Orchestration in Orc: A Deterministic Distributed Programming Model

William R. Cook and Jayadev Misra

Department of Computer Sciences, University of Texas at Austin
{wcook, misra}@cs.utexas.edu

Abstract. *Orc* is a new model of distributed programming which provides a strong theoretical foundation for internet computing based on compositions of web-services. Orc combines some of the power and flexibility of process algebra with the simplicity and determinism of synchronous programming models. We present an operational semantics of Orc and prove some laws analogous to those of Kleene algebra. We validate the deterministic operational semantics by proving it equivalent to a deterministic form of trace semantics.

1 Introduction

We propose a model for internet computing, called *Orc*. Orc assumes that there are basic services which provide computation and communication capabilities (i.e., data management, arithmetic and logical computation, and, even, channels for communication); we call such services *sites*. Orc includes only the machinery to call the sites in appropriate order, i.e., specify their *orchestration*, to carry out an internet computation.

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. Internet computing often requires a client to: invoke alternate sites for the same computation to guard against failure, poll a site until it supplies results which meet certain desired criteria, ask a site to notify the client when it acquires the appropriate data, or download an application and invoke it locally.

Consider a typical internet 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. He receives 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.

Orc has just three composition operators. Yet, it allows us to combine sites of arbitrary complexity in a computation, without making any assumptions about

their behavior. Orc includes no explicit constructs for time-out, thread synchronization or communication, features which are common in thread-based languages. These are easily programmed in Orc, as we show in our examples.

Contributions of the Paper Distributed programs are traditionally structured as networks of processes communicating over channels. The computation is nondeterministic because the channels have arbitrary delays and the process computations may be nondeterministic. The complexity of interaction among processes and channels does not readily facilitate practical programming and can complicate algebraic reasoning.

In Orc, we distinguish the client and its environment. The client initiates the computation by calling some sites. The responses from the sites may arrive in arbitrary order, but for each response the computation of the client is deterministic. Therefore, an Orc computation is a sequence of *rounds*, where in each round (except the very first one) some response is processed, and, possibly, site calls are made and values produced in a deterministic fashion. Orc is compact (see the syntax in Section 2.2) and for many real-life distributed applications, such as workflow coordination and managing an auction, the programs are concise and easily understood. The underlying theoretical model, the main subject of this paper, is also compact and permits derivations of several algebraic laws.

2 Overview of the Computation Model

In this section, we give a brief overview of the computation model using a series of examples. We give the formal syntax and an informal semantics sufficient for understanding the examples.

Orc can be added to a sequential programming language to by introducing a 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 a list of actual parameters. Evaluation of $E(L)$ may entail a wide-area computation and return zero or more results, the first one of which (if there is one) is assigned to z. If the evaluation yields no result, the statement execution does not terminate.

2.1 Site

The simplest Orc expression is a *site* call. A site is a separately defined procedure, like a web service. The site may be implemented on the client's machine or a remote machine. A site call elicits at most one response; it is possible that a site never responds to a call. Absence of response is treated as any other non-terminating computation. We will show how to alleviate this problem using time-outs.

Calling site $CNN(d)$, where CNN is a news service and d is a date, may download the newspage for the specified date. Calling $Email(a, m)$ sends message m to address a, causing permanent change in the state of the recipient's mailbox, and returns a signal to the client to denote completion of the operation. Calling an airline flight-booking site returns the booking information and causes a state change in the airline database.

Site calls are *strict*, i.e., a site is called only of all its parameters have values.

Some Fundamental Sites We define a few sites in Table 1 that are fundamental to effective programming in Orc.

$let(x, y, \cdots)$	publishes (i.e., returns) a tuple consisting of the values of its arguments.
$Rtimer(t)$	where t is integer and $t \geq 0$, returns a signal after exactly t time units.
$Signal$	returns a signal immediately. It is same as $Rtimer(0)$.
$if(b)$	where b is boolean, returns a signal if b is true, and it remains silent (i.e., does not respond) if b is false.

