Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Jayadev Misra and William R. Cook

Computation Orchestration

A Basis for Wide-Area Computing

Received: date / Accepted: date

Abstract The widespread deployment of networked applications and adoption of the internet has fostered an environment in which many distributed services are available. There is great demand to automate business processes and workflows among organizations and individuals. Solutions to such problems require *orchestration* of concurrent and distributed services in the face of arbitrary delays and failures of components and communication.

We propose a novel approach, called *Orc* for orchestration, that supports a structured model of concurrent and distributed programming. This model assumes that basic services, like sequential computation and data manipulation, are implemented by primitive *sites*. Orc provides constructs to orchestrate the concurrent invocation of sites to achieve a goal – while managing time-outs, priorities, and failure of sites or communication.

Keywords Wide-area Computing · Web Services · Computation Orchestration · Distributed Computing · Process Algebra · Thread-based Programming

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, such as sequential computation, data manipulation and communication. A web service 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 three composition operators: for parallel computation, sequencing and selective pruning. 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.

Works of the first and second author are partially supported by National Science Foundation grants CCR–0204323 and CCF-0448128, respectively.

Jayadev Misra and William R. Cook
The University of Texas at Austin
Austin, Texas 78712, USA
Tel.: 512-471-7316
Fax: 512-471-8885
E-mail: {misra,cook}@cs.utexas.edu

1.1 An Overview of the Orchestration Theory

1.1.1 Starting an orchestration

We propose a simple extension to a sequential programming language to invoke an 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*) and L is a list of actual parameters[1]. Evaluation of $E(L)$ may entail a wide-area computation involving, possibly, multiple servers. The evaluation outputs zero or more results, the first one of which (if there is one) is assigned to z, and further evaluation of E is terminated. If the evaluation yields no result, the statement execution does not terminate. The evaluation may initiate computations which have effects on other servers, and these effects may or may not be visible to the client.

Terminology A site *publishes* a value means that the site returns that value in response to a call. Similarly, an expression publishes a value means that its evaluation causes output of that value. A site/expression is *silent* if it never publishes.

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

1.1.2 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 publish 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. Evaluating $Email(a, m)$ causes 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 expression A returns the booking information and causes a state change in the airline database.

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[17] 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).

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.

1.1.3 Composition Operators

Orc has three composition operators to form expressions out of constituent subexpressions. Symmetric parallel composition of f and g, written as $f \mid g$, permits independent computations and publications (i.e., outputs) from f and g. The remaining two composition operators are inspired by the logical quantification operators: (1) for *all* x published by f do g, and (2) for *some* x published by f do g. We write the first expression as $f >x> g$ and the second as (g **where** $x :\in f$). We choose this syntax to better emphasize the algebraic properties of the operators (such as associativity of $>x>$) which are not evident in the quantified form.

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.

Evaluation of an expression calls some number of sites and publishes a (possibly empty) stream of values. Below, we give a brief summary of the composition operators using a series of examples; a detailed description appears in section 3.

- $(CNN \mid BBC)$ calls the two sites, CNN and BBC, simultaneously. Each site publishes at most one value. The output stream consists of the values published (i.e., returned) by the sites in time-order. Thus, there can be anywhere from zero to two values in the stream.

- Expression $(CNN >m> Email(a, m))$ first calls CNN. The value returned is named m, and $Email(a, m)$ is then called. The value returned by $Email(a, m)$ is the value published by 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 published in either case.
  Particularly interesting is an expression like

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

[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.

Here, $(CNN \mid BBC)$ may publish 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*.

– The operators $\gg$ and $\mid$ only initiate computations. The **where** operator prunes computations selectively. In $(Email(a, m)$ **where** $m :\in (CNN \mid BBC))$, the expression sends at most one email, with the first newspage received from either *CNN* or *BBC*. To evaluate this expression 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 has a value. The evaluation of $(CNN \mid BBC)$, as described above, may yield more than one value; 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.

Operator $>x>$ 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.

1.1.4 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.

$$MailOnce(a) \triangleq Email(a, m) \text{ where } m :\in (CNN \mid BBC)$$

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. Here, the value of x is not used.

An expression may call itself, as in

$$MailForever(a) \triangleq MailOnce(a) >x> MailForever(a)$$

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.

$$Ticker(a, t) \triangleq MailOnce(a) >x> Rtimer(t) >y> Ticker(a, t)$$

Site call $Rtimer(t)$ publishes 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 5 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.

1.2.1 Structure of the paper

The goal of this paper is to introduce the Orc programming model and illustrate its application in diverse areas of programming. 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 implementation-oriented semantics in section 3.2. A formal semantics is given in Section 4. Most programming is done by learning certain idioms. We develop a number of idioms in section 5, which show programming strategies for sequential computing, time-out, and communication and synchronization among computations. Section 6 contains a few laws, describing equivalences over Orc expressions. We have also developed a denotational semantics which offers alternative proofs of the algebraic laws [19].

We develop some longer examples in section 7. These are motivated by the intended application domain of Orc, web services orchestration.

2 Sites

2.1 Properties of sites

Each terminal element in an 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 or a variable. Variables may denote any value, including another site.

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 publish different values with the same input at different times. In particular, a site may publish 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 published by sites

A site is called with values of certain types and it publishes 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 published 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[12]; 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 published by a site can be assigned to a program variable.

We introduce a type, called *signal*, which has exactly one value. Its typical 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.

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 can not be rolled back. Any roll-back strategy is application dependent, say, by sending a cancellation message, which is interpreted by the recipient.

In this paper, we do not discuss *tentative* state changes, which can arise if a transaction is invoked as part of an Orc computation. Additional machinery is required to make tentative state changes permanent, a topic we will discuss in a forthcoming paper.

2.4 Some Fundamental Sites

We define a few sites in Table 1 that are fundamental to effective programming in Orc. The 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. Time is measured locally by the server on which the computation is performed. 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.

$$\begin{array}{lll} Atimer(t) & \equiv & Rtimer(t-c), \quad \text{provided } t \geq c \\ Rtimer(u) & \equiv & Atimer(u+c), \quad \text{provided } u \geq 0 \end{array}$$

3 Syntax and Informal Semantics

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

3.1 Syntax

The syntax of Orc appears in Table 2. Henceforth, we abbreviate $f \ >x>\ g$ to $f \gg g$ if the result published by x is not used in g.

Binding powers of the operators The operators in increasing order of precedence (binding power) are:
$\triangleq$, **where** , $:\in$, $\mid$, $>x>$.
Operator $>x>$ is right associative. So

$M \ >x>\ (N(x) \mid R) \ >x>\ S(x)$ is
$M \ >x>\ ((N(x) \mid R) \ >x>\ S(x))$.

Table 1 Fundamental Sites

$let(x, y, \cdots)$	publishes a tuple consisting of the values of its arguments.
$if(b)$	where b is boolean, publishes a signal if b is true, and it remains silent (i.e., does not respond) if b is false.
Signal	publishes a signal immediately. It is same as $if(true)$.
Clock	publishes the current time at the server of this site, as an integer.
$Atimer(t)$	publishes a signal at time t, where t is integer and $t \geq$ the value returned by *Clock*
$Rtimer(t)$	publishes a signal after exactly t time units, where t is integer and $t \geq 0$.

Table 2 Syntax of Orc

E	$\in$	Expression Name	
x, z	$\in$	Variable	
M	$\in$	Site $\subset$ Variable	
c	$\in$	Constant	
P	$\in$	$p_1, \ldots, p_n$	List of Actual Parameters
Q	$\in$	$q_1, \ldots, q_n$	List of Formal Parameters
Orc Statement	::=	$z :\in E(P)$	Evaluate $E(P)$ and assign to z
Expression Defn	::=	$E(Q) \triangleq f$	Define expression E
$f, g \in$ *Expression*	::=	$\mathbf{0}$	Zero expression
	$\parallel$	$M(P)$	Site call
	$\parallel$	$E(P)$	Expression call
	$\parallel$	$f \mid g$	Symmetric Parallel Composition
	$\parallel$	$f >x> g$	Sequential Composition
	$\parallel$	f **where** $x :\in g$	Asymmetric Parallel Composition
$p \in$ *Actual Parameter*	::=	x	Variable
	$\parallel$	c	Constant
$q \in$ *Formal Parameter*	::=	x	Variable

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.

