline at a site such as $c.get$. Calls are handled in arbitrary order and some caller may never receive a value even though values are being constantly put in the channel. However, if c is non-empty, the channel sends a value to some caller of $c.get$, and this value is eventually received by the caller. Therefore, a call to $c.get$ during an expression evaluation completes eventually if c is non-empty and this is the only caller.

Process A process is an expression which, typically, names channels which are shared with other expressions. Shown below is a simple process which reads items from its input channel c, calls site $Compute$ to do some computations with the item and then writes the result on output channel e.

$$
\begin{array}{l}
P(c,e) \ \underline{\Delta} \\
\quad c.get \ >x> \ Compute(x) \ >y> \ e.put(y) \gg P(c,e)
\end{array}
$$

Process $P(c,e)$ above produces no value, though it writes on channel e. To output every value which is also written on e, define

$$
\begin{array}{l}
Q(c,e) \ \underline{\Delta} \\
\quad c.get \ >x> \\
\quad Compute(x) \ >y> \\
\quad (let(y) \mid e.put(y) \gg Q(c,e))
\end{array}
$$

Define process P to read inputs from two input channels, c and d, independently, and write into e.

$$P \ \underline{\Delta} \ P(c,e) \mid P(d,e)$$

The following small example illustrates a dialog with a user process. The process reads a positive integer as input from a terminal called tty (which is a channel), checks if the number is prime and outputs the result to the terminal. It repeats these steps as long as input is provided to it.

$$
\begin{array}{lll}
Dialog \ \underline{\Delta} & & \\
\quad tty.get & >x> \\
\quad Prime?(x) & >b> \\
\quad tty.put(b) & \gg \\
\quad Dialog &
\end{array}
$$

Process Network A process network is a parallel composition of processes. There is no logical difference between a process and a network. For example, P is defined to be $P(c,e) \mid P(d,e)$ above, and it may be regarded as a process which executes two threads simultaneously or as a network of two processes.

Let us build a process which reads from a set of channels c_i, where i ranges over some set of indices, and outputs all the items read into channel e. That is, the process creates a fair merge of the input channels. The definition is a generalization of P, shown above, for multiple input channels, though the $Compute$ step is eliminated.

$$
\begin{array}{ll}
Multiplexor_i & \underline{\Delta} \\
\quad c_i.get \ >y> \ e.put(y) \gg Multiplexor_i & \\
Multiplexor & \underline{\Delta} \\
\quad (\ \mid i :: Multiplexor_i) &
\end{array}
$$

Mutual exclusion Consider a set of processes, Q_i, which share a resource, and access to the resource has to be exclusive. This is a mutual exclusion, or arbitration, problem and it is easily solved in Orc.

Process Q_i writes its own id i to channel c_i to request the resource. We employ the *Multiplexor*, above, to read the values from all c_i and write them to channel e. The arbiter reads a value i from e and calls site $i.Grant$ to permit Q_i to use the resource. After using the resource, Q_i returns a signal as the response of site call $i.Grant$. Expression *Mutex* orchestrates mutual exclusion.

$$\begin{array}{lll} Arbiter & \underline{\Delta} & e.get \; >i> \; i.Grant \gg Arbiter \\ Mutex & \underline{\Delta} & Multiplexor \mid Arbiter \end{array}$$

Note that the solution is starvation-free for each Q_i, because its request will be read eventually from c_i, put in channel e, read again from e and granted. This assumes that every process i releases the resource eventually by responding to $i.Grant$. The solution is easily modified to snatch the resource from an (unyielding) process after a time-out.

Synchronized Communications: Byzantine Protocol We can combine many of the earlier idioms to code more involved process behavior. Consider, for example, the Byzantine agreement protocol [21], which runs for a number of synchronized *rounds*. In each round, a process sends its own estimate (of the consensus value) to all processes, receives estimates from all processes (including itself), and computes a revised estimate, which it sends in the next round. The communications from process i to j use channel c_{ij}. We show the orchestration of the steps, though we omit (the crucial detail of) computing a new estimate, which we delegate to a site.

The sending of estimate v by process i to all processes is coded by

$$Send_i(v) \;\; \underline{\Delta} \;\; Signal \mid (\; \mid j :: c_{ij}.put(v) \gg \mathbf{0})$$

Evaluation of $Send_i(v)$ returns a signal immediately and appends v to all outgoing channels of i. The responses from $c_{ij}.put(v)$ are ignored (by using $\gg \mathbf{0}$).

Expression $Read_i$ encodes one round of message receipt by process i. Below, X is a vector of estimates and X_j is its j^{th} component.

$$Read_i \;\; \underline{\Delta} \;\; let(X) \; \mathbf{where} \; (\forall j :: X_j :\in c_{ji}.get)$$

Process i computes a new estimate from X by calling $Compute_i(X)$.