Table 1. Fundamental Sites

2.2 Syntax of Orc

In the following syntax, S ranges over site names, x over variables, F over expression names, and c over constants.

$$
\begin{aligned}
P \in Prog &::= \overline{D}\ e \\
D \in Decl &::= F(x) \ \underline{\Delta}\ \ e \\
e, f, g \in Expr &::= \mathbf{0} \parallel S(p) \parallel F(p) \parallel\ !p \parallel f > x > g \parallel f \mid g \parallel g \ \mathbf{where}\ x :\in f \\
p \in Param &::= x \parallel c
\end{aligned}
$$

A program ($Prog$) is a list of declarations ($Decl$). Declaration $F(x) \ \underline{\Delta}\ \ e$ defines expression F whose formal parameter is x and body is expression e. An expression ($Expr$) is either elementary or is a composition of two expressions. An elementary expression is either: (1) $\mathbf{0}$, which is treated as a site which is never called, (2) a site call $S(p)$, (3) an expression call $F(p)$, or (4) a site call $!p$ that publishes the value of p; it is equivalent to $let(p)$, see Table 1. Orc has three composition operators: (1) $>x>$ for sequential composition, (2) $\mid$ for symmetric parallel composition, and (3) **where** for asymmetric parallel composition. The operators in increasing order of precedence (binding power) are: $\underline{\Delta}$, $:\in$, **where** , $\mid$, $>x>$. Operator $>x>$ is right associative.

Evaluation of an expression calls some number of sites and publishes a (possibly empty) stream of values, as we explain next.

Sequential Composition To evaluate ($CNN > m > Email(a, m)$), first call CNN. The value it publishes (i.e., returns) is named m, and $Email(a, m)$ is then called. The value returned by $Email(a, m)$ is the value of the expression. If either site fails to respond, then the evaluation returns no value.

We write $M \gg \cdots$, without a parameter name in $\gg$, if the value published by M is of no significance.

Symmetric Parallel Composition To evaluate $(CNN \mid BBC)$, call the two sites simultaneously. The output stream consists of the values returned by *both* sites in time-order. Thus, there can be anywhere from zero to two values in the stream. Particularly interesting is an expression like

$$(\mathit{CNN} \mid \mathit{BBC}) \;\; >m> \;\; \mathit{Email}(a, m)$$

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

Asymmetric Parallel Composition Operators $>x>$ and $\mid$ only create threads. We introduce the **where** operator to prune threads selectively. For example, $(\mathit{Email}(a, m)$ **where** $m :\in (\mathit{CNN} \mid \mathit{BBC}))$ sends at most one email, with the first value received from either CNN or BBC. For this expression, we proceed by evaluating both $\mathit{Email}(a, m)$ and $(\mathit{CNN} \mid \mathit{BBC})$ simultaneously. Assume that initially a has a value, but m does not. Because of strictness in site call, evaluation of $\mathit{Email}(a, m)$ is suspended until m gets a value. Evaluation of $(\mathit{CNN} \mid \mathit{BBC})$, as described under Symmetric Parallel Composition, may yield up to two values; the first value is assigned to m and further evaluation of that expression is then terminated. At this point, $\mathit{Email}(a, m)$ is called and its response, if any, is the value of the whole expression.

Notes and Conventions Expressions $!p$ and $\mathit{let}(p)$ are equivalent; we use the former in describing the semantics and the latter in the examples. In this paper, we do not formally treat expressions of the form $!(p, q)$, which is $\mathit{let}(p, q)$, though the semantics can be easily extended. The expression $((f$ **where** $x :\in g)$ **where** $y :\in h)$ is also written as $(f$ **where** $x :\in g,\ y :\in h)$, or by writing the assignments in separate lines without the comma. We do not specify the types of parameters; parameters can be of any type which can be assigned to variables of the host language. In our examples, we use integer, site, string and list as parameter type.

2.3 Expression Definition

An expression is defined like a procedure, with a name and possible parameters. Below, $\mathit{MailOnce}(a)$ emails the first newspage from CNN or BBC to address a.

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

A more interesting expression emails a newspage to a, receives a confirmation from Email, waits for t time units, and then repeats these steps forever.

$$\mathit{Ticker}(a, t) \;\underline{\Delta}\; \mathit{MailOnce}(a) \gg \mathit{Rtimer}(t) \gg \mathit{Ticker}(a, t)$$

Recall from Table 1 that $\mathit{Rtimer}(t)$ returns a signal after t time units (the signal value itself is of no significance, only the time delay is).

The following expression emits a signal every time unit, starting immediately.

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

Suppose site *Query* returns a value (different ones at different times) and $Accept(x)$ returns x if x is acceptable, and remains silent otherwise. Publish all acceptable values by calling *Query* at unit intervals forever.