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

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 $E(Q) \triangleq f$ is well-formed if the free variables of f is a subset of Q.

Values, Tuples and Sites Sites publish values. The value could be of any type; in particular, it could be a tuple of values. We overload the definition of *let* such that $let(3)$ and $let(3, 5)$ publish 3 and tuple $(3, 5)$, respectively. We also allow tuples of variable names where a variable x may appear; this binds each variable to the corresponding component of the tuple.

Sites are also values. A site may be used as a parameter to another site or published by a site as a value. Thus, in

$$Find >M> M(x) \gg N(M)$$

Find publishes a site M, which is called and then N is called with M as a parameter.

In typical programming languages, $sqrt(4)$ and 2 are interchangeable in all contexts. That is not so in Orc. Given that *sqrt* is a site, $sqrt(4)$ is an Orc expression, but 2 is a value. Orc expressions that publish 2 are $sqrt(4)$ and $let(2)$. Only constants and variables which have values may appear as actual parameters of site calls, not Orc expressions. And, only Orc expressions may be combined using the composition operators.

Notational conventions We write

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

also as

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

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

3.2 Informal semantics

In this section, we describe the semantics of Orc informally, though rigorously, in operational terms. This semantics provides an abstract execution model which corresponds closely to our prototype implementation of Orc,

and is useful as a model for programmers to understand the execution of Orc expressions.

Imagine that a single client machine evaluates the Orc expression. It sends messages to call (remote) sites. On receiving response v to a call, it either publishes v and/or calls other sites with v as a parameter. It may start several computations simultaneously. Time-based sites, *Clock*, *Atimer* and *Rtimer*, are implemented on the client, so that they respond with exact values at exact moments (as measured by the client's clock).

The actual evaluation of an Orc expression may involve a distributed network of computers. In fact, process networks are easily represented as Orc expressions.

3.2.1 Overview of Expression Evaluation

We describe the operational semantics of expression evaluation in terms of *threads*. A thread defines a portion of the computation during an evaluation. Threads are used in this section only, as a convenient way of explaining expression evaluation. The Orc language does not have a notion of threads, and programming in Orc does not entail reasoning about threads. Moreover, our implementation of Orc does not use operating system threads.

To evaluate an expression, we create and run a thread. A thread may call sites, publish values and assign values to certain variables. Also, it may spawn sub-threads. The set of threads form a tree where any sub-thread of a thread is its child. The root of the tree is the thread which evaluates the main expression.

Associated with each thread is a *context*, bindings between variables and their values which are to be used in running the thread. The initial context (at the root) binds the values of the global variables to their values. Variables may also be defined when a thread is created. For example, in evaluating $M >x> f$, the thread to evaluate f starts with a given value of v of x. We write such a thread as $f_{(x,v)}$. In evaluating (f **where** $x :\in g$), the thread that evaluates f starts with name x being defined, though x has no value. We write such a thread as $f_{(x,\perp)}$. Later, this context is modified when x is assigned a value. If no new variable is defined when a thread is created, its context is empty.

During evaluation of a well-formed expression, any reference to variable x in a thread implies that x is defined in the context of this or some ancestor thread; x may not yet have a value. If (x, u) is in the context of this thread and $u \neq \perp$, then the value of x is u, and if $u = \perp$, then x has no value. If x is not defined in the context of this thread, i.e., (x, u) is not in the context for any u, then repeat the procedure starting at the parent thread to determine the value of x. Henceforth, we say that x has value v in a thread to mean that (x, v) is in the closest context in which x is defined and $v \neq \perp$.

Below, we describe the semantics of an expression in terms of its structure. Each expression has an **eval** part and **publish** part. The former specifies the threads that are created to evaluate this expression. The context of a created thread includes the context of the thread from which it is created and any additional variable bindings as given below. The publish part specifies what the thread publishes as its values.

A value is published by an expression as soon as it is published by a component thread.

1. $M(x)$, where M is a site, $M \neq \mathbf{0}$:
   eval: if x has value v, run a thread which calls $M(v)$.
   publish: value, if any, returned by site M.

2. $E(x)$, where E is a defined expression:
   eval:
   substitute actual parameter names for formal parameters in the definition of E;
   run a sub-thread to evaluate this definition.
   publish: all publications from this sub-thread.

3. $(f \mid g)$:
   eval: run sub-threads for f and g.
   publish: interleave all publications from both sub-threads in time order.

4. $(f >x> g)$:
   eval:
   run a sub-thread for f;
   for each publication v of f, run $g_{(x,v)}$ as a sub-thread.
   publish: interleave all publications of all g-threads in time order.

5. (f **where** $x :\in g$):
   eval:
   run sub-threads $f_{(x,\perp)}$ and g;
   for the first publication v of g do:
   modify context of f from $(x, \perp)$ to (x, v);
   terminate g-thread and all its descendants;
   publish: all publications of f-thread.

The rule for site call ensures that the call is made only if the actual parameter has a value. Otherwise, the call is deferred until the parameter has a value. We have considered site calls with just one parameter; for more parameters, *all* actual parameters must have values. Observe that no action is taken for expression **0**; no site is called, nor is there any publication corresponding to this expression.

For an expression call, the actual parameter names are substituted for the formal ones, and then the expression is evaluated. Note that some of the actual parameters may not have values, and the site calls may have to be deferred.

The rule for $f \mid g$ is straight-forward. No new context is created for either thread in this case, and publication of either thread is a publication for the whole expression.

The rule for $f >x> g$ merely creates a g-thread for each publication of f, with the appropriate context. For

$f \gg g$, simply create a g-thread for each publication of f, without additional context.

The first value published by g in (f **where** $x :\in g$) is relevant; subsequent values are ignored and the g-thread and all its descendants are then terminated. Any response received subsequently from a site in response to an earlier call from g (or its descendants) is ignored. If the first value is v, the value of x becomes v, and this is recorded by modifying the context from $(x, \bot)$ to (x, v) in the f-thread.

Synchronous Semantics The proposed semantics has internal events (creations of threads, making site calls, etc.) which it can process, and external events (responses from sites) which are beyond its control. We require that *all internal events be processed as soon as possible*, though they may be processed in any order. Absence of this requirement may delay processing internal events arbitrarily, or process an external response while there are outstanding internal events. Under synchronous semantics, no response is processed if there is an outstanding internal event. And, sub-threads created in $f \mid g$, for instance, are run simultaneously.

Events involving fundamental sites *let*, *if* and *Signal* are treated as internal events. Consequently, these sites publish their values before any external event is processed. For example, in $M \mid Signal$, the signal from *Signal* is published before the response from M.

3.2.2 Site call

The simplest expression is a site name without parameters. To evaluate the expression, call the site and the value published 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 have values. The parameters of a site call and the value published by the site can be of any type (see section 2.2), including a site name which can be called later during the evaluation.

3.2.3 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 publishes at most one value whereas an expression may publish 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, an expression call is non-strict; evaluation of an expression begins *when it is called*, even if some of its actual parameters are undefined. Site calls are strict in that its actual parameters must be defined before the site is called. See Section 3.2.7 for elaboration.

3.2.4 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 published by N. Site N cannot reference the value published by M. Operator $\gg$ is associative.

Consider

$$Rtimer(1) \gg Email(address, message)$$

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

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

sends two emails in sequence and then calls *Notify*.

The examples we have shown so far each publish 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 publishes a set of values at specific times, and each value causes a fresh evaluation of g at that time; this instance of g runs in parallel with f and other instances of g. The values published by all instances of g are the publications of $f \gg g$.

3.2.5 Value passing

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

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

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 published by f. Each value is referenced in a different instance of g as x. For example, suppose f has published three values, 0, 1 and 2. There will be three instances of g in which x set to 0, 1 and 2, respectively, and evaluations of all three instances of g and of f may be concurrent.

3.2.6 Symmetric parallel composition

We introduce | to permit symmetric parallel computations. Evaluation of $(M \mid N)$ calls both M and N, and publishes the values published by M and N (in the same order in which M and N publish). Given that *CNN* and *BBC* are two sites that publish newspages, *CNN* | *BBC* may potentially publish two newspages. It may publish 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, starts evaluations of f and g, which may, in turn, start yet more evaluations. Each evaluation publishes a stream of values. The publications of $f \mid g$ is the merge of these two streams in time order. If both publish values simultaneously, their merge order is arbitrary. Operator | is commutative and associative.