A round at process i is a sequence of $Send_i$, $Read_i$ and $Compute_i$. Define $Round_i(v,n)$ as n rounds of computation at process i starting with v as the initial estimate. The result of $Round_i(v,n)$ is a single estimate.

$$\begin{array}{lll} Round_i(v,0) & \underline{\Delta} & let(v) \\ Round_i(v,n) & \underline{\Delta} & \\ \quad Send_i(v) \gg & & \\ \quad Read_i \; >X> & & \\ \quad Compute_i(X) \; >u> & & \\ \quad Round_i(u, n-1) & & \end{array}$$

The entire algorithm is coded by $Byz(V,n)$, where V is the vector of initial estimates and n is the number of rounds. Below, i ranges over process indices.

$$Byz(V,n) \;\; \underline{\Delta} \;\; \{ \mid i :: Round_i(V_i, n)\}$$

Dining Philosophers The dining philosophers is a fundamental problem of shared resource allocation. We give a solution in Orc which resembles a process-based solution given in Hoare [17]. In this example, processes communicate using bounded buffers.

There are N processes, called *Philosophers*, where the i^{th} process is denoted by P_i. The philosophers are seated around a table where the right neighbor of P_i is $P_{i'}$ (henceforth, i' is $(i+1) \bmod N$). Every pair of neighbors share a fork. The fork to the left of P_i is $Fork_i$ and to his right is $Fork_{i'}$. Philosopher i can eat only if it holds *both* its left and right forks. Assume that a philosopher's life cycle consists of repeating the following steps: acquire the two adjacent forks, eat, and release the forks. Because of the seating arrangement, neighboring philosophers are prevented from eating simultaneously.

Each $Fork_i$ is a channel which is either empty (if some philosopher holds the corresponding fork) or has one signal (if no philosopher holds the fork). We write $Fork_i.put$ to send a signal along the channel. Initially, each channel holds a signal.

A philosopher's life is depicted by the following expression. Below, *Eat* returns a signal on completion of eating.

$$\begin{array}{l} P_i \;\; \underline{\Delta} \\ \qquad \{let(x,y) \gg Eat \gg Fork_i.put \gg Fork_{i'}.put \\ \qquad\quad \mathbf{where} \;\; x :\in Fork_i.get \\ \qquad\qquad\qquad y :\in Fork_{i'}.get \\ \qquad \} \\ \quad \gg P_i \end{array}$$

Represent the ensemble of philosophers by

$$DP \;\; \underline{\Delta} \;\; (\; \mid i : 0 \le i < N : P_i)$$

Deadlock It is well known that the given solution for dining philosophers has the potential for deadlock. To avoid deadlock, philosophers pick up their forks in a specific order: all except P_0 pick up their left and then their right forks, and P_0 picks up its right and then its left fork.

$$\begin{array}{l} P_0 \;\; \underline{\Delta} \\ \qquad Fork_1.get \gg Fork_0.get \gg Eat \gg \\ \qquad Fork_1.put \gg Fork_0.put \gg P_0 \end{array}$$

$$\begin{array}{l} P_i, \, 1 \le i < N, \;\; \underline{\Delta} \\ \qquad Fork_i.get \gg Fork_{i'}.get \gg Eat \gg \\ \qquad Fork_i.put \gg Fork_{i'}.put \gg P_i \end{array}$$

This example illustrates that the evaluation of an Orc expression may lead to deadlock when it spawns threads which wait for each other. Since the threads communicate only through sites, deadlock can be avoided if each site call is guaranteed to return a result. Many distributed applications communicate with web services, like a stock quote service, which have this property; so deadlock avoidance is easily established in these cases. For other site calls, like $c.get$ on channel c, there is no guarantee of receiving a result. But by judiciously using time-outs as alternatives of site calls in Orc expressions, we can ensure that a result is always delivered, and deadlock is avoided.

4.10 Backtrack Search

For problems which are traditionally implemented by backtracking, we exploit angelic non-determinism of Orc to express their solutions succinctly. The evaluation of the Orc expression will create multiple threads, which may be implemented by backtracking. Among the problems which are easily coded are parsing problems in language theory and combinatorial search. We show the solution to one well-known search problem below.

A classical backtracking problem: Eight queens The eight queens problem is to place 8 queens on a chess board so that no queen can capture another. Many interesting solutions appear in "Eight Queens In Many Programming Languages" [25].

A placement of queens in the last i rows of the board, $0 \le i < 8$, is called a *configuration*. A configuration is represented by a list of integers in the range 0 through 7, denoting the column in which the corresponding queen is placed. A configuration is *valid* if none of the queens in it can capture any other. Site call $check(x: xs)$, where $(x: xs)$ is a non-empty configuration and xs is valid, returns $(x: xs)$ provided it is valid; if $(x: xs)$ is not valid, it remains silent. We can implement *check* easily; determine if the queen at x can capture any of the queens represented by xs.

Expression $extend(x, n)$, where x is a valid configuration, n is an integer, $1 \le n$ and $|x| + n \le 8$, produces *all* valid extensions of x by placing n additional queens. The original problem is solved by calling $extend([], 8)$, which yields all possible solutions.