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

2.4 Small Examples

Simple Time-out Suppose site M returns a positive integer. Assign to z the value from M if it is received before t time units, else set z to 0.

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

Priority 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.

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

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

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

Recursive definition with time-out Call a list of sites and tally the number of responses received in 10 time units. Below, $tally(L)$ implements this specification where L is a list of sites and m is a (fixed) argument for each site call. We denote an empty list by $[\,]$, and a list with head x and tail xs by $(x : xs)$.

$$\begin{array}{lll} tally([\,]) & \triangleq & let(0) \\ tally(x : xs) & \triangleq & add(u, v) \qquad \text{— } add(u,v) \text{ returns the sum of } u \text{ and } v \\ & & \quad \textbf{where} \\ & & \qquad u :\in x(m) \gg let(1) \mid Rtimer(10) \gg let(0) \\ & & \qquad v :\in tally(xs) \end{array}$$

Kleene Star In the theory of regular expressions, M^* denotes the set of strings formed by concatenating zero or more M symbols. Analogously, define $Mstar(x)$

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

which returns the stream of results by successively calling M starting with x

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

We may use $Mstar(x)$ to compute successive approximations starting with x.

Arbitration A fundamental problem in concurrent computing is *arbitration*: to choose between two threads and let only one proceed. In CCS [8], $\alpha.P+\beta.Q$ is a process which behaves as process P if action α happens and as Q if β happens. In Orc terms, α and β correspond to sites *Alpha* and *Beta* and P and Q are expressions. We wish to evaluate P or Q depending on which of *Alpha* and *Beta* responds first. (This is similar, though not identical, to the CCS expression.) Below, boolean variable *flag* encodes which of *Alpha* and *Beta* responds first.

$$if(flag) \gg P \mid if(\neg flag) \gg Q$$
$$\textbf{where } flag :\in Alpha \gg let(true) \mid Beta \gg let(false)$$

Fork-join Parallelism In *fork-join* parallelism, we spawn two independent threads at a point in the computation, and resume the computation after both threads complete. There is no special construct for fork-join in Orc, but it is easy to code such computations. Below, we call sites M and N in parallel and return their values as a tuple after they both complete their executions.

$$let(u,v) \textbf{ where } u :\in M,\ v :\in N$$

Synchronization Synchronization of threads is fundamental in concurrent computing. Consider two threads $M \gg f$ and $N \gg g$; we wish to execute them independently, but synchronize f and g by starting them only after *both* M and N have completed; see the left solution in Figure 1. The solution in the right of Figure 1 passes on the values returned by M and N to f and g.

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

Fig. 1. Synchronization of Threads

Interrupt There is no mechanism in Orc to interrupt an expression evaluation. 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.

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

To allow interruption of f, use

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

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

To determine if there has been an interrupt, in order to initiate roll back, return a tuple whose first component is the value from f (if any) and the second component is a boolean to indicate whether there has been an interrupt:

$$let(z,b) \textbf{ where } (z,b) :\in f >y> let(y,true) \mid Interrupt.get >y> let(y,false)$$

Non-strict Evaluation; Parallel-or Suppose sites M and N return booleans. Compute the *parallel-or* of the two booleans, i.e., (in a non-strict fashion) return *true* as soon as either site returns *true* and *false* only if both sites return *false*. In the left solution in Figure 2, site $or(x, y)$ returns $x \vee y$. This solution may return up to three different values depending on how many of x and y are *true*. To return just one value, use the right solution in Figure 2.

$if(x) \mid if(y) \mid or(x, y)$ **where** $x :\in M$ $y :\in N$	$let(z)$ **where** $z :\in if(x) \mid if(y) \mid or(x, y)$ $x :\in M$ $y :\in N$

Fig. 2. Parallel-Or

Communicating Processes 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.

We introduce channels for communication among processes. Each channel has to be implemented as a site. We assume in our examples that channels are FIFO and unbounded, though other kinds of channels (including rendezvous-based communications) are easily implemented through sites.