Consider the expression $(M \mid N) >x> R$. The evaluation starts by calling M and N. Suppose M publishes a value first. This value is called x and R is called. If N publishes a value next, R is called again with a new value of x. That is, each value from $(M \mid N)$ starts a new instance of R.

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 where M publishes different values on each call, and N and R use those values. The two computations are depicted pictorially in figure 1.

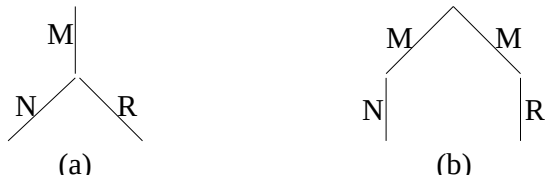

Fig. 1 (a) $M \gg (N \mid R)$ and (b) $M \gg N \mid M \gg R$

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*.

Alternative Semantics We discuss two alternative interpretations of $f \mid g$, each of which publishes a single value. One possibility is to let $f \mid g$ publish the first value published by either f or g and then terminate. This semantics incorporates arbitration, which we model directly in standard Orc in Section 5.3. Another possible interpretation of $f \mid g$ is that it publishes a tuple of first values, one from f and the other from g, and then terminates. That is, it implements fork-join parallelism in which both threads, f and g, have to publish a value for the computation to continue. We show how fork-join can be encoded in standard Orc in Section 5.5. The standard semantics is preferable because its formal definition is simpler, and we have been able to program a number of distributed programming paradigms more succinctly.

3.2.7 Asymmetric parallel composition

An expression with a **where** clause (henceforth, called a **where** expression), has the form $(f$ **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 publishes its first value, assign it to x and terminate further evaluation of g. Termination of g means: (1) any subsequent value received in response to an earlier site call is ignored, and (2) there are no further site calls or publications.

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 values published by f under this evaluation strategy are the publications of $(f$ **where** $x :\in g)$.

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

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

In section 3.2.6, expression

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

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

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

Expression calls are non-strict because the semantics of **where** demand it. In $(F(x)$ **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 a reference to x (where the value of x will be stored) to F.

Alternative Semantics We discuss two alternative interpretations of (g **where** $x :\in f$), and why they were rejected. In the first semantics, start evaluation of f; when f publishes its first value, name it x, start evaluation of g and terminate f. This computation can be expressed in standard Orc as ($let(x) \gg g$ **where** $x :\in f$). Additionally, if f never publishes, (g **where** $x :\in f$) and ($f >x> g$) have the same computations under the alternative semantics. We would like them to have different computations which is motivated by a formal analogy. These Orc operators are inspired by the quantification operators of predicate calculus: (1) ($f >x> g$) evaluates g for *all* x published by f, and (2) (g **where** $x :\in f$) evaluates g for *some* (the first) x published by f. Non-publication by f can be regarded as evaluating g over an empty domain, similar to formulae $(\forall x : x \in \{\} : p(x))$ and $(\exists x : x \in \{\} : p(x))$, where p is a predicate over variable x. These two predicate calculus formulae have different values.

A second possible interpretation of (g **where** $x :\in f$) is to evaluate f lazily, i.e., only when the value of x is needed. Then, in ($M \gg N(x)$ **where** $x :\in f$) if M does not respond, f is never evaluated. We can achieve the same effect by using $M \gg (N(x)$ **where** $x :\in f)$ in standard Orc. Translation of general expressions is more elaborate: call site *eval.set* to set a bit when x is needed and evaluation of f begins only when *eval.get* returns a signal denoting that the bit has been set.

3.2.8 Site **0**

Site **0** never publishes.

Use ($Email(address1, message) \gg \mathbf{0} \mid Notify$) to send an email and call *Notify* simultaneously. The first alternative never publishes a value. (A site like *Email* is called an *asynchronous* procedure in polyphonic *C#* [2]; no response is needed from it to proceed with the main computation.)

3.2.9 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 formal parameters, and an expression which serves as its body. As an example, consider the definition

$$Asynch(address, message, N) \triangleq Email(address, message) \gg \mathbf{0} \mid N$$

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

$$Asynch(a, m, Notify).$$

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

$$PmeetQ(u, v) \triangleq P(u) \gg Q(u) \mid P(v) \gg Q(v)$$

3.2.10 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.

$$Metronome \triangleq Signal \mid Rtimer(1) \gg Metronome$$

Parameters may appear in recursive calls in the usual fashion. Define a bounded metronome to generate n signals at unit intervals, starting immediately. Below, n is greater than 0 in the second definition.

$$BMetronome(0) \triangleq \mathbf{0}$$
$$BMetronome(n) \triangleq Signal \mid Rtimer(1) \gg BMetronome(n-1)$$

Site *Query* publishes a value (different ones at different times). Site $Accept(x)$ publishes x if x is acceptable, and is silent otherwise. It is required to publish all acceptable values by calling *Query* at unit intervals forever.

$$RepeatQuery \triangleq Metronome \gg Query >x> Accept(x)$$

Or, publish all acceptable values by calling *Query* at unit intervals n times.

$$RepeatQuery(n) \triangleq BMetronome(n) \gg Query >x> Accept(x)$$

Using only the basic composition operators, an expression can publish 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 through out this paper.

3.2.11 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 evaluation of E with actual parameter values substituted for the formal ones, assign the *first* value published to z, and then terminate the evaluation. If evaluation of E publishes no value, the execution of the statement does not terminate.

In many distributed programming applications, expression evaluation never publishes a value though it affects the external world through site calls. Several such examples appear in sections 5 and 7. In such a case, the Orc statement should be placed within a thread of the host language program with the expectation of nontermination.

3.3 Angelic vs. Demonic Non-determinism

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 published by M *and* N. And R may respond after, say, N has published its value, but fail to respond after M. One pleasing outcome of this evaluation strategy is that we have the identity (see section 6),

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

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 5.10 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[16], the **where** operator provides a convenient mechanism for program structuring and efficient evaluations 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,

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

is *not* equivalent to

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

In the first case, M is called at most once. In the second case, each value published by $(N \mid R)$ forces a fresh evaluation of M, thus possibly calling it twice. 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 when only some of the paths need to be explored.

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 7.

Multiple time-based computation Make four requests to site M, in intervals of one time unit each.

$$\begin{array}{l} M \\ \mid\; Rtimer(1) \gg M \\ \mid\; Rtimer(2) \gg M \\ \mid\; Rtimer(3) \gg M \end{array}$$

Time-out If site M publishes value v before t time units, publish v; if after t (or never), publish 0; if at t, publish either value.

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

Selective time-based computation Receive N's response as soon as possible, but no earlier than 1 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 receiving its response until a time unit has passed.

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

We can use this expression to give priority to M over N. Request M and N for values, but give priority to M by publishing its response if it arrives within the first time unit, even though after N's response.

$$let(x) \;\; \textbf{where}\; x :\in M \mid Delay(N)$$

Flow rate calculation Count the number of values published 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 publishes a signal, and *count.read* publishes the counter value. In this solution, the value published by *count.inc* is explicitly ignored, because we are interested in producing a single value after 10 time units.

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

In the given expression f continues to compute and call *c.inc* even after the expression publishes a value. Use

$$let(x) \;\; \mathbf{where}\; x :\in \\ f \gg count.inc \gg \mathbf{0} \;\mid\; Rtimer(10) \gg count.read$$

to terminate the computation of f after publication of a value.

We may compare the rates at which two sources (say, expressions f and g) are producing values and then choose one source over another when both are producing the same stream. 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 [16] notation for lists, denoting an empty list by $[\,]$, and a list with head x and tail xs by $(x : xs)$. Below, site call $add(u, v)$ publishes the sum of u and v.

$$\begin{array}{l} tally([\,]) \quad \triangleq \; let(0) \\ tally(x : xs) \; \triangleq \\ \quad (add(u, v) \\ \qquad \mathbf{where} \\ \qquad\quad u :\in x(m) \gg let(1) \mid Rtimer(10) \gg let(0) \\ \qquad\quad v :\in tally(xs)) \end{array}$$

4 Operational Semantics

We develop a formal semantics of Orc in this section. The semantics is operational, and it is based on labeled transition systems. First, in Section 4.1, we propose an asynchronous semantics in which processing of internal actions and external responses are interleaved in arbitrary order. Next, in Section 4.2, we obtain a synchronous semantics by restricting the asynchronous semantics as follows: external responses are processed only if there are no internal actions which can be processed.