We define $extend(x, n)$ as the n-fold application of $extend(x, 1)$. And $extend(x, 1)$ returns all valid extensions of x by one-position.

$$\begin{aligned} extend(x,n) &\;\Delta\; extend(x,1) \;>y>\; extend(y, n-1) \\ extend(x,1) &\;\Delta\; (\forall i : 0 \le i < 8 : check(i : x)) \end{aligned}$$

5 Laws about Orc Expressions

We list a number of laws about Orc expressions. These laws are also valid for regular expressions of language theory, which is a Kleene algebra [20]. Some Orc

expressions can be regarded as regular expressions. An Orc term is a symbol of the alphabet. Constant $\mathbf{0}$ corresponds to the empty set. For this section, we introduce symbol $\mathbf{1}$ which corresponds to the set that contains the empty string. Define $(f >x> \mathbf{1})$ as $(f >x> let(x))$, i.e., $\mathbf{1}$ is a site which reproduces the result of the last computation. However, $\mathbf{1} \gg f$ is not well defined, because the initial value of a computation is not defined. Therefore, the law (Left unit of $\gg$) holds only within a context where $\mathbf{1}$ is defined.

Below, we treat only $\gg$, the special case of $>x>$. Operator $\gg$ is associative. Operator $>x>$ is right associative; and it is fully associative if both sides in the following identity are well-formed.

$$(f \;>x>\; g) \;>y>\; h \;=\; f \;>x>\; (g \;>y>\; h)$$

Operators $\mid$ and $\gg$ mimic alternation and concatenation. There is no operator in Orc corresponding to $*$ of regular expressions, which we simulate using recursion. Additionally, Orc includes the **where** operator, which has no correspondence in language theory.

All Orc expressions including **where** expressions obey the laws given in this section. They can be proved using the formal semantics of Orc [18].

5.1 Kleene laws

Below f, g and h are Orc expressions.

(Zero and $\mid$)	$f \mid \mathbf{0} = f$
(Commutativity of $\mid$)	$f \mid g = g \mid f$
(Associativity of $\mid$)	$(f \mid g) \mid h = f \mid (g \mid h)$
(Left zero of $\gg$)	$\mathbf{0} \gg f = \mathbf{0}$
(Left unit of $\gg$)	$\mathbf{1} \gg f = f$
(Right unit of $\gg$)	$f \gg \mathbf{1} = f$
(Associativity of $\gg$)	$(f \gg g) \gg h = f \gg (g \gg h)$
(Distr. of $\gg$ over $\mid$)	$(f \mid g) \gg h = (f \gg h \mid g \gg h)$

Some of the axioms of Kleene algebra do not hold in Orc. First is the *idempotence* of $\mid$, $f \mid f = f$. Consider M and $M \mid M$. These are different in Orc, because we make two calls to M in $M \mid M$, and just one in M. Also, M may return two different results for the two calls made in $M \mid M$.

In Kleene algebra, $\mathbf{0}$ is both a right and a left zero. In Orc, it is only a left zero; that is, $f \gg \mathbf{0} = \mathbf{0}$ does not hold. Even though neither $f \gg \mathbf{0}$ nor $\mathbf{0}$ produces a value, evaluation of $f \gg \mathbf{0}$ may cause changes in the external world, but $\mathbf{0}$ has no such effect.

Another axiom of Kleene algebra which does not hold in Orc is the left distributivity of $\gg$ over $\mid$,

$$f \gg (g \mid h) \;=\; (f \gg g) \mid (f \gg h)$$

To see why, consider $M \gg (N \mid R)$. Here, M is called once and the value it returns is used in the evaluations of both N and R. In $(M \gg N \mid M \gg R)$, evaluations of $M \gg N$ and $M \gg R$ are treated independently, M being called once for each subexpression. The left distributivity axiom holds if f is a function; in this case, it has no impact on the external world, and it always returns the same value.

5.2 Laws about where expressions

The following identities for **where** expressions hold when both sides of an identity are well-formed.

(Distributivity over $\gg$)
$\{f \gg g \textbf{ where } x{:}\in h\} \;=\; \{f \textbf{ where } x{:}\in h\} \gg g$

(Distributivity over $|$)
$\{f \mid g \textbf{ where } x{:}\in h\} \;=\; \{f \textbf{ where } x{:}\in h\} \mid g$

(Distributivity over **where**)
$\{\{f \textbf{ where } x{:}\in g\} \textbf{ where } y{:}\in h\} \;=$
$\{\{f \textbf{ where } y{:}\in h\} \textbf{ where } x{:}\in g\}$

For the laws given above, both sides of an identity must be checked syntactically for well-formedness. Unlike the laws in section 5, both sides may not be well-formed if only one side is. Consider

$p = (M \mid N(x) \textbf{ where } x{:}\in g)$

According to (Distributivity over $|$), p is equal to q and r where

$q = (M \textbf{ where } x{:}\in g) \mid N(x)$
$r = (N(x) \textbf{ where } x{:}\in g) \mid M$

Expression r is well-formed though q is not, because x in the term $N(x)$ is not bound to any variable. So, $p \neq q$ though $p = r$.