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

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 to output channel e.

$$P(c, e) \;\underline{\Delta}\; c.get \;>x>\; Compute(x) \;>y>\; e.put(y) \gg P(c, e)$$

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

$$Q(c, e) \;\underline{\Delta}\; c.get \;>x>\; Compute(x) \;>y>\; (let(y) \mid e.put(y) \gg Q(c, e))$$

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

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

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

$$Dialog \;\underline{\Delta}\; tty.get \;>x>\; Prime?(x) \;>b>\; c.put(b) \gg Dialog$$

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

$$!c \overset{!c}{\hookrightarrow} \mathbf{0} \quad (\textsc{Pub}) \qquad \frac{[\![F(x) = e]\!] \in D}{F(p) \overset{\tau}{\hookrightarrow} [p/x]e} \quad (\textsc{Def})$$

$$\frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g} \quad (\textsc{Sym1}) \qquad \frac{g \overset{l}{\hookrightarrow} g'}{f \mid g \overset{l}{\hookrightarrow} f \mid g'} \quad (\textsc{Sym2})$$

$$\frac{f \overset{!c}{\hookrightarrow} f'}{f >x> g \overset{\tau}{\hookrightarrow} (f' >x> g) \mid [c/x]g} \quad (\textsc{Seq1V}) \qquad \frac{f \overset{l}{\hookrightarrow} f' \quad l \neq !c}{f >x> g \overset{l}{\hookrightarrow} f' >x> g} \quad (\textsc{Seq1N})$$

$$\frac{f \overset{!c}{\hookrightarrow} f'}{g \textbf{ where } x :\in f \overset{\tau}{\hookrightarrow} [c/x]g} \quad (\textsc{Asym1V}) \qquad \frac{f \overset{l}{\hookrightarrow} f' \quad l \neq !c}{g \textbf{ where } x :\in f \overset{l}{\hookrightarrow} g \textbf{ where } x :\in f'} \quad (\textsc{Asym1N})$$

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

Fig. 3. Asynchronous Operational Semantics of Orc

3 Asynchronous Operational Semantics

We first develop an asynchronous operational semantics, which allows transitions to be performed in any order. Asynchronous evaluation allows arbitrary interleaving of internal actions and external responses. The asynchronous semantics is refined in later sections to define a deterministic semantics.

In the syntax of Orc, defined in Section 2.2, x is bound in g for the expressions $f >x> g$ and g **where** $x :\in f$. Free variables and substitution of a constant c for variable x in e, written $[c/x]e$, are defined in the standard way. 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. We extend the definition of *Expr* to include $?u$:

$$e, f, g \in \mathit{Expr} ::= \mathbf{0} \parallel S(p) \parallel F(p) \parallel !p \parallel f >x> g \parallel f \mid g \parallel g \textbf{ where } x :\in f \parallel ?u$$

The transition relation $e \overset{l}{\hookrightarrow} e'$, defined in Figure 3, states that expression e transitions with event l to expression e'. There are four kinds of events:

$$l \in \mathit{Event} ::= S\langle c, u\rangle \parallel u?c \parallel \; !c \; \parallel \tau$$

$S\langle c, u\rangle$ is a site call with argument c and handle u. A new handle u is created for each site call – the environment generates a response, $u?c$, containing the result value c. Similarly, $!c$ publishes the output value c from a process. As is traditional, τ denotes internal events.

The rule SiteCall defines site calls. The transition label $S\langle c, u\rangle$ notifies the environment of the call. The call is replaced by the expression $?u$, which waits for the call to return. A site call occurs only when its parameters are constants; in $S(x)$, 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, which is then output. If the environment never produces a site return event, then the call blocks indefinitely. The Pub rule generates a publish action $!c$. If a variable is to be published, as in $!x$, the expression blocks until x is defined. Function expressions are evaluated using call-by-name in the Def rule. We assume a single global set of definitions D.

Sym1 and Sym2 are the standard rules for parallel composition.

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.

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

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 $S\langle c, u\rangle$, SiteRet introduces $u?c$, and Pub 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 $S\langle c, u\rangle$ or site returns $u?c$. This is because these events are only handled (eliminated) by the environment. The environment can also eliminate $!c$ events to obtain the output(s) of an expression.

4 Laws

Algebraic laws facilitate reasoning about process definitions. While we use advanced proof techniques to prove the laws, a programmer can use the laws directly to reason about programs without detailed knowledge of the underlying bisimulations. We begin with a few basic laws.

Proposition 1.