4.1 Asynchronous Semantics

As is common in small-step operational semantics, the language must be extended to represent intermediate states. We introduce $?u$ to denote an instance of a site call that has not yet returned a value, where u is a unique handle that identifies the call instance.

$$\begin{array}{rl} f, g \in Expr ::= & \mathbf{0} \parallel M(p) \parallel E(p) \parallel f >x> g \\ & \parallel f \mid g \parallel g \;\mathbf{where}\; x :\in f \parallel ?u \end{array}$$

The variable x is bound in g for the expressions $f >x> g$ and $(g \;\mathbf{where}\; x :\in f)$. Free variables and substitution of c for variable x in e, written $[c/x]e$, are defined in the standard way. We restrict the language to sites and definitions with a single argument. In the future we will extend the formal semantics to include multiple arguments, tuples, and other data structures.

The transition relation $f \overset{l}{\hookrightarrow} f'$, defined in Figure 2, states that expression f transitions with event l to expression f'. There are four kinds of events:

$$l \in Event ::= M\langle c, u\rangle \parallel u?c \parallel \;!c\; \parallel \tau$$

A *site call* event, $M\langle c, u\rangle$, represents a call to site M with argument c and handle u, as explained below. A *response* event, $u?c$, contains a site call handle u and the result value c. A *publish* event, $!c$, specifies a result c from an expression. As is traditional, τ denotes an *internal* event.

While the semantics is for the most part straightforward, the handling of site calls and the difference between sequential composition and asymmetric parallel composition deserve discussion.

Site Calls Although the syntax of a site call resembles a synchronous function call, it is given an asynchronous interpretation by the semantics. In particular, a site call involves three steps: invocation of the site, response from the site, and publication of the result. These steps can be arbitrarily interleaved with other site calls, or delayed indefinitely. The three steps in a site call are defined by the SiteCall, SiteRet, and Let rules.

Rule SiteCall specifies that a site call $M(c)$, where c is a constant, transitions to $?u$ with event $M\langle c, u\rangle$. The label u connects a site call to a site return – a fresh label is created for each call to identify that call instance. The resulting expression, $?u$, represents an expression that is blocked waiting for the return from the call. If the site call is nested within an expression, then the event is propagated to the top of the transition derivation, where it is visible to the environment. A site call occurs only when its parameters are constants; in $M(x)$, where x is a variable, the call is blocked until x is defined.

In SiteRet a pending site call $?u$ receives a result c from the environment and transitions to a publish expression $let(c)$. There is no assumption that all site calls eventually return. If the environment never produces a response event, then the call blocks indefinitely.

$$\frac{u \text{ fresh}}{M(c) \overset{M\langle c,u\rangle}{\hookrightarrow} ?u} \quad (\textsc{SiteCall})$$

$$?u \overset{u?c}{\hookrightarrow} let(c) \quad (\textsc{SiteRet})$$

$$let(c) \overset{!c}{\hookrightarrow} \mathbf{0} \quad (\textsc{Let})$$

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g} \quad (\textsc{Sym1})$$

$$\frac{g \overset{l}{\hookrightarrow} g'}{f \mid g \overset{l}{\hookrightarrow} f \mid g'} \quad (\textsc{Sym2})$$

$$\frac{[\![E(q) \triangleq f]\!] \in D}{F(p) \overset{\tau}{\hookrightarrow} [p/q]f} \quad (\textsc{Def})$$

$$\frac{f \overset{l}{\hookrightarrow} f' \qquad l \neq !c}{f >x> g \overset{l}{\hookrightarrow} f' >x> g} \quad (\textsc{Seq1N})$$

$$\frac{f \overset{!c}{\hookrightarrow} f'}{f >x> g \overset{\tau}{\hookrightarrow} (f' >x> g) \mid [c/x]g} \quad (\textsc{Seq1V})$$

$$\frac{f \overset{l}{\hookrightarrow} f' \qquad l \neq !c}{g \textbf{ where } x :\in f \overset{l}{\hookrightarrow} g \textbf{ where } x :\in f'} \quad (\textsc{Asym1N})$$

$$\frac{f \overset{!c}{\hookrightarrow} f'}{g \textbf{ where } x :\in f \overset{\tau}{\hookrightarrow} [c/x]g} \quad (\textsc{Asym1V})$$

$$\frac{g \overset{l}{\hookrightarrow} g'}{g \textbf{ where } x :\in f \overset{l}{\hookrightarrow} g' \textbf{ where } x :\in f} \quad (\textsc{Asym2})$$

Fig. 2 Asynchronous Operational Semantics of Orc

The Let rule generates a publish event $!c$. If a variable is to be published, as in $let(x)$, the expression blocks until x is defined.

Composition Rules Evaluation of sequential composition depends on whether or not the left side publishes a value. If the left expression publishes $!c$, Seq1V creates a new instance of the right side, $[c/x]g$, which is run in parallel with the main expression. If the left expression does not publish a value, then sequential composition uses the rule Seq1N. Sequential composition only publishes values from the right hand side; any values generated by the left side are hidden. No transitions are allowed on the right hand side until it is instantiated.

All of these expressions, the left-hand side and all the instances of the right-hand side, are executed in parallel. Because the semantics is asynchronous, there is no guarantee that the values published by the first instance will precede the values of later instances. Instead, the values produced by all instances of g are interleaved arbitrarily.

Asymmetric parallel composition uses rules Asym1N and Asym2 to allow transitions on the left and right, but only if the right expression does not publish a value. When the right side publishes a value $!c$, Asym1V terminates the right expression and the c is bound into the left expression. One subtlety of these rules is that the left expression may contain both active and blocked subexpressions – any subexpression that uses x is blocked until the right side publishes a value.

Sym1 and Sym2 are the standard rules for parallel composition. Expressions are evaluated using call-by-name in the Def rule. This ensures the non-strict call semantics required for expression calls. We assume a single global set of definitions D.

The traditional classification of rules into *introduction* and *elimination* forms is useful in understanding the distinction between Orc and its environment. The three main events which are introduced (appear in the conclusions of the rules) are: SiteCall introduces $M\langle c, u\rangle$, SiteRet introduces $u?c$, and Let introduces $!c$. The rules Seq1V and Asym1V eliminate $!c$. Unlike most process calculi, some events do not have corresponding elimination rules. For example, there are no *elimination* rules for site calls $M\langle c, u\rangle$ or site returns $u?c$. This is because these events are only handled (eliminated) by the environment.

4.2 Synchronous Semantics

As is typical of most process algebras, the asynchronous semantics of Orc given in Section 4.1 allows arbitrary delays in processing events. It does not specify *when* particular events take place, nor any specific order in processing the events. For instance, in evaluating $M \mid Rtimer(1)$, the two sites, M and $Rtimer$, may be called at vastly different times. Consequently, all we can assert about the call to $Rtimer$ is that the client will receive a signal *sometime* after unit delay. It is impossible to program time-out or any other time-based computation based on such weak guarantee.

We develop a *synchronous* semantics in this section whose essence is: process internal events, i.e., all events other than external response, as soon as possible. Therefore, initially, all sites which can be called will be called, and the client becomes *quiescent* waiting for an external response. Subsequently, on receiving an external response, Rule SiteRet (from Figure 2) is applied, which

$$
\begin{array}{lll}
\text{Asynchronous} & \hookrightarrow : \mathit{Expr} \times \mathit{Event} \times \mathit{Expr} & \{\text{Defined in Figure 2}\} \\
\text{Response} & \hookrightarrow_{\mathrm{R}} : \mathit{QExpr} \times \mathit{Response} \times \mathit{Expr} & = \{(q, r, e) \mid q \overset{r}{\hookrightarrow} e\} \\
\text{Action} & \hookrightarrow_{\mathrm{A}} : \mathit{Expr} \times \mathit{Action} \times \mathit{Expr} & = \{(f, a, f') \mid f \overset{a}{\hookrightarrow} f'\} \\
\text{Synchronous} & \hookrightarrow_{\mathrm{S}} : \mathit{Expr} \times \mathit{Event} \times \mathit{Expr} & = \hookrightarrow_{\mathrm{R}} \cup \hookrightarrow_{\mathrm{A}}
\end{array}
$$

Fig. 3 Synchronous Semantics

may make some internal events ready for processing. These internal events have to be processed before any other external response, until the client becomes *quiescent* again. Therefore, the evaluation proceeds in *rounds*, where each round consists of processing internal events, and each round, except the very first, is initiated by an external response.