6 Longer Examples

6.1 Workflow coordination

In this section, we consider a typical workflow application, where a number of activities have to be coordinated by having them occur in a designated sequence. The problem, which appears in Choi et. al.[7], is to arrange a visit of a speaker. An office assistant contacts the speaker, proposing a set of possible dates for the visit. The speaker responds by choosing one of the dates. The assistant then contacts *Hotel* and *Airline* sites. He sends the hotel and airline information to the speaker who sends an acknowledgment. Only after receiving the acknowledgment, the assistant confirms both the hotel and the airline reservations. The assistant then reserves a room for the lecture, announces the lecture (by posting

it at an appropriate web-site) and requests the audio-visual technician to check the equipment in the room prior to the lecture.

In our solution, we employ the following sites.

$GetDate(p, s)$: contact speaker p with a list of possible dates s; the response is a single date from s.

$Hotel(d)$: contact several hotels for a 2-night stay, leaving on date d. The response is the name of the chosen hotel, its location, price for the room and the confirmation number. This site implements the preferences of the speaker and the organization.

$Airline(d)$: similar to $Hotel$.

$Ack(p, t)$: same as $GetDate$ except tuple t is sent and only an acknowledgment is expected as a response.

$Confirm(t)$: confirm reservation t (for a hotel or airline).

$Room(d)$: reserve a room for one hour on date d. The response is the room number and the time of the day.

$Announce(p, q)$: announce the lecture with speaker information (from p), and room and time (from q).

$AV(q)$: contact the audio-visual technician with room and time (in q).

We have structured the solution as a sequence: (1) contact the speaker and acquire a date of visit, d, (2) make both hotel (h) and airline (a) reservations (3) acquire the acknowledgment from the speaker for h and a, (4) confirm the hotel and the airline, (5) reserve a room (q), and (6) announce the visit and contact the audio-visual technician. The value produced by evaluating the expression is of no significance.

$Visit(p, s) \;\underline{\Delta}$
$\quad GetDate(p, s) \quad >d>$
$\quad \{let(h, a) \textbf{ where } h{:}\in Hotel(d),\; a{:}\in Airline(d)\}$
$\quad\quad >(h, a)>$
$\quad Ack(p, (h, a)) \quad \gg$
$\quad \{let(x, y) \textbf{ where } x{:}\in Confirm(h),\; y{:}\in Confirm(a)\}$
$\quad\quad \gg$
$\quad Room(d) \quad >q>$
$\quad \{let(x, y) \textbf{ where } x{:}\in Announce(p, q),\; y{:}\in AV(q)\}$

The problem of arranging a visit is typically more elaborate than what has been shown: the speaker needs to be picked up at the airport and the hotel, lunches and dinners have to be arranged, and meetings with the appropriate individuals have to be scheduled. These additional tasks add no complexity, just bulk, to the solution. They would be coded as separate sites and orchestrated by the top-level solution. Also, we have not considered failure in this solution, which would be handled through time-outs and retries.

6.2 Orchestrating an auction

We consider an example of a typical web-based application, running an auction for an item. First, the item is advertised by calling site *Adv*, which posts its description and a minimum bid price at a web site. Bidders put their bids on specific channels, and we use the *Multiplexor* from page 30 to merge all the bids into a single channel, c.

We consider three variations on the auction strategy, $Auction_i(v)$, $1 \le i \le 3$. We start the auction by executing

```
z :∈ Auction_i(V)
```

where $1 \le i \le 3$ and V is the minimum acceptable bid.

A Non-terminating auction Our first solution continually takes the next bid from channel c which exceeds the current (highest) bid and posts it at a web site by calling *PostNext*.

Below, $nextBid(v)$ returns the next bid from c exceeding v. (the site call $if(x > v)$ returns a signal if $x > v$ and remains silent otherwise.)

```
nextBid(v) ≜
        c.get
  >x>
        { if(x > v) ≫ let(x)
        | if(x ≤ v) ≫ nextBid(v)
        }
```

Below, $Bids(v)$ returns a stream of bids from c where the first bid exceeds v and successive bids are strictly increasing.

```
Bids(v) ≜ nextBid(v) >y> (let(y) | Bids(y))
```

The following strategy starts the auction by advertising the item, and posts successively higher bids at a web site. But the expression evaluation never terminates.

```
Auction_1(v) ≜
   Adv(v) ≫ Bids(v) >y> PostNext(y) ≫ 0
```

A terminating auction We modify the previous program so that the auction terminates if no higher bid arrives for h time units (say, h is an hour). The winning bid is then posted by calling *PostFinal*, and the goal variable is assigned the value of the winning bid.