Commutativity of $\mid$:	$f \mid g \equiv g \mid f$
Identity for $\mid$:	$f \equiv f \mid \mathbf{0} \equiv \mathbf{0} \mid f$
Left identity for $>x>$:	$\mathbf{0} \equiv \mathbf{0} >x> g$

Proof. Direct from operational semantics

Briefly, we prove bisimulations using safe functions involving particular kinds of contexts [12]. For these proofs, we use *parallel composition contexts*, in which the context hole occurs only in a parallel composition. The function $\mathcal{F}_C$ is a function on process relations defined as follows:

$$\mathcal{F}_C(P,Q) = \{(C[P], C[Q]) \mid C \text{ is a parallel composition context and } (P,Q) \in \mathcal{R}\}$$

A function $\mathcal{F}$ on process relations is *safe* if $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightsquigarrow \mathcal{S}$ ($\mathcal{R}$ progresses to $\mathcal{S}$) implies $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightsquigarrow \mathcal{F}(\mathcal{S})$.

Lemma 1. *$\mathcal{F}_C$ is a* safe function*.*

A safe function $\mathcal{F}$ is useful because $\mathcal{R} \rightsquigarrow \mathcal{F}(\mathcal{R})$ implies that $\mathcal{R}$ is included in bisimilarity.

Proposition 2. *Associativity of* $|$

$$(f \mid g) \mid h \;\equiv\; f \mid (g \mid h)$$

Proof. Define the relation R as relating the above terms for all substitutions of f, g, and h: $\langle (f \mid g) \mid h, f \mid (g \mid h)\rangle \in R$. It is easy to show that R progresses to R by analysis of the rules Sym1 and Sym2. Thus R is a bisimulation and is included in bisimilarity.

Proposition 3. *Right distributivity of* $>x>$ *over* $|$

$$(f \mid g) >x> h \;\equiv\; (f >x> h) \mid (g >x> h)$$

Proof. Define the relation R as relating the above terms for all substitutions of f, g, and h. Show R progresses to $\mathcal{F}_C(R)$. The only transitions that can occur, on both sides, are Seq1N and Seq1V. For Seq1N, the derivations are simple congruences. For Seq1V, define the context $C[\bullet] = ([c/x]h \mid \bullet)$.

$$\begin{array}{ll} & (f \mid g) >x> h \\ \overset{!c}{\hookrightarrow} & \{\text{Seq1V and Sym1 applied to } f\} \\ & [c/x]h \mid ((f' \mid g) >x> h) \\ = & C[(f' \mid g) >x> h] \end{array}$$

$$\begin{array}{ll} & (f >x> h) \mid (g >x> h) \\ \overset{!c}{\hookrightarrow} & \{\text{Sym1 and Seq1V applied to } f\} \\ & ([c/x]h \mid (f' >x> h)) \mid (g >x> h) \\ = & \{\text{Associativity of } |\} \\ & [c/x]h \mid ((f' >x> h) \mid (g >x> h)) \\ = & C[(f' >x> h) \mid (g >x> h)] \end{array}$$

The case for Seq1V applied to g is analogous. Thus R progresses to $\mathcal{F}_C(R)$.

Proposition 4. *Associativity of* $>x>$ *. If $x \notin FV(h)$ then*

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

Proof. Included in the full paper.

5 Deterministic Operational Semantics

Determinism means that the outputs of a process are determined by the inputs to the process from the environment. As is typical of most process algebras, the asynchronous semantics of Orc given in Section 3 is highly nondeterministic. In this section we refine this semantics to create a deterministic semantics while preserving the essential concurrent and distributed nature of Orc computation. The key is to distinguish internal Orc actions from external events: internal evaluation must be confluent, so that only external events cause divergence (choice) in behavior. We must also prevent output race conditions during internal evaluation. The deterministic semantics eliminates internal choice from Orc, which simplifies the programming model significantly – without reducing the practical power of the language to support internet programming. All the examples in Section 2 function properly under the deterministic semantics.

The deterministic semantics models expression evaluation in *rounds*. A round consists of all possible internal actions —site calls, outputs and τ transitions— which are executed *eagerly*, i.e., as soon as possible, after receiving some response. The initial round proceeds without receiving any response; subsequently, a response is required at the start of each round. Thus, all actions corresponding to a response are executed before the next response is considered. After completing all possible internal actions in a round, the process becomes *quiescent*. The set of quiescent expressions, $Expr_Q$, is defined as follows:

$$q \in Expr_Q ::= \mathbf{0} \parallel S(x\,) \parallel\, !x \parallel q >x> e \parallel q \mid q \parallel q \textbf{ where } x :\in q \parallel ?u$$

The set of events is partitioned into *actions* and *responses*. Actions are initiated by an Orc process, while responses are initiated by the environment.