Consider the problem of publishing value 0 followed by 1. Expression $let(0) \gg let(1)$ is incorrect (it publishes only 1) and $let(0) \mid let(1)$ gives no guarantee on the order of publication. However, $let(0) \mid Rtimer(0) \gg let(1)$ guarantees that $let(0)$ will be processed in the first round and $let(1)$ in a subsequent round (after zero time delay); therefore, the order of publication is guaranteed.

We treat *let*, which is a fundamental site in Table 1, as an internal event by defining a transition rule, Rule (Let), for it. We can treat other sites, such as *if* and *Signal*, also as internal events. The rule for *if* is

$$\mathit{if}(\mathit{true}) \overset{!\mathit{signal}}{\hookrightarrow} \mathbf{0} \qquad \text{(If)}$$

which says that $\mathit{if}(\mathit{true})$ only publishes a signal; since there is no rule for $\mathit{if}(\mathit{false})$, it simply blocks. Site *Signal* in Table 1 is merely $\mathit{if}(\mathit{true})$; so it always publishes a signal. The sites *let*, *if* and *Signal* can immediately return a result (or decide, in the case of $\mathit{if}(\mathit{false})$, that it will never return a value). Therefore, it is possible to treat them as internal events. In contrast, calls to *Rtimer* are treated as external events because the response is, in general, not immediate.

Formal Description of the synchronous semantics For a formal description of the synchronous semantics, we start with the asynchronous semantics of Figure 2. We partition the set of events into *actions*, which are internal events, and *responses*, which are external. Actions are initiated by an Orc expression, while responses are initiated by the environment.

$$
\begin{array}{ll}
a \in \mathit{Action} & ::= \tau \parallel\ !c \parallel M\langle c, u\rangle \\
r \in \mathit{Response} & ::= u?c
\end{array}
$$

A *quiescent expression*, q, is an expression that cannot perform an action. It is defined by

$$
\begin{array}{ll}
q \in \mathit{QExpr} & ::= \mathbf{0} \parallel M(x) \parallel q >x> e \parallel q \mid q \\
& \quad \parallel q \ \mathbf{where}\ x :\in q \parallel ?u
\end{array}
$$

Observe that no action can be applied to a quiescent expression; so its evaluation has to wait for a response from the environment. A site call involving a variable, $M(x)$, is quiescent because it is blocked until the variable becomes defined. A site call with a constant argument, $M(c)$ is not quiescent.

In the asynchronous semantics of Figure 2, relation $\hookrightarrow$ maps an expression and event to an expression. For synchronous semantics, we partition $\hookrightarrow$ into two subrelations: $\hookrightarrow_{\mathrm{R}}$ for responses, and $\hookrightarrow_{\mathrm{A}}$ for actions, as shown formally in Figure 3. Now, $\hookrightarrow_{\mathrm{R}}$ maps a quiescent expression and a response to an expression, not necessarily quiescent. And $\hookrightarrow_{\mathrm{A}}$ maps an expression and an action to an expression. Observe that an action has no effect on a quiescent expression. Therefore, $\hookrightarrow_{\mathrm{A}}$ does not contain any triples (q, a, e) where q is quiescent, a is an action, and e is an expression. The synchronous evaluation relation is the union of $\hookrightarrow_{\mathrm{R}}$ and $\hookrightarrow_{\mathrm{A}}$.

An execution is a sequence of events. A new round is started initially and after each response event. A round may be infinite (does not terminate) because there may be an unending sequence of actions to perform. Then the expression never becomes quiescent and accepts no further responses from the environment.

Although the synchronous semantics constrains the order of operations during evaluation, it does not provide a formal model of absolute or relative *time*, which is necessary to accurately model the behavior of *Rtimer*. Developing a temporal semantics of Orc is left for future work.

5 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 publishes the 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 5).

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,

$$(f \textbf{ where } (\forall i : 0 \leq i \leq 2 : x_i :\in g_i))$$

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.

5.1 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 publishes the (single) value of e. This encoding also supports reassignments of variables.

Conditional execution A typical if-then-else statement,

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

is coded in Orc as

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

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

$$\mathit{absolute}(x) \triangleq \mathit{if}(x \geq 0) \gg \mathit{let}(x) \mid \mathit{if}(x < 0) \gg \mathit{let}(-x)$$

Iteration A typical loop in an imperative program has the form

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

where x may be a list of variables. The iteration condition B and $S(x)$ in the assignment statement may both depend on x. The purpose of the loop is to iterate until $B(x)$ becomes false and then publish the value of x. We simulate this code fragment in Orc by the following expression where B and S are written as Orc expressions which return at most one value each.

$$\mathit{loop}(x) \triangleq B(x) >b> (\mathit{if}(b) \gg S(x) >y> \mathit{loop}(y) \mid \mathit{if}(\neg b) \gg \mathit{let}(x))$$

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

$x := x_0$;
while b **do** $x := S(x)$ **od**;
return $T(x)$

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

5.2 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 publishes the set of values

$$x,\ M(x),\ M(x) >y> M(y),\ M(x) >y> M(y) >z> M(z),\ \ldots$$

Our definition of this expression is

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

Note that the values are published in the proper order because publications by *let* are treated as internal events, which take precedence over publications from M.

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

$$M(x),\ M(x) >y> M(y),\ M(x) >y> M(y) >z> M(z),\ \ldots$$

Define

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

More general expressions which take M as parameter are,

$$\mathit{Star}(M, x) \triangleq \mathit{let}(x) \mid M(x) >y> \mathit{Star}(M, y)$$
$$\mathit{Plus}(M, x) \triangleq M(x) >y> (\mathit{let}(y) \mid \mathit{Plus}(M, y))$$

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 published value for a convergence criterion, and stops the computation once a *convergent* value is found (i.e., one that meets the convergence criterion).

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

$$\mathit{RefineStream}(x) \triangleq \mathit{Star}(\mathit{Refine}, x)$$
$$\mathit{RefineConverge}(x) \triangleq \mathit{RefineStream}(x) >y> \mathit{Converge}?(y)$$

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

5.3 Arbitration

A fundamental problem in concurrent computing is *arbitration*: to choose between two computations 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. Below, *flag* records which of *Alpha* and *Beta* responds first.

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

This expression is not quite identical to $\alpha.P + \beta.Q$ in its effect because *both Alpha* and *Beta* may change their states even though only one of the publications is used in further computations. We can overcome this problem by encoding *Alpha* and *Beta* such that they first respond without changing their states, and then a second call elicits the value to be actually used.

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

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

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 $\mathit{Rtimer}(1)$. 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.

5.4 Time-out

Expression $(\mathit{let}(z)$ **where** $z :\in f \mid \mathit{Rtimer}(t) \gg \mathit{let}(3))$ either publishes the first publication of f, or times out after t units and publishes 3. A typical programming paradigm is to call site M and publish a pair (x, b) as the value, where b is true if M publishes x before the time-out, and false if there is a time-out. In the latter case, x is irrelevant. Below, z is the pair (x, b).

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

As a more involved example, call *Refine* repeatedly starting with some initial argument, and use a publication as the argument for the next call. Publish the last value (the most refined) that is received before time t. Below, $\mathit{BestRefine}(t, x)$ implements this specification. It publishes x if there is a time-out; else it publishes $\mathit{BestRefine}(t, y)$, where y is the value published by *Refine* before the time-out.

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

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

$$\begin{array}{l} \mathit{BestRefineRel}(h, x) \ \triangleq \\ \quad \mathit{Clock} \ >y> \ \mathit{BestRefine}(y + h, x) \end{array}$$

5.5 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 publishes their values as a tuple after they both complete their executions.

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

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 3.2.10, page 9) to generate a signal at every unit, then start computations to acquire the image, keyboard inputs and the mouse position, and on completion of all three computations, call *Draw*. We code this as

$$\begin{array}{lll} & \mathit{Metronome} & \\ \gg & (\mathit{Draw}(i, k, m) & \\ & \quad \textbf{where} \ i :\in \mathit{Image} & \\ & \quad\quad\quad\quad\ \ k :\in \mathit{Keyboard} & \\ & \quad\quad\quad\quad\ \ m :\in \mathit{Mouse} & \\ &) & \end{array}$$

We implicitly assume that i, k and m are evaluated faster than the refresh rate of one time unit.