Expression $Tbids(v)$, where v is a bid, returns a stream of pairs $(x, flag)$, where x is a bid value, $x \ge v$, and *flag* is boolean. If *flag* is *true*, then x exceeds its previous bid, and if *false* then x equals its previous bid, i.e., no higher bid has been received in an hour.


```
Tbids(v) ≜
  let(x, flag) | if(flag) ≫ Tbids(x)
    where
      (x, flag) :∈
         nextBid(v) >y> let(y, true)
       | Rtimer(h) ≫ let(v, false)
```

The full auction is given by

```
Auction_2(v) ≜
      Adv(v)
  ≫  Tbids(v)
  >(x, flag)>
      { if(flag)   ≫ PostNext(x)  ≫ 0
       |if(¬flag)  ≫ PostFinal(x) ≫ let(x)
      }
```

Batch processing Our previous solution posts every higher bid as it appears in channel c. It is reasonable to post higher bids only once each hour. So, we collect the best bid over an hour and post it. If this bid does not exceed the previous posting, i.e., no better bid has arrived in an hour, we close the auction, post the winning bid and return its value as the result.

Analogous to $nextBid(v)$, we define $bestBid(t, v)$ where t is an absolute time and v is a bid. And $bestBid(t, v)$ returns x, $x \ge v$, where x is the best bid received up to t. If $x = v$ then no better bid than v has been received up to t.

The code for $bestBid(t, v)$ (see *BestRefine* of section 4.4) can be understood as follows. First call $nextBid(v)$. If it returns y before t then $y > v$, and $bestBid(t, y)$ is the desired result. If $nextBid(v)$ times out then return v.

```
bestBid(t, v) ≜
  if(b) ≫ bestBid(t, y) | if(¬b) ≫ let(v)
    where
      (y, b) :∈
         nextBid(v) >y> let(y, true)
       | Atimer(t) >y> let(y, false)
```

Analogous to $Tbids(v)$, we define $Hbids(v)$ to return a stream of pairs $(x, flag)$, where x is the best bid received so far and *flag* is *true* iff x is received in the last hour. Expression *Hbids* calls *bestBid* every hour until it receives no better bid. Below, the value of $flag$ is simply the boolean $x = v$.

```
Hbids(v) ≜
  clock >y> bestBid(y + h, v) >x>
  (let(x, x = v) | if(x ≠ v) ≫ Hbids(x))
```

The code of $Auction_3$ is identical to that of $Auction_2$ except that *Tbids* in the latter is replaced by *Hbids*.

$$\begin{array}{lllll}
\multicolumn{5}{l}{Auction_3(v)\ \ \underline{\Delta}} \\
 & \multicolumn{4}{l}{Adv(v)} \\
\gg & \multicolumn{4}{l}{Hbids(v)} \\
\multicolumn{5}{l}{>(x, flag)>} \\
 & \{\ if(flag) & \gg\ PostNext(x) & \gg\ \mathbf{0} & \\
 & \ \ |\, if(\neg flag) & \gg\ PostFinal(x) & \gg\ let(x) & \\
 & \} & & &
\end{array}$$

6.3 Arranging and Monitoring a meeting

We write a program to arrange and monitor a meeting at (absolute) time T among a group of professors. First, send a message to all professors requesting the meeting. If N responses are received within 10 time units, then proceed with the meeting arrangement, otherwise cancel the meeting and inform *all* professors (not just those who have responded). To proceed with the meeting arrangement, reserve a room for time T. If room reservation succeeds, announce the meeting time and room to all professors. If room reservation fails, cancel the meeting and inform all of them.

It is given that a room can be preempted (by the department chairman) until one hour (h units) before its scheduled time. No meeting is preempted more than once. If the room is preempted (before $T - h$), attempt to reserve another room. If it succeeds, inform all professors that the meeting has been moved to another room. If room reservation fails, inform all of them that the meeting is now cancelled. The value of the entire computation is a boolean, *false* if the meeting is cancelled, *true* otherwise. This value can be computed only at $T - h$ or shortly thereafter.

Messages The computation sends several kinds of messages to the professors, which we list below. A message includes certain parameters.

$msg_1(t)$: Please respond if you can attend a meeting at time t.

$msg_2(t)$: The meeting planned for time t is cancelled due to poor response.

$msg_3(t)$: The meeting planned for time t is cancelled because no room is available then.

$msg_4(t, r)$: A meeting is scheduled at time t in room r.

$msg5(t, r, s)$: The meeting scheduled at time t in room r moved to room s.

$msg_6(t, r)$: The meeting scheduled at time t is cancelled because it was preempted from room r and no room is available at t.

Site $Broadcast_i(p)$, where $1 \leq i \leq 6$ and p is a list of parameters, sends the i^{th} message with parameters p to all professors, and returns a signal.

Specifications of the main components Main expressions are $Arrange$, *Room* and $Monitor$. Their specifications are as follows.

$Arrange(t)$: Send message $msg_1(t)$ to the professors and count the number of responses received in 10 time units. If this number is at least N, return *true*, otherwise call $Broadcast_2(t)$ and return *false*.

$Room(t)$: Reserve a room, r, for time t by calling the site $RoomReserve(t)$. If this fails (i.e., $r = 0$), call $Broadcast_3(t)$. If room reservation succeeds ($r \neq 0$), call $Broadcast_4(t, r)$. In all cases return value r.