Actions	$a \in Act ::= \tau \parallel\ !c\ \parallel S\langle c, u\rangle$
Responses	$r \in Rsp ::= u?c$

Evaluation in rounds is formalized by partitioning the base transition relation into two sub-relations: $\hookrightarrow_{\mathrm{X}}$ for environmental events, and $\hookrightarrow_{\mathrm{N}}$ for internal actions.

Asynchronous	$\hookrightarrow\ : Expr \times Event \times Expr$	$\{$Defined in Figure 3$\}$
External	$\overset{l}{\hookrightarrow}_{\mathrm{X}} : Expr \times Event \times Expr$	$= \{(q, r, e) \mid q \overset{r}{\hookrightarrow} e\} \cup \{(\hat{q}, \tau, \hat{q})\}$
Internal	$\overset{a}{\hookrightarrow}_{\mathrm{N}} : Expr \times Act^* \times Expr_Q$	$= \{(e, \bar{a}, q) \mid e\ (\overset{a}{\hookrightarrow})^*\ q\}$
Round	$\overset{l\bar{a}}{\hookrightarrow}_{\mathrm{R}} : Expr \times Event^* \times Expr_Q$	$= \overset{l}{\hookrightarrow}_{\mathrm{X}} \circ \overset{\bar{a}}{\hookrightarrow}_{\mathrm{N}}$

The external transition relation $\hookrightarrow_{\mathrm{X}}$ is the identity relation on non-quiescent expressions, denoted by $\hat{q} : Expr \backslash Expr_Q$. To ensure that all possible internal actions are performed before any external events are accepted $\hookrightarrow_{\mathrm{N}}$ must produce a quiescent expression. It is possible that internal evaluation of an expression may not terminate; then the expression never becomes quiescent and accepts no further responses from the environment.

To prevent race conditions, we require that expressions be *functional*. An expression is functional if all values published in a single round are identical.

Proposition 5. *Internal evaluation of functional expressions is confluent.*

Proof. Proof included in full paper.

Functional behavior can be ensured by several means, including syntactic restrictions or semantic analysis. For example, if *let* is interpreted as a site rather than a publish expression $!p$, then all expressions are functional. Another way to ensure that all expressions are functional is to prevent more than one output in a round. For example, consider the following extension of Orc: expression $!p$ is prohibited in source programs, but a variant of sequential composition, $>!x>$, is allowed, where

$$f >!x> g = f >x> (!x \mid g)$$

This extension guarantees that at most one $!p$ will be executed in a round, thus avoiding a race for the output order.

The deterministic semantics, based on rounds and functional expressions, may seem overly complex. Actually, it is easier to write programs under this semantics, because actions are guaranteed to happen immediately rather than being arbitrarily delayed.

6 Trace Semantics

We develop a denotational semantics for Orc using event trees as the semantic domain. Traces are easily derived from these denotations. The nodes of a tree are labeled by action events, and edges are labeled by response events or variables. An edge labeled by a variable represents a blocked process – when the variable becomes bound the nodes below the edge are merged with the node above the variable edge. A node is represented as a $Tree$ containing a set of actions, $Set(Act)$, for the node label and a map, $ResponseMap$, which defines labeled edges and subtrees.

$$\begin{aligned} t \in Tree &= Set(Act) \times ResponseMap \\ m \in ResponseMap &= Response \mapsto Tree \\ r \in Response &::= u?c \parallel x \end{aligned}$$