5.6 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. Con-

sider $M \gg f$ 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.

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

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

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

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 straight-forward 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), use $Refine(x)$ to publish matrix y, the temperature at the next moment. Site $Next$ computes the temperature at a point p from the previous temperatures of p and its neighbors. Typically, it would publish the average temperature of 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 publishes the previous temperature.

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

As in section 5.2, we can get a convergent value by using *RefineConverge*. Using this strategy, the heat transfer computation is run by

$$z :\in RefineConverge(I)$$

where I is the initial temperature matrix.

5.7 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 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* publishes 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

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

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 interrupt sites, and publishing specific codes for each kind of interrupt.

Often we wish to determine if there has been an interrupt. Then we publish 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:

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

An easy generalization is to interrupt a stream. Below, expression $callM$ calls M repeatedly until it is interrupted. It publishes a stream of tuples: $(x, true)$ for value x received from M and $(-, false)$ for interrupt. It does not publish after receiving an interrupt.

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

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.

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

The value published by $callM$ is a tuple whose first component is either a value published by M or a signal. Orc is a dynamically typed language which supports this form of type discrimination; the value of the second component determines the type of the first component.

5.8 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 non-strict evaluation of $x \vee y$ publishes *true* if either variable value is *true*; therefore, the 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 publish booleans. Compute the *parallel-or* of the two booleans, i.e., (in a non-strict fashion) publish *true* as soon as either site publishes *true* and *false* only if both sites publish *false*. In the following solution, site $or(x,y)$ publishes $x \vee y$. And $ift(b)$ publishes *true* if b is true and remains silent otherwise; $ift(b) = if(b) \gg let(true)$.

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

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

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

We can use the strategy of parallel-or to eagerly 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 publishes 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 publishes the minimum of its arguments.

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

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

$$(A \ \mid \ B) \ >y> \ threshold(y)$$

To obtain at most one such quote, we write

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

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

$$\begin{array}{l} (threshold(x) \ \mid \ threshold(y) \ \mid \ Min(x,y) \\ \quad \textbf{where} \\ \quad\quad x :\in A \\ \quad\quad y :\in B) \end{array}$$

5.9 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.

5.9.1 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) can also be implemented as 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 publishes a signal. Calling *c.get* publishes the value at the head of c and removes it from c if the channel is non-empty; if the channel is empty, *c.get* suspends the caller until the channel becomes non-empty.

5.9.2 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.

5.9.3 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.

$$P(c,e) \;\triangleq\; c.get \;>x>\; Compute(x) \;>y>\; e.put(y) \;\gg\; P(c,e)$$

This process publishes no value, though it writes on channel e. To publish every value which is also written on e, define

$$Q(c,e) \;\triangleq\; c.get \;>x>\; Compute(x) \;>y>\; (let(y) \mid e.put(y) \gg Q(c,e))$$

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

$$N \;\triangleq\; P(c,e) \mid P(d,e)$$

We may regard N as a network of two processes, $P(c,e)$ and $P(d,e)$.

The following small example illustrates a dialog with a user. The process reads from channel c into which the user writes a positive integer, checks if the integer is prime and writes the result to channel d. It repeats these steps as long as input is provided to it.

$$\begin{array}{l} Dialog \;\triangleq \\ \quad c.get \quad >x> \\ \quad Prime?(x) \;>b> \\ \quad d.put(b) \quad \gg \\ \quad Dialog \end{array}$$

5.9.4 Process Network

A process network is a parallel composition of processes. There is no logical difference between a process and a network. For example, N is defined to be $P(c,e) \mid P(d,e)$ above, and it may be regarded as a network which includes 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 publishes all the items read into channel e. That is, the process creates a fair merge of the values in the input channels. The definition is a generalization of N, shown above, for multiple input channels, without the *Compute* step.

$$\begin{array}{l} Multiplexor_i \;\triangleq \\ \quad c_i.get \;>y>\; e.put(y) \;\gg\; Multiplexor_i \\ Multiplexor \;\triangleq \\ \quad (\mid i :: Multiplexor_i) \end{array}$$

5.9.5 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.

Process Q_i writes a site name 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 site name M from e and calls M to permit the associated process to use the resource. After the process finishes using the resource, site M publishes a signal. Expression *Mutex* orchestrates mutual exclusion.

$$\begin{array}{ll} Arbiter & \triangleq\; e.get \;>M>\; M \;\gg\; Arbiter \\ Mutex & \triangleq\; Multiplexor \mid Arbiter \end{array}$$

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 releases the resource eventually, i.e., the corresponding site publishes eventually. The solution is easily modified to snatch the resource from an (unyielding) process after a time-out.

5.9.6 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 [23] 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

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

Evaluation of $Send_i(v)$ appends v to all outgoing channels of i. The responses from $c_{ij}.put(v)$ are ignored (by using $\gg \mathbf{0}$). There is no publication from $Send_i(v)$.

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.

$Read_i \ \triangleq \ let(X)$ **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 consists of evaluating $Send_i$ and $Read_i$ in parallel, and then evaluating $Compute_i$. 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.

$$
\begin{array}{ll}
Round_i(v, 0) & \triangleq \ let(v) \\
Round_i(v, n) & \triangleq \\
\quad (Send_i(v) \ | \ Read_i) & >X> \\
\quad Compute_i(X) & >u> \\
\quad Round_i(u, n-1) &
\end{array}
$$

Observe that process i can not begin $Compute_i$ in a round until all processes have completed their previous round, because the $Read_i$ waits until it receives inputs from all processes.

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.

$$Byz(V, n) \ \triangleq \ (\ |\ i :: Round_i(V_i, n))$$

5.9.7 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 in Hoare [18]. 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 $(i+1)$ mod N). Every pair of neighbors share a fork. The fork to the left of P_i is $Fork_i$ and to its 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.

Philosopher i's life is depicted by expression P_i. Below, Eat publishes a signal on completion of eating.

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

Represent the ensemble of philosophers by

$$DP \ \triangleq \ (\ |\ i : 0 \le i < N : P_i)$$

5.9.8 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.

$$
\begin{array}{l}
P_0 \ \triangleq \\
\quad Fork_1.get \gg Fork_0.get \gg Eat \gg \\
\quad Fork_0.put \gg Fork_1.put \gg P_0
\end{array}
$$

$$
\begin{array}{l}
P_i,\ 1 \le i < N, \ \triangleq \\
\quad Fork_i.get \gg Fork_{i'}.get \gg Eat \gg \\
\quad Fork_i.put \gg Fork_{i'}.put \gg P_i
\end{array}
$$

This example illustrates that the evaluation of an Orc expression may lead to deadlock when it spawns computations which wait for each other. Since such computations communicate only through sites, deadlock can be avoided if each site call is guaranteed to publish 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 avoided.

5.10 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 initiate multiple computations which may be implemented by backtracking. Among the problems which can be coded in this style 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" [29].

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. Configuration $(x : xs)$ is an *extension* of xs. A configuration is *valid* if none of the queens in it can capture any other. Site $check(x : xs)$, where $(x : xs)$ is a non-empty configuration and xs is valid, publishes $(x : xs)$ 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, $1 \leq n$ and $|x| + n \leq 8$, publishes *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 design $extend(x, 1)$ to publish all valid extensions of x by one-position. And, $extend(x, n)$ is the n-fold application of $extend(x, 1)$.

$$extend(x, 1) \;\triangleq\; (\forall i : 0 \leq i < 8 : check(i : x))$$
$$extend(x, n) \;\triangleq\; extend(x, 1) \;>y>\; extend(y, n-1)$$

6 Laws about Orc Expressions

We list a number of laws about Orc expressions. Many of these laws are also valid for regular expressions of language theory, which is a Kleene algebra [22]. Some Orc expressions can be regarded as regular expressions. An Orc term, a site or expression call, is a symbol of the alphabet. Constant **0** corresponds to the empty set. Operators $|$ and $\gg$ mimic alternation and concatenation. There is no unit symbol in Orc, but *Signal* acts as a left unit of $\gg$ and $let(x)$ as a right unit of $>x>$. 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.