$Monitor(t, r)$: Site $RoomCancel(r).get$ returns a signal if room r has been preempted. In case of preemption before time t, attempt to reserve a room, s. If reservation succeeds ($s \neq 0$), call $Broadcast_5(t, r, s)$ and return *true*. If room reservation fails ($s = 0$), call $Broadcast_6(t, r)$ and return *false*.

The computation structure The overall structure of the computation is

$$z{:}\in\ MeetingMonitor(T)$$

$$\begin{array}{lll}
\multicolumn{3}{l}{MeetingMonitor(t)\ \ \underline{\Delta}} \\
 & \multicolumn{2}{l}{Arrange(t)} \\
>b> & \multicolumn{2}{l}{(\ if(\neg b) \gg let(false)} \\
 & \multicolumn{2}{l}{\ \ |\, if(b)\ \ \gg\ Room(t)\ \ >r>} \\
 & & (\ \ if(r = 0)\ \gg\ let(false) \\
 & & \ \ |\, if(r \neq 0)\ \gg\ Monitor(t - h, r) \\
 & &) \\
 &) &
\end{array}$$

Codes of the main components We give the Orc code of the main components, $Arrange$, *Room* and $Monitor$.

The code for $Arrange$ uses $tally$ from section 3.4, page 20. Message m in $tally$ is $msg_1(t)$, and *prof* is a list of sites, one site for each professor. Expression $Arrange$ sends a cancellation message if the number of responses, n, is below N. It returns the value of $n \geq N$ in all cases. The codes of $Room(t)$ and $Monitor(t, r)$ follow easily from their specifications.

$$\begin{array}{ll}
\multicolumn{2}{l}{Arrange(t)\ \ \underline{\Delta}} \\
 & tally(prof) \\
>n> & (\ \ if(n \geq N)\ \gg\ let(true) \\
 & \ \ \ |\, if(n < N)\ \gg\ Broadcast_2(t)\ \gg\ let(false) \\
 &)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{Room(t)\ \ \underline{\Delta}} \\
 & RoomReserve(t) \\
>r> & (\ \ if(r = 0) \gg Broadcast_3(t)
\end{array}$$

$$
\begin{array}{ll}
 & \quad |\, \mathit{if}\,(r \neq 0) \gg \mathit{Broadcast}_4(t,r) \\
 & \quad) \\
\gg & \quad \mathit{let}(r)
\end{array}
$$

$$
\begin{array}{l}
\mathit{Monitor}(t,r) \;\underline{\Delta} \\
\quad \mathit{Atimer}(t) \gg \mathit{let}(\mathit{true}) \\
|\; (\quad \mathit{RoomCancel}(r).\mathit{get} \\
\quad\quad \gg \mathit{RoomReserve}(t) \quad >s> \\
\quad\quad\quad (\;\; \mathit{if}\,(s \neq 0) \gg \\
\quad\quad\quad\quad \mathit{Broadcast}_5(t,r,s) \quad \gg \mathit{let}(\mathit{true}) \\
\quad\quad\quad |\, \mathit{if}\,(s = 0) \gg \\
\quad\quad\quad\quad \mathit{Broadcast}_6(t,r) \quad \gg \mathit{let}(\mathit{false}) \\
\quad\quad\quad) \\
\quad)
\end{array}
$$

7 Concluding Remarks

7.1 Programming Language Design

The notation proposed in this paper provides a minimal language to express interesting multi-threaded computations. It is not intended as a serious programming language yet, because many language-related issues, from lexical to hierarchical structuring, have been ignored. We consider some below.

A number of programming paradigms appear repeatedly in Orc programming. We have listed some of them as idioms in section 4. Some coding patterns are so frequent that special notation should be designed for them. We consider a few such issues below.

Adding Code and data to expressions The absence of any arithmetic facility in an Orc expression is a nuisance (though not a disaster) when writing actual programs. To add x and y within an Orc expression we have to call the site $add(x,y)$, where add implements the addition procedure. We have adopted the convention of writing $x+y$, which a preprocessor can translate to $add(x,y)$. A number of sequential programming features, including conditional statements and some form of iteration, should be allowed within Orc. Also, the programming language should allow most data type manipulations, including array indexing, within Orc expressions, which can then be converted to site calls. And programmers may find it more pleasing to use longer names for the cryptic symbols $\gg$ and $|$.

Nested site calls The current syntax requires that the parameters of site calls be variables. We do not allow expressions as parameters, because they produce streams of values, not just one. But $M(N(x), R(y))$, where M, N and R are sites, makes sense. It is $\{M(u,v) \textbf{ where } u :\in N(x),\; v :\in R(y)\}$. There is no technical difficulty in allowing nested site calls.

An expression like $M(N(x), N(x))$ poses semantic ambiguity. It is not clear if N should be called twice for the two arguments of M or just once, with the value being used for both arguments. These options can be coded, respectively, as

$$
\begin{array}{l}
\{M(u,v) \textbf{ where } u :\in N(x), v :\in N(x)\} \\
\{M(u,u) \textbf{ where } u :\in N(x)\}
\end{array}
$$

We have to study a large number of examples to decide which of these should be picked as the default semantics. The other semantics will have to be coded explicitly.