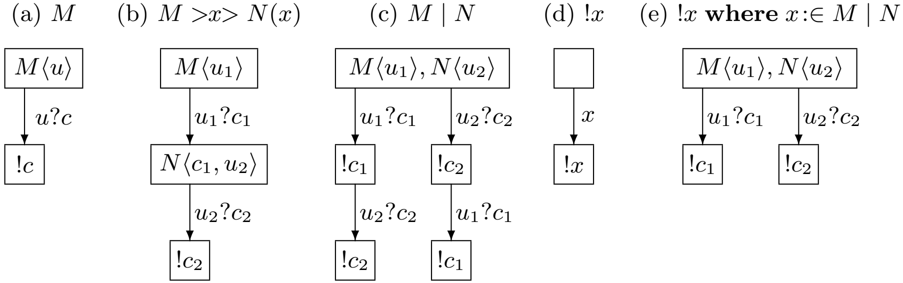

Fig. 4. Example trees and tree composition

$$
\begin{aligned}
\mathcal{T} &: \ Expr \rightarrow Tree \\
\mathcal{T}[\![\, F_i(x_i) \triangleq g_i; e \,]\!] &= \mathcal{T}r[\![\, e \,]\!]\rho_0 \text{ where } \rho_0 = Y(\lambda\rho.\{F_i \mapsto \lambda p.[p/x_i]\mathcal{T}r[\![\, g_i \,]\!]\rho) \\
\mathcal{T}r[\![\, \mathbf{0} \,]\!]\rho &= (\emptyset, \emptyset) \\
\mathcal{T}r[\![\, S(p) \,]\!]\rho &= Wait(p, (\{S\langle p, u\rangle\}, \{u?c \mapsto\, !c\,\})) \text{ where } u \text{ new} \\
\mathcal{T}r[\![\, !p \,]\!]\rho &= Wait(p, (\{\, !p\,\}, \emptyset)) \\
\mathcal{T}r[\![\, f \mid g \,]\!]\rho &= \mathcal{T}r[\![\, f \,]\!] \mid \mathcal{T}r[\![\, g \,]\!] \\
\mathcal{T}r[\![\, f > x > g \,]\!]\rho &= \mathcal{T}r[\![\, f \,]\!] > x > \mathcal{T}r[\![\, g \,]\!] \\
\mathcal{T}r[\![\, g \textbf{ where } x{:}{\in}\, f \,]\!]\rho &= \mathcal{T}r[\![\, g \,]\!] \ :x{\in}\ \mathcal{T}r[\![\, f \,]\!] \\
\mathcal{T}r[\![\, F(p) \,]\!]\rho &= \rho F(p) \\
Wait(c, t) &= t \\
Wait(x, t) &= (\emptyset, \{x \mapsto t\})
\end{aligned}
$$

Fig. 5. Trace semantics

Composition operators:

$$
\begin{aligned}
(s, m) \mid (s', m') &= (s \cup s', (m \triangleright m') \cup (m' \triangleright m)) \\
(\{n_i,\ !c_j\,\}, \{r_k \mapsto u_k\}) > x > t &= (\{n_i\}, \{r_k \mapsto u_k > x > t\}) \mid [c_j/x]t \\
(s, m)\ :x{\in}\ (s', m') &= \begin{cases} (s \cup s', (m \triangleright_x m') \cup (m \triangleleft_x m')) & !c\ \notin s' \\ [c/x](s, m) & !c\ \in s' \end{cases}
\end{aligned}
$$

Interleaving functions:

$$
\begin{aligned}
\{r_i \mapsto t_i\} \triangleright m &= \{r_i \mapsto (t_i \mid (\emptyset, m))\} \\
\{r_i \mapsto t_i\} \triangleright_x m &= \{r_i \mapsto (t_i\ :x{\in}\ (\emptyset, m))\} \\
m \triangleleft_x \{r_i \mapsto t_i\} &= \{r_i \mapsto ((\emptyset, m)\ :x{\in}\ t_i)\}
\end{aligned}
$$

Substitution:

$$
\begin{aligned}
[c/x](s, \{x \mapsto t, r_i \mapsto t_i\}) &= ([c/x]s, \{r_i \mapsto [c/x]t_i\}) \mid [c/x]t \\
[c/x](s, \{r_i \mapsto t_i\}) &= ([c/x]s, \{r_i \mapsto [c/x]t_i\}) \text{ where } x \notin r_i
\end{aligned}
$$

Fig. 6. Operations on traces

Figure 4 shows some Orc expressions and their corresponding trees. In (a), site M is called, the process waits for the response from the environment, and then the result c is published. The generic label $u?c$ denotes a family of edges, one for each particular value of c. In (b) the output of site M is used as an argument to site N. In (c), M and N are called together; the tree has two paths depending on whether M or N returns first. Part (d) illustrates the simplest blocked process, which waits for x to be defined and then publishes its value. In (e), the **where** expression terminates execution of $M \mid N$ after the first site returns a value.