Below, $\gg$ is associative, but $>x>$, by definition, is right associative. It is fully associative if both sides in the following identity are well-formed, i.e., h does not reference x.

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

All Orc expressions including **where** expressions obey the laws given in this section. They can be proved using bisimulation on the formal semantics of Orc in Section 4.

6.1 Kleene laws

Below f, g and h are Orc expressions.

(Zero and $\mid$)	$f \mid \mathbf{0} \;=\; f$
(Commut. of $\mid$)	$f \mid g \;=\; g \mid f$
(Assoc. 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 $\gg$)	$Signal \gg f \;=\; f$
(Right unit of $\gg$)	$f >x> let(x) \;=\; f$
(Assoc. of $\gg$)	$(f \gg g) \gg h \;=\; f \gg (g \gg h)$
(Distributivity)	$(f \mid g) \gg h \;=\; (f \gg h \mid g \gg h)$

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 publish two different results for the two calls made in $M \mid M$.

In Kleene algebra, **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 **0** publishes a value, evaluation of $f \gg \mathbf{0}$ may call sites and cause state changes, but **0** has no such effect.

Another axiom of Kleene algebra, the left distributivity of $\gg$ over $\mid$,

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

does not hold in Orc. To see why, consider $M \gg (N \mid R)$. Here, M is called once; if it responds, both N and R are called, and if it does not respond, neither is called. 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. Therefore, it is possible that N is called though R is never called. The left distributivity law holds if f is a function; in this case, f does not change any state, and it always publishes the same value.

6.2 Laws about **where** expressions

The following identities have no counterpart in Kleene algebra. Below, expression f is x-free means that f has no free occurrence of x, i.e., $x \notin free(f)$ (see Section 3.1 for the definition of *free*).

$(f \gg g \;\mathbf{where}\; x :\in h) \;=\; (f \;\mathbf{where}\; x :\in h) \gg g$, if g is x-free

$(f \mid g \;\mathbf{where}\; x :\in h) \;=\; (f \;\mathbf{where}\; x :\in h) \mid g$, if g is x-free

$((f \;\mathbf{where}\; x :\in g) \;\mathbf{where}\; y :\in h) =$
$((f \;\mathbf{where}\; y :\in h) \;\mathbf{where}\; x :\in g)$,
if g is y-free and h x-free

$(f \;\mathbf{where}\; x :\in M) \;=\; f \mid M \gg \mathbf{0}$, *if* f *is* x*-free*

7 Longer Examples

7.1 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 appro-

priate 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, (4) reserve a room (q), and (5) announce the visit and contact the audio-visual technician. The value published by the expression is of no significance.

$$
\begin{array}{l}
\mathit{Visit}(p,s) \;\triangleq \\
\quad \mathit{GetDate}(p,s) \quad >d> \\
\quad (\mathit{let}(h,a) \textbf{ where } h :\in \mathit{Hotel}(d),\ a :\in \mathit{Airline}(d)) \\
\qquad\qquad >(h,a)> \\
\quad \mathit{Ack}(p,(h,a)) \quad \gg \\
\quad (\mathit{let}(x,y) \textbf{ where } x :\in \mathit{Confirm}(h),\ y :\in \mathit{Confirm}(a)) \\
\qquad\qquad \gg \\
\quad \mathit{Room}(d) \qquad >q> \\
\quad (\mathit{let}(x,y) \textbf{ where } x :\in \mathit{Announce}(p,q),\ y :\in \mathit{AV}(q))
\end{array}
$$

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.

7.2 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 Section 5.9.4 to merge all bids into a single channel, c.

We consider three variations on the auction strategy, $\mathit{Auction}_i(v)$, $1 \le i \le 3$. We start the auction by calling

$$\mathit{Auction}_i(V)$$

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

7.2.1 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, $\mathit{nextBid}(v)$ publishes the next bid from c exceeding v. (the site call $\mathit{if}(x > v)$ publishes a signal if $x > v$ and remains silent otherwise.)

$$
\begin{array}{l}
\mathit{nextBid}(v) \;\triangleq \\
\qquad c.\mathit{get} \\
>x> \\
\qquad (\ \mathit{if}(x > v) \gg \mathit{let}(x) \\
\qquad\ \mid \mathit{if}(x \le v) \gg \mathit{nextBid}(v) \\
\qquad)
\end{array}
$$

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

$$\mathit{Bids}(v) \;\triangleq\; \mathit{nextBid}(v) >y> (\mathit{let}(y) \mid \mathit{Bids}(y))$$

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.

$$
\begin{array}{l}
\mathit{Auction}_1(v) \;\triangleq \\
\quad \mathit{Adv}(v) \gg \mathit{Bids}(v) \;>y>\; \mathit{PostNext}(y) \gg \mathbf{0}
\end{array}
$$

7.2.2 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 value of the winning bid is published.

Expression $\mathit{Tbids}(v)$, where v is a bid, publishes 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, and no higher bid has been received in an hour.

$$
\begin{array}{l}
\mathit{Tbids}(v) \;\triangleq \\
\quad \mathit{let}(x, \mathit{flag}) \mid \mathit{if}(\mathit{flag}) \gg \mathit{Tbids}(x) \\
\qquad \textbf{where} \\
\qquad\quad (x, \mathit{flag}) :\in \\
\qquad\qquad \mathit{nextBid}(v) \;>y>\; \mathit{let}(y, \mathit{true}) \\
\qquad\qquad \mid \mathit{Rtimer}(h) \gg \mathit{let}(v, \mathit{false})
\end{array}
$$

The full auction is given by

$$
\begin{array}{l}
Auction_2(v) \ \underline{\Delta} \\
\quad Adv(v) \\
\gg \ Tbids(v) \\
>(x, flag)> \\
\quad (\ if(flag) \quad \gg PostNext(x) \ \gg \mathbf{0} \\
\quad \ |if(\neg flag) \quad \gg PostFinal(x) \ \gg let(x) \\
\quad)
\end{array}
$$

7.2.3 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 publish its value as the result.

Analogous to $nextBid(v)$, define $bestBid(t, v)$ where t is an absolute time and v is a bid. And $bestBid(t, v)$ publishes x, $x \geq v$, 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.

Expression $bestBid(t, v)$ (see *BestRefine* of section 5.4) can be understood as follows. First call $nextBid(v)$. If it publishes y before t then $y > v$, and $bestBid(t, y)$ is the desired result. If $nextBid(v)$ times out then publish v.

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

Analogous to $Tbids(v)$, we define $Hbids(v)$ to publish 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 $x = v$.

$$
\begin{array}{l}
Hbids(v) \ \underline{\Delta} \\
\quad Clock \ >y> \ bestBid(y + h, v) \ >x> \\
\quad (let(x, x \neq v) \mid if(x \neq v) \gg Hbids(x))
\end{array}
$$

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

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

7.3 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.

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 that the meeting has been moved to another room. If room reservation fails, inform all 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.

7.3.1 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 has been 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 publishes a signal.

7.3.2 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, publish *true*, otherwise call $Broadcast_2(t)$ and publish *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 publish value r.

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

7.3.3 The computation structure

The main computation orchestrates the evaluations of $Arrange(t)$, $Room(t)$ and $Monitor(t, r)$ by evaluating the expression

$let(z)$ **where** $z :\in MeetingMonitor(T)$

Note that the computation is terminated after the first publication of *MeetingMonitor*.

```
MeetingMonitor(t) Δ
        Arrange(t)
   >b> ( if(¬b) ≫ let(false)
       | if(b)  ≫ Room(t)  >r>
                  ( if(r = 0) ≫ let(false)
                  | if(r ≠ 0) ≫ Monitor(t, r)
                  )
       )
```

7.3.4 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 11. 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 publishes the value of $n \geq N$ in all cases. The codes of $Room(t)$ and $Monitor(t, r)$ follow easily from their specifications.

```
Arrange(t) Δ
        tally(prof)
   >n>  ( if(n ≥ N) ≫ let(true)
        | if(n < N) ≫ Broadcast2(t) ≫ let(false)
        )

Room(t) Δ
        RoomReserve(t)
   >r>  ( if(r = 0) ≫ Broadcast3(t)
        | if(r ≠ 0) ≫ Broadcast4(t, r)
        )
   ≫    let(r)
```

```
Monitor(t, r) Δ
     Atimer(t − h) ≫ let(true)
   | (   RoomCancel(r).get
       ≫ RoomReserve(t)   >s>
         ( if(s ≠ 0) ≫
             Broadcast5(t, r, s)  ≫ let(true)
         | if(s = 0) ≫
             Broadcast6(t, r)     ≫ let(false)
         )
     )
```

8 Concluding Remarks

8.1 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 5. Some coding patterns are so frequent that special notation should be designed for them. We consider a few such issues below.