Fork-Join Parallelism It is common to call two sites, M and N, in parallel, name their values u and v, respectively, and continue computation only after both return their values. We would code this as

$$
\begin{array}{l}
\quad (\; \mathit{let}(u,v) \\
\quad\quad \textbf{where } u :\in M \\
\quad\quad\quad\quad v :\in N \\
\quad) \\
>(u,v)> \; \{\text{Remaining Computation}\}
\end{array}
$$

A convenient notational alternative is

$$
\begin{array}{l}
\quad (u \leftarrow M \;\; || \;\; v \leftarrow N) \\
\gg \{\text{Remaining Computation}\}
\end{array}
$$

Using this notation, the screen-refresh program of section 4.5 (page 24) looks much cleaner.

$$
\begin{array}{ll}
 & \mathit{Metronome} \\
\gg & (i \leftarrow \mathit{Image} \;\; || \;\; k \leftarrow \mathit{Keyboard} \;\; || \;\; m \leftarrow \mathit{Mouse}) \\
\gg & \mathit{Draw}(i,k,m)
\end{array}
$$

We can also remove a variable name which is never referenced. So

$$
\begin{array}{l}
\quad (M \;\; || \;\; v \leftarrow N) \\
\gg \{\text{Remaining Computation}\}
\end{array}
$$

is a shorthand for

$$
\begin{array}{l}
\quad (\; \mathit{let}(u,v) \gg \mathit{let}(v) \\
\quad\quad \textbf{where } u :\in M \\
\quad\quad\quad\quad v :\in N \\
\quad) \\
>v> \; \{\text{Remaining Computation}\}
\end{array}
$$

The workflow coordination example (section 6.1, page 35) now becomes much simpler.

$$
\begin{array}{ll}
\multicolumn{2}{l}{Visit(p,s) \;\underline{\Delta}} \\
& d \leftarrow\; GetDate(p,s) \\
\gg & \{h \leftarrow\; Hotel(d) \;\;||\;\; a \leftarrow\; Airline(d)\} \\
\gg & Ack(p,(h,a)) \\
\gg & \{Confirm(h) \;\;||\;\; Confirm(a)\} \\
\gg & q \leftarrow\; Room(d) \\
\gg & \{Announce(p,q) \;\;||\;\; AV(q)\}
\end{array}
$$

Hierarchical definitions The current definition of expressions treats all sites named in it as external sites. In many cases, an expression calls sites which are completely local to it, in that no other expression can (or should) call those sites. For example, consider the expressions

$$
\begin{array}{lll}
F & \underline{\Delta} & f \;>y>\; c.put(y) \gg \mathbf{0} \\
G & \underline{\Delta} & c.get \gg M \;>y>\; (let(y) \mid G) \\
E & \underline{\Delta} & F \mid G
\end{array}
$$

in which F is a producer that writes to channel c, G a consumer from c, and E the process network consisting of F and G. Here, channel c is local to E (so are the names F and G).

The following proposal allows structuring both expressions and sites into hierarchies. An expression definition consists of: (1) its name and formal parameters, (2) definitions of local sites (such as *c.put* and *c.get*, which are written in the host language, not Orc), (3) definitions of local expressions (such as F and G), (4) the body of the expression. When an expression is instantiated, its local sites are instantiated, and the local sites are terminated (garbage-collected) when the expression is terminated. Remote sites can still be called from an expression; a remote site name is either hard-coded as a constant or passed as a parameter to an expression.

Observe that having local expressions within an expression definition allows considerable information hiding.

7.2 Related work

This work draws upon a number of areas of computer science; we give a very brief outline of a few selected pieces of the relevant literature.

The work of the W3C group is of particular importance. The Semantic Web[13], a standard for the representation of data on the World Wide Web, is a collaborative effort led by W3C, which integrates a variety of applications using XML for syntax and URIs for naming. We expect that our model will be particularly suitable for processing metadata and making decisions based on their values.

Monadic computations in functional programming languages, like Haskell [15], have the same flavor as Orc computations. Monads allow a functional program to call external agents, like I/O devices, which behave as sites; see Elliott [9] for some particularly interesting applications. Concurrent Haskell [19] additionally

includes sophisticated constructs for thread creation and manipulation. Peyton Jones has pointed out the connection of those concurrency operators to the Orc constructs. Galen Menzel [22] has developed a compact implementation of Orc on Haskell based on his suggestions.

There is a huge amount of literature on the process network model of distributed computing. Our interest in this area derives from its formal semantics, and the possibility that Orc may be a viable alternative to some of these models. A recent work of considerable importance is Benton, Cardelli and Fournet[1]. It extends the $C\#$ programming language with new asynchronous concurrency abstractions based on the join calculus[11]. The language is applicable both to multi-threaded applications running on a single machine and to the orchestration of asynchronous, event-based applications communicating over a wide area network.