The tree semantics is defined by the function $\mathcal{T}$ in Figure 5. The function $\mathcal{T}r$ defines traces relative to an environment ρ containing interpretations of any process definitions F. The base environment, ρ_0, is created by taking the fixed point of the mutually recursive definitions. The semantics is fundamentally compositional, although free variables are represented explicitly in response maps

and eliminated via substitution, rather than λ abstraction. The function $Wait$ creates a blocked process for any expression that involves a free variable.

Each composition operator has a corresponding operation on traces, as defined in Figure 6. These combine trees using the interleaving functions, which correspond closely to the operational semantics. For example, the two cases $(m \triangleright m')$ and $(m' \triangleright m)$ in the definition of $|$ correspond to rules SYM1 and SYM2. The set of external responses in the two sub-trees must be disjoint; this is always true because response labels u are fresh for each call. The requirement for functional behavior is visible in the definition of $:x\in$, which is not a function unless there is a unique c for which $!c \in s'$. Processes are unblocked by substitution on trees: when a substitution provides a value for a response variable x in a tree, the corresponding subtree is instantiated and merged with the main tree.

The function $Traces$ computes the set of traces for a tree by constructing all paths through the tree. The traces of an Orc expression are given by the composition of $Traces$ and $\mathcal{T}$:

$$\begin{aligned} Traces(a_i, \emptyset) &= Perm(a_i) \\ Traces(a_i, \{x_k \mapsto t_k\}) &= \{s \,.\, x_k \,.\, s' \mid s \in Perm(a_i), s' \in Traces(t_k)\} \\ Traces_{Orc} &= Traces \circ \mathcal{T} \end{aligned}$$

The trace semantics is closely related to the operational semantics: It characterizes the set of all possible sequences of actions that an expression can perform. The set of traces of a term e is derived from the transition relation:

$$Traces_{\hookrightarrow}[\![e]\!] = \{s_1 ... s_n \mid e \overset{s_1}{\hookrightarrow}_{\text{R}} e_1 \overset{s_2}{\hookrightarrow}_{\text{R}} e_2 \, ... \, e_{n-1} \overset{s_n}{\hookrightarrow}_{\text{R}} e_n\}$$

Proposition 6. *For any functional term e and substitution σ,*

$$Traces_{\hookrightarrow}[\![\sigma e]\!] = \sigma(Traces_{Orc}[\![e]\!])$$

Proof. By structural induction on Orc terms. Included in full paper.

Lemma 2. *(Equivalence of operational semantics with rounds and deterministic trace semantics.) For any closed functional term e, $Traces_{\hookrightarrow}[\![e]\!] = Traces_{Orc}[\![e]\!]$*

Lemma 3. *The operational semantics is deterministic with respect to external events.*

7 Related Work

We give a brief overview of the work most related to that described in this paper; a more complete account will appear in the final paper. An extended discussion of Orc programming model along with more real-world examples appears in [10] and a denotational semantics for it in [7].

Orc draws extensively from experience with process algebras, particularly CCS [8], and CSP [6] and π-calculus [9]. These formalisms represent a multi-threaded computation by an expression which has interesting algebraic properties. The operational semantics and the proof by bisimulation in this paper are

directly motivated by CCS. The asynchronous version of Orc can be translated to a variant of π-calculus [9] with support for process termination. Although process termination has been studied extensively [1, 11], we do not know of any work that ties termination to communication as in Orc. We are studying the translation of the deterministic semantics of Orc to π-calculus, which seems non-trivial because of eager evaluation. Also, we are now comparing the Orc programming model with the work of Benton, Cardelli and Fournet[2] which employs join calculus[4] as its basis. Orc is similar in its treatment of time and round-based computations to synchronous programming models [3, 5].

References

1. Amadio and Prasad. Localities and failures. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 14, 1994.
2. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *TOPLAS*, 26(5):769 – 804, September 2004.
3. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
4. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the POPL*. ACM, 1996.
5. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.
6. C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.
7. T. Hoare, G. Menzel, and J. Misra. A tree semantics of an orchestration language. In M. Broy, editor, *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`.
8. R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International, 1989.
9. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
10. J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series, Marktoberdorf, Germany, 2004.
11. J. Riely and M. Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, 2001.
12. D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.