8.1.1 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 $|$.

8.1.2 Nested site calls

The current syntax requires that the parameters of site calls be variables. We do not allow expressions as parameters, because they publish streams of values, not just one. But $M(N(x), R(y))$, where M, N and R are sites, makes sense. It is ($M(u, v)$ **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

$(M(u,v)$ **where** $u :\in N(x), v :\in N(x))$
$(M(u,u)$ **where** $u :\in N(x))$

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.

8.1.3 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 publish their values. We would code this as

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

A convenient notational alternative is

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

Using this notation, the screen-refresh program of section 5.5 (page 15) looks much cleaner.

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

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

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

is a shorthand for

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

The workflow coordination example (section 7.1, page 20) now becomes much simpler.

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

8.1.4 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

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

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.

8.2 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.

Process calculi, including CSP [18], CCS [26] and π-calculus [27,33], provide fundamental models of concurrency in which processes communicate over channels. Orc has much in common with the philosophy of process algebras. They all represent a multi-threaded computation by an expression which has useful algebraic properties. But unlike these process algebras, Orc permits integration of arbitrary components (sites) in a computation. This introduces a distinction between the Orc expression and the environment in which it runs. We do not assume that the environment is modeled in Orc. Process calculi have evolved to be more and more focused on channels as fundamental communication mechanisms. Orc takes a different approach, in that it describes the structure of a distributed computation using primitives that define common communication patterns. We believe that Orc may be a viable alternative to process calculi.

Although process calculi are not intended to be programming languages, practical programming languages have been designed or influenced by process calculi. The

language Pict [31] is based on π-calculus [27]. A recent work of considerable importance is Benton, Cardelli and Fournet[2]. It extends the $C\#$ programming language with new asynchronous concurrency abstractions based on the join calculus[14]. 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. Channels are used in concurrent ML [32] and concurrent Haskell [21].

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 [8] suggests that Orc operators can be represented in a slightly extended version of Pi-calculus [27]. Galen Menzel [25] has developed a compact implementation of Orc in Concurrent Haskell that uses the same approach.

Simon Peyton-Jones has pointed out a connection between Orc constructs and the List monad, as used in functional programming languages, including Haskell [16]. The list monad is often used to express non-deterministic computations: the list represents a set of possible results to a computation, and the subsequent steps of the program may create new possible results or eliminate results that are no longer valid. The sequential composition operator, $>x>$, is analogous to the bind operator $>>=$ in Haskell. The **where** operator can also be modeled as taking the first item from a lazy list. The standard list monad always produces values in a specific order, while the publication order in Orc is non-deterministic.

Orc has some similarity with synchronous languages [4,3]. Sites calls and returns are similar to output and input signals. Harel and his co-workers [15] 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.

Orc shares many of the goals of business process orchestration languages, like BPEL[20]. Both BPEL and Orc make an explicit distinction between a process and the services it orchestrates. The `invoke` tag in BEPL is similar to calling a site in Orc. The `response` tag is similar to binding a variable to the name of a site call. Both languages provide mechanism for parallel execution and sequencing. BPEL defines a graph structure to specify how results are combined, rather than using composition of expressions. Since BPEL is based on XML it is easily read by machines. BPEL also does not have a formal semantics, although it has does have a detailed informal specification.

One important area where Orc may be applied is in the development of workflow systems. There is no commonly-accepted theory of workflow; instead there are models based on communication and speech act theories [13,24], extensions of Petri nets [10,11], and UML [9]. A π-Calculus model of an electronic marketplace is developed in Padget and Bradford [30]. More recently Aalst has developed a catalog of 19 workflow patterns [1]. This report shows that commercial workflow products have difficulty expressing many of the patterns. The examples of given in this paper cover many of the workflow patterns. A full evaluation of the ability of Orc, or other process calculi, to model workflow patterns is a subject for future work.

A preliminary version of the current paper was presented at Marktoberdorf in 2004 [28].

Acknowledgements We are 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. Simon Peyton-Jones provided insight into the connection between Orc and monads that was essential to the Haskell implementation. We are grateful to Natarajan Shankar for pointing out the similarity between Orc operators and the quantification operators of predicate calculus. Comments and suggestions from Luca Cardelli, Ankur Gupta, Gerard Huét, Amir Husain, Mathai Joseph, Greg Lavender, Jose Meseguer, Elaine Rich, Todd Smith, Reino Kurki-Suonio and Greg Plaxton have enriched the paper.

References

1. Aalst, W.M.P.V.D., Hofstede, A.H.M.T., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
2. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. TOPLAS **26**(5), 769 – 804 (2004)
3. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**(1), 64–83 (2003)
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19**(2), 87–152 (1992)
5. Cardelli, L.: Transitions in programming models (microsoft research european faculty summit '03). `http://research.microsoft.com/Users/luca/Slides/2003-07-16%20Transitions%20in%20Programming%20Models%20(MSR%20Faculty%20Summit).pdf`
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., Misra, J.: A structured orchestration language. Available for download at `http://www.cs.utexas.edu/users/wcook/projects/orc` (2005)
9. Eriksson, H.E., Penker, M.: Business Modeling with UML: Business Patterns at Work. John Wiley (2000)
10. Eshuis, R., Dehnert, J.: Reactive Petri nets for workflow modeling. In: W.M.P. van der Aalst, E. Best (eds.) Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), *Lecture Notes in Computer Science*, vol. 2679, pp. 296–315. Springer-Verlag (2003)
11. Eshuis, R., Wieringa, R.: Comparing Petri net and activity diagram variants for workflow modelling - a quest for reactive Petri nets. In: H. Ehrig, W. Reisig, G. Rozenberg, H. Weber (eds.) Petri Net Technology for Communication-Based Systems, *Lecture Notes in Computer Science*, vol. 2472, pp. 321–351. Springer-Verlag (2003)
12. Main page for World Wide Web Consortium (W3C) XML activity and information. `http://www.w3.org/XML/` (2001)
13. Flores, F.F., Winograd, T.: Understanding Computers and Cognition: A New Foundation for Design. Intellect Books (1986)
14. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join-calculus. In: Proceedings of the POPL. ACM (1996)
15. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts. McGraw-Hill (1998)
16. Haskell 98: A non-strict, purely functional language. Available at `http://haskell.org/onlinereport` (1999)
17. Hoare, C.: Monitors: an operating system structuring concept. Communications of the ACM **17**(10), 549–557 (1974)
18. Hoare, C.: Communicating Sequential Processes. Prentice Hall International (1984)
19. 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`
20. IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems: Business Process Execution Language for Web Services version 1.1. Available for download at `http://www-128.ibm.com/developerworks/library/specification/ws-bpel`
21. 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)
22. 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)
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. TOPLAS **4**(3), 382–401 (1982)
24. McCarthy, J.: Elephant 2000: A programming language based on speech acts. `http://www-formal.stanford.edu/jmc/elephant/elephant.html`
25. Menzel, G.: Implementation of Orc on Concurrent Haskell (2004). Under preparation
26. Milner, R.: Communication and Concurrency. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International (1989)
27. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
28. Misra, J.: Computation orchestration: A basis for wide-area computing. In: M. Broy (ed.) Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series. Marktoberdorf, Germany (2004)
29. Osgood, I., Sheppard, D., Wright, C., Merritt, D., Geiger, B.: Eight queens in many programming languages. `http://c2.com/cgi/wiki?EightQueensInManyProgrammingLanguages`
30. Padget, J.A., Bradford, R.J.: A π-calculus model of a spanish fish market - preliminary report. In: AMET '98: Selected Papers from the First International Workshop on Agent Mediated Electronic Trading on Agent Mediated Electronic Commerce, pp. 166–188. Springer-Verlag, London, UK (1999)
31. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: G. Plotkin, C. Stirling, M. Tofte (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (2000)
32. Reppy, J.H.: CML: A higher-order concurrent language. SIGPLAN Notices **26**(6), 293–305 (1991)
33. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)