Process algebras, particularly CCS [23] and CSP [17], have much in common with the philosophy of Orc. All three represent a multi-threaded computation by an expression which has interesting algebraic properties. But unlike these process algebras, Orc permits integration of arbitrary components (sites) in a computation. This is both an advantage in that we can orchestrate heterogeneous components, and a disadvantage in that we are unable to decide equivalence of arbitrary Orc expressions, by using bisimulation, for example.

Orc differs in a major way from process algebras in its basic operators and the evaluation procedure. We permit arbitrary sequential compositions of expressions, $f \gg g$, which is not supported in CCS or CSP. Some recent work by Cook [8] suggests that Orc operators can be represented in a slightly extended version of Pi-calculus [24].

Transaction processing has a massive number of references; a comprehensive survey appears in Gray and Reuter[12]. Our approach has the flavor of nested transactions (see Chapter 4 of[12]) though there is considerable difference in semantics. Of special importance in our work are compensating transactions, and, particularly, their use in business process orchestration languages, like BPEL[2, 3]. Butler and Hoare[4] are developing a theoretical model and algebra for process interaction which includes compensating transactions.

Harel and his co-workers [14] have developed a very attractive visual notation, *Statecharts*, to encode computations of interacting processes. Their approach has met with considerable practical success. They have also developed a rigorous semantics of the visual notation.

Acknowledgment I am extremely grateful to C.A.R. Hoare for extensive discussions and many key insights. Galen Menzel has carried out implementations of Orc in Java and Haskell, and has contributed significantly to the programming model. William Cook has inspired me by exploring several deep issues in the formal semantics of Orc, and has shown me connections to other process algebras. Simon Peyton-Jones provided insight into the connection between Orc and monads that was essential to the Haskell implementation. Elaine Rich has helped me focus on the big issues by being skeptical at the appropriate moments. I am grateful to José Meseguer for extremely detailed comments and criticisms, and to Albert Benveniste for undertaking a substantive implementation effort which is partially inspired by Orc. Comments and suggestions from Luca Cardelli, Ankur Gupta, Gerard Huét, Amir Husain, Mathai Joseph, Todd Smith, Reino Kurki-Suonio and Greg Plaxton have enriched the paper.

References

[1] Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. TOPLAS **26**(5), 769 – 804 (2004)

[2] web site on BPEL, A.: Available for download at `http://www-106.ibm.com/developerworks/webservices/library/ws-autobp`

[3] web site on BPEL, A.: Available for download at `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`

[4] Butler, M., Hoare, C.: Personal communication

[5] Cardelli, L.: Transitions in programming models (microsoft research european faculty summit '03). http://research.microsoft.com/Users/luca/Slides/2003-07-16n

[6] Cardelli, L., Davies, R.: Service combinators for web computing. IEEE Transactions on Software Engineering **25**(3), 309–316 (1999)

[7] Choi, Y., Garg, A., Rai, S., Misra, J., Vin, H.: Orchestrating computations on the world-wide web. In: R.F. B. Monien (ed.) Parallel Processing: 8th International Euro-Par Conference, vol. LNCS 2400, pp. 1–20. Springer-Verlag Heidelberg (2002)

[8] Cook, W.: Semantics of Orc (2004). Under preparation

[9] Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 Haskell Workshop. Snowbird, Utah, USA (2004)

[10] Main page for World Wide Web Consortium (W3C) XML activity and information. `http://www.w3.org/XML/` (2001)

[11] Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join-calculus. In: Proceedings of the POPL. ACM (1996)

[12] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)

[13] Group, W.: Semantic web. http://www.w3.org/2001/sw/

[14] Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts. McGraw-Hill (1998)

[15] Haskell 98: A non-strict, purely functional language. Available at `http://haskell.org/onlinereport` (1999)

[16] Hoare, C.: Monitors: an operating system structuring concept. Communications of the ACM **17**(10), 549–557 (1974)

[17] Hoare, C.: Communicating Sequential Processes. Prentice Hall International (1984)

[18] Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. In: M. Broy (ed.) Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series. Marktoberdorf, Germany (2004). Also available at `http://www.cs.utexas.edu/users/psp/Semantics.Orc.pdf`

[19] Jones, S.P.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In: T. Hoare, M. Broy, R. Steinbruggen (eds.) Proc. of the NATO Advanced Study Institute, Engineering theories of software construction, pp. 47–96. IOS Press; ISBN: 1 58603 1724, 2001, Marktoberdorf, Germany (2000)

[20] Kozen, D.: On Kleene algebras and closed semirings. In: Proceedings, Math. Found. of Comput. Sci., *Lecture Notes in Computer Science*, vol. 452, pp. 26–47. Springer-Verlag (1990)

[21] Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. TOPLAS **4**(3), 382–401 (1982)

[22] Menzel, G.: Implementation of Orc on Concurrent Haskell (2004). Under preparation

[23] Milner, R.: Communication and Concurrency. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International (1989)

[24] Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)

[25] Osgood, I., Sheppard, D., Wright, C., Merritt, D., Geiger, B.: Eight queens in many programming languages. http://c2.com/cgi/wiki?EightQueensInManyProgrammingLanguages